

Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И
КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Податочни структури

Владимир Трајковиќ, Слободан Калајџиски,
Магдалена Костоска Горчевска, Христина
Михајлоска Трпческа,
Ефтим Здравевски, Петре Ламески
Бојана Котеска, Илинка Иваноска, и
Славе Темков

Скопје, Септември 2023

Благодарност

Благодарност до колегите Ана Маџевска Богданова и Анастас Мишев за нивниот придонес во развојот на предметите и материјалите каде се изучуваат податочните структури, како и сите колеги кои во некој момент во времето биле дел од овие предмети.

Содржина

Благодарност	ii
Листа на слики	vii
Листа на табели	x
1 Вовед	1
1.1 Асимптотска анализа	3
1.2 Правила	6
1.2.1 Циклуси	7
1.2.2 Вгнездени циклуси	8
1.2.3 Последователни извршувања	9
1.2.4 Условни извршувања	9
1.2.5 Рекурзии	9
1.3 Принципи на споредба	10
1.4 Нотации	15
2 Низи	16
2.1 Операции со низи	18
2.2 Проблеми со низи	22
2.3 Задачи за вежбање	26
2.4 ArrayList во Java	26
3 Листи	29
3.1 Еднострano поврзани листи	30
3.1.1 Операции со еднострano поврзани листи	32
3.1.2 Едноставни проблеми со еднострano поврзани листи	39
3.1.3 Напредни проблеми со еднострano поврзани листи	44
3.1.4 Задачи за вежбање	49
3.2 Двострано поврзани листи	51
3.2.1 Операции со двострано поврзани листи	52

3.2.2	Едноставни проблеми со двострано поврзани листи	59
3.2.3	Напредни проблеми со двострано поврзани листи	64
3.2.4	Задачи за вежбање	69
3.3	LinkedList во Java	71
4	Еднодимензионални податочни структури	75
4.1	Апстрактни податочни типови	75
4.2	Стек (stack, магацин)	76
4.2.1	Операции со стек	77
4.2.2	Имплементација на стек со низа	77
4.2.3	Имплементација на стек со листа	80
4.2.4	Stack во Java	82
4.2.5	Едноставни проблеми со стек	83
4.2.6	Напредни проблеми со стек	93
4.2.7	Задачи за вежбање	101
4.3	Редица (ред)	106
4.3.1	Операции со редица	106
4.3.2	Имплементација на редица со низа	106
4.3.3	Имплементација на редица со листа	112
4.3.4	Queue во Java	114
4.3.5	Едноставни проблеми со редица	115
4.3.6	Напредни проблеми со редица	129
4.3.7	Задачи за вежбање	142
4.3.8	Проблеми со комбинирање на стек и редица	146
4.3.9	Задачи за вежбање	159
4.4	Приоритетна редица	162
5	Хеш табели	164
5.1	Хеш табела со затворени „кофички”	166
5.1.1	Операции со хеш табела со затворени „кофички”	168
5.1.2	Едноставни проблеми со хеш табели со затворени „кофички”	171
5.1.3	Напредни проблеми со хеш табели со затворени „кофички”	182
5.1.4	Задачи за вежбање	197
5.2	Хеш табела со отворени „кофички”	203
5.2.1	Операции со хеш табела со отворени „кофички”	206
5.2.2	Двојно хеширање	212
5.2.3	Едноставни проблеми со хеш табели со отворени „кофички”	215
5.2.4	Напредни проблеми со хеш табели со отворени „кофички” .	228
5.2.5	Задачи за вежбање	238

5.3	Hashmap и Hashtable во Java	244
6	Дрва	250
6.1	Теоретски основи за дрва	250
6.1.1	Вовед	250
6.1.2	Терминологија	250
6.1.3	Репрезентација на дрвата	253
6.1.4	Бинарни дрва	254
6.1.5	Трансформација на шума од произволни дрва во бинарно дрво	255
6.1.6	Претставување на бинарните дрва со Java код	259
6.1.7	Изминување на бинарните дрва	260
6.1.8	Основни манипулации со јазлите во бинарните дрва	263
6.1.9	Задачи	268
6.1.10	Задачи за вежбање	289
6.2	Пребарувачки дрва	291
6.2.1	Бинарни пребарувачки дрва	291
6.2.2	Основни манипулации со јазлите во бинарните пребарувачки дрва	292
6.2.3	Сложеност на операциите кај бинарни дрва наспроти бинарни пребарувачки дрва	299
6.2.4	Балансирали бинарни дрва	300
6.2.5	AVL дрва	302
6.2.6	В дрва	313
6.2.7	В+ дрва	318
6.2.8	Задачи	322
6.2.9	Задачи за вежбање:	329
7	Графови	331
7.1	Што е тоа Граф?	331
7.2	Дефиниции и својства	332
7.3	Репрезентација на графови	334
7.3.1	Матрица на соседност	334
7.3.2	Листа на соседи	337
7.3.3	Листа на работи	339
7.4	Видови патеки кај графовите	341
7.5	Некои видови на графови	343
7.6	Алгоритми за изминување и пребарување на графовите	346
7.6.1	Изминување по длабочина - DFS	346

7.6.2	Изминување по широчина - BFS	348
7.6.3	Двонасочно пребарување	349
7.6.4	Алгоритмот Дијикстра (Dijikstra)	350
7.6.5	Алгоријатм на Флојд-Варшал (Floyd-Warshall)	351
7.7	Проблеми во кои се применуваат алгоритми со графови	353
7.7.1	Наоѓање на најкраток пат	353
7.7.2	Поврзаност на граф	353
7.7.3	Мрежен проток	354
7.7.4	Детекција на циклуси	354
7.7.5	Партиционирање и боене	354
7.7.6	Споредување на графови	354
7.7.7	Хамилтонови и Ојлерови патеки и циклуси	354
7.7.8	Артикулациони точки	355
7.7.9	Проблем на покриеност на темињата	355
7.7.10	Детекција на планарни графови	355
7.7.11	Изоморфизам на графови	355
7.7.12	Проблем на патувачки трговец - Traveling Salesman Problem (TSP)	355
7.7.13	Тополошко сортирање	355
7.7.14	Минимално распнувачко дрво (Minimum spanning tree - MST)	357
7.8	Едноставни проблеми со графови	359
7.9	Напредни проблеми со графови	370
7.10	Задачи за вежбање	381

Листа на слики

1-1 Одредување асимптотска горна граница	5
1-2 Можни горни асимптотски граници на $f(n)$	6
2-1 Визуелен приказ на низа	17
3-1 Визуелен приказ на јазел на едностррано поврзана листа	30
3-2 Пример на едностррано поврзана листа	31
3-3 Пример на вметнување на јазел на почеток на листа	34
3-4 Пример на вметнување на јазел после даден јазел	35
3-5 Пример на бришење на даден јазел	37
3-6 Визуелен приказ на јазел на двоострано поврзана листа	51
4-1 Стек од книги.	76
4-2 Имплементација на стек со низа.	77
4-3 Имплементација на стек со листа.	80
4-4 Имплементација на редица со низа.	107
4-5 Пример за последователни додавања и вадења на елементи во редица.	107
4-6 Визуелизација на циклична низа.	108
4-7 Имплементација на редица со циклична низа.	108
4-8 Пример за последователни додавања и вадења на елементи во редица имплементирана со циклична низа.	109
4-9 Имплементација на редица со едностррано поврзана листа.	112
4-10 Приказ на структурата.	146
5-1 Визуелен приказ на процесот на хеширање.	165
5-2 Хеш табела со затворени „кофички“ без колизии.	167
5-3 Хеш табела со затворени „кофички“ со колизии.	167
5-4 Хеш табела со отворени „кофички“ без колизии.	204
5-5 Хеш табела со отворени „кофички“ со колизии.	205
5-6 Пребарување во ОВНТ.	208
5-7 Додавање во ОВНТ.	210
5-8 Бришење во ОВНТ.	211

5-9 Двојно хеширање во ОВНТ.	213
6-1 Три еквивалентни претстави на едно дрво	251
6-2 Корен на дрво со поддрва	252
6-3 Пример на дрво	253
6-4 Примери на подредени дрва	254
6-5 Нецелосно и целосно бинарно дрво	255
6-6 Репрезентација на едноставни дрва со бинарно дрво	256
6-7 Процес на трансформација на произволно дрво во бинарно дрво	257
6-8 Репрезентација на шума од дрва со бинарно дрво	258
6-9 Репрезентација на шума од дрва со бинарно дрво	261
6-10 Вметнување јазел со покажувач q под јазел со покажувач p во лево и десно поддрво од едно бинарно дрво	263
6-11 Пример на бинарно дрво	265
6-12 Пример на бинарно дрво	275
6-13 Пример на бинарно дрво	289
6-14 Пример на бинарно дрво	290
6-15 Пример на бинарно дрво	290
6-16 Бинарно пребарувачко дрво (а) и обично бинарно дрво (б)	291
6-17 Бинарно пребарувачко дрво пред и по вметнување на јазел со клуч	294
6-18 Бришење на терминален јазел пред (а) и после операцијата (б)	295
6-19 Бришење на јазел со едно дете (клуч 4) пред (а) и после операцијата (б)	296
6-20 Бришење на јазел со две деца (клуч 2) пред (а) и после операцијата (б)	297
6-21 Балансирано бинарно дрво (а) и небалансирано бинарно дрво (б)	301
6-22 AVL дрво (а) и бинарно пребарувачко дрво (б)	303
6-23 Еднократна ротација на десно од (а) кон (б) и еднократна ротација на лево од (б) кон (а)	304
6-24 Пример за балансирање на дрво со помош на еднократна ротација во десно	305
6-25 Пример за балансирање на дрво со помош на еднократна ротација во лево при додавање на јазлите со клучеви 1, 2, 3, 4, 5, 6 и 7 последователно	306
6-26 Пример кога еднократната ротација не помага во балансирање на дрвото	307
6-27 Двократна ротација десно-лево	308
6-28 Двократна ротација лево-десно	308
6-29 Двократна ротација лево-десно	308

6-30 Постапка на градење на AVL дрво	309
6-31 В дрво од ред 3	315
6-32 В дрво од ред 4	315
6-33 Внесување на клуч во непополнет јазел од В дрво од ред 3	316
6-34 Внесување на клуч во пополнет јазел од В дрво од ред 3 со помош на делење на терминален јазел	317
6-35 Недозволена состојба за В дрво од ред 3	317
6-36 Делење на внатрешен јазел кај В дрво од ред 3	317
6-37 Делење на внатрешен јазел кај В дрво од ред 3 со што се предиз- викува зголемување на висината на дрвото	320
6-38 Пример за В+ дрво	321
6-39 Пример на бинарно пребарувачко дрво	322
7-1 Мостовите на паркот во Konigsburg (Калининград)	331
7-2 Илустрација на различни видови на графови	333
7-3 Пример за комплетен граф со 5 темиња	343
7-4 Пример за граф Циклус	343
7-5 Пример за граф ѕвезда	344
7-6 Пример за граф патека	344
7-7 пример за бипартитен граф	344
7-8 Граф тркало	345
7-9 Дрво	345
7-10 Петерсен граф	345
7-11 Илустрација на Прим	358
7-12 Илустрација на Крускал	360
7-13 Репрезентација на лавиринтот со граф	366

Листа на табели

1.1	Споредба на време на извршување за различни кардиналности	14
1.2	Најчести кардиналности	15
6.1	Сложеност на најчестите операции кај бинарно дрво со N јазли	300
6.2	Сложеност на најчестите операции кај бинарно пребарувачко дрво со N јазли	300
6.3	Споредба помеѓу B дрва и B+ дрва	319

Глава 1

Вовед

Пишувањето на програми може да биде тежок процес, посебно ако се прави без да се следат добри препораки или упатства. Структурното програмирање е еден принцип за развој на програми кои го “разбива“ кодот на помали делови што соодветствува на решавањето на сложен проблем со негово делење на поедноставни проблеми. Со овој процес се дава одредена структура на развојот на програми, од каде потекнува и името на овој пристап.

Информациите кои треба да бидат процесирани од страна на програмата се чуваат во **податочни структури - data structures** (arrays, records, lists, stacks, trees, files) поддржани од соодветниот програмски јазик. Податочните структури ги групираат податоците. Добар избор на податочни структури може да ја поедностави изработката на некоја програма, но важи и обратното. Изборот на податочната структура влијае на јасноста, проширливоста, брзината и мемориските побарувања за одредено програмско решение.

Пример: Споредете го наоѓањето на телефонски број за познато презиме и наоѓањето на презиме за познат телефонски број со користење на хартиен телефонски именик.

Податочните структури можат да бидат **статични** (да дозволуваат промени само на вредностите – како низи (arrays) или записи (records)) и **динамични** (да дозволуваат промени и во изгледот, посебно големината – како магацини (stacks), листи (lists), дрва (trees), датотеки (files)).

Типични акции кои се извршуваат над податочните структури се:

- пронаоѓање,
- пребарување,
- пребројување,
- вметнување,
- сортирање и
- бришење.

Пред да се започне со пишување на програма треба да се разбере проблемот и да се креира решение или алгоритам, чекор по чекор. Ако не знаете како да опишете решение за некој проблем, тогаш е сосема извесно дека нема да знаете ниту да напишете програма која ќе го решава тој проблем. За да биде корисен, алгоритамот треба да нуди решение за генерален, добро специфициран проблем. Алгоритамски гледано еден проблем се специфицира со утврдување на сите можни влезни податоци со кои тој алгоритам ќе работи, поведенија или постапки за нивна обработка, како и сите можни излезни податоци кои алгоритамот ќе ги креира. Важно е да се направи разлика помеѓу правилно поведение за група влезни податоци и правилно поведение за сите можни влезни податоци.

Според тоа алгоритам е процес кој прима било кој можен влез и го трансформира во посакуван излез.

За да се описат алгоритмите потребно е да се користи некаква нотација за опишување на последователните чекори кои треба да се извршат при негова реализација. Такви нотации може да бидат: говорен јазик – македонски, английски; псевдокод; програмски јазик.

Псевдокодот може да се дефинира како програмски јазик кој не се грижи за синтаксата. Тој е попрецизен од говорните јазици, но понепрецизен од програмските јазици.

Еден можен пристап за опис на алгоритам би бил: опишување на идејата во говорен јазик, опишување на концептот на алгоритамот во псевдокод, детализирање на алгоритамот во конкретен програмски јазик.

Алгоритмите треба да бидат коректни, ефикасни и лесни за имплементација.

Коректноста на алгоритмите треба да се докаже (со неможност да се најде валиден влезен податок за кој алгоритамот не работи правилно, со потврда дека спротивниот алгоритам дава грешни резултати, со математичка индукција). Важно е да се разбере дека исказот “решението е очигледно“ најчесто води во заблуда и создавање на погрешни алгоритми.

Не постои алгоритам за наоѓање на алгоритми. Смислувањето на алгоритам е обично многу потешко од пишување програма (за познат алгоритам).

Изкусен дизајнер на алгоритми е всушност експерт за ефикасност на алгоритмите. Ефикасни се оние коректни алгоритми кои за дадени влезни податоци завршуваат побрзо од други коректни алгоритми кои работат со истите влезни податоци. При тоа треба се напомене дека наоѓањето на најефикасен алгоритам не е секогаш од пресудно значење. Имено статистички гледано само 20% од кодот на даден програм се извршува 80% од времето додека тој програм е стартиран. Од тука следи дека на вкупната ефикасност на програмата значително влијае само 20% од кодот, па доволно е само тој да се оптимизира.

Иако брзината на извршување на некој алгоритам реално зависи од хардверот

над кој тој се извршува, па забрзување на алгоритамот може да се постигне со негово извршување на побрз хардвер, тоа може да биде лимитирано со технички ограничувања (не постои побрз хардвер) и економски ограничувања (нема пари за да се купи побрз хардвер).

Нé интересираат одговорите на следните прашања:

- Како да се процени ефикасноста на алгоритмите?
- На кој начин да се споредат два алгоритми?

1.1 Асимптотска анализа

За да се одговори на претходните прашања, потребно е да се користи модел на извршување кој не зависи од хардверот. Вообичаено се претпоставува дека алгоритмите се извршуваат на хипотетички компјутер наречен “*x*“ или **RAM**. Моделот на случаен пристап (Random Access Machine - RAM) е теоретски компјутерски модел кој се користи за анализа на алгоритми. Овој модел апстрактира детали на конкретната компјутерска архитектура и обезбедува идеален простор за анализа на временска комплексност на алгоритми. При тоа се претпоставува дека:

- Секоја едноставна операција ($+$, $*$, $-$, $=$, if, call, return) се извршува во една временска единица.
- Циклусите (јамките), процедурите и функциите се извршуваат во онолку временски единици колку што содржат итерации.
- Постои неограничена меморија.

Во овој модел **времето потребно за извршување** на алгоритамот за дадени влезни податоци е **еднакво со бројот на едноставните операции**. Овој модел е идеален и служи како теоретски алат за анализа, без влијание на конкретни аспекти на хардверската архитектура.¹

Хипотетичкиот компјутер RAM ќе го користиме за споредба на ефикасноста на два алгоритми. За секој од алгоритмите се бројат операциите при извршувањето како што беше објаснето во воведниот дел. Во следниот пример ќе се илустрира како се определува вкупниот број операции потребни за извршување на алгоритамот, односно како се определува времетраењето на алгоритамот што го решава дадениот проблем.

Пример: Да се напише псевдокод за пресметување на функцијата $\sum_{i=1}^n x^i$

sumX(x, n)	Број на операции
{	

```

1 sum = 0;           1
2 for (int i=1; i<=n; ++i)   1+(n+1)+(n)=2n+2
3     sum = x*(sum+1);   n(1+1+1)=3n
4 return sum;        1
}

```

Анализата на овој код е едноставна. Линиите 1 и 4 траат една временска единица, а линијата 3 трае три временски единици (едно собирање, едно множење и едно доделување). Линијата 3 се извршува n пати (поради наредбата for). Линијата 2 содржи една иницијализација, $n + 1$ споредба и n инкрементирања или вкупно: $2n + 2$. Вкупниот број операции е $5n + 4$.

Значи $f(n) = 5n + 4$. Бидејќи времето на извршување е еднакво на бројот на извршените операции, може да се запише и $T(n) = f(n) = 5n + 4$.

Уште од почетокот на оваа глава зборувавме за можност за споредба на два алгоритми – алгоритам А и алгоритам В кои решаваат еден ист проблем. Да ја разгледаме следната постапка. Нека е направена анализа за потребниот број операции за секој од алгоритмите и нека се добиени величините $T_A(n)$ и $T_B(n)$ соодветно. Да се потсетиме дека n е мерка за големината на проблемот и го претставува бројот на влезните податоци. Останува само да се споредат двете функции $T_A(n)$ и $T_B(n)$ и така ќе се одреди кој е поефикасниот алгоритам.

Сепак, не е толку едноставно. Во општи случај не се знае колку може да биде n ($10, \dots, 100, \dots, 100\ 000, \dots, 1000\ 000$). Ако може да се покаже дека $T_A(n) \leq T_B(n)$ за сите $n \geq 0$, тогаш алгоритамот А побргу се извршува и тој е подобар од алгоритамот В.

Но, бројот на влезни податоци n не е однапред познат и потребна е друга алатка за да се изврши споредувањето. Одговорот лежи во одредување на асимптотска граница. Ова значи дека функцијата (n) за поголеми вредности на n ќе се доближува до асимптотската граница, но никогаш нема да ја пресече.

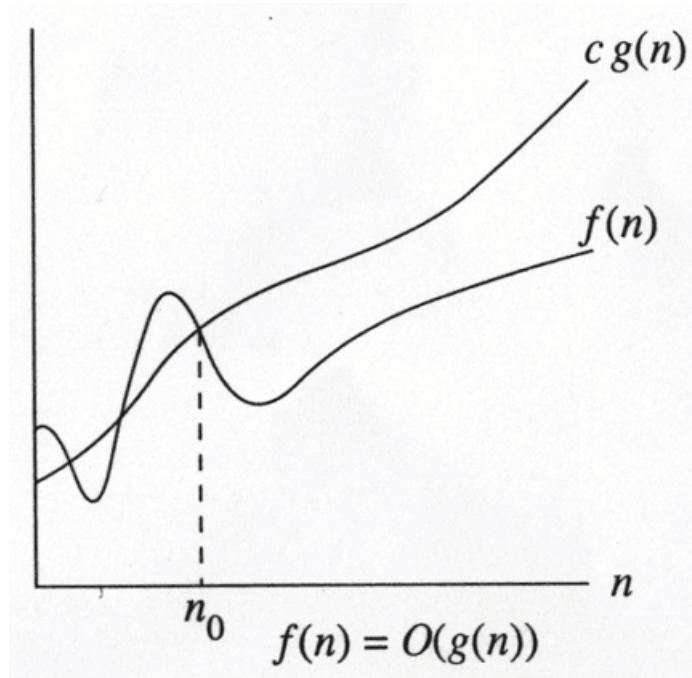
Заради полесно објаснување, ќе го искористиме фактот дека во RAM моделот може да се изедначат $f(n)$, функцијата добиена со броене на операциите во алгоритамот, и (n) . Воведуваме поим за **асимптотска горна граница О** (се чита “**големо о**”).

Дефиниција за асимптотска горна граница: велиме дека $f(n) = O(g(n))$ ако постои n_0 и некоја константа $c > 0$ така што за сите:

$$n \geq n_0, \text{ важи } f(n) \leq cg(n).$$

Ова може да се види на слика 1-1

Поинаку кажано: за функцијата $f(n)$ постои функција $g(n)$ која секогаш “е над” неа, за било колкав број податоци n почнувајќи од некое n_0 т.е. $f(n)$ може



Слика 1-1: Одредување асимптотска горна граница

да го опишеме со помош на $g(n)$.

Пример: Да ја разгледаме функцијата $f(n) = 8n + 128$. Да се провери дали може $f(n) = O(n^2)$, т.е. дали може функцијата n^2 да биде асимптотска горна граница на $f(n)$:

$$f(n) \leq cn^2, c = 1 \quad (1.1)$$

$$\Rightarrow 8n + 128 \leq n^2 \quad (1.2)$$

$$\Rightarrow 0 \leq n^2 - 8n - 128 \quad (1.3)$$

$$\Rightarrow 0 \leq (n - 16)(n + 8) \quad (1.4)$$

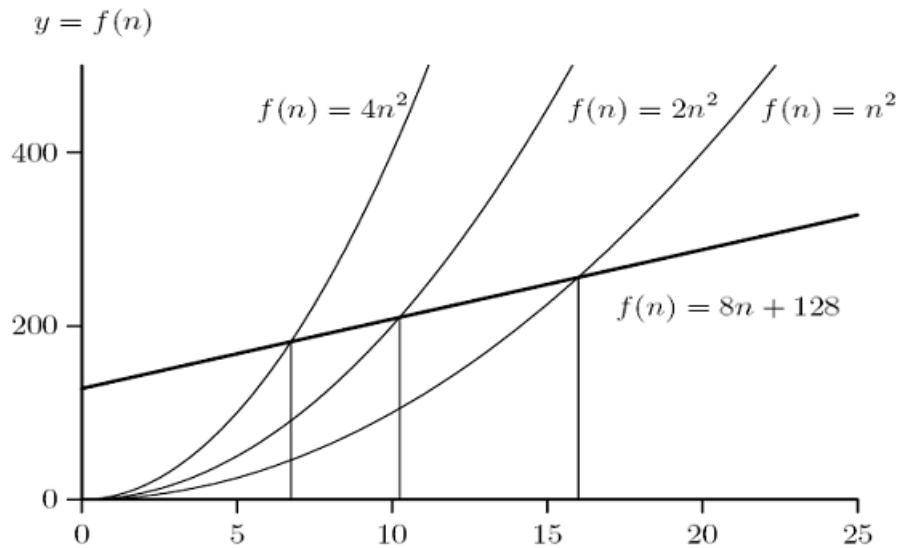
$$\forall n \geq 0, (n + 8) > 0 \quad (1.5)$$

$$(n_0 - 16) \geq 0 \Rightarrow n_0 = 16 \quad (1.6)$$

Значи, една можна асимптотска горна граница на $f(n)$ е $O(n^2)$. На слика 1-2 се прикажани неколку можни горни асимптотски граници за $f(n)$ со различни вредности на n_0 . Малку подоцна ќе биде описано како таа се одредува кога се работи за определување сложеност на алгоритмите.

Постојат неколку правила за работа на нотацијата “тоглемо $O(\cdot)$ ”. Тие ќе ни помогнат при определувањето на сложеноста на даден алгоритам.

Правило 1: Се отстрануваат сите членови, освен членот со највисок степен,

Слика 1-2: Можни горни асимптотски граници на $f(n)$

т.е. најголема кардиналност

Пример: $O(n^2 + n \log n + n) \rightarrow O(n^2)$

Правило 2: Се отстрануваат константите

Пример: $O(3n^2) \rightarrow O(n^2)$ $O(1024) \rightarrow O(1)$

Да се вратиме на почетокот на овој дел – определување сложеност на даден алгоритам и споредување различни алгоритми.

Според RAM моделот, се споредува времето за извршување на алгоритмите во:

- најдобро време на извршување,
- средно време на извршување,
- најлошо време на извршување на даден алгоритам.

Вообичаено, од интерес е најлошото време на извршување на даден алгоритам. Тоа дава претстава за времето (односно ресурсите) потребно да се изврши алгоритамот при најлошиот можен влез. Средното време на извршување на алгоритамот не ја нуди таа информација, додека најдоброто време на извршување обично бара специјален влез.

1.2 Правила

Да разгледаме едноставен фрагмент со псевдокод со кој се реализира сумата:

$$\sum_{i=1}^n i^3$$

sumP(n)

Број на операции

```

{
1 partial_sum = 0;           1
2 for (int i=1; i<=n; ++i) 2n+2
3     partial_sum += i*i*i; 4n
4 return partial_sum;       1
}

```

Според анализата во претходниот пример, $f(n) = 6n + 4$. Горната асимптотска граница се одредува според правилата 1 и 2, па се добива $O(n)$. Велиме дека овој алгоритам е од ред (n) или дека има **линеарна сложеност**. Горната асимптотска граница, $O(.)$, ни кажува како се однесува алгоритамот во зависност од n во “најлош случај”.

Следуваат неколку правила кои ќе ни помогнат да ги опишеме алгоритмите со помош на броене на инструкциите, да ја определиме горната асимптотска граница $O(.)$ и на тој начин да ја увидиме комплексноста, односно сложеноста на алгоритмите.

Примерите во оваа глава ќе бидат дадени во псевдо-код.

1.2.1 Циклуси

Времето на извршување на циклуси е еднакво на сумата на времињата потребни да се извршат операциите во циклусот помножено со бројот на итерации.

Пример 1: Следниот програмски сегмент го бара најголемиот елемент во низа со n елементи и е од ред $O(n)$

```

find_max(arrayIn){
    maxElement = arrayIn[0]
    for(i=1; i<n; i++){
        if(arrayIn[i]>maxElement)
            maxElement = arrayIn[i]
    }
    return maxElement
}

```

Пример 2: Следниот програмски сегмент врши бинарно пребарување даден елемент во (подредена во растечки редослед) низа, односно бара дали дадениот елемент се наоѓа во низата. Алгоритмот напрво го гледа средишниот елемент на низата (просторот на пребарување е целата низа) и притоа го споредува елементот кој го бара со средишниот:

- доколку средишниот елемент е бараниот, се враќа

- доколку елементот кој го бараме е помал од средишниот, ќе го бараме во првата половина на низата, однодно просторот на пребарување сега ќе биде пола низа, од почеток до средина), и повторно продолжуваме на сличен начин: го бараме средишниот елемент на новиот простор на пребарување, споредуваме итн...
- доколку елементот кој го бараме е поголем од средишниот, ќе го бараме во втората половина на низата, однодно просторот на пребарување сега ќе биде пола низа, од средина до крај), и повторно продолжуваме на сличен начин: го бараме средишниот елемент на новиот простор на пребарување, споредуваме итн..

Бинарното пребарување е ефикасно бидејќи го делува просторот за пребарување на пола во секој чекор. Во најлошиот случај, временската комплексност е $O(\log(n))$

```
binary_search(arrayIn, targetElement){
    start = 0
    end = n-1
    while(start<=end){
        mid = (start + end)/2
        if(targetElement == arrayIn[mid])
            return mid
        else if (targetElement < arrayIn[mid])
            end = mid - 1
        else
            start = mid + 1
    }
    return -1
}
```

1.2.2 Вгнездени циклуси

Времето на извршување на вгнездените циклуси е еднакво на сумата на времињата потребни да се извршат операциите во циклусот помножено со производот од бројот на итерации на циклусите.

Пример: Следниот програмски сегмент е од ред $O(n^2)$

```
for( i=0; i<n; i++ )
    for( j=0; j<n; j++ )
        k++;
```

1.2.3 Последователни извршувања

При последователни извршувања доволно е да се земе најголемата кардиналност. На пример следниот програмски сегмент има кардиналност $O(n)$ следена со $O(n^2)$ и вкупно се зема дека има кардиналност $O(n^2)$:

```
for( i=0; i<n; i++)
    a[i] = 0;
for( i=0; i<n; i++ )
    for( j=0; j<n; j++ )
        a[i] += a[j] + i + j;
```

1.2.4 Условни извршувања

Фрагментот

```
if( condition )
    S1
else
    S2
```

има времетраење кое е еднакво на времетраењето на тестот (condition) собрано со подолгото од времетраењата на S1 или S2. Со ова некогаш се зема далеку полошо времетраење, но никогаш не се зема пократко времетраење.

1.2.5 Рекурзии

Рекурзиите мора да се разгледуваат од случај до случај. Понекогаш тие имаат линеарно времетраење $O(n)$, како во следниот програмски сегмент:

```
factorial( n )
{
    if( n <= 1 )
        return 1;
    else
        return n * factorial(n-1);
}
```

но, понекогаш пресметувањето на нивното времетраење е покомплексно како во следниот случај (пресметка на фибоначи):

```

fib( n )
{
1   if( n <= 1 )
2       return 1;
else
3       return fib(n-1) + fib(n-2);
}

```

Нека (n) е времетраењето на извршување на функцијата $\text{fib}(n)$. Очигледно е дека времетраењето е 1 (за споредбата) собрано со делот после else-от кој пак има едноставна операција и два рекурзивни повика со должина (според предходната претпоставка) $T(n - 1)$ и $T(n - 2)$ соодветно. Така добиваме: $T(n) = T(n - 1) + T(n - 2) + 2$

Со едноставна анализа (со примена на истото размислување за првите два члена на сумата во следниот чекор добиваме по два нови и така натаму) се заклучува дека растот (кардиналноста) на оваа функција е експоненционален – $O(2^n)$. Ова претставува типичен пример за неефикасен алгоритам. Лесно се утврдува дека со првиот повик на линија 3 се пресметува вредноста $\text{fib}(n-2)$, која потоа повторно се пресметува при вториот повик на линија 3.

1.3 Принципи на споредба

Да го разгледаме следниот проблем: Да се најде најголемата позитивна подсума во низа од n цели броеви!

Пример: За влез: -2, 11, -4, 13, -5, -2, одговорот е 20 (вториот до четвртиот елемент).

Ќе бидат разгледани 4 различни по ефикасност (и разбираливост) коректни алгоритми. Првиот алгоритам е претставен со следниот програмски код:

```

1 max_subsequence_sum( a[], n )
2 {
3     max_sum = 0; best_i = best_j = -1;
4     for( i=0; i<n; i++ )
5         for( j=i; j<n; j++ )
6         {
7             this_sum = 0;
8             for( k = i; k<=j; k++ )
9                 this_sum += a[k];
10            if( this_sum > max_sum )
11                { //update max_sum, best_i, best_j

```

```

12         max_sum = this_sum;
13         best_i = i;
14         best_j = j;
15     }
16 }
17 return max_sum;
18 }
```

Времето на извршување е $O(n^3)$ и се должи на линијата 6, со големина $O(1)$ која е вметната во три вгнездени *for* циклуси. Првиот циклус (линија 2) има n итерации, вториот (линија 3) има $n - i + 1$ односно сигурно помалку од n итерации, додека третиот циклус (линија 5) има $j - i + 1$ односно повторно помалку од n итерации. Чекорите од 7 до 10 имаат кардиналност $O(n^2)$ бидејќи се наоѓаат во два вгнездени циклуса, па поради правилото на последователност не влијаат на вкупната кардиналност. За да се потврди точноста на пристапот ќе биде направена и подетална анализа која разгледува колку пати се извршува чекорот 6 односно пресметува колку пати се извршува: $\sum_{i=1}^n \sum_{j=i}^n \sum_{k=i}^j$.

$$\begin{aligned}
& \sum_{k=i}^j 1 = j - i + 1 \\
& \sum_{j=i}^n (j - i + 1) = \frac{(n - i + 1)(n - i + 2)}{2} \\
& \sum_{i=1}^n \frac{(n - i + 1)(n - i + 2)}{2} = \\
& \frac{1}{2} \sum_{i=1}^n i^2 - (n + \frac{3}{2}) \sum_{i=1}^n i + \frac{1}{2}(n^2 + 3n + 2) \sum_{i=1}^n 1 = \\
& \frac{1}{2} \frac{n(n+1)(2n+1)}{6} - (n + \frac{3}{2}) \frac{n(n+1)}{2} + \frac{n^2 + 3n + 2}{2} n = \frac{n^3 + 3n^2 + 2n}{6}
\end{aligned}$$

Оваа анализа ја потврдува кардиналноста од големина n^3 !

Ефикасноста на алгоритамот може да се подобри со исфрлање на еден од трите вгнездени циклуси како што е направено со следниот алгоритам каде е избегната непотребната редунданција на сумите (повторно пресметување) чекорите 5 и 6 во првиот алгоритам, или математички кажано: $\sum_{k=i}^j a_k = a_i + \sum_{k=i}^{j-1} a_k$.

Вториот алгоритам има кардиналност $O(n^2)$ што се утврдува со слична (но поедноставна) анализа).

```

1 max_subsequence_sum( a[], n )
2 {
```

```

3     max_sum = 0; best_i = best_j = -1;
4     for( i=0; i<n; i++ )
5     {
6         this_sum = 0;
7         for( j=i; j<n; j++ )
8         {
9             this_sum += a[k];
10            if( this_sum > max_sum )
11            {   //update max_sum, best_i, best_j
12                max_sum = this_sum;
13                best_i = i;
14                best_j = j;
15            }
16        }
17        return max_sum;
18    }
19 }
```

Трето решение на дадениот проблем е рекурзивно (и најкомплицирано). Пристапот што се користи се нарекува "раздели и владеј". Основната идеја е во делењето на низата во две поднизи (лева и десна) кои имаат приближно ист број на елементи. Најголемата можна подсума ќе биде или во левата подниза, или во десната подниза, или ќе биде сума на последователни елементи што ќе почнат во левата подниза, а ќе завршат во десната подниза. Доколку ги најдеме најголемите суми на последователни елементи во левата и десната подниза, дополнително треба да ја најдеме најголемата сума на елементи на крајот на левата подниза собрана со најголемата сума на елементи на почетокот на десната подниза. Бараното решение е најголемата од трите пресметани суми.

На пример во низата

Лева подниза Десна подниза

4 -3 5 -2 -1 2 6 -2

Максималниот збир во левата подниза е 6 (од a_1 до a_3), а на десната е 8 (од a_6 до a_7). Најголемата сума на крајот од левата подниза е 4 (од a_1 до a_4), а на почетокот на десната подниза е 7 (од a_5 до a_7). Следствено бараното решение е $4 + 7 = 11$ (од a_1 до a_7).

```

1 max_sub_sequence_sum( a[],n )
2 {   return max_sub_sum( a, 0, n-1 ); }
3 int max_sub_sum( a[], left, right )
4 {
```

```

5   if ( left == right ) /* Base Case */
6     if( a[left] > 0 )
7       return a[left];
8     else
9       return 0;
10    center = (left + right )/2;
11    max_left_sum = max_sub_sum( a, left, center );
12    max_right_sum = max_sub_sum( a, center+1, right );
13    max_left_border_sum = 0; left_border_sum = 0;
14    for( i=center; i>=left; i-- )
15    {
16      left_border_sum += a[i];
17      if( left_border_sum > max_left_border_sum )
18        max_left_border_sum = left_border_sum;
19    }
20    max_right_border_sum = 0; right_border_sum = 0;
21    for( i=center+1; i<=right; i++ )
22    {
23      right_border_sum += a[i];
24      if( right_border_sum > max_right_border_sum )
25        max_right_border_sum = right_border_sum;
26    }
27    return max3( max_left_sum, max_right_sum, max_left_border_sum +
28      max_right_border_sum );
}

```

Нека $T(n)$ биде бараната кардиналност за n влезни податоци. За $n = 1$, се извршуваат чекорите од 1 до 4 од што следи дека $T(1) = 1$. Во сите други случаи програмот извршува два рекурзивни повици, два циклуса со кардиналност $O(n)$ - (чекори 9 до 17) и неколку едноставни операции (чекори 5 и 18) кои поради предходната кардиналност на циклусите може да се игнорираат. Остатокот од програмата се извршува во чекорите 6 и 7. Во тие чекори бројот на влезните податоци се преполовува, па според тоа тие се со кардиналност $T(n/2)$ или вкупно $2T(n/2)$. Тогаш вкупното време на извршување на алгоритамот може да се претстави со следната рекурентна равенка:

$$T(1) = 1 \quad T(n) = 2T(n/2) + O(n) = 2T(n/2) + n$$

тогаш

$$T(2) = 4 = 2 * 2,$$

$$T(4) = 12 = 4 * 3,$$

$$T(8) = 32 = 8 * 4,$$

$$T(16) = 80 = 16 * 5.$$

или за $n = 2$, $T(n) = n * (k + 1) = n \log_2 n + n = O(n \log_2 n)$.

Четвртиот алгоритам е поедноставен за имплементација, но уште поважно, тој е и поефикасен!

```

1 max_subsequence_sum( ia[], n )
2 {
3     i = this_sum = max_sum = 0; best_i = best_j = -1;
4     for( j=0; j<n; j++ )
5     {
6         this_sum += a[j];
7         if( this_sum > max_sum )
8             { /* update max_sum, best_i, best_j */
9                 max_sum = this_sum;
10                best_i = i;
11                best_j = j;
12            }
13        else
14            if( this_sum < 0 )
15            {
16                i = j + 1;
17                this_sum = 0;
18            }
19        }
20    return max_sum;
21 }
```

Ефикасноста на овој алгоритам се утврдува едноставно и таа е еднаква на (n) . Утврдување на коректноста на алгоритамот е задача за читателите на текстот.

Во Табела 1.1 се дадени времињата за извршување на предходните четири алгоритми за различен број на влезни податоци:

Алгоритам	1	2	3	4
кардиналност	$O(n^3)$	$O(n^2)$	$O(n \log_2 n)$	$O(n)$
$n = 10$	0.00103	0.00045	0.00066	0.00034
$n = 100$	0.47015	0.01112	0.00486	0.00063
$n = 1,000$	448.77	1.1233	0.05843	0.00333
$n = 10,000$	NA	111.13	0.68631	0.03042
$n = 100,000$	NA	NA	8.0113	0.29832

Табела 1.1: Споредба на време на извршување за различни кардиналности

Најчестите кардиналности, или "класи" функции со кои опишуваме било кој алгоритам се дадени во Табела 1.2:

Функција	Име
C	Константа
$\log_2 n$	Логаритамска
$\log^2 n$	Квадратно-логаритамска
n	Линеарна
n^2	Квадратна
n^3	Кубна
2^n	Експоненцијална

Табела 1.2: Најчести кардиналности

1.4 Нотации

Покрај O -нотацијата, постојат и други нотации кои помагаат уште поточно да се опишат алгоритмите.

Ω -нотација

Ω -(големо омега од ен) опишува асимптотска долна граница (најдобро извршување) на некоја функција.

За дадено $g(n)$, $\Omega(g(n))$ се дефинира со:

$\Omega(g(n)) = f(n)$: постојат позитивни константи c и n_0 такви што $0 \leq cg(n) \leq f(n)$ за секое $n \geq n_0$.

θ -нотација

θ -(големо тета од ен) опишува асимптотска средна граница (средно време на извршување) на некоја функција.

За дадено $g(n)$, $\theta(g(n))$ се дефинира со:

$\theta(g(n)) = f(n)$: постојат позитивни константи c_1 , c_2 и n_0 такви што $c_1g(n) \leq f(n) \leq c_2g(n)$ за секое $n \geq n_0$.

Глава 2

Низи

Прва податочна структура што ќе биде изучувана е структурата наречена низа. Низата може да се дефинира како множество на подредени парови (индекс, вредност), при што за секое појавување на индекс, постои соодветна вредност асоцирана за тој индекс. Индексите (особено во реализациите подржани од програмските јазици) вообичаено се последователни цели броеви.

Во реализацијата на низите на било кој конкретен компјутер, мора да се дефинира максимален број на елементи на низата, а физички во меморијата низата зафаќа последователни мемориски локации со должина соодветна на големината на низата. Секоја вредност од низата може директно да се пристапи со користење на нејзиниот индекс во $O(1)$ време.

Пример: Да ја дефинираме променливата `vozrast` во која ќе се чуваат вредностите на возрастите од 10 студенти.

1 `int vozrast[10];`

При самата дефиниција на променливата низа `vozrast`, мора да се спцифицира максималниот број на елементи, а во нашиот случај, тоа е 10.

Пристапот до елементите е директен, т.е. директно може да се внесе и/или промени вредноста за даден индекс во низата.

1 `vozrast[3] = 19;`
2 `vozrast[5] = 18;`
3 `vosrast[7] = 22;`
4 `vozrast[3] = 20;`
5 `vozrast[15] = 21; //грешка`

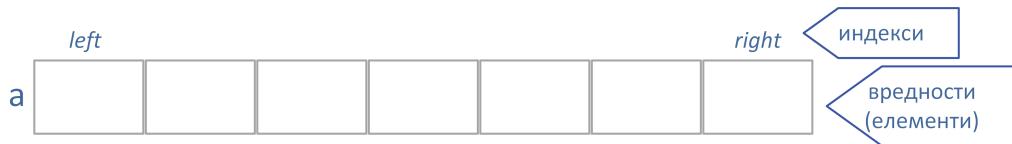
Во линијата 1, најпрвин за четвртиот елемент во низата `vozrast` се доделува вредност 19, а потоа во линијата 4 истата вредност се променува со вредност 20. Во линијата 5 има грешка, бидејќи пробуваме да внесеме вредност за елемент од низата што излегува надвор од границите на низата. Затоа мора да се внимава

кога се користат елементите од низата, секогаш да се запазуваат границите на низата.

Низите ги имаат следните својства:

- Низите чуваат повеќе елементи од ист тип под заедничко име
- Елементите во низата може да се пристапуваат по случаен редослед, користејќи го индексот
- Меморијата за низата е предефинирана, нема потреба од дополнителен мемориски простор
- Низите се статички структури, нивната големина е фиксна и не може да се промени после декларација

Во продолжение ќе разгледаме општ тип на низи во кои индексите имаат опсег помеѓу две гранични вредности [left ; right] како што е покажано на сликата 2-1:



Слика 2-1: Визуелен приказ на низа

Дефиниција на низа ќе се изврши преку класа Array во Јава и притоа оваа класа ќе содржи низа од генерички тип, како и големина на листата (моментален број на елементи сместени во низата и притоа максимален број на елементи е капацитетот на низата). Креирањето на објект низа (конструктор за тип Array) е со влезен параметар - капацитет на низата и иницијални низата нема елементи (size=0).

```

1 public class Array<E> {
2     private E data[];
3     private int size;
4
5     public Array(int capacity) {
6         this.data = (E[]) new Object[capacity];
7         this.size = 0;
8     }
9
10    //методи на низа
11 }
```

2.1 Операции со низи

Функциите (операциите) кои може да се применат кога се работи со низа се:

- изминување и печатење на низа,
- пребарување вредност според индекс,
- пребарување индекс според вредност,
- додавање нов пар (индекс, вредност) и
- бришење пар (индекс, вредност).

Иако овие операции делуваат тривијално, може да бидат комплексни, особено ако сакаме да ја задржиме последователноста на индексите и да ги запазиме граничните случаи (недозволување на промена или бришење на пар од празна низа и додавање на нов пар во низа со пополнет максимален број на парови).

Операцијата **изминување и печатење** на низа се имплементира со прекривање на методата `toString()` во Java која враќа `String`. Бидејќи методата линерно ги изминува сите елементи во низата, нејзината сложеност е $O(n)$.

```

1   @Override
2   public String toString() {
3       String ret = new String();
4       if(size>0) {
5           ret = "[";
6           ret += data[0];
7           for(int i=1;i<size;i++) {
8               ret += "," + data[i];
9           }
10          ret+="]";
11          return ret;
12      }
13      else {
14          ret = "Prazna niza!";
15      }
16      return ret;
17  }

```

Вметнувањето вредности е една од основните операции кога се работи со низи. Потребно е да се направи место за новиот елемент, на било која позиција на низата. Вметнување елемент во низа може да се изведе на крај на низата или пак на било кој индекс во низата.

Во генерален случај за дадена низа $a[left \dots right]$, да се вметне вредност val во $a[ins]$ (каде $left \leq ins \leq right$), потребно е да се поместат вредностите надесно, за

да се направи место за новиот елемент. Методата `insertLast(E o)` вметнува нов елемент `E` после последниот елемент на низата. Методата `insert(int position, E o)` вметнува нов елемент `E` на дадена позиција во низата, и притоа сите елементи од дадената позиција натаму ги поместува за едно место во десно. И во двете методи се проверува капацитетот на низата, и доколку не е доволен, се удвојува капацитетот со аликација на нов, дупло поголем, мемориски простор. Сложеноста на методата за вметнување на елемент е $O(n)$ бидејќи во најлош случај се поместуваат сите елементи на низата во десно, односно се вршат N копирања.

```

1  public void insertLast(E o) {
2      //проверка дали има доволен капацитет низата, ако нема - resize
3      if(size + 1 > data.length)
4          this.resize();
5      data[size++] = o;
6  }
7
8  public void insert(int position, E o) {
9      //проверка дали позицијата е во даден ранг
10     if (position >= 0 && position <= size) {
11         //проверка дали има доволен капацитет низата, ако нема - resize
12         if(size + 1 > data.length)
13             this.resize();
14         //копирање на податоците за една позиција во десно
15         for(int i=size;i>position;i--) {
16             data[i] = data[i-1];
17         }
18         data[position] = o;
19         size++;
20     } else {
21         System.out.println("Ne mozhe da se vmetne element na taa
22             pozicija");
23     }
24
25     //методата го удвојува капацитетот на низата
26     public void resize() {
27         //се алокира нов дупло поголем мемориски простор
28         E[] newData = (E[]) new Object[size*2];
29         //се копираат податоците во новиот простор
30         for (int i = 0; i < size; i++)
31             newData[i] = data[i];
32         //се заменува стариот мемориски простор со новиот

```

```

33     this.data = newData;
34 }
```

Бришење елемент во низа е процес на отстранување на конкретен елемент и реорганизација на низата. Потребно е да се изврши движење на елементите во низата налево (освен кога се брише последната вредност) за да се прегрупира и со тоа да ја намали својата големина.

Во генерален случај за дадена низа $a[left\dots right]$, да се избрише вредност val во $a[del]$, каде (каде $left \leq del \leq right$), потребно е да се поместат вредностите налево, за да се пополни празнината. Методата `delete(int position)` брише елемент од дадена позиција во низата, и притоа сите елементи после елементот кој се брише се поместуваат за една позиција во лево (да се пополни празнината на избришаниот елемент).

```

1  public void delete(int position) {
2      //проверка дали позицијата е во даден ранг
3      if (position >= 0 && position < size) {
4          //се копираат податоците после дадената позиција за едно место во
5          //лево
6          for (int i = position + 1; i < size; i++)
7              data[i - 1] = data[i];
8          size--;
9      }
}
```

Елементот кој се наоѓа на дадена позиција може да се добие со `get(int position)` методата, а замена/поставување на елемент на дадена позиција се врши со методата `set(int position, E o)`, која го поставува (заменува постоечкиот елемент со) E на дадената позиција $position$. Методата `getSize()` го враќа моменталниот број на елементи сместени во низата. Методата `find(E o)` ја враќа првата позиција каде се наоѓа вредноста E во низата.

```

1  public E get(int position) {
2      if (position >= 0 && position < size)
3          return data[position];
4      else
5          System.out.println("Ne e validna dadenata pozicija");
6      return null;
7  }
8
9  public void set(int position, E o) {
10     if (position >= 0 && position < size)
11         data[position] = o;
```

```

12     else
13         System.out.println("Ne moze da se vmetne element na dadenata
14             pozicija");
15
16     public int getSize() {
17         return size;
18     }
19
20     public int find(E o) {
21         for (int i = 0; i < size; i++) {
22             if(o.equals(data[i]))
23                 return i;
24         }
25         return -1;
26     }

```

Користење на дефинираните методи

Следниот код дефинира класа ArrayTester во која во main методата се дефинира низа која содржи цели броеви (integers) и ја прикажува употребата на дефинираните методи.

```

1 public class ArrayTester {
2
3     public static void main(String[] args) {
4         Array<Integer> myArray = new Array<Integer>(4);
5
6         myArray.insertLast(4);
7         System.out.print("Nizata по вметнуванje на 4 како последен елемент:
8             ");
9         System.out.println(myArray.toString());
10
11         myArray.insertLast(7);
12         myArray.insertLast(13);
13         System.out.print("Nizata по додаване на 7 и 13 како елементи: ");
14         System.out.println(myArray.toString());
15
16         myArray.insert(1, 3);
17         System.out.print("Nizata по додаване на 3 како елемент на pozicija
18             1: ");
19         System.out.println(myArray.toString());

```

```

18
19     myArray.set(2, 6);
20     System.out.print("Nizata po menuvanje na vrednosta na elementot na
21         pozicija 2 vo 6: ");
22     System.out.println(myArray.toString());
23
24     myArray.delete(0);
25     System.out.print("Nizata po brishenje na elementot na pozicija 0
26         (prviot element): ");
27     System.out.println(myArray.toString());
28
29     System.out.print("Na pozicija 2 vo nizata sega se naogja: ");
30     System.out.println(myArray.get(2));
31
32     System.out.print("Brojot 3 sega se naogja vo nizata na pozicija: ");
33     System.out.println(myArray.find(3));
34
35 }
36 }

```

Излезот по извршувањето на овој код е:

```

Nizata po vmetnuvanje na 4 kako posleden element: {4}
Nizata po dodavanje на 7 и 13 како elementi: {4,7,13}
Nizata по dodavanje на 3 како element на pozicija 1: {4,3,7,13}
Nizata по menuvanje на vrednosta na elementot na pozicija 2 vo 6: {4,3,6,13}
Nizata по brishenje na elementot na pozicija 0 (prviot element): {3,6,13}
Na pozicija 2 vo nizata sega se naogja: 13
Brojot 3 sega se naogja vo nizata na pozicija: 0
Sega na krajot goleminata na nizata e: 3

```

2.2 Проблеми со низи

Следат неколку примери за употреба од овој податочен тип.

Задача 1. Просек на низа

За дадена низа од N ($1 \leq N \leq 50$) природни броеви, да се најде бројот кој е најблиску до нивниот просек. Ако постојат два броја со исто растојание до просекот, да се врати помалиот од нив. Во низата може да има дупликати.

На пример за низата 1, 2, 3, 4, 5 просекот е $(1 + 2 + 3 + 4 + 5) / 5 = 15 / 5 = 3$, што значи дека бројот кој треба да се врати и е најблиску до просекот е 3.

За низата 1, 2, 3, 4, 5, 6 просекот е 3.5 и двата броја 3 и 4 се на исто растојание од просекот. Точната вредност која треба да се врати е помалиот од нив, а тоа е 3.

Првиот број од влезот е бројот на елементи во низата N, а потоа во секој ред се дадени броевите.

Решение

Решението е дадено во brojDoProsek методата во ArrayMeanValue класата и примената на методата е прикажана во main методата.

```

1 import java.util.Scanner;
2 import java.lang.Math;
3 public class ArrayMeanValue {
4
5     public static int elementClosestToMean(Array<Integer> arr) {
6         int sum = 0, index = 0;
7         for(int i = 0; i < arr.getSize(); i++){
8             sum += arr.get(i);
9         }
10        int avg = sum / arr.getSize();
11        int minDifference = Math.abs((arr.get(0) - avg));
12        for(int i = 1; i < arr.getSize(); i++){
13            if(Math.abs(arr.get(i) - avg) < minDifference){
14                minDifference = Math.abs(arr.get(i) - avg);
15                index = i;
16            }
17            if(Math.abs((arr.get(i) - avg)) == minDifference){
18                if(arr.get(i) < arr.get(index)){
19                    index = i;
20                }
21            }
22        }
23        return arr.get(index);
24    }
25
26    public static void main(String[] args) {
27        Scanner input = new Scanner(System.in);
28        int N = input.nextInt();
29        Array<Integer> arr = new Array<>(N);

```

```

30         for(int i=0;i<N;i++) {
31             arr.insertLast(input.nextInt());
32         }
33         System.out.println(elementClosestToMean(arr));
34     }
35 }
```

Задача 2. Промени низи

Нека се дадени две низи, кои треба да бидат со иста големина. Да се напише функција која ќе прави промени во двете низи така што доколку на дадена позиција тие имаат еднакви елементи, истите треба да се избришат и во двете низи.

Решение

Решението е дадено во compareAndChangeArrays методата во ChangeArrays класата и примената на методата е прикажана во main методата.

```

1  public class ChangeArrays<E> {
2      public void compareAndChangeArrays(Array<E> arr1, Array<E> arr2) {
3          if(arr1.getSize() != arr2.getSize()) {
4              System.out.println("Nizite ne se so ista golemina!");
5              return;
6          }
7          int size = arr1.getSize();
8          int i = 0;
9          while(i < size) {
10              if(arr1.get(i).equals(arr2.get(i))) {
11                  arr1.delete(i);
12                  arr2.delete(i);
13                  size--;
14              } else {
15                  i++;
16              }
17          }
18      }
19
20      public static void main(String[] args) {
21          Array<String> arr1 = new Array<String>(4);
22          arr1.insertLast("nb11");
23          arr1.insertLast("b1");
```

```
24     arr1.insertLast("b2");
25     arr1.insertLast("nb12");
26
27     Array<String> arr2 = new Array<String>(4);
28     arr2.insertLast("nb21");
29     arr2.insertLast("b1");
30     arr2.insertLast("b2");
31     arr2.insertLast("nb22");
32
33     System.out.println("Nizite pred primenuvanjeto na funkcijata: ");
34     System.out.println(arr1.toString());
35     System.out.println(arr2.toString());
36
37     ChangeArrays<String> pom = new ChangeArrays<String>();
38     pom.compareAndChangeArrays(arr1, arr2);
39
40     System.out.println("Nizite po primenuvanjeto na funkcijata: ");
41     System.out.println(arr1.toString());
42     System.out.println(arr2.toString());
43
44     ArrayList<Integer> arr3 = new ArrayList<Integer>(3);
45     arr3.add(10);
46     arr3.add(13);
47     arr3.add(7);
48
49     ArrayList<Integer> arr4 = new ArrayList<Integer>(3);
50     arr4.add(5);
51     arr4.add(13);
52     arr4.add(3);
53
54     System.out.println("Nizite pred primenuvanjeto na funkcijata: ");
55     System.out.println(arr3.toString());
56     System.out.println(arr4.toString());
57
58     ChangeArrays<Integer> pom2 = new ChangeArrays<Integer>();
59     pom2.compareAndChangeArrays(arr3, arr4);
60
61     System.out.println("Nizite po primenuvanjeto na funkcijata: ");
62     System.out.println(arr3.toString());
63     System.out.println(arr4.toString());
64
```

```
65     }
66 }
```

2.3 Задачи за вежбање

Задача 1. Бриши дупли

За дадена низа од N ($1 \leq N \leq 50$) природни броеви да се избришат дупликат вредностите кои се јавуваат на соседни позии. Односно доколку две и повеќе вредности се една до друга во низата, да се остави само едно појавување.

На пример низата 1, 2, 2, 2, 3, 2, 4, 4, 1 по обработката ќе биде 1, 2, 3, 2, 4, 1.

Првиот број од влезот е бројот на елементи во низата N , а потоа во секој ред се дадени броевите.

Задача 2. Додади средни вредности

За дадена низа од N ($1 \leq N \leq 50$) природни броеви меѓу секои два соседи да се внесе нов елемент кој е (целоброен, заокружен) просек од двата соседи.

На пример низата 1, 3, 5, 6 по обработката ќе биде 1, 2, 3, 4, 5, 6, 6.

Првиот број од влезот е бројот на елементи во низата N , а потоа во секој ред се дадени броевите.

2.4 ArrayList во Java

Класата **ArrayList** во Java моделира низа на која може да и се промени големината (со алокација на нов простор и копирање на елементите во тој простор). Класата употребува генерици за дефинирање на тип на елементи кои ќе ги содржи. Имплементацијата може да биде побавна од стандардните низи, но може да биде корисна во програми каде што се потребни многу манипулации во низата.

Постојат неколку конструктора за **ArrayList**:

- **ArrayList()** - Креира празна низа со капацитет десет
- **ArrayList(int capacity)** - Креира празна низа со иницијален капацитет *capacity*

Постојат голем број на методои кои ги нуди класат, дел од нив се:

- **boolean add(E e)** - Се користи за додавање на наведениот елемент *E* на крајот од низата

- **void add(int index, E element)** - Се користи за вметнување на наведениот елемент E на дадената позиција index во низата
- **void clear()** - Се користи за да се отстрanат сите елементи од низата
- **void ensureCapacity(int requiredCapacity)** - Го зголемува капацитетот на овој инстанцата на ArrayList, доколку е потребно, за да се осигура дека може да го задржи барем бројот на елементи наведени со аргументот requiredCapacity за минимален капацитет.
- **E get(int index)** - Го враќа елементот на позиција index
- **boolean isEmpty()** - Враќа true ако низата е празна, инаку false
- **int lastIndexOf(Object o)** - Се користи за враќање на индексот на последното појавување на наведениот елемент во оваа низа, или -1 ако низата не го содржи овој елемент
- **int indexOf(Object o)** - Се користи за враќање на индексот на првото појавување на наведениот елемент во оваа низа, или -1 ако низата не го содржи овој елемент
- **E remove(int index)** - Го отстранува елементот кој се наоѓа на позиција index во низата
- **boolean remove(Object o)** - Го отстранува првото појавување на дадениот објект од низата
- **int size()** - Го враќа бројот на елементи присутни во низата
- **void trimToSize()** - Се користи за да се намали капацитетот на низата за да биде моменталната големина (бројот на елементи)

Користење на дефинираните методи

Следниот код дефинира класа ArrayTesterArrayList во која во main методата се дефинира низа која содржи цели броеви (integers) и ја прикажува употребата на дефинираните методи (односно истото тестирање кое се употребуваше за претходно дефинирана класа Array).

```

1 public class ArrayTesterArrayList {
2
3     public static void main(String[] args) {
4         Array<Integer> myArray = new Array<Integer>(4);
5
6         myArray.insertLast(4);
7         System.out.print("Nizata po vmetnuvanje na 4 kako posleden element:
8             ");
9         System.out.println(myArray.toString());
10        myArray.insertLast(7);

```

```
11     myArray.insertLast(13);
12     System.out.print("Nizata po dodavanje na 7 i 13 kako elementi: ");
13     System.out.println(myArray.toString());
14
15     myArray.insert(1, 3);
16     System.out.print("Nizata po dodavanje na 3 kako element na pozicija
17         1: ");
18     System.out.println(myArray.toString());
19
20     myArray.set(2, 6);
21     System.out.print("Nizata po menuvanje na vrednosta na elementot na
22         pozicija 2 vo 6: ");
23     System.out.println(myArray.toString());
24
25     myArray.delete(0);
26     System.out.print("Nizata po brishenje na elementot na pozicija 0
27         (prviot element): ");
28     System.out.println(myArray.toString());
29
30     System.out.print("Na pozicija 2 vo nizata sega se naogja: ");
31     System.out.println(myArray.get(2));
32
33     System.out.print("Brojot 3 sega se naogja vo nizata na pozicija: ");
34     System.out.println(myArray.find(3));
35 }
36 }
```

Излезот по извршувањето на овој код е:

```
Nizata po vmetnuvanje na 4 kako posleden element: [4]
Nizata po dodavanje na 7 i 13 kako elementi: [4, 7, 13]
Nizata po dodavanje na 3 kako element na pozicija 1: [4, 3, 7, 13]
Nizata po menuvanje na vrednosta na elementot na pozicija 2 vo 6: [4, 3, 6, 13]
Nizata po brishenje na elementot na pozicija 0 (prviot element): [3, 6, 13]
Na pozicija 2 vo nizata sega se naogja: 13
Brojot 3 sega se naogja vo nizata na pozicija: 0
Sega na krajot goleminata na nizata e: 3
```

Глава 3

Листи

Листите или поврзаните листи се податочни структури кои ги чуваат елементите линеарно. За разлика од низите, обработени во претходната глава, кај поврзаните листи елементи не мора да се чуваат во последователни мемориски локации. Секој елемент во листата се нарекува јазел и содржи податоци. Подредувањето на елементите се изведува така што секој елемент има и врска/показувач (кон мемориската локација) на следниот елемент во листата. За разлика од низите, каде што големината на низата е ограничена, листите немаат ограничување на големина.

Листите имаат неколку предности како податочни структури:

- вметнувањето и бришењето на елементи е поедноставно за разлика од низите, бидејќи треба само да се креира/избрише еден јазел и да се променат покажувачите соодветно
- големината на листите е лесно променлива, бидејќи нов јазел може да се смести на било кое слободно место во меморијата, па така претставуваат динамички податочни структури
- имаат ефикасна употреба на меморијата за разлика од низите

Воедно листите имаат и неколку недостатоци:

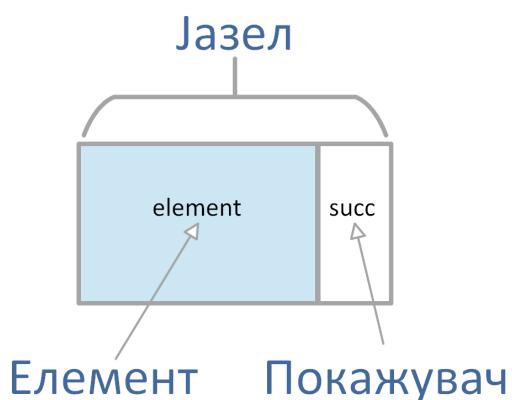
- не можеме директно да пристапиме до било кој елемент на листата, туку мораме да ја изминеме за да стигнеме до соодветен јазел
- се употребува дополнителна меморија за чување на покажувачите

Во зависност од тоа какви покажувачи содржи секој јазел, генерално има два типа на листи:

- еднострano поврзани листи
- двострано поврзани листи

3.1 Еднострano поврзани листи

Јазлите на еднострano поврзаните листи содржат информации и покажувач кон наредниот елемент односно јазел во листата. На слика 3-1 е даден визуелен приказ на јазел. Елементот ја содржи информацијата која ја доделуваме, додека покажувачот ја дава мемориската локација на следниот јазел во листата. Java класата која моделира еден ваков јазел дадена е во кодот SLLNode. Елементот на листата (element во кодот) е од генерички тип кој според потребата се наведува при декларација на листата.



Слика 3-1: Визуелен приказ на јазел на еднострano поврзана листа

```

1 public class SLLNode<E> {
2     protected E element; //елементот на јазелот
3     protected SLLNode<E> succ; //показувач кон следниот јазел
4
5     public SLLNode(E elem, SLLNode<E> succ) {
6         this.element = elem;
7         this.succ = succ;
8     }
9 }
```

При дефиниција на еднострano поврзана листа потребно ни е да го знаеме каде се наоѓа првиот јазел на листата. Потоа тргнувајќи од тој јазел може да ги измнинеме сите јазли. Затоа соодветно при дефиниција на податочната структура чуваме само покажувач кон првиот јазел. Ова е прикажано во кодот SLL кој ја моделира структурата и има само еден јазел first од тип SLLNode.

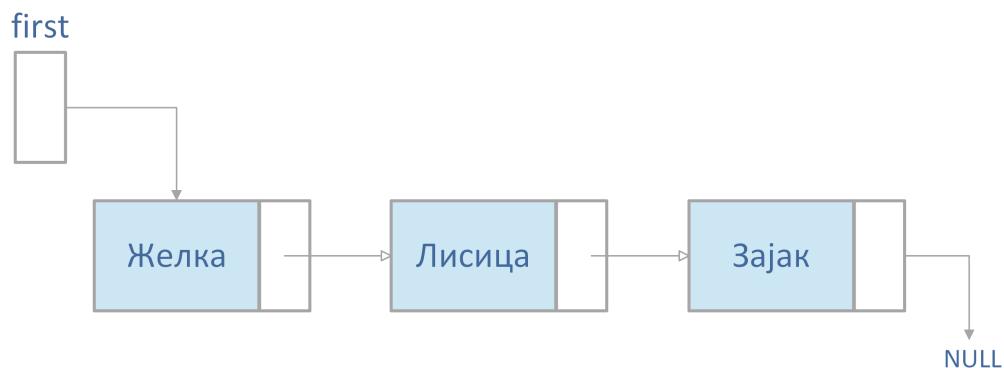
```

1 public class SLL<E> {
2     private SLLNode<E> first; //првиот јазел во листата
```

```

3
4     public SLL() {
5         //креирање на празна листа
6         this.first = null;
7     }
8
9     //методи на SLL
10    ...
11 }
```

Секој јазел во листата има следбеник (освен последниот јазел), и секој јазел во листата има претходник (освен првиот јазел). Должината на листата е бројот на јазлите кои ги содржи. Празната листа не содржи ниту еден јазел. Слика 3-2 нуди визуелен приказ на пример на едностррано поврзана листа која содржи имиња на животни. Листата е со должина 3 бидејќи има 3 јазли. Прв јазел во листата, односно покажувачот на првиот јазел на листата води до јазелот кој како информација содржи Желка. Преку тој јазел и покажувачот за следбеник стигаме со вториот јазел во листата кој содржи информација Лисица. Преку покажувалот за следбеник следно стигаме до јазел кој содржи информација Зајак. Бидејќи овој јазел е последен во листата и нема следбеник, покажувачот на следниот јазел е NULL. Кога би ја декларирале оваа листа во код ви ставиле тип `SLL<String>` бидејќи содржината на секој јазел во овој случај е стринг. Предноста на употребата на генерици ни овозможува при декларација на листата да употребиме било кој податочен тип (вклучувајки и сопствено дефинирани типови/класи).



Слика 3-2: Пример на едностррано поврзана листа

3.1.1 Операции со еднострano поврзани листи

Базични операции со еднострano поврзани листи вклучуваат:

- изминување и печатење на листа
- вметнување на елемент
- бришење на елемент
- наоѓање на елемент
- бришење на листа

Операцијата **изминување и печатење** на листа се имплементира со препокривање на методата `toString()` во Java која враќа `String`. Се започнува со пристап до првиот јазел во листата преку параметарот `first` (односно се дефинира привремен јазел `tmp` кој на почетокот покажува на првиот елемент во листата, ред 5 од кодот). Потоа се додека има следен јазел (ред 7 од кодот) се зема информацијата на моменталниот јазел (т.е. `tmp` јазелот), се додава во резултантниот `String` и се преминува на следниот јазел (ред 8 од кодот). Бидејќи методата линерно ги изминува сите јазли во листата, нејзината сложеност е $O(n)$.

```

1  @Override
2  public String toString() {
3      String ret = new String();
4      if (first != null) {
5          SLLNode<E> tmp = first;
6          ret += tmp.element;
7          while (tmp.succ != null) {
8              tmp = tmp.succ;
9              ret += "->" + tmp.element;
10         }
11     } else
12         ret = "Prazna lista!!!";
13     return ret;
14 }
```

Операцијата за **вметнување** на нов јазел во листата вклучува 4 различни случаи:

- вметнување на елемент на почеток во листата
- вметнување на елемент на крај во непразна листа
- вметнување на елемент по даден јазол
- вметнување на елемент пред даден јазол

Имплементацијата на **вметнување на елемент на почеток во листата** е дадена во методот `insertFirst(E o)`. Имплементацијата на методата започнува со

креирање на нов јазел ins кој ја содржи информацијата за новиот елемент (ред 2). Потоа за следбеник на новиот јазел се доделува првиот јазел во листата (ред 3). Финално новиот јазел се поставува како прв јазел во листата.

```

1  public void insertFirst(E o) {
2      SLLNode<E> ins = new SLLNode<E>(o, null);
3      ins.succ = first;
4      first = ins;
5  }

```

На слика 3-3 е прикажано чекор по чекор извршување на методата `insertFirst("Волк")` на листата од примерот на слика 3-2.

Имплементацијата на **вметнување на елемент по даден јазол** е дадена во методот `insertAfter(E o, SLLNode<E> node)`, чии влезни параметри се новиот елемент o кој сакаме да го вметнеме после јазелот node. Имплементацијата започнува со проверка дали јазелот node е null, и доколку условот е исполнет се креира нов јазел ins чија вредност е дадената влезна вредност E o, а следбеник е следбеникот на јазелот node. Како следбеник на јазелот node се поставува новиот јазел ins. Сликата 3-4 го прикажува извршувањето на методата `insertAfter("Волк",node)` на листата од примерот на слика 3-2, каде јазелот node е јазелот кој ја содржи вредноста Лисица.

```

1  public void insertAfter(E o, SLLNode<E> node) {
2      if (node != null) {
3          SLLNode<E> ins = new SLLNode<E>(o, node.succ);
4          node.succ = ins;
5      } else {
6          System.out.println("Dadenot jazol e null");
7      }
8  }

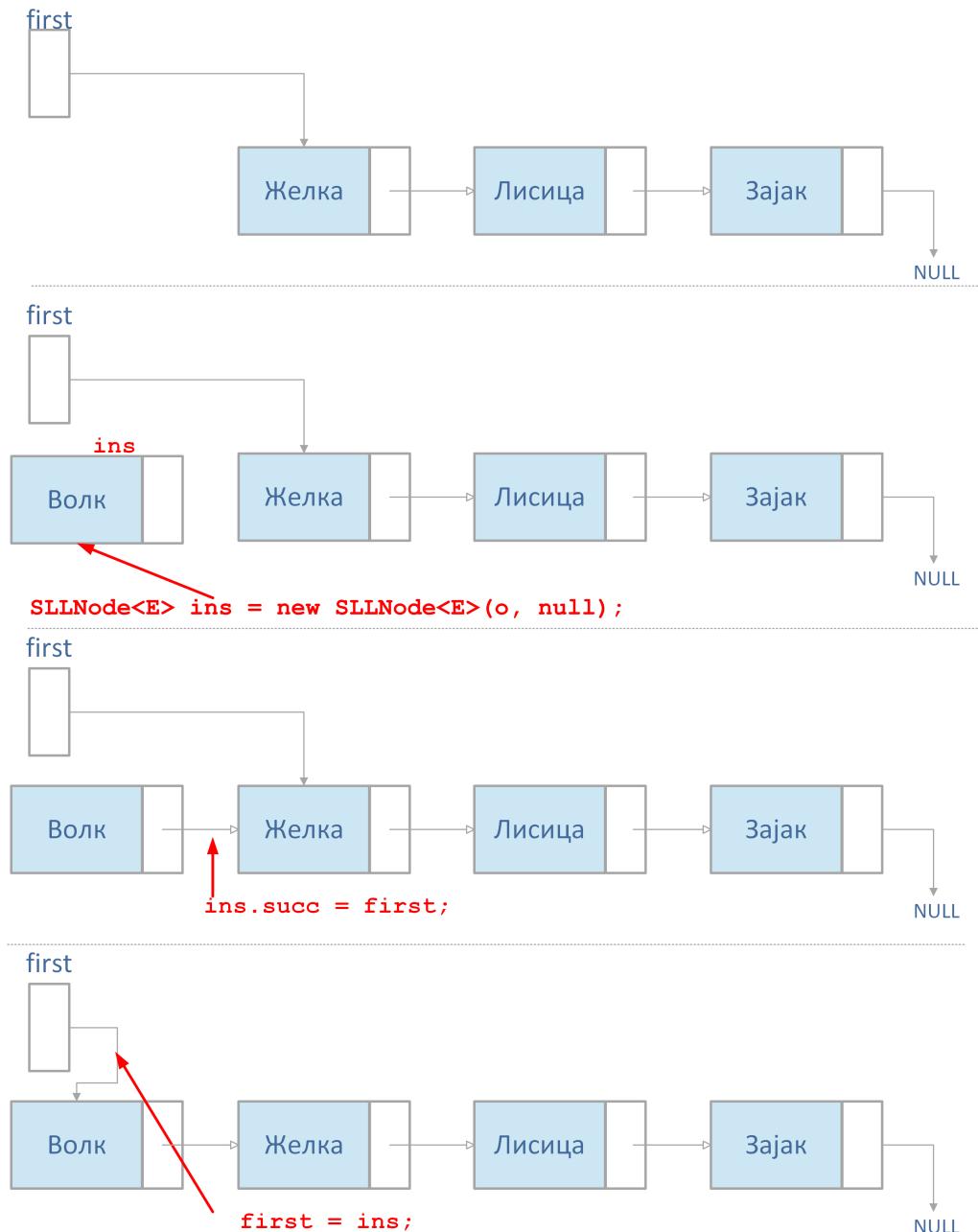
```

Имплементациите на **вметнување на елемент на крај во листа** како и **вметнување на елемент пред даден јазол** се дадени во методите `insertLast(E o)` и `insertBefore(E o, SLLNode<E> node)` соодветно.

```

1  public void insertLast(E o) {
2      if (first != null) {
3          SLLNode<E> tmp = first;
4          while (tmp.succ != null)
5              tmp = tmp.succ;
6          tmp.succ = new SLLNode<E>(o, null);
7      } else {
8          insertFirst(o);

```

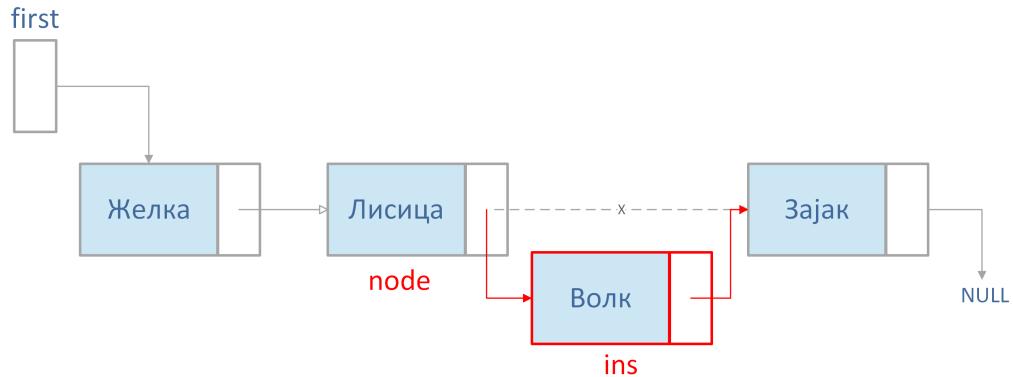


Слика 3-3: Пример на вметнување на јазел на почеток на листа

```

9
10 }
11
12 public void insertBefore(E o, SLLNode<E> before) {
13     if (first != null) {
14         SLLNode<E> tmp = first;
15         if(first==before){

```



Слика 3-4: Пример на вметнување на јазел после даден јазел

```

16         this.insertFirst(o);
17         return;
18     }
19     //ако first!=before
20     while (tmp.succ != before && tmp.succ!=null)
21         tmp = tmp.succ;
22     if (tmp.succ == before) {
23         tmp.succ = new SLLNode<E>(o, before);
24     } else {
25         System.out.println("Elementot ne postoi vo listata");
26     }
27 } else {
28     System.out.println("Listata e prazna");
29 }
30 }
```

Операцијата за **бришење** на јазел во листата вклучува два различни случаи:

- бришење на првиот јазел во листата
- бришење на даден јазол во листата

Имплементацијата на **бришење на првиот јазел во листата** е дадена во методот `deleteFirst()`. Имплементацијата на методата започнува проверка дали постои прв јазел и доколку постои се става неговиот следбеник како прв јазел. Притоа се враќа како повратна вредност вредноста на избришаниот јазел.

```

1  public E deleteFirst() {
2      if (first != null) {
3          SLLNode<E> tmp = first;
4          first = first.succ;
```

```

5         return tmp.element;
6     } else {
7         System.out.println("Listata e prazna");
8         return null;
9     }
10    }

```

Имплементацијата на **бришење на даден јазел во листата** е дадена во методот delete(SLLNode<E> node). Имплементацијата во tmp го бара јазелот што се наоѓа пред јазелот кој се брише, и потоа го премостува јазелот кој треба да биде избришан со тоа што на јазелот tmp како следбеник го става следбеникот на јазелот кој се брише. Сликата 3-5 го прикажува извршувањето на методата delete("Волк",node), каде јазелот tmp (или претходникот на јазелот кој го бришеме) е јазелот кој ја содржи вредноста Лисица.

```

1  public E delete(SLLNode<E> node) {
2      if (first != null) {
3          SLLNode<E> tmp = first;
4          if(first == node) {
5              return this.deleteFirst();
6          }
7          while (tmp.succ != node && tmp.succ.succ != null)
8              tmp = tmp.succ;
9          if (tmp.succ == node) {
10             tmp.succ = tmp.succ.succ;
11             return node.element;
12         } else {
13             System.out.println("Elementot ne postoi vo listata");
14             return null;
15         }
16     } else {
17         System.out.println("Listata e prazna");
18         return null;
19     }
20 }

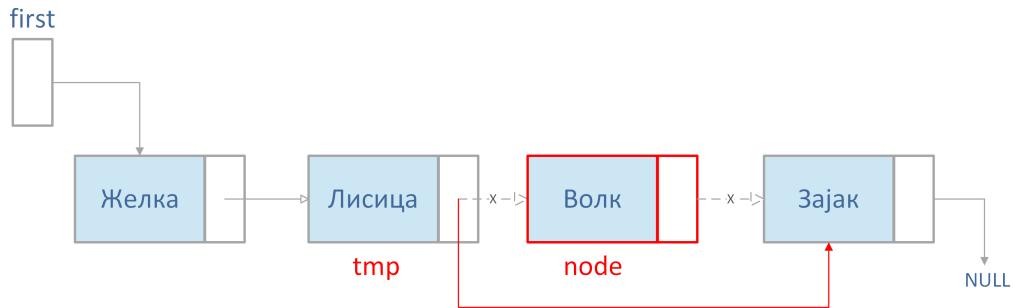
```

Имплементацијата на **наоѓање на јазел** во листата со дадена вредност е дадена во методот find(E o). Имплементацијата на методата ја изменува листата и о секој јазел проверува дали ја содржи бараната вредност. Во првиот момент кога ќе наиде на јазол кој ја содржи бараната вредност, го враќа тој јазел.

```

1  public SLLNode<E> find(E o) {

```



Слика 3-5: Пример на бришење на даден јазел

```

2     if (first != null) {
3         SLLNode<E> tmp = first;
4         while (!tmp.element.equals(o) && tmp.succ != null)
5             tmp = tmp.succ;
6         if (tmp.element.equals(o)) {
7             return tmp;
8         } else {
9             System.out.println("Elementot ne postoi vo listata");
10        }
11    } else {
12        System.out.println("Listata e prazna");
13    }
14    return null;
15 }
```

Имплементацијата на **бришење листа** е дадена во методот `deleteList()` и само го поставува `first` елементот на `null`. Со тоа се губи пристап до сите претходни јазли и вредности.

Дополнителни помошни методи се `size()`, која ја враќа **големината** (т.е. бројот на јазли во) на листата и `getFirst()` која го враќа **првиот јазол во листата**.

```

1  public void deleteList() {
2      first = null;
3  }
4
5  public SLLNode<E> getFirst() {
6      return first;
7  }
8
9  public int size() {
10     int listSize = 0;
```

```

11     SLLNode<E> tmp = first;
12     while(tmp != null) {
13         listSize++;
14         tmp = tmp.succ;
15     }
16     return listSize;
17 }
```

Користење на дефинираните методи

Следниот код дефинира класа SLLTester во која во main методата се дефинира листа која содржи цели броеви (integers) во јазлите и ја прикажува употребата на дефинираните методи

```

1 public class SLLTester {
2
3     public static void main(String[] args) {
4         SLL<Integer> lista = new SLL<Integer>();
5         lista.insertLast(5);
6         System.out.print("Listata po vmetnuvanje na 5 kako posleden element:
7             ");
8         System.out.println(lista.toString());
9
10        lista.insertFirst(3);
11        System.out.print("Listata po vmetnuvanje na 3 kako prv element: ");
12        System.out.println(lista.toString());
13
14        lista.insertLast(1);
15        System.out.print("Listata po vmetnuvanje na 1 kako posleden element:
16             ");
17        System.out.println(lista.toString());
18
19        lista.deleteFirst();
20        System.out.print("Listata po brishenje na prviot element: ");
21        System.out.println(lista.toString());
22
23        SLLNode<Integer> pom = lista.find(5);
24        lista.insertBefore(2, pom);
25        System.out.print("Listata po vmetnuvanje na elementot 2 pred
          elementot 5: ");
26        System.out.println(lista.toString());
```

```

26     pom = lista.find(1);
27     lista.insertAfter(3, pom);
28     System.out.print("Listata po vmetnuvanje na elementot 3 posle
29         elementot 1: ");
30     System.out.println(lista.toString());
31     System.out.println("Momentalna dolzina na listata: " + lista.size());
32
33     pom = lista.find(2);
34     lista.delete(pom);
35     System.out.print("Listata po brishenje na elementot 2: ");
36     System.out.println(lista.toString());
37     System.out.println("Momentalna dolzina na listata: " + lista.size());
38
39     lista.deleteList();
40     System.out.print("Pecatenje na listata po nejzino brishenje: ");
41     System.out.println(lista.toString());
42     System.out.println("Momentalna dolzina na listata: " + lista.size());
43 }
```

Излезот по извршувањето на овој код е:

```

Listata po vmetnuvanje na 5 kako posleden element: 5
Listata po vmetnuvanje na 3 kako prv element: 3->5
Listata po vmetnuvanje na 1 kako posleden element: 3->5->1
Listata po brishenje na prviот element: 5->1
Listata po vmetnuvanje na elementот 2 пред elementот 5: 2->5->1
Listata po vmetnuvanje на elementот 3 посle elementот 1: 2->5->1->3
Momentalna dolzina na listata: 4
Listata po brishenje на elementот 2: 5->1->3
Momentalna dolzina на listata: 3
Pecatenje на listata по nejzino brishenje: Prazna lista!!!
Momentalna dolzina на listata: 0
```

3.1.2 Едноставни проблеми со едностррано поврзани листи

Еднострраните листи имаат голема примена за градба на посложени податочни структури, како и за моделирање на различни случаи од реалниот свет.

Следат неколку примери за употреба од овој податочен тип.

Задача 1. Избриши последно појавување на број

Дадена е едностррано поврзана листа чии што јазли содржат цели броеви. За даден број од тастатура, потребно е да се отстрани неговото последно појавување (да се избрише јазолот што го содржи бројот).

Влез: Во првата линија е даден бројот на елементи n . Во следните n линии се дадени елементите на листата. Во последната линија е даден бројот кој треба да се отстрани (неговото последно појавување).

Излез: На излез треба да се испечатат јазлите на резултантната листа.

Пример.

Влез:

```
5
4
6
4
9
3
3
4
```

Излез:

```
4->6->9->3
```

Решение

Решението е дадено во main методата во IzbrishiPosleden класата.

```

1 import java.util.*;
2 public class IzbrishiPosleden {
3     public static void main(String[] args)
4     {
5         Scanner sc = new Scanner(System.in);
6         int n = sc.nextInt();
7         SLL<Integer> lista=new SLL<Integer>();
8
9         for(int i=0;i<n;i++)
10        {
11             int el=sc.nextInt();
12             lista.insertLast(el);
13         }
14         int todelete=sc.nextInt();
15
16         SLLNode<Integer> node=lista.getFirst();
17         SLLNode<Integer> brisi = null;
```

```

18     while(node!=null)
19     {
20         if(node.element==todelete)
21         {
22             brisi=node;
23         }
24         node=node.succ;
25     }
26     if(brisi!=null)
27         lista.delete(brisi);
28     System.out.println(lista.toString());
29 }
30 }
```

Задача 2. Замени соседи

Дадена е еднострano поврзана листа чии што јазли содржат по еден природен број. Да се трансформира листата така што секој соседен пар јазли ќе си ги заменат местата (првиот со вториот, па третиот со четвртиот итн...).

Влез: Во првата линија е даден бројот на елементи n. Во следните n линии се дадени елементите на листата.

Излез: На излез треба да се испечатат јазлите на резултантната листа.

Пример.

Влез:

```

4
1
2
3
4
```

Излез:

```
2->1->4->3
```

Решение

Решението е дадено во main методата во SwapPairs класата.

```

1 import java.io.BufferedReader;
2 import java.io.InputStreamReader;
3
4 public class SwapPairs {
5     public static void main(String[] args) throws Exception {
```

```

6     SLL<Integer> lista = new SLL<Integer>();
7     BufferedReader stdin = new BufferedReader(new
8         InputStreamReader(System.in));
9     int n = Integer.parseInt(stdin.readLine());
10    for (int i = 0; i < n; i++) {
11        lista.insertLast(Integer.parseInt(stdin.readLine()));
12    }
13
14    SLLNode<Integer> jazol = lista.getFirst();
15    while (jazol != null && jazol.succ != null) {
16        Integer pom = jazol.element;
17        jazol.element = jazol.succ.element;
18        jazol.succ.element = pom;
19        jazol = jazol.succ.succ;
20    }
21    System.out.print(lista.toString());
22 }

```

Задача 3. Раздели листа

Дадена е еднострено поврзана листа со природни броеви. Да се креираат две резултантни еднострено поврзани листи т.ш. во првата листа ќе се земаат само јазлите што содржат парни број, при што доколку во првичната листа има повеќе соседни јазли со парни броеви се зема само последниот јазел. Слична процедура се применува и за втората резултантна листа, при што овде се земаат само јазлите што содржат непарни броеви, при што ако има повеќе соседни јазли со непарни броеви се зема само последниот јазел.

Влез: Во првата линија е даден бројот на елементи n . Во втората линија се даваат броевите во листата одделени со празно место.

Излез: Прво се печати резултантната листа со прости броеви, а потоа во нов ред таа со непрости. Доколку некоја од листите е празна се печати: Prazna lista.

Пример.

Влез:

8

1 3 2 4 5 7 6 8

Излез:

4->8

3->7

Решение

Решението е дадено во main методата во RazdeliLista класата.

```

1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4
5 public class RazdeliLista {
6     public static void main(String[] args) throws IOException {
7
8         SLL<Integer> lista = new SLL<Integer>();
9         SLL<Integer> parni = new SLL<Integer>();
10        SLL<Integer> neparni = new SLL<Integer>();
11        BufferedReader stdin = new BufferedReader(new
12            InputStreamReader(System.in));
13        String s = stdin.readLine();
14        int N = Integer.parseInt(s);
15        s = stdin.readLine();
16        String[] pomniza = s.split(" ");
17        for (int i = 0; i < N; i++) {
18            lista.insertLast(Integer.parseInt(pomniza[i]));
19        }
20
21        SLLNode<Integer> pom = lista.getFirst();
22
23        while(pom!=null){
24            while(pom.succ!=null && pom.element%2==0 && pom.succ.element%2==0){
25                pom=pom.succ;
26            }
27            while(pom.succ!=null && !(pom.element%2==0) &&
28                !(pom.succ.element%2==0)){
29                pom=pom.succ;
30            }
31            if(pom.element%2==0)
32                parni.insertLast(pom.element);
33            else
34                neparni.insertLast(pom.element);
35            pom = pom.succ;
36        }
37
38        if(parni.size()==0) System.out.println("Prazna lista");
39        else System.out.println(parni);

```

```

38
39     if(neparni.size()==0) System.out.println("Prazna lista");
40     else System.out.println(neparni);
41 }
42 }
```

3.1.3 Напредни проблеми со едностррано поврзани листи

Задача 1. Преврти листа

Да се напише функција/метода во склоп на SLL класата што ги превртува сите врски во едностррано поврзана листа.

Решение

Решението е дадено во методата mirror() и употребата на истата е прикажана во main методата.

```

1  public void mirror() {
2      if (first != null) {
3          //m=nextsucc, p=tmp,q=next
4          SLLNode<E> tmp = first;
5          SLLNode<E> newsucc = null;
6          SLLNode<E> next;
7
8          while(tmp != null){
9              next = tmp.succ;
10             tmp.succ = newsucc;
11             newsucc = tmp;
12             tmp = next;
13         }
14         first = newsucc;
15     }
16 }
17
18 public static void main(String[] args) {
19     SLL<String> listaIminja = new SLL<String>();
20     listaIminja.insertLast("Ana");
21     listaIminja.insertLast("Bojana");
22     listaIminja.insertLast("Cece");
23     listaIminja.insertLast("Dina");
24     System.out.print("Listata pred prevrtuvanje: ");
```

```

25     System.out.println(listaIminja.toString());
26     listaIminja.mirror();
27     System.out.print("Listata po prevrtuvanje: ");
28     System.out.println(listaIminja.toString());
29 }
```

Излезот по извршувањето на овој код е:

```
Listata pred prevrtuvanje: Ana->Bojana->Cece->Dina
Listata po prevrtuvanje: Dina->Cece->Bojana->Ana
```

Задача 2. Спои сортирани листи

Нека се дадени две еднострено поврзани листи чии јазли се сортирани во растечки редослед. Да се напише функција која ќе ги спои двете листи во една така што резултантната листа да е сортирана. Сортирањето е подредување со слевање.

Решение

Решението е дадено во методата join() во JoinSortedLists класата и употребата е прикажана во main методата. Притоа сметаме дека генеричкиот тип E е од тип Comparable, за да може да се изврши споредба на вредностите на јазлите.

```

1 public class JoinSortedLists<E extends Comparable<E>> {
2
3     public SLL<E> join(SLL<E> list1, SLL<E> list2){
4         SLL<E> rezultat = new SLL<E>();
5         SLLNode<E> jazol1 = list1.getFirst(), jazol2 = list2.getFirst();
6         //SLLNode<E> jazol2 = list2.getFirst();
7
8         while(jazol1 != null && jazol2 != null){
9             if(jazol1.element.compareTo(jazol2.element)<0){ //jazol1<jazol2
10                 rezultat.insertLast(jazol1.element);
11                 jazol1 = jazol1.succ;
12             }
13             else{
14                 rezultat.insertLast(jazol2.element);
15                 jazol2 = jazol2.succ;
16             }
17
18         }
19
20         if(jazol1 != null){
```

```

21         while(jazol1 != null){
22             rezultat.insertLast(jazol1.element);
23             jazol1 = jazol1.succ;
24         }
25     }
26
27     if(jazol2 != null){
28         while(jazol2 != null){
29             rezultat.insertLast(jazol2.element);
30             jazol2 = jazol2.succ;
31         }
32     }
33
34     return rezultat;
35 }
36
37 public static void main(String[] args){
38     SLL<String> lista1 = new SLL<String>();
39     lista1.insertLast("Ana");
40     lista1.insertLast("Bojana");
41     lista1.insertLast("Dejan");
42     SLL<String> lista2 = new SLL<String>();
43     lista2.insertLast("Andrijana");
44     lista2.insertLast("Biljana");
45     lista2.insertLast("Darko");
46
47     JoinSortedLists<String> js = new JoinSortedLists<String>();
48     System.out.println(js.join(lista1, lista2));
49 }
50 }
```

Излезот по извршувањето на овој код е:

Ana->Andrijana->Biljana->Bojana->Darko->Dejan

Задача 3. Преуреди листа

Дадена е еднострено поврзана листа $L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$. Преуредете ги јазлите во листата така што новата листа ќе биде : $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \dots$

Влез: Во првата линија е даден бројот на елементи n. Во втората линија се даваат броевите во листата одделени со празно место.

Излез: На излез треба да се испечати преуредената листа
Пример.

Влез:

5
1 2 3 4 5

Излез:

1->5->2->4->3

Решение

Решението е дадено со методата rearrange() во SLL класата. Идејата е прво да се најде средината на листата, па да се преврти втората половина на листата (од средината до крај, со употреба на методата reverselist(SLLNode<E> node)) и со сливање (наизменично) да се додаваат јазли од првата и втората половина.

```

1   SLLNode<E> reverselist(SLLNode<E> node) {
2       SLLNode<E> prev = null, curr = node, next;
3       while (curr != null) {
4           next = curr.succ;
5           curr.succ = prev;
6           prev = curr;
7           curr = next;
8       }
9       node = prev;
10      return node;
11  }
12
13 void rearrange() {
14     //1) Најди ја средината на листата
15     SLLNode<E> sredina = this.getFirst();
16     for(int i=1;i<this.size()/2;i++)
17         sredina = sredina.succ;
18     System.out.println(sredina.element());
19
20     //2) Подели ја листата на две половини
21     //node1, првиот јазел од првата половина 1 -> 2 -> 3
22     //node2, првиот јазел од втората половина 4 -> 5
23     SLLNode<E> node1 = this.getFirst();
24     SLLNode<E> node2 = sredina.succ;
25     sredina.succ = null;
26
27     //3) Преврти ја втората половина т.е. 5 -> 4

```

```
28     node2 = reverselist(node2);
29
30     //4) Наизменично спојувај ги јазлите
31     SLLNode<E> node = new SLLNode<E>(null, null); //помошен јазол
32
33     // curr е покажувачот на помошниот јазол
34     // од каде ќе се формира новата листа
35     SLLNode<E> curr = node;
36     while (node1 != null || node2 != null) {
37
38         // Прво додад јазол од првата листа
39         if (node1 != null) {
40             curr.succ = node1;
41             curr = curr.succ;
42             node1 = node1.succ;
43         }
44
45         // Пак додад јазол од втората листа
46         if (node2 != null) {
47             curr.succ = node2;
48             curr = curr.succ;
49             node2 = node2.succ;
50         }
51     }
52
53     // Отстрани го помошниот јазел
54     node = node.succ;
55 }
56
57 public static void main(String[] args) {
58     Scanner sc = new Scanner(System.in);
59     int n = sc.nextInt();
60     SLL<Integer> lista=new SLL<Integer>();
61
62     for(int i=0;i<n;i++)
63     {
64         int el=sc.nextInt();
65         lista.insertLast(el);
66     }
67
68     lista.rearrange();
```

```

69     System.out.println(lista.toString());
70 }
```

3.1.4 Задачи за вежбање

Задача 1. Раздели min-max листа

Дадена е еднострено поврзана листа чии што јазли содржат по еден природен број. Во дадената листа треба да се пронајдат елементите со најмала и најголема вредност и потоа листата треба да се подели на две резултантни еднострено поврзани листи, т.ш. во првата листа треба да се сместат сите јазли кои содржат броеви поблиски до најмалиот елемент отколку до најголемиот елемент, а во втората сите јазли кои содржат броеви поблиски до најголемиот елемент отколку до најмалиот. Доколку елементот е на исто растојание од најмалиот и најголемиот елемент тогаш се сместува во листата на елементи поблиски до најмалот елемент. Јазлите во резултантните листи се додаваат според редоследот по кој се појавуваат во дадената листа. (Помош: бројот 3 е на растојание 2 од бројот 1 и на растојание 4 од бројот 7. Следува дека бројот 3 е поблиску до бројот 1 отколку до бројот 7).

Влез: Во првата линија е даден бројот на елементи n . Во втората линија се даваат броевите во листата одделени со празно место.

Излез: На излез во првиот ред треба да се испечатат јазлите по редослед на првата резултантната листа (која содржи елементи кои се поблиску до најмалиот елемент). Во вториот ред треба да се испечатат јазлите на по редослед на втората резултантната листа (која содржи елементи кои се поблиску до најголемиот елемент).

Пример.

Влез:

9

1 5 7 3 2 9 4 8 6

Излез:

1->5->3->2->4

7->9->8->6

Задача 2. Бриши јазли од листа

Дадена е еднострана поврзана листа со цели броеви. Ваша задача е да бришете јазли т.ш. прво ќе оставите еден јазол, еден ќе бришете, па ќе оставите 2 јазли еден ќе бришете, па ќе оставите 3 јазли па еден ќе бришете итн... Односно од вас

се бара да бришете преку 1, па преку 2, па преку 3 јазли итн... додека е возможно да се брише.

Влез: Во првата линија е даден бројот на елементи n. Во втората линија се даваат броевите во листата одделени со празно место.

Излез: Резултатната листа. Доколку листата е празна испечатете: Prazna lista

Пример.

Влез:

9

4 6 8 3 1 3 5 7 2

Излез:

4->8->3->3->5->7

Задача 3. Наизменично спои листи

Дадени се две еднострano поврзани листи чии што јазли содржат по еден природен број. Треба да се спојат двете листи во една резултантна на тој начин што наизменично прво ќе се даваат првите два јазли од првата листа во резултантната, па првите два од втората листа, па следните два од првата, па следните два од втората итн. Јазлите што ќе останат треба да се додадат на крај во резултантната листа, прво оние што останале од првата листа, потоа оние што останале од втората листа.

Влез: Во првата линија е даден бројот на елементи на првата листа, па втората линија се даваат броевите во листата одделени со празно место. Во третата линија бројот на елементи на втората листа, па четвртата линија се даваат броевите во листата одделени со празно место.

Излез: На излез треба да се испечатат јазлите по редослед во резултантната споена листа.

Пример.

Влез:

5

5 7 9

8

1 1 4 5 6 8 9 4

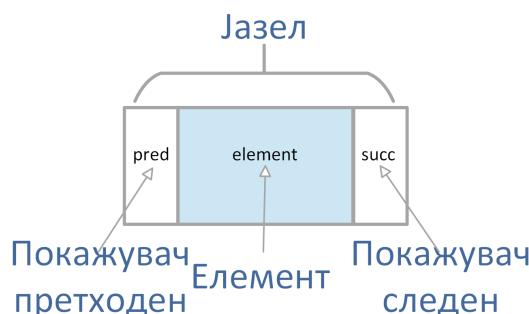
Излез:

5->7->1->1->9->4->5->6->8->9->4

3.2 Двострано поврзани листи

Од анализата на операциите на бришење и вметнување на елементи кај едностррано поврзаните листи, произлегува проблемот со одредување на предходникот на даден јазел. Оваа операција во ошт случај има линеарна комплексност затоа што е потребно да се позиционираме на претходникот на јазелот кој треба да се избрише или кога треба да се внесе нов јазол пред него. Овој проблем би имал тривијално решение доколку дефинираме податочна структура чии елементи би имале дополнително линк поле (вкупно две линк полиња) кои би покажувале на предходниот и следниот елемент во поврзаната листа. Таквата структура вообичаено се нарекува двојно поврзана листа.

Јазлите на двострано поврзаните листи содржат информации и покажувачи кон наредниот и претходниот јазел во листата. На слика 3-6 е даден визуелен приказ на јазел. Елементот ја содржи информацијата која ја доделуваме, додека покажувачите ја даваат мемориската локација на следниот и претходниот јазел во листата. Java класата која моделира еден ваков јазел дадена е во кодот `DLLNode`. Елементот на листата (`element` во кодот) е од генерички тип кој според потребата се наведува при декларација на листата.



Слика 3-6: Визуелен приказ на јазел на двоострано поврзана листа

```

1 public class DLLNode<E> {
2     protected E element; //елементот на јазелот
3     protected DLLNode<E> pred,succ; //покажувачи кон претходен и следен јазел
4
5     public DLLNode(E elem, DLLNode<E> pred, DLLNode<E> succ) {
6         this.element = elem;
7         this.pred = pred;
8         this.succ = succ;
9     }
10 }
```

При дефиниција на двострано поврзана листа потребно ни е да ги знаеме каде се наоѓаат првиот и последниот јазел на листата. Потоа тргнувајќи од било кој од тие јазли може да ја изминеме целата листа. Затоа соодветно при дефиниција на податочната структурачуваме само покажувачи кон првиот и последниот јазел. Ова е прикажано во кодот DLL кој ја моделира структурата и има два јазли first и last од тип DLLNode.

```

1 public class LL<E> {
2     private DLLNode<E> first, last; //првиот и последниот јазел во листата
3
4     public DLL() {
5         //креирање на празна листа
6         this.first = null;
7         this.last = null;
8     }
9
10    //методи на SLL
11    ...
12 }
```

3.2.1 Операции со двострано поврзани листи

Базични операции со двострано поврзани листи се исти како и на едностррано поврзани листи, и вклучуваат:

- изминување и печатење на листа
- вметнување на елемент
- бришење на елемент
- наоѓање на елемент
- бришење на листа

Операцијата **изминување и печатење** на листа се имплементира со препокривање на методата `toString()` во Java која враќа `String`. Како и кај едностррано поврзани листи, се започнува со пристап до првиот јазел во листата преку параметарот `first` (односно се дефинира привремен јазел `tmp` кој на почетокот покажува на првиот елемент во листата. Потоа се додека има следен јазел се зема информацијата на моменталниот јазел (т.е. `tmp` јазелот), се додава во резултантниот `String` и се преминува на следниот јазел. Бидејќи методата линерно ги изминува сите јазли во листата, нејзината сложеност е $O(n)$. Слично методата `toStringR` ја изминува листата и враќа `String` како резултат, но започнува со изминување од последниот елемент во листата и се враќа наназад (односно се

движи со употреба на покажувачите на претходници, за разлика од `toString()` која ги употребува покажувачите на следбеници на јазли).

```

1   @Override
2   public String toString() {
3       String ret = new String();
4       if (first != null) {
5           DLLNode<E> tmp = first;
6           ret += tmp.toString();
7           while (tmp.succ != null) {
8               tmp = tmp.succ;
9               ret += "<->" + tmp.toString();
10          }
11      } else
12          ret = "Prazna lista!!!";
13      return ret;
14  }
15
16  public String toStringR() {
17      String ret = new String();
18      if (last != null) {
19          DLLNode<E> tmp = last;
20          ret += tmp.toString();
21          while (tmp.pred != null) {
22              tmp = tmp.pred;
23              ret += "<->" + tmp.toString();
24          }
25      } else
26          ret = "Prazna lista!!!";
27      return ret;
28  }

```

Операцијата за **вметнување** на нов јазел во двострано поврзана листа ги вклучува 4те различни случаи како и кај едностррано поврзана листа:

- вметнување на елемент на почеток во листата
- вметнување на елемент на крај во непразна листа
- вметнување на елемент по даден јазол
- вметнување на елемент пред даден јазол

Имплементациите на методи се слични, со таа разлика што кај двострана треба да се погрижиме и за покажувачите кон претходниците дополнително.

Така имплементацијата на **вметнување на елемент на почеток во листата** е дадена во методот insertFirst(E o) е слична со таа за едностррано поврзанат листа, со тоа што дополнително новиот јазел се поставува како претходник на досегашниот прв јазел.

```

1  public void insertFirst(E o) {
2      DLLNode<E> ins = new DLLNode<E>(o, null, first);
3      if (first == null)
4          last = ins;
5      else
6          first.pred = ins;
7      first = ins;
8  }

```

Имплементацијата на **вметнување на елемент по даден јазол** е дадена во методот insertAfter(E o, DLLNode<E> after), чии влезни параметри се новиот елемент о кој сакаме да го вметнеме после јазелот after. Во имплементацијата се креира нов јазел ins чија вредност е дадената влезна вредност E o, претходник е јазекот after, а следбеник е следбеникот на јазелот after. Новиот јазел ins се поставува како претходник на следбеникот на јазелот after. Како следбеник на јазелот after се поставува новиот јазел ins.

```

1  public void insertAfter(E o, DLLNode<E> after) {
2      if (after == last) {
3          insertLast(o);
4          return;
5      }
6      DLLNode<E> ins = new DLLNode<E>(o, after, after.succ);
7      after.succ.pred = ins;
8      after.succ = ins;
9  }

```

Имплементациите на **вметнување на елемент на крај во листа** како и **вметнување на елемент пред даден јазол** се дадени во методите insertLast(E o) и insertBefore(E o, DLLNode<E> node) соодветно.

```

1  public void insertLast(E o) {
2      if (first == null)
3          insertFirst(o);
4      else {
5          DLLNode<E> ins = new DLLNode<E>(o, last, null);
6          last.succ = ins;
7          last = ins;

```

```

8         }
9     }
10
11    public void insertBefore(E o, DLLNode<E> before) {
12        if (before == first) {
13            insertFirst(o);
14            return;
15        }
16        DLLNode<E> ins = new DLLNode<E>(o, before.pred, before);
17        before.pred.succ = ins;
18        before.pred = ins;
19    }

```

Операцијата за **бришење** на јазел во листата вклучува два различни случаи:

- бришење на првиот јазел во листата
- бришење на последниот јазел во листата
- бришење на даден јазол во листата

Имплементацијата на **бришење на првиот јазел во листата** е дадена во методот `deleteFirst()`. Имплементацијата на методата започнува проверка дали постои прв јазел и доколку постои се става неговиот следбеник како прв јазел. Притоа се враќа како повратна вредност вредноста на избришаниот јазел. Слично имплементацијата на **бришење на последниот јазел во листата** е дадена во методот `deleteLast()` кој, доколку има барем уште еден јазел во листата покрај првиот, како последен јазел го става претходникот на последниот јазел.

```

1   public E deleteFirst() {
2       if (first != null) {
3           DLLNode<E> tmp = first;
4           first = first.succ;
5           if (first != null) first.pred = null;
6           if (first == null)
7               last = null;
8           return tmp.element;
9       } else
10          return null;
11   }
12
13  public E deleteLast() {
14      if (first != null) {
15          if (first.succ == null)

```

```

16         return deleteFirst();
17     else {
18         DLLNode<E> tmp = last;
19         last = last.pred;
20         last.succ = null;
21         return tmp.element;
22     }
23 } else
24     return null;
25 }
```

Имплементацијата на **бришење на даден јазел во листата** е дадена во методот delete(DLLNode<E> node). Имплементацијата едноставно (за разлика од еднострano поврзаните листи) го премостува јазел што се брише со манипулација на покажувачите од неговиот претходник и следбеник .

```

1  public E delete(DLLNode<E> node) {
2      if (node == first) {
3          return deleteFirst();
4      }
5      if (node == last) {
6          return deleteLast();
7      }
8      node.pred.succ = node.succ;
9      node.succ.pred = node.pred;
10     return node.element;
11
12 }
```

Имплементацијата на **наоѓање на јазел** во листата со дадена вредност е дадена во методот find(E o). Имплементацијата на методата ја изменува листата и о секој јазел проверува дали ја содржи бараната вредност. Во првиот момент кога ќе наиде на јазол кој ја содржи бараната вредност, го враќа тој јазел.

```

1  public DLLNode<E> find(E o) {
2      if (first != null) {
3          DLLNode<E> tmp = first;
4          while (!tmp.element.equals(o) && tmp.succ != null)
5              tmp = tmp.succ;
6          if (tmp.element.equals(o)) {
7              return tmp;
8          } else {
9              System.out.println("Elementot ne postoi vo listata");
10 }
```

```

10         }
11     } else {
12         System.out.println("Listata e prazna");
13     }
14     return null;
15 }
```

Имплементацијата на **бришење листа** е дадена во методот `deleteList()` и само го поставува `first` елементот на `null`. Со тоа се губи пристап до сите претходни јазли и вредности.

Дополнителни помошни методи се `getSize()`, која ја враќа **големината** (т.е. бројот на јазли во) на листата и `getFirst()` и `getLast()` кои ги враќаат **првиот и последниот јазол во листата**.

```

1  public void deleteList() {
2      first = null;
3      last = null;
4  }
5
6  public DLLNode<E> getFirst() {
7      return first;
8  }
9
10 public DLLNode<E> getLast() {
11
12     return last;
13 }
14
15 public int getSize() {
16     int listSize = 0;
17     DLLNode<E> tmp = first;
18     while(tmp != null) {
19         listSize++;
20         tmp = tmp.succ;
21     }
22     return listSize;
23 }
```

Користење на дефинираните методи

Следниот код дефинира класа `DLLTester` во која во `main` методата се дефинира листа која содржи цели броеви (`integers`) во јазлите и ја прикажува употребата

на дефинираните методи

```

1 public class DLLTester {
2
3     public static void main(String[] args) {
4         DLL<Integer> lista = new DLL<Integer>();
5         lista.insertLast(5);
6         System.out.print("Listata po vmetnuvanje na 5 kako posleden element:
7             ");
8         System.out.println(lista.toString()+" i obratno "+lista.toStringR());
9
10        lista.insertFirst(3);
11        System.out.print("Listata po vmetnuvanje na 3 kako prv element: ");
12        System.out.println(lista.toString()+" i obratno "+lista.toStringR());
13
14        lista.insertLast(1);
15        System.out.print("Listata po vmetnuvanje na 1 kako posleden element:
16             ");
17        System.out.println(lista.toString()+" i obratno "+lista.toStringR());
18
19        lista.deleteFirst();
20        System.out.print("Listata po brishenje na prviot element: ");
21        System.out.println(lista.toString()+" i obratno "+lista.toStringR());
22
23        DLLNode<Integer> pom = lista.find(5);
24        lista.insertBefore(2, pom);
25        System.out.print("Listata po vmetnuvanje na elementot 2 pred
26             elementot 5: ");
27        System.out.println(lista.toString()+" i obratno "+lista.toStringR());
28
29        pom = lista.find(1);
30        lista.insertAfter(3, pom);
31        System.out.print("Listata po vmetnuvanje na elementot 3 posle
32             elementot 1: ");
33        System.out.println(lista.toString()+" i obratno "+lista.toStringR());
34
35        pom = lista.find(1);
36        lista.insertAfter(6, pom);
37        System.out.print("Listata po vmetnuvanje na elementot 6 posle
38             elementot 1: ");
39        System.out.println(lista.toString()+" i obratno "+lista.toStringR());

```

```

36     pom = lista.find(3);
37     lista.delete(pom);
38     System.out.print("Listata po brishenje na elementot 3: ");
39     System.out.println(lista.toString()+" i obratno "+lista.toStringR());
40
41     System.out.println("Momentalna dolzina na listata:
42         "+lista.getSize());
43
44     lista.deleteList();
45     System.out.print("Pecatenje na listata po nejzino brishenje: ");
46     System.out.println(lista.toString()+" i obratno "+lista.toStringR());
47     System.out.println("Momentalna dolzina na listata:
48         "+lista.getSize());
    }
}

```

Излезот по извршувањето на овој код е:

```

Listata po vmetnuvanje na 5 kako posleden element: 5 i obratno 5
Listata po vmetnuvanje na 3 kako prv element: 3<->5 i obratno 5<->3
Listata po vmetnuvanje na 1 kako posleden element: 3<->5<->1 i obratno
1<->5<->3
Listata po brishenje na prviот element: 5<->1 i obratno 1<->5
Listata po vmetnuvanje na elementот 2 пред elementот 5: 2<->5<->1
i obratno 1<->5<->2
Listata po vmetnuvanje на elementот 3 posle elementот 1: 2<->5<->1<->3
i obratно 3<->1<->5<->2
Listata po vmetnuvanje на elementот 6 posle elementот 1: 2<->5<->1<->6<->3
i obratно 3<->6<->1<->5<->2
Listata po brishenje на elementот 3: 2<->5<->1<->6 i obratно 6<->1<->5<->2
Momentalna dolzina na listata: 4
Pecatenje na listata по nejzino brishenje: Prazna lista!!!
i obratno Prazna lista!!!
Momentalna dolzina na listata: 0

```

3.2.2 Едноставни проблеми со двострано поврзани листи

Задача 1. Палиндром

Дадена е двојно поврзана листа со N јазли каде секој јазел содржи по еден број. Да се провери дали двојно поврзаната листа е палиндром: односно ако ја измиснете од почеток до крај и од крај до почеток, дали ќе добиете ист збор.

Влез: Во првиот ред од влезот даден е бројот на јазли во листата N, а во вториот ред се дадени броевите.

Излез: На излез треба да се испечати 1 ако листата е палиндром, -1 ако не е.
Пример.

Влез 1:

5
1 2 3 1 2

Излез:

-1

Влез 2:

5
1 2 3 2 1

Излез:

1

Решение

Решението е дадено во main методата во PalindromeDLL класата.

```

1  public class PalindromeDLL {
2
3      public static int isItPalindrome(DLL<Integer> list){
4          DLLNode<Integer> poceten = list.getFirst();
5          DLLNode<Integer> posleden = list.getLast();
6          while((poceten != posleden)&&(poceten.pred != posleden)){
7              if(!poceten.element.equals(posleden.element))
8                  return -1;
9              poceten = poceten.succ;
10             posleden = posleden.pred;
11         }
12         return 1;
13     }
14
15     public static void main(String[] args) {
16         Scanner in = new Scanner(System.in);
17         int n = in.nextInt();
18         DLL<Integer> list = new DLL<Integer>();
19         for (int i = 0; i < n; i++) {
20             list.insertLast(in.nextInt());
21         }
22         in.close();

```

```

23     System.out.println(isItPalindrome(list));
24 }
25
26 }
```

Задача 2. Подели според парност

Дадена е двострано поврзана листа чии јазли содржат по еден природен број. Листата треба да се подели на две резултантни листи, т.ш. во првата резултантна листа ќе бидат сместени јазли од влезната листа кои содржат парни броеви, а во втората – непарните. Јазлите во резултантните листи се додаваат наизменично почнувајќи од почетокот и крајот на влезната листа (т.е. прво се разгледува првиот елемент од листата и се додава во соодветната резултантна листа, па последниот, па вториот итн...).

Влез: Во првиот ред од влезот е даден бројот на јазли во листата, а во вториот ред се дадени броевите од кои се составени јазлите по редослед во листата.

Излез: Во првиот ред од излезот треба да се испечатат јазлите по редослед од првата резултантна листа (т.е. парните), во вториот ред од втората (т.е. непарните).

Пример.

Влез:

5

1 2 3 4 5 Излез:

2 4

1 5 3

Решение

Решението е дадено во main методата во PodeliSporedParnost класата.

```

1 public class PodeliSporedParnost {
2
3     public static void podeliParnost(DLL<Integer> lista, DLL<Integer>
4                                     lparni, DLL<Integer> lneparni) {
5
6         DLLNode<Integer> pom1 = lista.getFirst();
7         DLLNode<Integer> pom2 = lista.getLast();
8
9         while(pom1!=pom2 && pom2.succ!=pom1){
10             if(pom1.element%2==0)
11                 lparni.insertLast(pom1.element);
```

```

11     else
12         lneparni.insertLast(pom1.element);
13     if(pom2.element%2==0)
14         lparni.insertLast(pom2.element);
15     else
16         lneparni.insertLast(pom2.element);
17     pom1 =pom1.succ;
18     pom2=pom2.pred;
19 }
20 if(pom1==pom2){
21     if(pom1.element%2==0)
22         lparni.insertLast(pom1.element);
23     else
24         lneparni.insertLast(pom1.element);
25     return;
26 }
27 }
28
29 public static void main(String[] args) throws IOException {
30     DLL<Integer> lista = new DLL<Integer>(), parni = new DLL<Integer>(),
31     neparni = new DLL<Integer>();
32     BufferedReader stdin = new BufferedReader(new InputStreamReader(
33             System.in));
34     String s = stdin.readLine();
35     int N = Integer.parseInt(s);
36     s = stdin.readLine();
37     String[] pomniza = s.split(" ");
38     for (int i = 0; i < N; i++) {
39         lista.insertLast(Integer.parseInt(pomniza[i]));
40     }
41     podeliParnost(lista, parni, neparni);
42
43 // Pecatenje parni
44 DLLNode<Integer> tmp = parni.getFirst();
45 while (tmp != null) {
46     System.out.print(tmp.element);
47     if (tmp.succ != null)
48         System.out.print(" ");
49     tmp = tmp.succ;
50 }
```

```

51     System.out.println();
52     // Pecatenje neparni
53     tmp = neparni.getFirst();
54     while (tmp != null) {
55         System.out.print(tmp.element());
56         if (tmp.succ != null)
57             System.out.print(" ");
58         tmp = tmp.succ;
59     }
60     System.out.println();
61 }
62 }
```

Задача 3. Вметни просек меѓу соседи

Дадена е двострано поврзана листа која содржи природни броеви. Ваша задача е да ја преуредите влезната листата, т.ш. меѓу секои два соседни јазли од влезната листа ќе додадете нов јазел кој ќе ја содржи средната вредност од двата соседни јазли. Доколку средната вредност е децимална, тогаш бројот треба да биде заокружен на поголемиот (пр. Ако соседните јазли имаат вредност 1 и 2, нивната средна вредност е 1,5 и оваа вредност се заокружува на 2).

Влез: Од стандарден влез во првиот ред се дава цел број N, кој го претставува бројот на елементи во листата, а во вториот се даваат броевите во листата одделени со празно место.

Излез: Ваша задача е да ја испечатите резултантната листа.

Пример.

Влез:

3

1 2 4

Излез:

1 2 2 3 4

Решение

Решението е дадено во main методата во VmetniProsek класата.

```

1 public class VmetniProsek {
2     public static void main(String[] args) throws IOException {
3         // TODO Auto-generated method stub
4
5         DLL<Integer> lista = new DLL<Integer>();
```

```

6
7     BufferedReader br = new BufferedReader(new
8         InputStreamReader(System.in));
9     String s = br.readLine();
10    int N = Integer.parseInt(s);
11    s = br.readLine();
12    String[] pomniza = s.split(" ");
13    for (int i = 0; i < N; i++) {
14        lista.insertLast(Integer.parseInt(pomniza[i]));
15    }
16
17    DLLNode<Integer> tmp = lista.getFirst();
18    DLLNode<Integer> next = tmp.succ;
19    while (tmp != null && next != null) {
20        float a = tmp.element;
21        float b = next.element;
22        Integer nov = Math.round((a+b)/2);
23        lista.insertAfter(nov, tmp);
24        tmp = next;
25        next = tmp.succ;
26    }
27
28    tmp = lista.getFirst();
29    while (tmp != null) {
30        System.out.print(tmp.element + " ");
31        tmp = tmp.succ;
32    }
33}
34
35}

```

3.2.3 Напредни проблеми со двострано поврзани листи

Задача 1. Листа од листи

Дадена е двојно поврзана листа од двојно поврзани листи. Да се најде сума на секоја од подлистите, а потоа производ на овие суми.

Влез: Број N кој кажува колку листи има.

Број M кој кажува колку елементи има во секоја листа.

Во следните M линии се податоците $1 \leq A \leq 1000$ за секоја од листите

Излез: Еден број што е производот на сумите од низите. Со седум децимали.
Пример.

Влез:

```
3
4
1 2 3 4
2 3 4 5
6 7 8 9
```

Излез:

```
1400
```

Решение

Решението е дадено во main методата во ListaOdListi класата.

```

1 public class ListaOdListi {
2
3     public static long findMagicNumber(DLL<DLL<Integer>> list) {
4         DLLNode<DLL<Integer>> current = list.getFirst();
5         long prod = 1;
6         while (true) {
7             int sum = 0;
8             DLLNode<Integer> current1 = current.element.getFirst();
9             while (true) {
10                 sum += current1.element;
11                 if (current1 == current.element.getLast()) {
12                     break;
13                 }
14                 current1 = current1.succ;
15             }
16             prod *= sum;
17             if (current == list.getLast()) {
18                 break;
19             }
20             current = current.succ;
21         }
22         return prod;
23     }
24
25     public static void main(String[] args) {
26         Scanner in = new Scanner(System.in);
27         int n = in.nextInt();
```

```

28     int m = in.nextInt();
29     DLL<DLL<Integer>> list = new DLL<DLL<Integer>>();
30     for (int i = 0; i < n; i++) {
31         DLL<Integer> tmp = new DLL<Integer>();
32         for (int j = 0; j < m; j++) {
33             tmp.insertLast(in.nextInt());
34         }
35         list.insertLast(tmp);
36     }
37     in.close();
38     System.out.println(findMagicNumber(list));
39 }
40 }
```

Задача 2. Преврти ја листата

Дадена е двострано поврзана листа чии што јазли содржат по еден природен број. Листата треба да се преврти т.ш. прво се превртуваат јазлите кои содржат парни броеви, а потоа јазлите со непарни броеви. Листата се разгледува од назад. Право на користење имате само една дополнителна помошна двострано поврзана листа.

Влез: Во првиот ред од влезот е даден бројот на јазли во листа, потоа во вториот ред се дадени броевите од кои се составени јазлите по редослед во листата.

Излез: На излез треба да се испечатат јазлите по редослед во превртената листа

Пример.

Влез:

5

1 2 3 4 5

Излез:

4 2 5 3 1

Решение

Решението е дадено во main методата во PrevrtiLista класата.

```

1 public class PrevrtiLista {
2
3     public static void prevrtiLista(DLL<Integer> lista, DLL<Integer>
4                                     pomosna) {
```

```
5     DLLNode<Integer> pom = lista.getLast();
6
7     while (pom != null) {
8         if (pom.element % 2 == 0) {
9             pomasna.insertLast(pom.element);
10            if(pom==lista.getFirst()){
11                lista.deleteFirst();
12            }
13            else if(pom==lista.getLast()){
14                lista.deleteLast();
15            }
16            else {
17                (pom.pred).succ = pom.succ;
18                (pom.succ).pred = pom.pred;
19            }
20        }
21
22        pom = pom.pred;
23    }
24
25    pom = lista.getLast();
26    while (pom != null) {
27        pomasna.insertLast(pom.element);
28        pom = pom.pred;
29    }
30
31 }
32
33 public static void main(String[] args) throws IOException {
34     DLL<Integer> lista = new DLL<Integer>(), pomasna = new DLL<Integer>();
35     BufferedReader stdin = new BufferedReader(new InputStreamReader(
36             System.in));
37     String s = stdin.readLine();
38     int N = Integer.parseInt(s);
39     s = stdin.readLine();
40     String[] pomniza = s.split(" ");
41     for (int i = 0; i < N; i++) {
42         lista.insertLast(Integer.parseInt(pomniza[i]));
43     }
44
45     prevrtiLista(lista, pomasna);
```

```

46
47     // Pecatenje nova lista
48     DLLNode<Integer> tmp1 = pomasna.getFirst();
49     while (tmp1 != null) {
50         System.out.print(tmp1.element);
51         if (tmp1.succ != null)
52             System.out.print(" ");
53         tmp1 = tmp1.succ;
54     }
55     System.out.println();
56 }
57
58 }
```

Задача 3. Преврти листа

Да се напише функција/метода во склоп на DLL класата што ги превртува сите врски во едностррано поврзана листа.

Решение

Решението е дадено во методата mirror() и употребата на истата е прикажана во main методата.

```

1  public void mirror() {
2
3      DLLNode<E> tmp = null;
4      DLLNode<E> current = first;
5      last = first;
6      while(current!=null) {
7          tmp = current.pred;
8          current.pred = current.succ;
9          current.succ = tmp;
10         current = current.pred;
11     }
12
13     if(tmp!=null && tmp.pred!=null) {
14         first=tmp.pred;
15     }
16 }
17
18 public static void main(String[] args) {
```

```

19     DLL<String> lista = new DLL<String>();
20     lista.insertLast("ovaa");
21     lista.insertLast("lista");
22     lista.insertLast("kje");
23     lista.insertLast("bide");
24     lista.insertLast("prevrtena");
25
26     System.out.println("Listata pred da bide prevrtena: ");
27     System.out.println(lista.toString());
28
29     lista.mirror();
30
31     System.out.println("Listata otkako e prevrtena: ");
32     System.out.println(lista.toString());
33 }
```

Излезот по извршувањето на овој код е:

```

Listata pred da bide prevrtena:
ovaa<->lista<->kje<->bide<->prevrtena
Listata otkako e prevrtena:
prevrtena<->bide<->kje<->lista<->ovaa
```

3.2.4 Задачи за вежбање

Задача 1. Подели според просек

Дадена е двострано поврзана листа чии што јазли содржат по еден природен број. Листата треба да се подели на две резултантни листи, т.ш. во првата листа треба да се сместат сите јазли кои содржат броеви помали или еднакви на просекот на листата, а во втората сите јазли кои содржат броеви поголеми од просекот на листата. Јазлите во резултантните листи се додаваат според обратен редослед од оној по кој по кој се појавуваат во дадената листа (т.е. прво се започнува со разгледување на последниот јазол од влезната листа и се додава во соодветната резултантна листа, па претпоследниот итн...).

Влез: Во првиот ред од влезот е даден бројот на јазли во листата, а во вториот ред се дадени броевите од кои се составени јазлите по редослед во листата.

Излез: Во првиот ред од излезот треба да се испечатат јазлите по редослед од првата резултантна листа (броеви помали или еднакви на просекот на листата), во вториот ред од втората (броеви поголеми од просекот на листата).

Пример.

Влез:

5

4 2 1 5 3

Излез:

3 1 2

5 4

Задача 2. Бриши подлисти

Дадени се две двојно поврзани листи чии што јазли содржат по една природен број. Од првата листа треба да се избришат сите појавувања на втората листа (појавување на една листа во друга значи првата листа да е подлиста на втората). Јазлите што ќе останат во првата листа треба да се прикажат на излез. Ако не остане ниту еден јазел се печати Prazna lista.

Влез: Во првиот ред од влезот е даден бројот на јазли на првата листа, потоа во вториот ред се дадени броевите од кои се составени јазлите по редослед во првата листа разделени со празно место. Во третиот ред е даден бројот на јазли на втората листа, а во четвртиот ред броевите од кои се составени јазлите по редослед во втората листа.

Излез: На излез треба да се испечатат јазлите по редослед во резултантната (првата) листа. Ако не остане ниту еден јазел се печати Prazna lista.

Пример.

Влез:

22

1 2 3 4 5 6 1 2 3 4 5 6 1 2 6 5 1 3 4 1 5 2

3

4 5 6 Излез:

1 2 3 1 2 3 1 2 6 5 1 3 4 1 5 2

Задача 3. Војска

Пред командантот на војската наредени се сите војници и во двојно поврзана листа дадени се нивните ID-а. На командантот не му се допаѓа како се наредени војниците и решава да одбере два под-интервали од војници и да им ги замени местата, односно војниците што се наоѓаат во едниот под-интервал ќе ги смести во другиот, и обратно.

Влез: Во првиот ред даден е бројот на војници.

Во вториот ред дадени се ID-то на секој од војниците.

Во третиот ред дадени се два броеви, ID на првиот војник и ID на последниот војник од првиот интервал.

Во четвртиот ред дадени се два броеви, ID на првиот војник и ID на последниот војник од вториот интервал.

Излез: Да се испечати новиот редослед на војниците (т.е. на нивните ID-а)

Забелешка 1: Интервалите никогаш нема да се преклопуваат и ќе содржат барем еден војник. Целата низа ќе содржи најмалку два војника.

Забелешка 2: Обратете посебно внимание кога интервалите се еден до друг и кога некој од интервалите почнува од првиот војник или завршува со последниот војник.

Пример.

Влез:

10

1 2 3 4 5 6 7 8 9 10

1 5

6 10

Излез:

6 7 8 9 10 1 2 3 4 5

3.3 LinkedList во Java

Класата **LinkedList** во Java моделира двојно поврзана листа. Класата употребува генерици за дефинирање на тип на елементи кои ќе ги содржи.

Постојат повеќе конструктора за **LinkedList**, од кои еден е **LinkedList()** кој креира празна двострана листа.

Сите операции се извршуваат како што може да се очекува за двојно поврзана листа. Постојат голем број на методои кои ги нуди класат, дел од нив се:

- **boolean add(E e)** - Се користи за додавање на наведениот елемент E на крајот од листата
- **void add(int index, E element)** - Се користи за вметнување на наведениот елемент E на дадената позиција index во листата
- **void addFirst(E e)** - Се користи за да се додаде елемент на почеток од листата
- **void addLast(E e)** - Се користи за да се додаде елемент на крајот од листата
- **void clear()** - Се користи за да се отстрanат сите елементи од листата
- **boolean contains(Object o)** - Враќа true ако листата го содржи дадениот елемент, во спротивно false
- **E element()** - Го враќа првиот елемент во листата
- **E getFirst()** - Го враќа првиот елемент во листата

- **E getLast()** - Го враќа последниот елемент во листата
 - **E get(int index)** - Го враќа елементот на позиција index
 - **int lastIndexOf(Object o)** - Се користи за враќање на индексот на последното појавување на наведениот елемент во оваа листа, или -1 ако низата не го содржи овој елемент
 - **int indexOf(Object o)** - Се користи за враќање на индексот на првото појавување на наведениот елемент во оваа листа, или -1 ако низата не го содржи овој елемент
 - **E removeFirst()** - Го отстранува првиот елементот во листата
 - **E removeLast()** - Го отстранува последниот елементот во листата
 - **E remove(int index)** - Го отстранува елементот кој се наоѓа на позиција index ви низата
 - **boolean remove(Object o)** - Го отстранува првото појавување на дадениот објект од низата
 - **int size()** - Го враќа бројот на елементи присутни во низата

Користење на дефинираните методи

Следниот код дефинира класа `LinkedListTester` во која во `main` методата се дефинира листа која содржи цели броеви (`integers`) во јазлите и ја прикажува употребата на дефинираните методи (исто како што ја тестираме претходно дефинираната класа `DLL`).

```
1 public class LinkedListTester {  
2  
3     public static void main(String[] args) {  
4         LinkedList<Integer> lista = new LinkedList<Integer>();  
5         lista.addLast(5);  
6         System.out.print("Listata po vmetnuvanje na 5 kako posleden element:  
7             ");  
8         System.out.println(lista.toString());  
9  
10        lista.addFirst(3);  
11        System.out.print("Listata po vmetnuvanje na 3 kako prv element: ");  
12        System.out.println(lista.toString());  
13  
14        lista.addLast(1);  
15        System.out.print("Listata po vmetnuvanje na 1 kako posleden element:  
16            ");  
17        System.out.println(lista.toString());
```

```

17     lista.removeFirst();
18     System.out.print("Listata po brishenje na prviot element: ");
19     System.out.println(lista.toString());
20
21     //DLLNode<Integer> pom = lista.find(5);
22     int indeksNa5 = lista.indexOf(5);
23     lista.add(indeksNa5, 2);
24     System.out.print("Listata po vmetnuvanje na elementot 2 pred
25         elementot 5: ");
26     System.out.println(lista.toString());
27
28     int indeksNa1 = lista.indexOf(1);
29     lista.add(indeksNa1+1, 3);
30     System.out.print("Listata po vmetnuvanje na elementot 3 posle
31         elementot 1: ");
32     System.out.println(lista.toString());
33
34     int indeksNa3 = lista.indexOf(3);
35     lista.remove(indeksNa3);
36     System.out.print("Listata po brishenje na elementot 3: ");
37     System.out.println(lista.toString());
38
39     System.out.println("Momentalna dolzina na listata: "+lista.size());
40
41     lista.clear();
42     System.out.print("Pecatenje na listata po nejzino brishenje: ");
43     System.out.println(lista.toString());
44     System.out.println("Momentalna dolzina na listata: "+lista.size());
}
}

```

Излезот по извршувањето на овој код е:

```

Listata po vmetnuvanje na 5 kako posleden element: [5]
Listata po vmetnuvanje na 3 kako prv element: [3, 5]
Listata po vmetnuvanje na 1 kako posleden element: [3, 5, 1]
Listata po brishenje na prviot element: [5, 1]
Listata po vmetnuvanje na elementot 2 pred elementot 5: [2, 5, 1]
Listata po vmetnuvanje na elementot 3 posle elementot 1: [2, 5, 1, 3]
Listata po brishenje na elementot 3: [2, 5, 1]
Momentalna dolzina na listata: 3
Pecatenje na listata po nejzino brishenje: []

```

Momentalna dolzina na listata: 0

Глава 4

Еднодимензионални податочни структури

4.1 Апстрактни податочни типови

Пред да започнеме со проучувањето на еднодимензионалните податочни типови, да се осврнеме на поимот апстрактен податочен тип. За разлика од низите и листите, за кои велиме дека се фундаментални податочни типови, се појавува потреба од дефинирање на нови податочни типови за одредени намени. Кај фундаменталните типови, фокусот беше ставен на нивната имплементација. На пример, низата беше описана како последователни мемориски локации каде се чуваат податоците. Кај апстрактните податочни типови, фокусот не е на имплементацијата. Токму затоа и се викаат апстрактни, бидејќи во нивното дефинирање не се грижиме за нивната имплементација (ја апстрахираме), туку само за тоа какви податоци треба да зачуваме и како истите да ги обработуваме. За апстрактните податочни типови карактеристично е [1]:

- Специфицирање на множеството вредности за податочниот тип
- Специфицирање на можните операции врз тоа множество вредности

При тоа, воопшто не се наведува ништо за имплементацијата на податочниот тип.

За секој апстрактен податочен тип, всушност се прави одреден “договор”. Овој договор ги содржи главните карактеристики на типот: множеството податоци и операции со тие податоци. Во него се нагласува какви ќе бидат податоците и какво е посакуваното однесување од операциите дефинирани над тие податоци.

Овој концепт на апстрактни податочни типови овозможува јасно раздвојување на двете основни компоненти: спецификација и имплементација на податочниот тип.

Спецификацијата претставува опис на можните вредности, како и опис на операциите над нив (нивното посакувано однесување). Имплементацијата пак ги содржи деталите со претставувањето на податоците (со некоја од фундаменталните структури или преку други апстрактни типови), како и деталите поврзани со алгоритмите кои се користат за имплементирање на операциите врз тие податоци. Ваквата поделба овозможува постоење на алтернативни имплементации за еден ист договор. При тоа, доколку различните имплементации стриктно го почитуваат договорот, можно е нивно наизменично користење, во зависност од потребата и примената, остварувајќи при тоа баланс помеѓу користење на ресурси (на пример меморија) и перформанси. Најважна придобивка од апстракцијата е раздвојувањето на одговорностите. Оној кој бара одредена податочна структура да биде изработена за да може да ја користи во своите апликации, се што треба е добро да ја специфицира. Оној пак кој имплементира одредена структура, треба само да го испочитува постигнатиот договор. Јасното одвојување на одговорностите е особено важно кога станува збор за големи проекти и комплексни апликации.

4.2 Стек (stack, магацин)

Стекот претставува еднодимензионална линеарна секвенца од елементи која работи по принципот последен-внесен-прв-изведен (last-in-first-out). Ова значи дека додавањето на нови елементи и нивното вадење од линеарната секвенца се врши само од еден крај. Овој крај на структурата се нарекува врв на стекот. Аналогијата се прави со стог од сено, каде сеното се става и вади само на горната страна – на врвот на стогот.

Под длабочина на стек се подразбира бројот на елементи што се складирани во него. Празниот стек има длабочина 0. На слика 4-1 е даден еден пример на стек од книги. Под претпоставка дека се работи за тешки книги, додавањето и вадењето на книгите е можно само на горната страна - врвот на стекот.



Слика 4-1: Стек од книги.

4.2.1 Операции со стек

Основните операции за стек се:

- Празнење на целиот стек
- Проверка дали стекот е празен
- Додавање на елемент на врвот на стекот (операција "push")
- Вадење на елемент од врвот на стекот (операција "pop")
- Дополнително, проверка на првиот елемент во стекот без негово вадење (операција "peek").

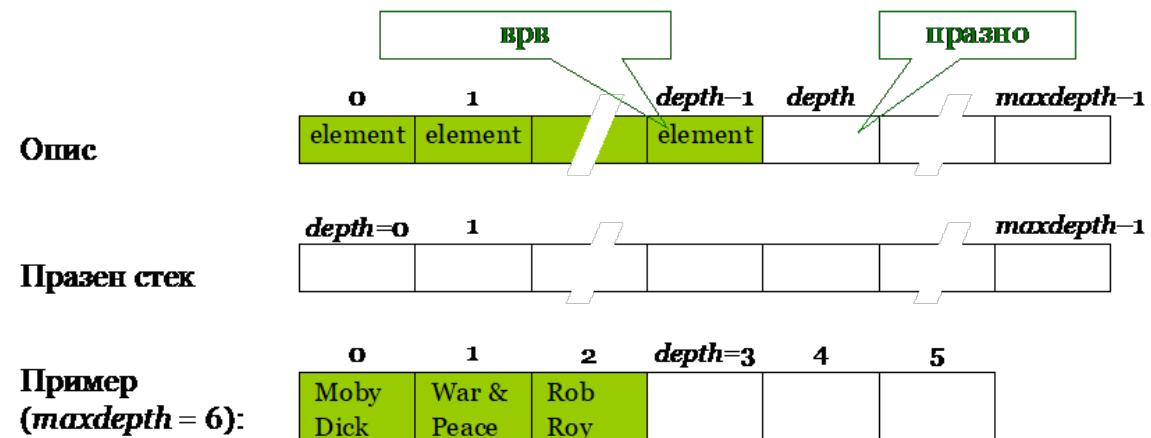
За стек ќе дадеме 2 имплементации, со двата различни фундаментални податочни типови: низа и поврзана листа.

4.2.2 Имплементација на стек со низа

Имплементацијата на стек со низа преставува имплементација на ограничен стек, чија максимална длабочина може да биде еднаква на големината на низата со која истиот се имплементира.

Имплементацијата содржи променлива *depth* која ја содржи тековната длабочина на стекот, како и низа од елементи *elems* со должина *maxdepth* во која се содржат елементите на позициите *elems[0..depth-1]*.

На слика 4-2 е даден изгледот на оваа имплементација, како и пример за празен стек и стек од неколку елементи.



Слика 4-2: Имплементација на стек со низа.

За да се моделира стек прво ќе користиме Java интерфејсна дефиниција за интерфејс на стек (Stack) структура на податоци. Во овој интерфејс, Е претставува параметризиран тип (генерички) кој може да биде било кој тип. Методите

во интерфејсот се: isEmpty() кој проверува дали стекот е празен; peek() кој го враќа елементот што се наоѓа на врвот (последниот додаден елемент) на стекот без да го отстранува; clear() кој го празни стекот; push(E x) кој го додава објектот x на врвот на стекот; и pop() кој го отстранува и враќа елементот што се наоѓа на врвот на стекот. Интерфејсот Stack ја дефинира основната функционалност на стек структурата на податоци и може да се имплементира во класи кои ги користат стековите во нивните програми.

```

1 public interface Stack<E> {
2     // Елементи на стекот се објекти од произволен тип.
3     // Методи за пристап:
4     public boolean isEmpty();
5     // Враќа true ако и само ако стекот е празен.
6
7     public E peek();
8     // Го враќа елементот на врвот од стекот.
9
10    // Методи за трансформација:
11    public void clear();
12    // Го празни стекот.
13
14    public void push(E x);
15    // Го додава x на врвот на стекот.
16
17    public E pop();
18    // Го отстранува и враќа елементот што е на врв на стекот.
19 }
```

Следната генеричка Java класа ArrayStack која го имплементира интерфејсот Stack<E> представува стек која користи низа за складирање на елементите. Оваа класа користи две приватни променливи за складирање на податоците: elems е низа за складирање на елементите на стекот, додека depth го чува бројот на моментално зачуваните елементи во стекот. Конструкторот на ArrayStack креира нов празен стек со максимална длабочина (максимален број на елементи). Методите од интерфејсот за стек се препокриени во оваа класа: isEmpty() кој го проверува стекот дали е празен, враќа true ако е празен, инаку false; peek() кој го враќа елементот на врвот од стекот без да го отстранува; clear() кој го празни стекот со поставување на сите елементи на null и поставува depth на 0; push(E x) кој го додава елементот x на врвот на стекот и го зголемува depth за 1; size() кој го враќа бројот на елементи на стекот (длабочината); и pop() кој го отстранува и враќа елементот на врвот од стекот и со отстранувањето, го поставува елементот

на врвот на null и го намалува depth за 1, а потоа го враќа отстранетиот елемент.

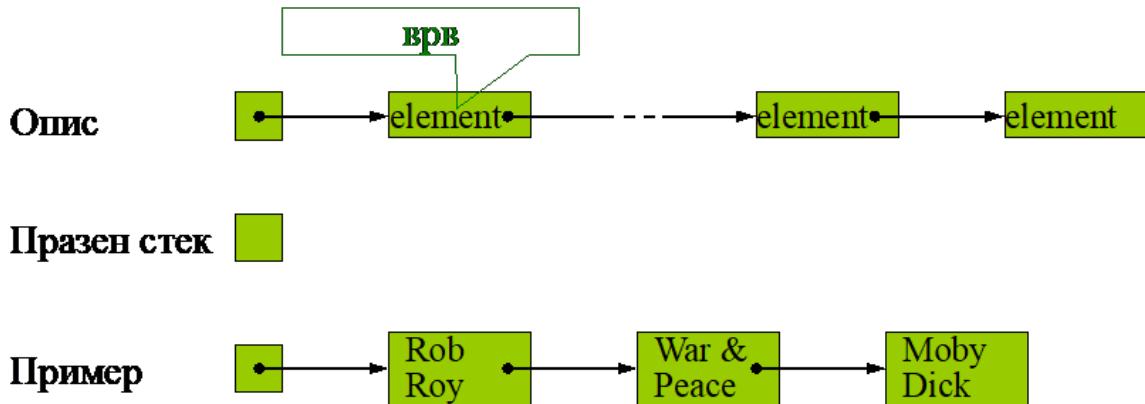
```
1 import java.util.NoSuchElementException;
2
3 public class ArrayStack<E> implements Stack<E> {
4     private E[] elems; // elems[0...depth-1] се неговите елементи.
5     private int depth; // depth е длабочината на стекот.
6
7     @SuppressWarnings("unchecked")
8     public ArrayStack(int maxDepth) {
9         // Конструкција на нов, празен стек.
10        elems = (E[]) new Object[maxDepth];
11        depth = 0;
12    }
13
14    public boolean isEmpty() {
15        // Враќа true ако и само ако стекот е празен.
16        return (depth == 0);
17    }
18
19    public E peek() {
20        // Го враќа елементот на врвот од стекот.
21        if (depth == 0)
22            throw new NoSuchElementException();
23        return elems[depth - 1];
24    }
25
26    public void clear() {
27        // Го празни стекот.
28        for (int i = 0; i < depth; i++) elems[i] = null;
29        depth = 0;
30    }
31
32    public void push(E x) {
33        // Го додава x на врвот на стекот.
34        elems[depth++] = x;
35    }
36
37    public int size() {
38        // Ја враќа должината на стекот.
39        return depth;
40    }
```

```

41
42     public E pop() {
43         // Го отстранува и враќа елементот што е на врвот на стекот.
44         if (depth == 0)
45             throw new NoSuchElementException();
46         E topmost = elems[--depth];
47         elems[depth] = null;
48         return topmost;
49     }
50 }
```

4.2.3 Имплементација на стек со листа

Алтернативна имплементација на стекот е со помош на поврзана листа. Оваа имплементација претставува неограничен стек, каде првиот елемент од листата го претставува врвот на стекот (односно сите операции се извршуваат на почетокот на листата). На слика 4-3 е претставена имплементацијата со едностррано поврзана листа.



Слика 4-3: Имплементација на стек со листа.

Следната генеричка Java класа LinkedStack која го имплементира интерфејсот Stack<E> представува стек која користи едностррано поврзана листа за складирање на елементите. Оваа класа користи две приватни променливи за складирање на податоците. top (од тип SLLNode<E>) е референца кон првиот јазол од едностррано поврзаната листа која ги содржи елементите на стекот, додека size го чува бројот на моментално зачуваните елементи во стекот. Конструкторот на LinkedStack креира нов празен стек со поставување на top на null и size на 0, што значи дека стекот е празен при иницијализација. toString() методот се користи за создавање текстуална репрезентација на стекот, каде секој елемент е разделен

со празно место. Другите методи кои се препокриени од интерфесот за стек во оваа имплементација на стек со листа се: `isEmpty()` кој го проверува стекот дали е празен; `clear()` кој го празни стекот со поставување на `top` на `null` и `size` на 0; `peek()` кој го враќа елементот на врвот од стекот без да го отстранива; `push(E x)` кој го додава елементот `x` на врвот на стекот со креирање нов јазол и поставување на `top` да реферира до новиот јазол, а потоа зголемува `size` за 1; `size()` го враќа бројот на елементи на стекот (длабочината); и `pop()` кој го отстранива и враќа елементот на врвот од стекот, а по отстранивањето, го намалува `size` за 1 и го поставува `top` да реферира до следниот јазол во листата.

```
1 import java.util.NoSuchElementException;
2
3 public class LinkedStack<E> implements Stack<E> {
4     // top е линк до првиот јазол од еднострено поврзаната листа која ги
5     // содржи елементите на стекот.
6     private SLLNode<E> top;
7     int size;
8
9     public LinkedStack() {
10         // Конструкција на нов, празен стек.
11         top = null;
12         size = 0;
13     }
14
15     public String toString() {
16         // Прави текстуална репрезентација на стекот.
17         SLLNode<E> current = top;
18         StringBuilder s = new StringBuilder();
19         while (current != null) {
20             s.append(current.element);
21             s.append(" ");
22             current = current.succ;
23         }
24         return s.toString();
25     }
26
27     public boolean isEmpty() {
28         // Враќа true ако и само ако стекот е празен.
29         return (top == null);
30     }
31
32     public void clear() {
```

```

32         // Го празни стекот.
33         top = null;
34         size = 0;
35     }
36
37     public E peek() {
38         // Го враќа елементот на врвот на стекот.
39         if (top == null)
40             throw new NoSuchElementException();
41         return top.element;
42     }
43
44     public void push(E x) {
45         // Го додава x на врвот на стекот.
46         top = new SLLNode<E>(x, top);
47         size++;
48     }
49
50     public int size() {
51         // Ја враќа должината на стекот.
52         return size;
53     }
54
55     public E pop() {
56         // Го отстранува и враќа елементот што е на врвот на стекот.
57         if (top == null)
58             throw new NoSuchElementException();
59         E topElem = top.element;
60         size--;
61         top = top.succ;
62         return topElem;
63     }
64
65 }
```

4.2.4 Stack во Java

Во Јава постои класата `Stack` која претставува LIFO стек на објекти и може да се користи во проблеми за готова имплементација на стек. Ја наследува и проширува Java класата `Vector` со пет операции кои дозволуваат вектор да биде третиран како стек:

```
1 public class Stack<E> extends Vector<E>
```

Методите на оваа класа се:

- empty() кој враќа: true ако и само ако стекот не содржи елементи, false во останатите случаи;
- peek() кој враќа: објектот кој е најгоре на стекот (последниот елемент од Vector објектот), без притоа да го вади од структурата;
- pop() кој враќа: објектот кој е најгоре на стекот (последниот елемент од Vector објектот и притоа го брише елементот);
- push(E item) кој го додава item влезниот аргумент;
- search(Object o) кој враќа: растојанието до првото појавување на o од почетокот на стекот; -1 доколку објектот не е во стекот.

4.2.5 Едноставни проблеми со стек

Постојат голем број практични примери на стек. Тесен магацин, со ширина еднаква на товарот кој во него се складира, и кој има еден влез, преставува стек. Во него, последно внесениот товар треба да се извади прв. Потоа, стекот има особено голема примена во самите компјутерски системи, бидејќи со негова помош е возможно реализирање на концептот на подпроцедури, нивно повикување и враќање. Други примери за употреба на стек се во backtracking алгоритми, back/forward функционалности во прелистувачи, undo/redo функционалности, превртување на поврзана листа, евалуација на аритметички изрази, проверка на коректност на загради во израз, парсери, имплицитно во рекурзија, повици на функции итн.

Следат неколку примери за корисноста од овој податочен тип.

Задача 1. Загради

Да се провери коректноста на заградите во еден израз. Еден израз има коректни загради ако:

- За секоја лева заграда, подоцна следува соодветна десна заграда - За секоја десна заграда претходно постои лева заграда

- Секој под-израз меѓу пар од две загради содржи коректен број на загради.

Примери на изрази со коректни и некоректни загради: $s \times (s - a) \times (s - b) \times (s - c)$ коректни

$(- b + \sqrt{b^2 - 4ac}) / 2a$ коректни

$s \times (s - a) \times (s - b) \times (s - c)$ некоректни

$s \times (s - a) \times s - b) \times (s - c)$ некоректни

$(-b + \sqrt{b^2 - 4ac}) / 2a$ некоректни

Влез: Во влезот е даден изразот кој се внесува.

Излез: На излез треба да се испечати дали заградите во изразот се коректни или не.

Пример:

Влез: $s \times (s - a) \times (s - b) \times (s - c)$

Излез: $s \times (s - a) \times (s - b) \times (s - c)$ има коректни загради.

Решение

Во дадената задача треба да се провери коректноста на заградите во даден израз. За решение мора да се избере соодветна податочна структура. Тука мора да се користи стек за да се одржи редоследот и вгнездувањето на заградите во изразите и за да се овозможи проверка дали заградите се коректно балансираны. Еве зошто тука се користи стек и како помага во овој контекст:

- Проверка на парови на загради: Заградите '()', '[]' и '{}', мораат да се спарени во парови за изразот да се смета за валиден. Стекот е природен избор за чување на овие парови. Кога се среќава отворачка заграда, таа се става на стекот, а кога се среќава затворачка заграда, се споредува со заградата на врвот на стекот за да се обезбеди дека се спарени.
- LIFO принципот: Оваа својство е важно за проверката на точноста на паровите на загради бидејќи сакаме најскоро отворената заграда да ја спариме со следната затворачка заграда.
- Вгнездени изрази со загради: Стековите можат да се справат и со вгнездени изрази со загради. На пример, ако имате израз како $(a + (b - c))$, стекот може правилно да утврди дека внатрешниот пар на загради $(b - c)$ е внесен во надворешниот пар $(a + ...)$. Внатрешните загради се обработуваат и се отстрануваат од стекот пред надворешните загради.
- Ефикасност: Стековите обезбедуваат ефикасен начин за следење на паровите на загради бидејќи имаат операции за додавање и отстранување на елементи со константна временска сложеност (push и pop операции).

Имплементацијата на решението е дадена подолу. Во оваа имплементација може да се користат две различни имплементации на стек: ArrayStack или LinkedStack (во коментар дадено). Изборот на која имплементација на стек да се користи зависи од специфичните барања на проблемот.

Во решението влезниот израз се обработува користејќи го методот `daliKorektni`, како и методот `daliSoodvetni` за проверка дали два знаци се соодветни загради. Методот `daliKorektni` користи ArrayStack (или LinkedStack) за

проверка на коректноста на заградите во изразот phrase. Го итерира секој знак од изразот и, ако знакот е отворачка заграда ((, [или {}, го става во стекот bracketStack. Ако знакот е затворачка заграда (),] или }), тогаш го споредува со последниот елемент (врвот) во стекот. Ако се совпаѓат, ги отстранува од стекот. Ако на крајот на итерацијата стекот е празен, тоа значи дека сите загради се коректно поставени и методот враќа true. Во спротивно, враќа false. Дополнително се користи методот daliSoodvetni кој споредува два знаци (left и right) за да провери дали се соодветни загради. На пример, ако left е отворачка заграда (() и right е затворачка заграда ()) , методата враќа true. Во секој друг случај, враќа false. Резултатот (дали заградите се коректни или не) се печати на излез во главниот main метод.

```

1 //Вметни класа SLLNode
2 //Вметни класа Stack
3 //Вметни класа ArrayStack
4 //Вметни класа LinkedStack
5
6 public class Zagradi {
7     public static boolean daliKorektni(String phrase) {
8         // Test whether phrase is well-bracketed.
9         ArrayStack<Character> bracketStack = new ArrayStack<Character>(100);
10        // LinkedStack<Character> bracketStack = new LinkedStack<>();// за
11        // решение со LinkedStack откоментирајте го овој ред
12
13        for (int i = 0; i < phrase.length(); i++) {
14            char cur = phrase.charAt(i);
15            if (cur == '(' || cur == '[' || cur == '{')
16                bracketStack.push(cur);
17
18            else if (cur == ')' || cur == ']' || cur == '}') {
19                if (bracketStack.isEmpty()) return false;
20
21                char left = bracketStack.pop();
22                if (!daliSoodvetni(left, cur)) return false;
23
24            }
25
26        }
27
28        public static boolean daliSoodvetni(char left, char right) {
29            // Провери дали left и right се совпаѓачки загради

```

```

30     // (со претпоставка дека left е отворачка заграда и right е
31     // затворачка заграда).
32     switch (left) {
33         case '(':
34             return (right == ')');
35         case '[':
36             return (right == ']');
37         case '{':
38             return (right == '}');
39     }
40     return false;
41 }
42 public static void main(String[] args) {
43     String phrase = "s x (s - a) x (s - b) x (s - c)";
44 //     String phrase = "s x (s - a) x s - b) x (s - c)";
45     System.out.println(phrase + " ima "
46         + (daliZagraditeSePravilni(phrase) ? "korektni" :
47             "nekorektni") + " zagradi.");
48 }
```

Задача 2. Постфикс

Да се напише алгоритам кој ќе врши евалуација на израз во постфикс нотација. Пример $5 \ 9 \ + \ 2 \ * \ 6 \ 5 \ * \ +$ изразот е во постфикс нотација, и го претставува изразот $(5 + 9) * 2 + 6 * 5$, со што по евалуацијата резултатот треба да биде $14 * 2 + 30 = 58$.

Влез: Во влезот е даден изразот кој се внесува.

Излез: На излез треба да се испечати резултатот од евалуацијата на изразот.

Пример:

Влез: $5 \ 9 \ + \ 2 \ * \ 6 \ 5 \ * \ +$

Излез: Резултатот е 58.0

Решение

Во дадената задача треба да се евалуира израз во постфикс нотација. Во постфикс нотацијата, операторите следат по operandите. Ова значи дека не е потребна употреба на загради или оценување на приоритет на операции. Стекот е идеална структура за да се чуваат и обработуваат operandите и operatorите во овој редослед.

Имплементацијата на решението е дадена подолу. Во оваа имплементација се користи LinkedStack за чување на броевите и пресметките. evaluiraj_postfix е методот кој го прима изразот во постфикс нотација како влезен аргумент и ја враќа евалуацијата на тој израз како Double вредност. Се итерира низ секој знак од влезниот израз. Ако знакот е празно место, се игнорира и продолжува со следниот знак., додека, ако знакот е цифра (број), тој се конвертира во Double и се става на стекот. Ако пак, знакот е оператор (+, *, x, -, /), се извршува соодветната операција врз последните два броеви на стекот (претпоследниот и последниот број). Резултатот се става на стекот. Пример: за изразот '5 9 +', на стекот прво ќе биде 5.0, потоа 9.0, и накрај резултатот од собирањето 14.0. Ако настане грешка, како на пример, ако има недостаток на оператори или недостасува операнд, може да се испечати соодветна порака за грешка. На крајот, ако на стекот има само еден елемент, тоа е резултатот од евалуацијата и се враќа како Double вредност. Резултатот од примерен израз во постфикс нотација ('5 9 + 2 * 6 5 * +') со повикување на evaluiraj_postfix методот се печати на излез во главниот main метод.

```

1 //Вметни класа SLLNode
2 //Вметни класа Stack
3 //Вметни класа LinkedStack
4
5 public class CalcPostfix {
6     public static Double evaluiraj_postfix(String izraz) {
7         LinkedStack<Double> stack = new LinkedStack<>();
8         Double r = null;
9         for (int i = 0; i < izraz.length(); i++) {
10             char c = izraz.charAt(i);
11             if (c == ' ') continue;
12             else if (Character.isDigit(c)) {
13                 // stack.push(c);
14                 stack.push(Character.digit(c, 10));
15                 stack.push((double) c - '0');
16             } else {
17                 // 6 2 - => 6-2
18                 // 5 2 / => 2
19                 // System.out.println("Pred promenata (top first):" + stack +
20                 // " operator:" + c);
21                 if (stack.size() >= 2) {
22                     Double posleden_broj = stack.pop();
23                     Double pretposleden_broj = stack.pop();
24                     switch (c) {

```

```

24         case '+':
25             stack.push(pretposleden_broj + posleden_broj);
26             break;
27         case '*':
28             stack.push(pretposleden_broj * posleden_broj);
29             break;
30         case 'x':
31             stack.push(pretposleden_broj * posleden_broj);
32             break;
33         case '-':
34             stack.push(pretposleden_broj - posleden_broj);
35             break;
36         case '/':
37             stack.push(pretposleden_broj / posleden_broj);
38             break;
39     }
40     // System.out.println("Po promenata (top first):" +
41     // stack);
42 } else {
43     System.out.println("Nevaliden vlez - nedostasuva operand
44     na pozicija:" + i);
45     return r;
46 }
47 if (stack.size() != 1) {
48     System.out.println("Nevaliden vlez - nedostasuva operator");
49 } else {
50     r = stack.pop();
51 }
52 return r;
53 }

54 public static void main(String[] args) {
55     String primer = "5 9 + 2 * 6 5 * +";
56     System.out.println("Rezultatot e " + evaluiraj_postfix(primer));
57 }
58 }
59 }
```

Задача 3. Нова структура

Да се направи имплементација за нова податочна структура QuasiStack која што ќе биде неограничена структура. Новата структура треба да може да ги задоволи следните операции:

- вметнување елемент – Вметнувањето елемент секогаш е вметнување на врвот на податочната структура. Оваа операција треба да се имплементира со сложеност $O(1)$.
- вадење елемент – Вадењето елемент од структурата може да биде вадење на елемент од врвот на структурата или од дното. Треба секогаш да се извади поголемиот елемент, при споредба на елементите на врвот и дното. Доколку се еднакви го вади елементот од дното. Оваа операција треба да се имплементира со сложеност $O(1)$.
- да се провери дали структурата е празна
- да се провери кој е елементот на врв на структурата
- да се провери кој е елементот на дното на структурата

Елементите се внесуваат во структурата во истиот редослед како што се читат од стандардниот влез. Имате право на користење на само една ваква структура. Немате право на користење други дополнителни структури. Ваша задача е да го испечатите редоследот на вадење на сите елементи од структурата QuasiStack.

Влез: Во првиот ред од влезот е даден бројот на елементи кои ќе се внесуваат, а во секој нов ред се дадени елементите кои се читаат.

Излез: Прво се печати елементот на врвот на структурата, а после овој на дното. Потоа во нов ред се печатат сите елементи од структурата QuasiStack во редослед како што се вадат од неа, одделени со празно место.

Пример:

Влез:

8 (број на елементи кои треба да се прочитаат, се ставаат на структурата последователно онака како што се читаат)

6 (елемент 1)
5 (елемент 2 итн.)
4
1
0
5
2
9

Излез:

9 (елементот на врвот на структурата)
 6 (елементот на дното на структурата)
 9 6 5 4 2 5 1 0 (сите елементи од структурата QuasiStack)

Решение

Во дадената задача треба да се имплементира нова податочна неограничена структура QuasiStack. За да се одговори на барањата од задачата мора да се искористи и трансформира структура стек имплементирана со листа, но тута бидејќи мора да се знаат и врвот и дното на стекот ќе користиме имплементација со двојно поврзана листа. Структура QuasiStack е слична на обичниот стек, но дополнително го чува и најголемиот елемент на стекот (врвот и дното на стекот). Вредностите кои ги чува мораат да бидат компарабилни (имплементираат интерфејсот Comparable) за да се споредуваат.

Имплементацијата на решението е дадена подолу. Во оваа имплементација во класата QuasiStack за да се задоволат барањата на задачата се дадени: конструктор за иницијализација на празен квази стек; isEmpty() метод кој враќа true ако и само ако квази стекот е празен; peekTop() метод кој го враќа елементот на врвот на квази стекот; peekBottom() метод кој го враќа елементот на дното на квази стекот; clear() метод кој го празни квази стекот; push(E x) метод кој додава елемент x на врвот на квази стекот (ако квази стекот е празен, x станува и дно и врв на стекот); и pop() кој го отстранува и враќа елементот што е на врвот и дното на квази стекот, споредувајќи ги најголемиот и најмалиот елемент и ги отстранува соодветно. Резултатот од примерен влез се печати на излез во главниот main метод.

```

1 import java.util.NoSuchElementException;
2 import java.io.BufferedReader;
3 import java.io.IOException;
4 import java.io.InputStreamReader;
5
6 //Вметни класа DLLNode
7 //Вметни класа Stack
8
9 class QuasiStack<E extends Comparable<E>> implements Stack<E> {
10     //Stekot e pretstaven na sledniot nacin: top e link do prviot jazol
11     // na ednostrano-povrzanata lista koja sodrzi gi elementite na stekot .
12     private DLLNode<E> top, bottom;
13
14     public QuasiStack () {
15         // Konstrukcija na nov, prazen stek.

```

```
16         top = null;
17         bottom = null;
18     }
19
20     public boolean isEmpty () {
21         // Vrakja true ako i samo ako stekot e prazen.
22         return (top == null);
23     }
24
25     public E peekTop () {
26         // Go vrakja elementot na vrvot od stekot.
27         if (top == null)
28             throw new NoSuchElementException();
29         return top.element;
30     }
31
32     public E peekBottom () {
33         // Go vrakja elementot na vrvot od stekot.
34         if (bottom == null)
35             throw new NoSuchElementException();
36         return bottom.element;
37     }
38
39     public void clear () {
40         // Go prazni stekot.
41         top = null;
42         bottom = null;
43     }
44
45     public void push (E x) {
46         // Go dodava x na vrvot na stekot.
47         DLLNode<E> ins = new DLLNode<E>(x, null, top);
48         if (top == null)
49             bottom = ins;
50         else
51             top.pred = ins;
52         top = ins;
53     }
54
55     public E pop () {
56         // Go otstranuva i vrakja pogolemiot element shto e na vrvot i dnoto
```

```
    na stekot.  
57     if (top == null)  
58         throw new NoSuchElementException();  
59     E topElem = top.element;  
60     E bottomElem = bottom.element;  
61  
62     if (top == bottom){  
63         top = null;  
64         bottom = null;  
65         return topElem;  
66     }  
67  
68     if(topElem.compareTo(bottomElem) == 1){  
69         top = top.succ;  
70         top.pred = null;  
71         return topElem;  
72     }  
73     else{  
74         bottom = bottom.pred;  
75         bottom.succ = null;  
76         return bottomElem;  
77     }  
78 }  
79 }  
80  
81 public class TestQuasiStack {  
82     public static void main(String[] args) throws IOException {  
83         // TODO Auto-generated method stub  
84         BufferedReader br = new BufferedReader(new  
85             InputStreamReader(System.in));  
86  
87         QuasiStack<Integer> qs = new QuasiStack<Integer>();  
88  
89         int brElementi = Integer.parseInt(br.readLine());  
90  
91         for(int i=0; i<brElementi; i++){  
92             int x = Integer.parseInt(br.readLine());  
93             qs.push(x);  
94         }  
95         System.out.println(qs.peekTop());
```

```

96     System.out.println(qs.peekBottom());
97
98     while(!qs.isEmpty()){
99         System.out.print(qs.pop()+" ");
100    }
101 }
102 }
```

4.2.6 Напредни проблеми со стек

Задача 1. Поништување топчиња

Да се напише алгоритам со кој ќе се имплементира играта “Поништување топчиња”. Во оваа игра на располагање имате топчиња во три различни бои (R-црвена, G-зелена и B-сина), обележани со знакот + или -. Поништување на топчиња може да настане само доколку тие се од иста боја и со спротивен знак. На почеток се генерира една случајна листа со топчиња. Ваша задача е од тој влез, како доаѓаат топчињата да направите поништување и да кажете колку, од каков тип (+ или -) и од која боја фалат за да се поништат сите топчиња од влезот.

Влез: Во влезот е дадена листа од случајни топчиња и тоа во облик: боја, знак.

Излез: На излез треба да се испечатат бројот на парови и паровите кои може да се формираат.

Пример:

Влез: R+ G- G+ G+ R+ B- B+ R- G+ R- B- B+ B+ R+

Парови кои може да се формираат од овој список се: (R+, R-); (B+, B-); (B- B+); (R+, R-); (G-, G+); (R+, R-) Остануваат без партнери: G+, G+, B+, R+

Излез:

4

R- G- G- B+

Решение

Во дадената задача треба да се имплементира игра со топчиња во три различни бои (R-црвена, G-зелена и B-сина), обележани со знакот + или -, каде се прави поништување на топчињата само доколку тие се од иста боја и со спротивен знак. За да се одговори на барањата од задачата мора да се искористат 3 податочни структури стек (crveni, zeleni и sini) за да се чуваат топчињата со соодветните бои (може било која имплементација на стек да се искористи). Од влезната листа кој содржи топчиња (topcinja) се поминува низ секое топче во листата и се прави

следново: ако топчето е црвено ('R'), се проверува дали стекот crveni е празен или не. Ако не е празен и горното топче на стекот е исто со тековното, тогаш се додава на стекот. Во спротивно, се отстранува од стекот и се игнорира. Истото се применува и за зелените ('G') и сините ('B') топчиња. По обработката на сите топчиња треба да се отстранат топчињата од стековите во низа која ги содржи отстранетите топчиња како текст во формат "Боја Знак" каде знакот е + или -, и се печати бројот на отстанети топчиња.

Имплементацијата на решението е дадена подолу. Резултатот од примерен влез се печати на излез во главниот main метод.

```

1 import java.util.NoSuchElementException;
2 import java.io.BufferedReader;
3 import java.io.IOException;
4 import java.io.InputStreamReader;
5 import java.util.LinkedList;
6 import java.util.List;
7
8 //Вметни класа Stack
9 //Вметни класа ArrayStack
10
11 public class Topcinja {
12     public static String ponistiTopcinja (List l) {
13         ArrayStack<String> crveni = new ArrayStack<String>(100);
14         ArrayStack<String> zeleni = new ArrayStack<String>(100);
15         ArrayStack<String> sini = new ArrayStack<String>(100);
16         String s;
17         String izlez = new String();
18         int n = 0;
19
20         for (int i = 0; i < l.size(); i++) {
21             s = (String) l.get(i);
22
23             if (s.charAt(0) == 'R') {
24                 if (!crveni.isEmpty())
25                     if (crveni.peek().equals(s))
26                         crveni.push(s);
27                 else
28                     crveni.pop();
29             else
30                 crveni.push(s);
31         }
32

```

```

33     if (s.charAt(0) == 'G') {
34         if(!zeleni.isEmpty())
35             if(zeleni.peek().equals(s))
36                 zeleni.push(s);
37             else
38                 zeleni.pop();
39             else
40                 zeleni.push(s);
41     }
42
43     if (s.charAt(0) == 'B') {
44         if(!sini.isEmpty())
45             if(sini.peek().equals(s))
46                 sini.push(s);
47             else
48                 sini.pop();
49             else
50                 sini.push(s);
51     }
52 }
53 while(!crveni.isEmpty()){
54     n++;
55     if(crveni.pop().charAt(1)=='+')
56         izlez+="R- ";
57     else
58         izlez+="R+ ";
59 }
60 while(!zeleni.isEmpty()){
61     n++;
62     if(zeleni.pop().charAt(1)=='+')
63         izlez+="G- ";
64     else
65         izlez+="G+ ";
66 }
67 while(!sini.isEmpty()){
68     n++;
69     if(sini.pop().charAt(1)=='+')
70         izlez+="B- ";
71     else
72         izlez+="B+ ";
73 }

```

```

74
75     System.out.println(n);
76
77     return izlez;
78 }
79
80 public static void main (String[] args) throws IOException {
81
82     BufferedReader br = new BufferedReader(new
83         InputStreamReader(System.in));
84     String vlez[] = new String[100];
85     vlez = br.readLine().split(" ");
86
87     List<String> topcinja = new LinkedList<String>();
88     for(int i=0; i<vlez.length; i++)
89         topcinja.add(vlez[i]);
90
91     System.out.println(ponistiTopcinja(topcinja));
92 }
93 }
```

Задача 2. Молекула на вода

Да се напише алгоритам со кој ќе се имплементира играта “Направи молекула на вода”. Во оваа игра на располагање имате два типа атоми (Н-водород, и О-кислород). За да се направи молекула на вода (H_2O) потребно е да имате два атоми на водород и еден атом на кислород. На почеток се генерира една случајна секвенца од атоми. Ваша задача е од тој влез, како доаѓаат атомите да генерирате молекули и да кажете колку такви молекули се креирале, и кои атоми останале несврзани.

Влез: Во влезот е дадена секвенца од случајни атоми

Излез: На излез треба да се испечати бројот на молекули H_2O , и несврзаните атоми од водород и кислород.

Пример:

Влез:

Н Н О Н Н О Н Н О Н Н Н Н Н Н Н Н Н Н

Излез:

8

Н

О

Решение

Во дадената задача треба да се имплементира обработка на атоми и креирање на молекули, притоа се симулира спојување на водородни и кислородни атоми за формирање вода (H_2O) молекули. За да се одговори на барањата од задачата мора да се искористат два стека: vodorod и kislorod, кои се користат за да се чуваат атомите со соодветните видови (водород и кислород) (може било која имплементација на стек да се искористи). Од влезната листа кој содржи атоми (atomi) се поминува низ секое атом во листата и се прави следново: Ако атомот е водород ("H"), се додава во стекот vodorod, а ако атомот е кислород ("O"), се додава во стекот kislorod. Треба да се провери дали има доволно водородни и кислородни атоми за создавање на водата (H_2O). Доколку има доволно водородни и кислородни атоми, треба да се отстрани по два водородни и еден кислороден атом од соодветните стекови, со зголемување на бројот на создадени молекули вода. Доколку нема доволно водородни и кислородни атоми за да се создаде вода, операцијата се завршува. По обработката на сите атоми треба да се отстрани сите останати водородни и кислородни атоми од стековите во низа која ги содржи сите атоми како текст во формат "H" за водород и "O" за кислород, и се печати бројот на создадените молекули на вода.

Имплементацијата на решението е дадена подолу. Резултатот од примерен влез се печати на излез во главниот main метод.

```

1 import java.util.NoSuchElementException;
2 import java.io.BufferedReader;
3 import java.io.IOException;
4 import java.io.InputStreamReader;
5 import java.util.LinkedList;
6 import java.util.List;
7
8 //Вметни класа Stack
9 //Вметни класа ArrayStack
10
11 public class Molekuli {
12
13     public static String napraviMolekula (ArrayQueue<String> l) {
14         ArrayStack<String> vodorod = new ArrayStack<String>(1000);
15         ArrayStack<String> kislorod = new ArrayStack<String>(1000);
16         String s;
17         String izlez = new String();

```

```
18     int br = 0;
19
20     while (!l.isEmpty()){
21         s = (String) l.dequeue();
22         if (s.equals("H"))
23             vodorod.push(s);
24         else
25             kislorod.push(s);
26     }
27
28     while (!kislorod.isEmpty()){
29         if(!vodorod.isEmpty()) {
30             vodorod.pop();
31             if(!vodorod.isEmpty()) {
32                 vodorod.pop();
33                 kislorod.pop();
34                 br++;
35             }
36             else {
37                 vodorod.push("H");
38                 break;
39             }
40         }
41         else
42             break;
43     }
44
45     while(!vodorod.isEmpty()){
46         vodorod.pop();
47         izlez+="H\n";
48     }
49     while(!kislorod.isEmpty()){
50         kislorod.pop();
51         izlez+="O\n";
52     }
53
54     System.out.println(br);
55
56     return izlez;
57 }
58
```

```

59  public static void main (String[] args) throws IOException {
60      BufferedReader br = new BufferedReader(new
61          InputStreamReader(System.in));
62      String[] vlez = new String[1000];
63      vlez = br.readLine().split(" ");
64
65      ArrayQueue<String> atomi = new ArrayQueue<String>(1000);
66      for(int i=0; i<vlez.length; i++)
67          atomi.enqueue(vlez[i]);
68
69      System.out.println(napraviMolekula(atomi));
70  }

```

Задача 3. Infix to Postfix

Користејќи ја податочната структура магацин (стек), променете го изразот од infix во postfix нотација.

Infix нотација: операторот се наоѓа помеѓу operandите $(5 + 9) * 2 + 6 * 5$.

Postfix нотација: операторите се запишуваат после operandите $5 9 + 2 * 6 5 *$ $+$.

Влез: Во влезот е даден изразот во infix нотација.

Излез: На излез треба да се испечати изразот во postfix нотација.

Забелешка: При реализација на задачата дозволено е да се користи само еден стек. Не е дозволено да се користат дополнителни структури како низи и листи.

Пример:

Влез: $a + b * (c^d - e)^f + g * h - i$

Излез: $abcd^e - fgh * +^ * +i -$

Решение

Во дадената задача треба да се промени даден израз од infix во postfix нотација. Во постфиксната нотација, операторите следат по operandите. Мора да се користи стек за да се организира обработката на операторите и да се изгради постфиксниот израз. Со користење на стек, можеме да напишеме ефикасен алгоритам за преведување на инфиксниот израз во постфиксна нотација, што овозможува подоцна лесно извршување на изразот без потреба од анализа на инфиксната структура.

Имплементацијата на решението е дадена подолу. Во оваа имплементација методот infixToPostfix враќа резултат во постфиксна нотација. Операторите во инфиксната нотација имаат различни приоритети (на пример, * и / имаат поголем приоритет од + и -). Стекот овозможува ефикасно следење на приоритетите на операторите и правилното редоследување на операторите во постфиксната нотација. Кога се сртне оператор во инфиксниот израз, ако има оператори на врвот на стекот со поголем или ист приоритет, тие се преместуваат на постфиксниот израз пред таа нова операција. Исто така, стекот се користи за да се следат заградите (и) во изразот и да се осигура дека операторите меѓу заградите се обработуваат соодветно. Резултатот од примерен израз во постфикс нотација со повикување на infixToPostfix методот се печати на излез во главниот main метод.

```

1 //Вметни класа Stack
2 //Вметни класа ArrayStack
3
4 public class InfixToPostfix {
5     public static String infixToPostfix(String infix) {
6         StringBuilder postfix = new StringBuilder();
7         ArrayStack<Character> operatorStack = new ArrayStack<>(1000);
8
9         for (char c : infix.toCharArray()) {
10            if (Character.isLetterOrDigit(c)) {
11                postfix.append(c); // Ако е операнд, додади го на постфикс
12                изразот.
13            } else if (c == '(') {
14                operatorStack.push(c); //Ако е (, стави го на стекот на
15                оператори.
16            } else if (c == ')') {
17                //Премести оператори од стекот на оператори на постфикс се
18                //додека не наидеш на (.
19                while (!operatorStack.isEmpty() && operatorStack.peek() !=
20                    '(') {
21                    postfix.append(operatorStack.pop());
22                }
23                operatorStack.pop(); //Отстрани го ( од стекот на оператори.
24            } else {
25                //Операторите имаат различни приоритети.
26                //Премести оператори од стекот на оператори на постфикс
27                //доколку имаат поголем или ист приоритет.
28                while (!operatorStack.isEmpty() && precedence(c) <
29                    precedence(operatorStack.peek())) {
30                    postfix.append(operatorStack.pop());
31                }
32            }
33        }
34    }
35 }
```

```

26         operatorStack.push(c);
27     }
28 }
29
30 //Премести ги сите преостанати оператори од стекот на оператори на
31 //постфикс.
32 while (!operatorStack.isEmpty()) {
33     postfix.append(operatorStack.pop());
34 }
35
36     return postfix.toString();
37 }
38
39 private static int precedence(char operator) {
40     switch (operator) {
41         case '^':
42             return 3;
43         case '*':
44         case '/':
45             return 2;
46         case '+':
47         case '-':
48             return 1;
49         default:
50             return 0;
51     }
52 }
53
54 public static void main(String[] args) {
55     String infixExpression = "a+b*(c^d-e)^(f+g*h)-i";
56     String postfixExpression = infixToPostfix(infixExpression);
57     System.out.println(postfixExpression);
58 }
```

4.2.7 Задачи за вежбање

Задача 1. Дупли загради

Со користење на податочна структура да се најде дали еден израз има дупли загради или не. Изразот е правилен и содржи само еден вид загради (). Докол-

ку има да се испечати соодветна порака. Задачата да се реши со временска и мемориска сложеност од $O(n)$.

Влез: Во влезот е даден аритметичкиот израз.

Излез: На излез треба да се испечати "Najdeni se dupli zagradi доколку се најдени дупли загради во изразот. Доколку нема се печати "/".

Забелешка: Задачата се решава исклучиво со една податочна структура и не е дозволено користење на дополнителни други структури.

Пример 1:

Влез:

$((a+(b)))+(c+d))$

Излез:

Najdeni se dupli zagradi

Пример 2:

Влез:

$((a)+(b))$

Излез:

/

Задача 2. Отровни растенија

Во една градина имаа одреден број на растенија. Секое од растенијата е третирано со одредена количина на пестициди. После секој ден, ако некое растение има повеќе пестициди од растението што е лево од него (односно е послабо), истото умира.

Со дадени иницијални вредности за количината на пестициди во секое растение, треба да се одреди после кој ден по ред (бројот на денот) нема веќе да умира ни едно растение, односно после кој ден нема да има растенија кои имаат повеќе пестициди од тие што се до нив од лева страна.

Влез: Во влезот во првиот ред е даден бројот на растенија, а потоа за секое растение е дадена количината на пестициди.

Излез: На излез треба да се испечати бројот на денот по кој веќе нема да умира ни едно растение.

Пример 1:

Влез:

7

6 5 8 4 7 10 9

Излез:

2

На почеток, сите растенија се живи. После првиот ден, умираат 3 растенија - Зтото (8 пестициди), 5тото (7 пестициди) и бтото (10 пестициди). Со тоа, остануваат 6 5 4 9.

После вториот ден, умира 1 растение - последното (9 пестициди). Со тоа, остануваат 6 5 4. И после ова веќе нема да умираат растенија бидејќи нема растение кое има повеќе пестициди од тоа лево од него.

Пример 2:

Влез:

5

1 1 1 1 1

Излез:

0

Задача 3. Испитна сесија

Додека Стефан ги потготвува испитите за полагање во јунска сесија, тој има навика да ги чува сите книги на еден куп, една врз друга. При пребарување на дадена книга која му е потребна, тој секогаш ги трга прво најгорните, една по една, се додека не ја земе книгата која му треба. Штом ќе ја извади таа книга, останатите кои биле над неа ги враќа во истиот редослед назад. Откако ќе го научи дадениот предмет, ја враќа книгата одозгора врз сите други.

Дадена е иницијалната поставеност на книгите на купот на Стефан (во редослед одоздола нагоре). Дадени се и испитите по распоред на полагање за јунска сесија. Ваша задача е да одредите колку пати секоја од книгите ќе биде извадена и ставена назад на купот.

Влез: Во првата линија од влезот се дадени два броја: M, број на книги и N, број на испити.

Во втората линија од влезот се дадени имињата на книгите, подредени одоздола нагоре.

Во третата линија од влезот се дадени испитите кои се полагаат по редослед.

Излез: На излез треба да се испечати за секоја книга колку пати ќе биде земена и вратена назад на купот (еден „настан“ на земање-враќање на книгата се брои еднаш, не два пати). Имињата на книгите се печатат во исти редослед во кој биле дадени на влезот.

Пример:

Влез:

7 3

APS OS Mrezhi AOK Objektno Strukturno Kalkulus

APS Objektno Mrezhi

Излез: APS 3

OS 1

Mrezhi 2

AOK 2

Objektno 3

Strukturno 3

Kalkulus 3

Објаснување: За да ја извадиме книгата за АПС, треба да ги извадиме и вратиме назад сите останати книги во купот. Откако ќе завршиме, ја враќаме најгоре на купот. Наредно полагаме Објектно, па за стигнеме до таа книга, треба да ги тргнеме прво книгите за АПС, калкулус и структурно. Ја враќаме книгата за Објектно најодозгора. За да дојдеме до книгата за мрежи, треба да ги извадиме и вратиме сите книги освен таа за ОС. На крај ја враќаме книгата за мрежи најгоре.

Забелешка: Не може да има дупликати наслови на книги. Еден испит може да се појави повеќе пати. На излез имињата на книгите се печатат во исти редослед во кој биле дадени на влезот.

Задача 4. Танцови двојки

Да се напише алгоритам со кој ќе се изврши креирање на танцови парови по соодветни танц-групи во една танцова школа. Танцов пар се формира од машко и женско запишани на иста танцова група. Во школата за танци на располагање има групи за основни танци O, стандардни танци S и латино танци L. Има уписан рок така што заинтересираните кандидати може да се упишат. Со завршување на уписниот рок се врши формирање на танцови двојки. Ваша задача е од добиениот список на сите запишани кандидати да направите соодветни парови и да кажете колку, од каков тип на кандидати (машко или женско) и за која танцова група фолат за да сите добијат свој партнери.

Влез: Во влезот е дадена листа од уписаните кандидати по редослед: прв дојден, прв запишан и тоа во облик: танцова група, пол.

Излез: На излез треба да се испечатат бројот на двојките кои се формираат и самите двојки.

Пример:

Влез:

LM OZ OM OM LM SZ SM LZ OM LZ SZ SM SM LM

Парови кои може да се формираат од овој список се: (LM,LZ); (SM, SZ); (SZ SM); (LM, LZ); (OZ, OM); (LM, LZ)

Остануваат без партнери: OM, OM, SM, LM

Излез:

4

LZ SZ OZ OZ

Задача 5. Игра со топчиња

Да се напише алгоритам со кој ќе се имплементира играта за мобилен телефон “Подреди топчиња според боја”. Во оваа игра на располагање имате топчиња во три различни бои (R-црвена, G-зелена и B-сина). На екран имате 3 кутии. Во првата кутија се наоѓаат топчињата кои што ви ги доделува апликацијата на почеток на играта. Играта завршува кога ќе се подредат топчињата според боја (во третата кутија) и тоа во следниот редослед RGB (односно прво ќе бидат црвени-те, па зелените и на крај сините топчиња), а другите кутии се празни. Втората кутија може да ја користите како помошна при распределбата на топчињата. Притоа треба да се води грижа дека во еден момент само едно топче може да се вади или става од врвот на кутијата. Исто така за оваа игра важи следново правило: Доколку на влез дојдат последователно три топчиња од црвена боја, тоа значи “бомба”. Ова значи поништување на сите топчиња последователно што се наоѓаат во таа кутија (што се црвена боја), се додека не се дојде до топче од различна боја.

Влез: Во влезот е даден прво вкупниот број на топчиња. Следно се дава во секој нареден ред соодветно секвенцата од топчиња која што треба да ја сместите во првата кутија.

Излез: На излез треба да се испечати состојбата на топчињата во третата кутија.

Пример 1:

Влез:

4

R

R

R

G

Излез:

G

Пример 2:

Влез:

3

B

B

В

Излез:

В В В

4.3 Редица (ред)

Редицата преставува еднодимензионална линеарна секвенца од елементи која работи по принципот прв-внесен-прв-изваден. Тоа значи дека елементите се додаваат на едниот крај од линеарната секвенца (опаш), а се вадат на другиот крај (глава). Должина на редицата преставува бројот на елементи што ги содржи во себе. Празната редица има должина 0.

4.3.1 Операции со редица

Основните операции за податочниот тип редица се:

- Празнење на целата редица
- Проверка дали редицата е празна
- Додавање на елемент на опашот од редицата (операција “en-queue”)
- Вадење елемент од главата на редицата (операција “de-queue”)
- Дополнително, проверка на елементот на главата на редицата без негово вадење (операција "peek").

И за редицата ќе дадеме 2 имплементации, со двата фундаментални податочни типови: низа и листа.

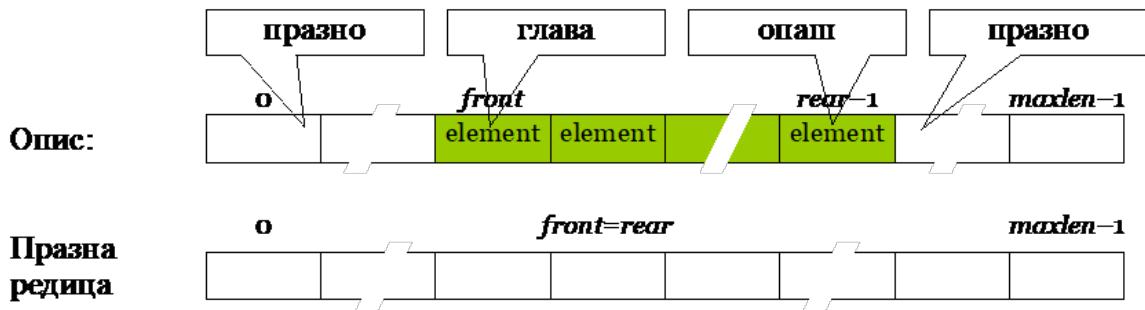
4.3.2 Имплементација на редица со низа

Редицата имплементирана со низа преставува ограничена редица, со должина помала или еднаква на должината на низата со која се имплементира maxlen. Содржи променлива length, која ја содржи моменталната должина, потоа променливи глава (front) и опаш (rear), како и низа од елементи elems, кои ги содржат елементите на редицата на позициите elems[front...rear-1].

На слика 4-4 е даден пример за имплементација на редица со низа.

На следната слика 4-5 е даден пример на последователни вметнувања и вадења на елементи од редица, имплементирана со низа од 6 елементи.

Разгледувајќи ја конечната состојба на примерот од сликата (по додавањето на Ралф), се наметнува прашањето што ако сега сакаме да додадеме уште еден елемент? Максималниот капацитет не е исполнет (низата има 6 елементи, а редицата моментално само 4). Каде треба да се додаде новиот елемент? Едно



Слика 4-4: Имплементација на редица со низа.

На почеток:	По додавање на Homer, Marge, Bart, Lisa:
0 1 2 3 4 5 <code>elems</code> [] [] [] [] [] [] <code>front</code> [0] <code>rear</code> [0] <code>length</code> [0]	0 1 2 3 4 5 <code>elems</code> [Homer] [Marge] [Bart] [Lisa] [] [] <code>front</code> [0] <code>rear</code> [4] <code>length</code> [4]
По додавање на Maggie:	По вадење на елемент од главата:
0 1 2 3 4 5 <code>elems</code> [Homer] [Marge] [Bart] [Lisa] [Maggie] [] <code>front</code> [0] <code>rear</code> [5] <code>length</code> [5]	0 1 2 3 4 5 <code>elems</code> [] [Marge] [Bart] [Lisa] [Maggie] [] <code>front</code> [1] <code>rear</code> [5] <code>length</code> [4]
По вадење на елемент од главата:	По додавање на Ralph:
0 1 2 3 4 5 <code>elems</code> [] [Bart] [Lisa] [Maggie] [] <code>front</code> [2] <code>rear</code> [5] <code>length</code> [3]	0 1 2 3 4 5 <code>elems</code> [] [] [Bart] [Lisa] [Maggie] [Ralph] <code>front</code> [2] <code>rear</code> [0] <code>length</code> [4]

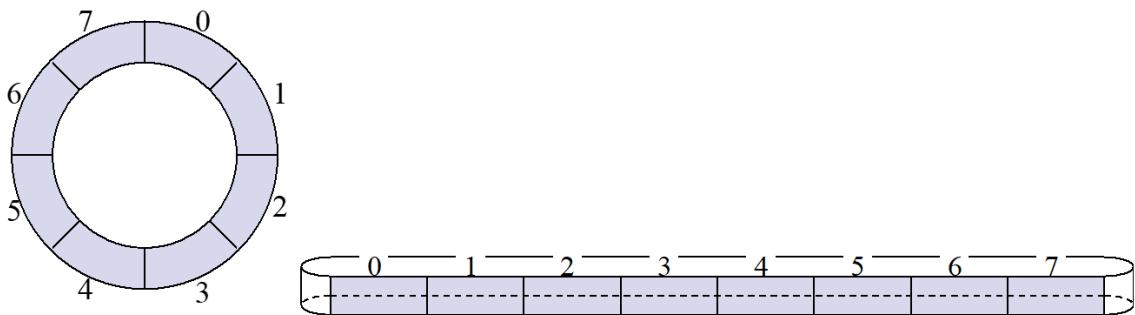
Слика 4-5: Пример за последователни додавања и вадења на елементи во редица.

можно решение на овој проблем е да ја модифицираме операцијата на вадење на елемент, при што за секој изваден елемент, останатите елементи би се поместувале во лево, за да се порамнат на почетокот на низата. Но ваквата операција би имала временска сложеност $O(n)$, иста со операцијата бришење на елемент од низа. Дали постои поефикасен начин?

Решение на овој проблем претставуваат цикличните низи. Циклична низа со должина n е низа во која секој елемент има и претходни и следбеник. При тоа, претходник на $a[0]$ е $a[n-1]$, а следбеник на $a[n-1]$ е $a[0]$.

На слика 4-6 е дадена визуелна престава на ваквите низи.

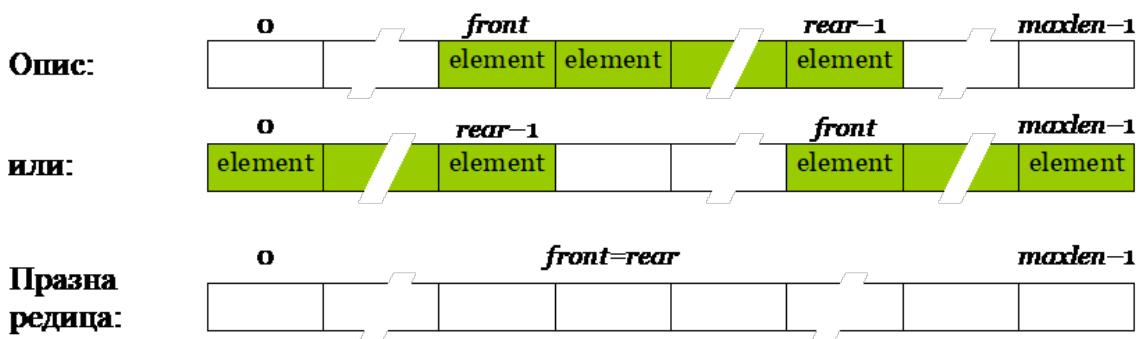
Во тој случај, имплементација преставува ограничена редица, со капацитет помал или еднаков на однапред дефинирана максимална должина ($maxlen$), содржи променлива $length$, која ја содржи моменталната должина, променливи глава ($front$) и опаш ($rear$), како и циклична низа од од елементи $elems$,



Слика 4-6: Визуелизација на циклична низа.

кои ги содржат елементите на редицата на позициите $\text{elems}[\text{front} \dots \text{rear}-1]$ или $\text{elems}[\text{front} \dots \text{maxlen}-1]$ и $\text{elems}[0 \dots \text{rear}-1]$.

На слика 4-7 е даден пример за имплементација со циклична низа



Слика 4-7: Имплементација на редица со циклична низа.

На слика 4-8 е даден пример на додавање и вадење на елементи во редица имплементирана со циклична низа.

За да се моделира редица прво ќе користиме Java интерфејсна дефиниција за интерфејс на редица (Queue) структура на податоци. Во овој интерфејс, Е претставува параметаризиран тип (генерички) кој може да биде било кој тип. Методите во интерфејсот се: `isEmpty()` кој проверува дали редицата е празна; `size()` кој го враќа бројот на елементи во редицата, односно ја должината на редицата; `peek()` кој го враќа елементот кој се наоѓа на почетокот на редицата без да го отстранува, т.е. го враќа елементот кој следен треба да биде избришан од редицата; `clear()` кој ја празни редицата; `enqueue(E x)` кој го додава елементот `x` на крајот на редицата; и `dequeue()` кој го отстранува и враќа првиот елемент (почетниот) од редицата. Интерфејсот Queue ја дефинира основната функционалност на редица како податочна структура и овозможува имплементација на разни видови редици во Java, вклучувајќи обични редици, приоритетни редици и други.

<p>На почеток:</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>elems</td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>front</td><td>0</td><td>rear</td><td>0</td><td>length</td><td>0</td></tr> </table>	0	1	2	3	4	5	elems						front	0	rear	0	length	0	<p>По додавање на Homer, Marge, Bart, Lisa:</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>elems</td><td>Homer</td><td>Marge</td><td>Bart</td><td>Lisa</td><td></td></tr> <tr><td>front</td><td>0</td><td>rear</td><td>4</td><td>length</td><td>4</td></tr> </table>	0	1	2	3	4	5	elems	Homer	Marge	Bart	Lisa		front	0	rear	4	length	4
0	1	2	3	4	5																																
elems																																					
front	0	rear	0	length	0																																
0	1	2	3	4	5																																
elems	Homer	Marge	Bart	Lisa																																	
front	0	rear	4	length	4																																
<p>По додавање на Maggie:</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>elems</td><td>Homer</td><td>Marge</td><td>Bart</td><td>Lisa</td><td>Maggie</td></tr> <tr><td>front</td><td>0</td><td>rear</td><td>5</td><td>length</td><td>5</td></tr> </table>	0	1	2	3	4	5	elems	Homer	Marge	Bart	Lisa	Maggie	front	0	rear	5	length	5	<p>По вадење на елемент од главата:</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>elems</td><td></td><td></td><td>Bart</td><td>Lisa</td><td>Maggie</td></tr> <tr><td>front</td><td>1</td><td>rear</td><td>5</td><td>length</td><td>4</td></tr> </table>	0	1	2	3	4	5	elems			Bart	Lisa	Maggie	front	1	rear	5	length	4
0	1	2	3	4	5																																
elems	Homer	Marge	Bart	Lisa	Maggie																																
front	0	rear	5	length	5																																
0	1	2	3	4	5																																
elems			Bart	Lisa	Maggie																																
front	1	rear	5	length	4																																
<p>По вадење на елемент од главата:</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>elems</td><td>Nelson</td><td></td><td>Bart</td><td>Lisa</td><td>Maggie</td></tr> <tr><td>front</td><td>2</td><td>rear</td><td>1</td><td>length</td><td>5</td></tr> </table>	0	1	2	3	4	5	elems	Nelson		Bart	Lisa	Maggie	front	2	rear	1	length	5	<p>По додавање на Ralph:</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>elems</td><td></td><td></td><td>Bart</td><td>Lisa</td><td>Maggie</td></tr> <tr><td>front</td><td>2</td><td>rear</td><td>0</td><td>length</td><td>4</td></tr> </table>	0	1	2	3	4	5	elems			Bart	Lisa	Maggie	front	2	rear	0	length	4
0	1	2	3	4	5																																
elems	Nelson		Bart	Lisa	Maggie																																
front	2	rear	1	length	5																																
0	1	2	3	4	5																																
elems			Bart	Lisa	Maggie																																
front	2	rear	0	length	4																																
<p>По додавање на Nelson:</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>elems</td><td>Nelson</td><td></td><td>Bart</td><td>Lisa</td><td>Maggie</td></tr> <tr><td>front</td><td>2</td><td>rear</td><td>1</td><td>length</td><td>5</td></tr> </table>	0	1	2	3	4	5	elems	Nelson		Bart	Lisa	Maggie	front	2	rear	1	length	5	<p>По додавање на Martin:</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>elems</td><td>Nelson</td><td>Martin</td><td>Bart</td><td>Lisa</td><td>Maggie</td></tr> <tr><td>front</td><td>2</td><td>rear</td><td>2</td><td>length</td><td>6</td></tr> </table>	0	1	2	3	4	5	elems	Nelson	Martin	Bart	Lisa	Maggie	front	2	rear	2	length	6
0	1	2	3	4	5																																
elems	Nelson		Bart	Lisa	Maggie																																
front	2	rear	1	length	5																																
0	1	2	3	4	5																																
elems	Nelson	Martin	Bart	Lisa	Maggie																																
front	2	rear	2	length	6																																
<p>По вадење на елемент од главата:</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>elems</td><td>Nelson</td><td>Martin</td><td></td><td>Lisa</td><td>Maggie</td></tr> <tr><td>front</td><td>3</td><td>rear</td><td>2</td><td>length</td><td>5</td></tr> </table>	0	1	2	3	4	5	elems	Nelson	Martin		Lisa	Maggie	front	3	rear	2	length	5																			
0	1	2	3	4	5																																
elems	Nelson	Martin		Lisa	Maggie																																
front	3	rear	2	length	5																																

Слика 4-8: Пример за последователни додавања и вадења на елементи во редица имплементирана со циклична низа.

```

1 public interface Queue<E> {
2     // Елементи на редицата се објекти од произволен тип.
3     // Методи за пристап:public boolean isEmpty();
4
5     // Враќа true ако и само ако редицата е празна.
6
7     public int size();
8     // Ја враќа должината на редицата.
9
10    public E peek();
11    // Го враќа елементот од почетокот на редицата.
12
13    // Методи за трансформација:

```

```

14  public void clear();
15  // Ја празни редицата.
16
17  public void enqueue(E x);
18  // Го додава x на крај на редицата.
19
20  public E dequeue();
21  // Го отстранува и враќа почетниот елемент на редицата.
22 }

```

Следната генеричка Java класа ArrayQueue представува редица која користи низа за складирање на елементите. Оваа класа ги користи променливите elems, length, front и rear за складирање на податоците. elems е низа за складирање на елементите на редицата, length го чува бројот на моментално зачуваните елементи во редицата, а front и rear ги користи за следење на почетниот и крајниот елемент на редицата. Конструкторот на ArrayQueue креира нова празна редица со максимална длабочина (максимален број на елементи). Методите од класата за редица се: isEmpty() кој ја проверува редицата дали е празна; size() кој го враќа бројот на елементи во редицата (должината); peek() кој го враќа елементот на почетокот на редицата без да го отстранува; clear() кој ја брише содржината на редицата и поставува front и rear на произволни вредности; enqueue(E x) кој го додава елементот x на крајот на редицата и ако rear стигне до последниот елемент од низата, го враќа на почетокот на низата (циклички); и dequeue() кој го отстранува и враќа првиот елемент (почетниот) на редицата и доколку front стигне до последниот елемент од низата, го враќа на почетокот на низата (циклички). Оваа класа овозможува имплементација на редици во Java користејќи низа како основа за складирање на елементите.

```

1 import java.util.NoSuchElementException;
2
3 class ArrayQueue<E> {
4     // Редицата е претставена на следниот начин:
5     // length го содржи бројот на елементи.
6     // Ако length > 0, тогаш елементите на редицата се зачувани во
7     // elems[front...rear-1]
8     // Ако rear > front, тогаш во elems[front...maxlength-1] и
9     // elems[0...rear-1]
10    E[] elems;
11    int length, front, rear;
12
13    // Конструктор ...
14    @SuppressWarnings("unchecked")

```

```
13     public ArrayQueue(int maxlen) {
14         elems = (E[]) new Object[maxlen];
15         clear();
16     }
17
18     public boolean isEmpty() {
19         // Враќа true ако и само ако редицата е празна.
20         return (length == 0);
21     }
22
23     public int size() {
24         // Ја враќа додлжината на редицата.
25         return length;
26     }
27
28     public E peek() {
29         // Го враќа елементот на почетокот на редицата.
30         if (length > 0)
31             return elems[front];
32         else
33             throw new NoSuchElementException();
34     }
35
36     public void clear() {
37         // Ја празни редицата.
38         length = 0;
39         front = rear = 0; // произволно
40     }
41
42     public void enqueue(E x) {
43         // Го додава x на крај на редицата.
44         if (length == elems.length)
45             throw new NoSuchElementException();
46         elems[rear++] = x;
47         if (rear == elems.length) rear = 0;
48         length++;
49     }
50
51     public E dequeue() {
52         // Го отстранува и враќа почетниот елемент на редицата.
53         if (length > 0) {
```

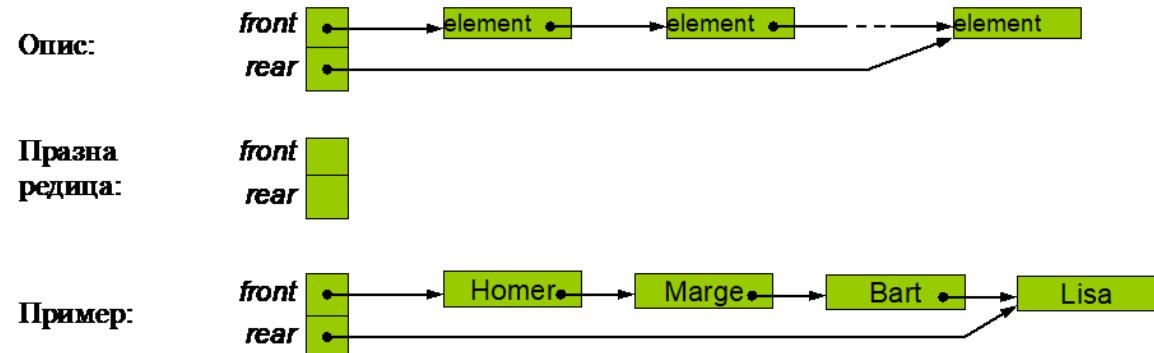
```

54     E frontmost = elems[front];
55     elems[front++] = null;
56     if (front == elems.length) front = 0;
57     length--;
58     return frontmost;
59 } else
60     throw new NoSuchElementException();
61 }
62 }
```

4.3.3 Имплементација на редица со листа

Втората имплементација на редицата е имплементацијата со поврзана листа. При тоа се добива неограничена редица, преставена со поврзана листа, каде првиот елемент е главата на редицата, а чие заглавие содржи два покажувачи: еден кон првиот елемент (глава - front) и еден кон последнот елемент (опаш - rear) и променлива length.

На слика 4-9 е даден пример за имплементација на редицата со едностррано поврзана листа



Слика 4-9: Имплементација на редица со едностррано поврзана листа.

Следната генеричка Java класа LinkedQueue имплементира редица која користи едностррано поврзана листа за складирање на елементите. Оваа класа ги користи променливите front, rear и length за складирање на податоците. front и rear се референци до првиот и последниот јазол од едностррано поврзаната листа која ги содржи елементите на редицата, а length го чува бројот на моментално зачуваните елементи во редицата. Конструкторот на LinkedQueue креира нова празна редица. Методите во оваа имплементација на редица со листа се: isEmpty() кој проверува дали редицата е празна; size() кој го враќа бројот на елементи во редицата (должината); peek() кој го враќа елементот на почетокот

на редицата без да го отстранива; clear() ја брише содржината на редицата со поставување на front и rear на null, а length се поставува на 0; enqueue(E x) кој го додава елементот x на крајот на редицата со креирање нов јазол и ако редицата веќе има елементи, го поврзува со последниот елемент преку rear.succ, а rear го поставува на новиот јазол, притоа доколку редицата е празна, и front и rear се поставуваат на новиот јазол; и dequeue() кој го отстранива и враќа првиот елемент (почетниот) на редицата поставувајќи го front на следниот јазол во листата и ако отстранетиот јазол е последниот, го поставува и rear на null, притоа намалувајќи length за 1 и го враќа отстранетиот елемент. Оваа класа овозможува имплементација на редици во Java користејќи едностррано поврзана листа како основа за складирање на елементите.

```

1 import java.util.NoSuchElementException;
2
3 public class LinkedQueue<E> implements Queue<E> {
4     // Редицата е претставена на следниот начин:
5     // length го содржи бројот на елементи.
6     // Елементите се зачувуваат во јазли од SLL
7     // front и rear се линкови до првиот и последниот јазел соодветно.
8     SLLNode<E> front, rear;
9     int length;
10
11    // Конструктор ...
12    public LinkedQueue () {
13        clear();
14    }
15
16    public boolean isEmpty () {
17        // Враќа true ако и само ако редицата е празна.
18        return (length == 0);
19    }
20
21    public int size () {
22        // Ја враќа должината на редицата.
23        return length;
24    }
25
26    public E peek () {
27        // Го враќа елементот на почетокот на редицата.
28        if (front == null)
29            throw new NoSuchElementException();

```

```

30         return front.element;
31     }
32
33     public void clear () {
34         // Ја празни редицата.
35         front = rear = null;
36         length = 0;
37     }
38
39     public void enqueue (E x) {
40         // Го додава x на крај на редицата.
41         SLLNode<E> latest = new SLLNode<E>(x, null);
42         if (rear != null) {
43             rear.succ = latest;
44             rear = latest;
45         } else
46             front = rear = latest;
47         length++;
48     }
49
50     public E dequeue () {
51         // Го отстранива и враќа почетниот елемент на редицата.
52         if (front != null) {
53             E frontmost = front.element;
54             front = front.succ;
55             if (front == null) rear = null;
56             length--;
57             return frontmost;
58         } else
59             throw new NoSuchElementException();
60     }
61 }
```

4.3.4 Queue во Java

Нема готова класа за редица во Java, тук постои само интерфејс. Ова е Queue интерфејсот во Java кој може да се користи за имплементација на класи за редица:

```

1  public interface Queue<E> extends Collection<E>
2 {
```

```

3   E element();
4   boolean offer(E e);
5   E peek();
6   E poll();
7   E remove();
8 }
```

Класи од Јава кои го имплементираат Queue интерфејсот се: AbstractCollection, LinkedList, PriorityQueue, LinkedBlockingQueue, BlockingQueue, ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue.

4.3.5 Едноставни проблеми со редица

Редицата исто така има голем број на примени. Секоја редица на чекање на добивање одреден сервис всушност се претставува со апстрактниот тип редица. Во компјутерските системи, редиците се користат кај мрежните печатачи, каде повеќе корисници испраќаат документи за печатење на делен печатач, потоа кај хард дисковите, односно во нивните драјвери, каде повеќе процеси бараат пристап до податоците од дискот. Уште една примена е во оперативните системи, каде повеќе процеси чекаат на ред да бидат опслужени од процесорот. Системот наречен распределувач (scheduler) ги распределува процесите кои чекаат во листата и им обезбедува пристап до процесорот.

Задача 1. Round Robin

Во оперативните системи за распределување на процеси се користи распределувачки алгоритам Round-Robin. Да се имплементира модифициран Round-robin алгоритам кој што работи на следниот принцип: Сите процеси кои треба да бидат распределени како атрибути имаат име, време на извршување и време на пристигнување. Алгоритамот ги распределува процесите според нивното време на пристигнување, односно логиката на извршување е “прв дојден, прв услужен”. Првиот процес кој ќе биде распределен од распределувачот е овој процес кој што има најмало време на пристигнување во системот. На овој процес распределувачот му дава определен квантум на време за кое може да се извршува. Откако времето ќе заврши, ако процесот не се извршил целосно, истиот се прекинува и притоа се менува неговото време на извршување (кое сега е првичното време – квантумот доделен од алгоритамот RR за извршување), и овој процес се става последен во процесите кои чекаат за распределување. Истата постапка се применува и за следните процеси кои чекаат за распределба. Оваа постапка се

применува се додека сите процеси не се извршат, односно нивното време на извршување не стане 0. Доколку два процеси имаат исто време на пристигнување, распределувачот ќе го земе оној процес кој што има поголемо време на извршување.

Влез: Во влезот е даден во првата линија цел број $N > 0$, кој претставува број на процеси кои треба да се распределуваат, а потоа се внесуваат во посебни линии соодветно: името на процесот (стринг), времето на извршување (цел број) и времето на пристигнување (цел број) за секој од процесите. Во последната линија се дава еден цел број, кој го означува квантумот на време T кој што алгоритамот RR им го дodelува на процесите за извршување.

Излез: На излез треба да се испечати редоследот на распределување на процесите со помош на алгоритамот RR, и тоа во иста линија само имињата на процесите одделени со празно место.

Пример:

Влез:

5
A 40 2
B 35 0
C 28 10
D 45 4
E 32 2
10

Излез:

B A E D C B A E D C B A E D D

Решение

Во дадената задача треба да се имплементира модифициран распределувачки алгоритам Round-Robin за да се распределат дадени процеси. Алгоритамот ги распределува процесите според нивното време на пристигнување, односно логиката на извршување е “прв дојден, прв услужен”. Поради ова, за да се одговори на барањата од задачата мора да се искористи соодветна податочна структура редица (може да се искористи било која имплементација) за чување на процесите кои треба да се извршат. За да се моделира еден процес, кој има информација за име, време на пристигнување (`arrival_time`) и време на извршување (`execution_time`) мора да се даде соодветна класа `Process` во која има метод `updateTime` за ажурирање на времето на извршување на процесот според даден квант (`quantum`). Прво, се вчитуваат информации за секој процес (име, време на извршување и време на пристигнување) и се креираат објектите за секој процес, по што се внесува

квантот (quantum) за Round Robin алгоритмот. Процесите се сортираат според времето на пристигнување и времето на извршување и се додаваат во редицата соодветно. Потоа, тие се извршуваат во редослед според времето на пристигнување и квантот. Доколку процесот не заврши со извршување, се додава поново во редицата за да има шанса да се изврши во идниот круг. Излезот е редоследот на извршување на процесите.

Имплементацијата на решението е дадена подолу. Резултатот од примерен влез се печати на излез во главниот main метод.

```

1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 import java.util.Collections;
5 import java.util.LinkedList;
6 import java.util.List;
7 import java.util.NoSuchElementException;
8 import java.util.Scanner;
9
10 //Вметни класа Queue
11 //Вметни класа SLLNode
12 //Вметни класа LinkedQueue
13
14 class Process implements Comparable<Process>{
15     private String name;
16     private int arrival_time, execution_time;
17
18     public Process(String n, int at, int et){
19         this.name = n;
20         this.arrival_time = at;
21         this.execution_time = et;
22     }
23
24     public void updateTime(int quantum){
25         if(this.execution_time < quantum)
26             this.execution_time = 0;
27         else
28             this.execution_time-=quantum;
29     }
30
31     public int get_arrivalTime(){
32         return this.arrival_time;
33     }

```

```
34
35     public int get_executionTime(){
36         return this.execution_time;
37     }
38
39     public String getName(){
40         return this.name;
41     }
42
43     public String toString(){
44         return name;
45     }
46
47     @Override
48     public int compareTo(Process o) {
49         // TODO Auto-generated method stub
50         if(this.arrival_time > o.get_arrivalTime())
51             return 1;
52         else if (this.arrival_time == o.get_arrivalTime()){
53             if(this.execution_time > o.get_executionTime())
54                 return -1;
55             else
56                 return 1;
57         }
58         else
59             return -1;
60     }
61 }
62
63 public class RoundRobin {
64
65     public static void main(String[] args) throws IOException {
66         // TODO Auto-generated method stub
67         BufferedReader br =new BufferedReader(new
68             InputStreamReader(System.in));
69         LinkedList<Process> pList = new LinkedList<Process>();
70         Process p = null;
71
72         LinkedQueue<Process> q = new LinkedQueue<Process>();
73
74         int N = Integer.parseInt(br.readLine());
```

```

74
75     String input;
76     for(int i=0;i<N;i++){
77         input = br.readLine();
78         String[] atrb = input.split(" ");
79         int arrivalT = Integer.parseInt(atrb[2]);
80         int executionT = Integer.parseInt(atrb[1]);
81         p = new Process(atrb[0], arrivalT, executionT);
82         pList.add(p);
83     }
84     int quantum = Integer.parseInt(br.readLine());
85
86     Collections.sort(pList);
87     for(int i=0;i<N;i++){
88         //System.out.println(pList.get(i).getName());
89         q.enqueue(pList.get(i));
90     }
91
92     while(!q.isEmpty()){
93         p = q.dequeue();
94         p.updateTime(quantum);
95         if(p.get_executionTime() != 0)
96             q.enqueue(p);
97         System.out.print(p.toString() + " ");
98     }
99 }
100 }
```

Задача 2. Нова структура

Да се направи имплементација за нова податочна структура QuasiQueue која што ќе биде ограничена структура. Новата структура треба да може да ги задоволи следните операции:

- вметнување елемент – Вметнувањето елемент секогаш е вметнување на крај на податочната структура. Доколку при вметнување елемент структурата е полна, треба да се овозможи новиот елемент да се вметне. Тој секогаш се поставува на почетна позиција на структурата. Оваа операција треба да се имплементира со сложеност $O(1)$.
- вадење елемент – Вадењето елемент од структурата може да биде вадење на елемент од почеток или крај на структурата. Треба секогаш да се извади

помалиот елемент, при споредба на елементите на почеток и крај. Доколку се еднакви, се вади елементот од почеток на структурата. Оваа операција треба да се имплементира со сложеност $O(1)$.

- да се провери дали структурата е празна
- да се провери кој е елементот на почеток на структурата
- да се провери кој е елементот на крај на структурата

Елементите се внесуваат во структурата во истиот редослед како што се читат од стандардниот влез. При имплементација на структурата QuasiQueue имате право на користење на само една основна структура. Немате право на користење други дополнителни структури. Ваша задача е да го испечатите редоследот на вадење на сите елементи од структурата QuasiQueue.

Влез: Во влезот е дадена во првиот ред големината на структурата. Во вториот ред од влезот е даден бројот на елементи кои ќе се внесуваат, а потоа во секој нов ред се наоѓаат елементите кои треба да се внесат.

Излез: На излез треба да се испечати прво елементот на почеток на структурата, а после оној на крајот. Потоа во нов ред се печатат сите елементи од структурата QuasiQueue во редослед како што се вадат од неа, одделени со празно место.

Пример:

Влез:

6 (големина на структурата)
8 (број на елементи кои треба да се прочитаат)
9
5
4
10
0
5
2
6

Излез:

6 (елементот на почеток на структурата)
5 (елементот на крај на структурата)
5 0 6 5 4 10 (сите елементи од структурата QuasiQueue)

Решение

Во дадената задача треба да се имплементира нова податочна неограничена структура QuasiQueue. За да се одговори на барањата од задачата мора да се

искористи и трансформира структура редица (тука имплементација со низа) во која се знаат почетокот и крајот на редицата. Структура QuasiQueue е слична на обичната редица, но дополнително мора да се модифицираат методите како што се бара. Вредностите кои ги чува мораат да бидат компарабилни (имплементираат интерфејсот Comparable) за да се споредуваат.

Имплементацијата на решението е дадена подолу. Во оваа имплементација во класата QuasiQueue за да се задоволат барањата на задачата се дадени: конструктор за иницијализација на празна квази редица; isEmpty() метод кој проверува дали редицата е празна; size() кој го враќа бројот на елементи во редицата; peekFirst() метод кој го враќа првиот елемент во редицата (без да го отстранува); peekLast() метод кој го враќа последниот елемент во редицата (без да го отстранува); clear() метод кој ја празни квази редицата; enqueue(E x) кој додава елемент x на крајот на редицата; и dequeue() кој го отстранува и враќа помалиот елемент, при споредба на елементите на почеток и крај од редицата. Резултатот од примерен влез се печати на излез во главниот main метод.

```

1 import java.util.NoSuchElementException;
2 import java.io.BufferedReader;
3 import java.io.IOException;
4 import java.io.InputStreamReader;
5
6 //Вметни класа Queue
7
8 interface Queue<E extends Comparable<E>> {
9     //Елементи на редицата се објекти од произволен тип.
10    //Методи за простап:
11    public boolean isEmpty ();
12    //Враќа true ако и само ако редицата е празна.
13
14    public int size ();
15    //Ја враќа должината на редицата.
16
17    public E peekFirst ();
18    //Го враќа елементот од почетокот на редицата.
19
20    public E peekLast ();
21    //Го враќа елементот од крајот на редицата.
22
23    //Методи за трансформација:
24    public void clear ();
25    //Ја празни редицата.

```

```
26
27     public void enqueue (E x);
28     //Го додава x на крај на редицата.
29
30     public E dequeue ();
31     //Го отстранува и враќа почетниот елемент на редицата.
32 }
33
34 class QuasiQueue<E extends Comparable<E>> implements Queue<E> {
35     //Редицата е претставена на следниот начин:
36     //length го содржи бројот на елементи.
37     //Ако length > 0, тогаш елементите на редицата се зачувани во
38     //elems[front...rear-1]
39     //Ако rear > front, тогаш во elems[front...maxlength-1] и
40     //elems[0...rear-1]
41     E[] elems;
42     int length, front, rear, maxlength;
43
44     @SuppressWarnings("unchecked")
45     public QuasiQueue (int maxlength) {
46         elems = (E[]) new Comparable[maxlength];
47         thismaxlength = maxlength;
48         clear();
49     }
50
51     public boolean isEmpty () {
52         //Враќа true ако и само ако редицата е празна.
53         return (length == 0);
54     }
55
56     public int size () {
57         //Ја враќа должината на редицата.
58         return length;
59     }
60
61     public E peekFirst () {
62         //Го враќа елементот од почетокот на редицата.
63         if (length > 0)
64             return elems[front];
65         else
66             throw new NoSuchElementException();
67 }
```

```
66
67     public E peekLast () {
68         //Го враќа елементот од крајот на редицата.
69         if (length > 0)
70             return elems[rear-1];
71         else
72             throw new NoSuchElementException();
73     }
74
75     public void clear () {
76         //Ја празни редицата.
77         length = 0;
78         front = rear = 0; //произволно
79     }
80
81     public void enqueue (E x) {
82         //Го додава x на крај од редицата.
83         //elems[rear++] = x;
84         if (rear == maxlen)
85             elems[0] = x;
86         else{
87             elems[rear++] = x;
88             length++;
89         }
90     }
91
92     public E dequeue () {
93         //Го отстранува и враќа помалиот од почетниот и крајниот елемент на
94         //редицата.
95         if (length > 0) {
96             E frontmost = elems[front];
97             E rearmost = elems[rear-1];
98             if (front == rear){
99                 front = 0;
100                rear = 0;
101                elems[front] = null;
102                length--;
103                return frontmost;
104            }
105            if(frontmost.compareTo(rearmost) == -1 ||
106                frontmost.compareTo(rearmost) == 0){
```

```
106         elems[front++] = null;
107         length--;
108         return frontmost;
109     }
110     else{
111         elems[rear-1] = null;
112         rear--;
113         length--;
114         return rearmost;
115     }
116 } else
117     throw new NoSuchElementException();
118 }
119 }
120
121 public class TestQuasiQueue {
122     public static void main(String[] args) throws IOException {
123         BufferedReader br = new BufferedReader(new
124             InputStreamReader(System.in));
125
126         int maxElementi = Integer.parseInt(br.readLine());
127         //System.out.println(maxElementi);
128
129         QuasiQueue<Integer> qq = new QuasiQueue<Integer>(maxElementi);
130         int brElementi = Integer.parseInt(br.readLine());
131
132         for(int i=0; i<brElementi; i++){
133             int x = Integer.parseInt(br.readLine());
134             qq.enqueue(x);
135             //System.out.println(qq.peekFirst());
136         }
137
138         System.out.println(qq.peekFirst());
139         System.out.println(qq.peekLast());
140
141         while(!qq.isEmpty()){
142             System.out.print(qq.dequeue()+" ");
143         }
144     }
```

Задача 3. Возови

На една железничка станица дошло до расипување на еден воз. За да се искористат вагоните кои се во функционална состојба, потребно е да се направи прераспределување на истите за да се добие нов воз. Откачувањето на вагоните од расипаниот воз се прави еден по еден од страна на последниот вагон. На истата шина во спротивна насока поставена е новата локомотива на која ќе се прикачуваат вагоните од стариот воз кои се во функционална состојба (прикачувањето секогаш се врши на последниот вагон), т.е. со оваа нова локомотива ќе прави новиот воз. Нека вагоните на расипаниот воз се обележани со сериски броеви, освен оние кои се во нефункционална состојба, тие се обележани се со 0. При формирање на новиот воз треба да се внимава вагоните да бидат сортирани според сериските броеви и тоа во опаѓачки редослед, гледајќи од страна на локомотивата.

За да се изврши прераспределба на вагоните се користи една помошна кружна шина (паралелна на онаа кај што се поставени стариот и новиот воз). На оваа помошна кружна шина вагоните се вадат по истиот редослед по кој се додаваат. Вагоните кои излегуваат од оваа помошна шина може или да се прикачуваат на новиот воз на крај, или на стариот воз на крај или на крајот на самата шина (т.е. да се передат од почеток на крај на истата шина зошто е кружна). Вагоните од стариот воз (кои се откачуваат) може да се прикачуваат на крај на помошната шина или на крај на новиот воз. Истото важи и за вагоните на новиот воз. Ваша задача е да направите алгоритам кој ќе го формира новиот воз со вагони во функционална состојба.

Влез: Во влезот е даден прво се вкупниот број на вагони на расипаниот воз. Следно се дава во секој нареден ред соодветно сериските броеви на вагоните од расипаниот воз.

Излез: На излез треба да се испечати состојбата на новиот воз со сериските броеви (почнувајќи од последниот вагон).

Пример:

Влез:

30
55
100
44
33
0
0
22

5
11
8
60
4
21
90
12
56
108
404
3
0
0
22
0
110
0
6
0
17
0
71

Излез:

3 4 5 6 8 11 12 17 21 22 22 33 44 55 56 60 71 90 100 108 110 404

Решение

Во дадената задача треба да се имплементира прераспределување на вагони и создавање на нов воз. За да се одговори на барањата од задачата мора да се искористат два стека: starVoz - стек кој содржи вагоните на стариот воз (влезен параметар) и novVoz - стек кој ќе ги содржи вагоните на новиот воз (може било која имплементација на стек да се искористи), како и една редица shina - редица која се користи како временска редица за вагоните кои треба да бидат преместени од еден стек во друг. Од влезните информации кои содржат информации за еден вагон се создава листа која се превртува и додава во стекот starVoz. МОра да се изврши прераспределба на вагоните според следниве правила: Прво се преместуваат сите ненултови вагони од starVoz во novVoz. Ако има последователни нули во starVoz, тие се изоставуваат. Доколку има нули вагон во starVoz, се

преместуваат сите последователни нулти вагони во shina. Се извршува прераспределба на вагоните во starVoz и shina така што се осигура дека вагоните во novVoz се во опаѓачки редослед. Бидејќи, во процесот на преместување, потребно е да се сортираат вагоните според нивните броеви се користи структурата за редица shina, за да се изврши сортирањето на вагоните. Редицата осигурува правилно сортирање на вагоните пред да бидат додадени во новиот воз. На крајот, сите вагони се преместуваат од novVoz и печатат како резултат.

Имплементацијата на решението е дадена подолу. Резултатот од примерен влез се печати на излез во главниот main метод.

```

1 //Вметни класа Stack
2 //Вметни класа ArrayStack
3 //Вметни класа Queue
4 //Вметни класа ArrayQueue
5
6 public class Vozovi {
7     public static String preraspredeli(String[] vlez) {
8         ArrayStack<Integer> starVoz = new ArrayStack<Integer>(vlez.length);
9         ArrayStack<Integer> novVoz = new ArrayStack<Integer>(vlez.length);
10        ArrayQueue<Integer> shina = new ArrayQueue<Integer>(vlez.length);
11        int top;
12        String out = new String("");
13
14        for(int i=0; i<vlez.length; i++)
15            starVoz.push(Integer.parseInt(vlez[i]));
16        while(true){
17            if(!starVoz.isEmpty()&&starVoz.peek()!=0)
18                novVoz.push(starVoz.pop());
19            else{
20                while(!starVoz.isEmpty() && starVoz.peek()==0)
21                    starVoz.pop();
22                if(starVoz.isEmpty())
23                    out+="Site vagoni se rasipani";
24                else
25                    novVoz.push(starVoz.pop());
26            }
27
28            while(!starVoz.isEmpty()){
29                while(!starVoz.isEmpty() && starVoz.peek()==0){
30                    starVoz.pop();
31                }
32                if (starVoz.isEmpty())

```

```

33             break;
34         else{
35             top = starVoz.pop();
36             if (novVoz.peek() < top){
37                 shina.enqueue(novVoz.pop());
38                 novVoz.push(top);
39             }
40             else
41                 shina.enqueue(top);
42         }
43     }
44     if(!shina.isEmpty())novVoz.push(shina.dequeue());
45     while(!shina.isEmpty()){
46         top = shina.dequeue();
47         if (novVoz.peek() < top){
48             starVoz.push(novVoz.pop());
49             novVoz.push(top);
50         }
51         else
52             starVoz.push(top);
53     }
54     if (starVoz.isEmpty() && shina.isEmpty()) break;
55 }
56
57 while(!novVoz.isEmpty()){
58     out+=novVoz.pop()+" ";
59 }
60
61 return out;
62 }

63
64 public static void main (String[] args) throws IOException {
65     BufferedReader br = new BufferedReader(new
66         InputStreamReader(System.in));
67     String s = br.readLine();
68     int n = Integer.parseInt(s);
69     String[] vlez = new String[n];
70
71     for(int i=0;i<n;i++){
72         vlez[i] = br.readLine();
73     }

```

```

73
74     System.out.println(preraspredeli(vlez));
75     br.close();
76 }
77 }
```

4.3.6 Напредни проблеми со редица

Задача 1. Колоквиум

Се организира прв колоквиум по предметот Алгоритми и структури на податоци. За таа цел се отвара анкета по предметот на која студентите се пријавуваат. Анкетата има дадено 2 избори:

- 1) Полагам во било кој термин
- 2) Испитот ми се преклопува со Математика

Студентите се поставуваат во термините според редоследите во кои се применени (почнувајќи од првиот). Сите студенти сакаат да полагаат колку е можно порано па затоа дел од студентите мамат и во анкетата наведуваат дека истиот ден полагаат и Математика. Асистентите бараат список на студенти кои полагаат Математика и добиваат. Потоа се започнува со распределба на студентите во термини: прво во термините се доделуваат студентите кои се пријавиле дека полагаат и Математика (по редоследот по кој се пријавиле), меѓутоа секој од овие студенти се проверува дали навистина полага и Математика и ако мамел се сместува на крај од списокот на студенти кои избрале дека полагаат било кој термин. Потоа се изминуваат останатите студенти и се доделуваат во термини.

Влез: Во влезот е даден прво капацитетот на студенти по термин (т.е. по колку студенти во еден термин може да полагаат). Следно се дава бројот и списокот на студенти кои истиот ден полагаат и Математика (според редоследот по кој се пријавиле). Потоа се дава бројот и списокот на останатите студенти (според редоследот по кој се пријавиле). На крај се дава број и список на студенти кои навистина полагаат Математика.

Излез: На излез треба да се испечати број на термин, па студентите кои полагаат во тој термин.

Пример:

Влез:

2

4

IlinkaIvanoska

IgorKulev

MagdalenaKostoska

HristinaMihajloska

3

VladimirTrajkovik

SlobodanKalajdziski

AnastasMisev

1

IlinkaIvanoska

Излез: 1

IlinkaIvanoska

VladimirTrajkovik

2

SlobodanKalajdziski

AnastasMisev

3

IgorKulev

MagdalenaKostoska

4

HristinaMihajloska

Решение

Во дадената задача треба да се имплементира распределба на полагање на студенти на прв колоквиум по предметот Алгоритми и структури на податоци по термини. За да се одговори на барањата од задачата мора да се искористат 2 податочни структури редица (redMath и redOstanati) за да се чуваат студентите соодветно според тоа што се пријавиле дека полагаат (АПС и Математика, или пак само АПС) (може било која имплементација на редица да се искористи). На влезот кој се обработува дадени се бројот на студенти за распределба по термин, број на студенти кои исто така полагаат и Математика во истиот термин како и колоквиумот по АПС, како и нивните имиња кои се ставаат во соодветната редица, па бројот на студенти кои полагаат само колоквиум по АПС, без Математика, како и нивните имиња кои со кои се пони втората редица. Потоа, се дава бројот на студенти кои вистински полагаат Математика, како и нивните имиња за кои мора да се искористи дополнителна податочна структура да се зачуваат како листа listRealMath. Распределба на студентите се прави според следните услови: Студентите од redMath се проверува дали се вистински пријавени за Математика, и ако не се во listRealMath, тогаш се преместуваат во redOstanati. Потоа, студентите се распределуваат во термините. Во секој термин се додаваат студен-

ти онолку колку е дозволено, ако има достапни студенти во redMath. Ако нема повеќе студенти во redMath, се користи redOstanati за дополнително пополнување на термините. Накрај, за секој термин се прикажува листата на студенти кои се пријавиле за тој термин.

Имплементацијата на решението е дадена подолу. Резултатот од примерен влез се печати на излез во главниот main метод.

```

1 import java.util.NoSuchElementException;
2 import java.io.BufferedReader;
3 import java.io.IOException;
4 import java.io.InputStreamReader;
5 import java.util.*;
6
7 //Вметни класа Queue
8 //Вметни класа ArrayQueue
9
10 public class Kolokvium {
11     public static void main (String[] args) throws IOException {
12         BufferedReader br = new BufferedReader(new
13             InputStreamReader(System.in));
14         String vlez;
15
16         ArrayQueue<String> redMath = new ArrayQueue<String>(100);
17         ArrayQueue<String> redOstanati = new ArrayQueue<String>(100);
18         LinkedList<String> listRealMath = new LinkedList<String>();
19
20         int i;
21         int brStudentiTermin, brStudentiMath, brOstanati, brRealMath;
22
23         brStudentiTermin = Integer.parseInt(br.readLine());
24         brStudentiMath = Integer.parseInt(br.readLine());
25         for (i = 0; i<brStudentiMath; i++)
26         {
27             vlez = br.readLine();
28             redMath.enqueue(vlez);
29         }
30
31         brOstanati = Integer.parseInt(br.readLine());
32         for (i = 0; i<brOstanati; i++)
33         {
34             vlez = br.readLine();
35             redOstanati.enqueue(vlez);
36         }
37
38         System.out.println("Red Math: " + redMath);
39         System.out.println("Red Ostanati: " + redOstanati);
40         System.out.println("List Real Math: " + listRealMath);
41     }
42 }
```

```
35 }
36
37     brRealMath = Integer.parseInt(br.readLine());
38     for (i = 0; i<brRealMath; i++)
39     {
40         vlez = br.readLine();
41         listRealMath.add(vlez);
42     }
43
44     String elem;
45     int t = 1;
46
47     while (!redMath.isEmpty())
48     {
49         System.out.println(t);
50         for (i = 0; i < brStudentiTermin;)
51         {
52             if (!redMath.isEmpty())
53             {
54                 elem = redMath.peek();
55                 if (!listRealMath.contains(elem))
56                     red0stanati.enqueue(redMath.dequeue());
57                 else {
58                     elem = redMath.dequeue();
59                     i++;
60                     System.out.println(elem);
61                 }
62             }
63             else if (!red0stanati.isEmpty())
64             {
65                 elem = red0stanati.dequeue();
66                 i++;
67                 System.out.println(elem);
68             }
69             else break;
70         }
71         t++;
72         if (redMath.isEmpty())
73             break;
74     }
75     if (redMath.isEmpty())
```

```

76     {
77         while (!red0stanati.isEmpty())
78         {
79             System.out.println(t);
80             for (i = 0; i < brStudentiTerenin;)
81             {
82                 if (!red0stanati.isEmpty())
83                 {
84                     elem = red0stanati.dequeue();
85                     i++;
86                     System.out.println(elem);
87                 }
88                 else break;
89             }
90             t++;
91         }
92     }
93 }
94 }
```

Задача 2. Консултации

Кај асистентот Игор се одржуваат консултации по два предмети АСП и MMC во еден термин. Бидејќи по АПС колоквиумот е следниот ден, Игор им рекол на студентите кои што чекаат дека прво ќе ги услужи студентите по АПС, а после студентите по MMC. Студентите се подготвиле со прашања и прашањата за АПС можат да бидат од тип A, B, C или D. Асистентот им напоменал на студентите по АПС, ако дојде некој студент и праша прашање од тип X (X е A,B,C или D) и веднаш после него дојде студент со прашање од тип X (т.е. со прашање од ист тип), вториот студент ќе биде ставен на крајот од редот и истовремено ќе биде пуштен еден студент од другата редица за MMC (ако таа редица не е празна). Генерално, ако последното одговорено прашање по АПС е од тип X, и дојде студент со прашање од тип X, тој се преместува на крајот од редот и се пушта еден студент од другата редица за MMC (ако таа редица не е празна). Кој ќе биде конечниот распоред за влегување?

Влез: Во влезот е даден прво бројот на студенти кои се пријавиле за консултации АПС, а потоа се наведуваат студентите според редоследот на пријавување и се дава за кој тип прашање се пријавиле (A, B, C или D). Следно се дава бројот на студенти кои се пријавиле за консултации MMC, а потоа се наведуваат студентите според редоследот на пријавување.

Излез: На излез треба да се испечатат студентите според редоследот по кој влегле на консултации.

Пример:

Влез:

3

IlinkaIvanoska A

MagdalenaKostoska A

HristinaMihajloska B

1

IgorKulev

Излез:

IlinkaIvanoska

IgorKulev

HristinaMihajloska

MagdalenaKostoska

Решение

Во дадената задача треба да се имплементира распоред по кој студенти ќе влегуваат на консултации, по два предмета АПС и MMC во ист термин. Редоследот на студенти треба да е според типот на прашање. За да се одговори на барањата од задачата мора да се искористат 3 податочни структури редица (redAPS, redTip и redMMS) за да се чуваат студентите кои се за АПС, нивниот тип на прашање и студентите кои се за MMC (може било која имплементација на редица да се искористи). Од дадениот влез прво се чита број на студенти кои имаат прашања за АПС, како и нивните имиња и тип на прашање, со што се додаваат информациите во редиците redAPS и redTip, соодветно. Потоа, се внесува бројот на студенти за MMC, со нивните имиња за додавање во редицата redMMS. За секој студент за АПС треба да се спореди типот на неговото прашање со претходното од типот за АПС. Ако типот на прашање е ист како и претходниот, студентот се додава во крајот на редицата redAPS и ако има студенти од redMMS, тогаш се печати првиот студент од redMMS. Ако типот на прашање е различен од претходниот, студентот се печати и типот се ажурира со новиот тип на прашање. На крај, сите студенти останати од redMMS треба да се испечатат.

Имплементацијата на решението е дадена подолу. Резултатот од примерен влез се печати на излез во главниот main метод.

```

1 import java.util.NoSuchElementException;
2 import java.io.BufferedReader;
3 import java.io.IOException;
```

```
4 import java.io.InputStreamReader;
5
6 //Вметни класа Queue
7 //Вметни класа ArrayQueue
8
9 public class Konsultacii {
10     public static void main (String[] args) throws IOException {
11
12         BufferedReader br = new BufferedReader(new
13             InputStreamReader(System.in));
14         String vlez;
15
16         ArrayQueue<String> redAPS = new ArrayQueue<String>(100);
17         ArrayQueue<String> redTip = new ArrayQueue<String>(100);
18         ArrayQueue<String> redMMS = new ArrayQueue<String>(100);
19
20         int i;
21         int brStudentiAPS, brStudentiMMS;
22
23         brStudentiAPS = Integer.parseInt(br.readLine());
24         for (i = 0; i<brStudentiAPS; i++)
25         {
26             vlez = br.readLine();
27             String[] pom = vlez.split(" ");
28             redAPS.enqueue(pom[0]);
29             redTip.enqueue(pom[1]);
30         }
31
32         brStudentiMMS = Integer.parseInt(br.readLine());
33         for (i = 0; i<brStudentiMMS; i++)
34         {
35             vlez = br.readLine();
36             redMMS.enqueue(vlez);
37         }
38
39         String pom, pomTip, tip="";
40         if(!redAPS.isEmpty())
41         {
42             pom = redAPS.dequeue();
43             System.out.println(pom);
44             tip = redTip.dequeue();
```

```
44         i++;
45     }
46
47     while(!redAPS.isEmpty())
48     {
49         pom = redAPS.dequeue();
50         pomTip = redTip.dequeue();
51         if(tip.equals(pomTip))
52         {
53             redAPS.enqueue(pom);
54             redTip.enqueue(pomTip);
55             if(!redMMS.isEmpty())
56             {
57                 pom = redMMS.dequeue();
58                 System.out.println(pom);
59             }
60         }
61         else
62         {
63             System.out.println(pom);
64             tip=pomTip;
65         }
66
67     }
68     while(!redMMS.isEmpty())
69     {
70         pom = redMMS.dequeue();
71         System.out.println(pom);
72     }
73 }
74 }
```

Задача 3. Организација на испит

Треба да се организира испитот по еден предмет на ФИНКИ. Поради тоа што полагањето се состои од теоретски дел (е-тест) и практичен дел (задачи на компјутер), а не сите студенти ги полагаат и двата дела, асистентите објавиле анкета на курсот за да се пријавуваат студентите за тоа што ќе полагаат. На анкетата ги имаат следните избори:

1. Полагам само е-тест
2. Полагам само задачи

3. Полагам и е-тест и задачи

Испитот се одржува во лаб. З каде што капацитетот на лабораторијата е 20 места. Студентите се примаат да полагаат според следниот редослед: прво се полни лабораторијата според редоследот по кој се пријавиле студентите кои ќе полагаат само е-тест. Ако лабораторијата не се исполнi од овие студенти се пуштаат студенти кои пријавиле и теорија и писмено, ама овој пат да полагаат само е-тест. Откако овие студенти ќе завршат со полагање ја напуштаат лабораторијата и само оние кои ги пријавиле двета дела застануваат на крај на редицата која чека за задачи. Потоа се почнува кон спроведување на полагањето на задачите и тоа во оној редослед како што се пријавиле студентите на анкетата. Кој ќе биде конечниот редослед на полагање на студентите за е-тест, а кој за задачи?

Влез: Во влезот е даден прво бројот на студенти кои се пријавиле само за е-тест, а потоа се наведуваат студентите според редоследот на пријавување за е-тест, потоа истото за студентите кои се пријавиле само за задачи, па на крај студентите кои се пријавиле и за двете.

Излез: На излез треба да се испечатат студентите според редоследот по кој влегле да полагаат прво за е-тест, потоа за задачи, соодветно по термини (еден термин има 20 студенти).

Пример:

Влез:

5

Ристовска Моника

Ристова Ивана

Николчев Иван

Мановска Адријана

Буразер Маријан

5

Стојменовска Емилија

Пановски Ангелче

Трајковска Елена

Србиноска Ивана

Битирнова Лусијана

5

Милошевски Дамјан

Павловик Вуксан

Лупши Беса

Рајчин Мартин

Поповски Александар

Излез:

Polagaat e-test:
termin 1
РистовскаМоника
РистоваИвана
НиколчевИван
МановскаАдријана
БуразерMariјан
МилошевскиДамјан
ПавловикВуксан
ЛушиБеса
РајчинMartin
ПоповскиАлександар
Polagaat zadaci:
termin 1
СтојменовскаЕмилија
ПановскиAngelче
ТрајковскаЕлена
СрбиноскаИвана
БитирноваLусијана
МилошевскиДамјан
ПавловикВуксан
ЛушиБеса
РајчинMartin
ПоповскиАлександар

Решение

Во дадената задача треба да се имплементира организација на полагање на студенти на е-тестови и задачи во рамките на дадени термини. Секој студент може да полага е-тест, задачи или и двата видови на испити. Студентите кои полагаат е-тест и задачи треба да бидат распределени во термини. За да се одговори на барањата од задачата мора да се искористат 3 податочни структури редица (redEtest, redZadachi и redEtestZadachi) за да се чуваат студентите соодветно според тоа што полагаат (може било која имплементација на редица да се искористи). Од дадениот влез прво се чита колку студенти полагаат е-тест, а потоа се читаат нивните имиња и се додаваат во редицата redEtest. Потоа се читаат колку студенти полагаат задачи, а потоа се читаат нивните имиња и се додаваат во редицата redZadaci. На крај, се читаат колку студенти полагаат и е-тест и задачи, а потоа се читаат нивните имиња и се додаваат во редицата redEtestZadaci.

Прво се одредуваат термините за полагање на е-тест (termin 1, termin 2, итн.). Секој термин може да има најмногу 20 студенти. Потоа се врши распределба на студентите кои полагаат и е-тест и задачи. Прво полагаат студентите кои полагаат и двата видови на испити. Се врши проверка дали има достапни студенти во редицата redEtest, а ако не, се земаат студенти од редицата redEtestZadaci и се додаваат во редицата redZadaci. По ова, следува распределбата на студентите кои полагаат само е-тест и студентите кои полагаат само задачи. На крај, се печати во кој термин кои студенти полагаат, прво за студентите кои полагаат е-тест и потоа за студентите кои полагаат задачи.

Имплементацијата на решението е дадена подолу. Резултатот од примерен влез се печати на излез во главниот main метод.

```

1 import java.util.NoSuchElementException;
2 import java.io.BufferedReader;
3 import java.io.IOException;
4 import java.io.InputStreamReader;
5
6 //Вметни класа Queue
7 //Вметни класа ArrayQueue
8
9 public class Polaganje {
10     public static void main (String[] args) throws IOException {
11
12         BufferedReader br = new BufferedReader(new
13             InputStreamReader(System.in));
14         String vlez;
15
16         ArrayQueue<String> redEtest = new ArrayQueue<String>(100);
17         ArrayQueue<String> redZadaci = new ArrayQueue<String>(100);
18         ArrayQueue<String> redEtestZadaci = new ArrayQueue<String>(100);
19
20         int i;
21         int brStudentiEtest, brStudentiZadaci, brStudentiEtestZadaci;
22
23         brStudentiEtest = Integer.parseInt(br.readLine());
24         for (i = 0; i<brStudentiEtest; i++)
25         {
26             vlez = br.readLine();
27             redEtest.enqueue(vlez);
28         }

```

```
29     brStudentiZadaci = Integer.parseInt(br.readLine());
30     for (i = 0; i<brStudentiZadaci; i++)
31     {
32         vlez = br.readLine();
33         redZadaci.enqueue(vlez);
34     }
35
36     brStudentiEtestZadaci = Integer.parseInt(br.readLine());
37     for (i = 0; i<brStudentiEtestZadaci; i++)
38     {
39         vlez = br.readLine();
40         redEtestZadaci.enqueue(vlez);
41     }
42
43     String elem;
44     int t = 1;
45     System.out.println("Polagaat e-test:");
46     while (!redEtest.isEmpty())
47     {
48         System.out.println("termin "+ t);
49         for (i = 0; i < 20;)
50         {
51             if (!redEtest.isEmpty())
52             {
53                 //elem = redEtest.peek();
54                 elem = redEtest.dequeue();
55                 i++;
56                 System.out.println(elem);
57             }
58             else if (!redEtestZadaci.isEmpty())
59             {
60                 elem = redEtestZadaci.dequeue();
61                 i++;
62                 System.out.println(elem);
63                 redZadaci.enqueue(elem);
64             }
65             else break;
66         }
67         t++;
68         if (redEtest.isEmpty())
69             break;
```

```
70 }
71 if (redEtest.isEmpty())
72 {
73     while (!redEtestZadaci.isEmpty())
74     {
75         System.out.println("termin "+t);
76         for (i = 0; i < 20;)
77         {
78             if (!redEtestZadaci.isEmpty())
79             {
80                 elem = redEtestZadaci.dequeue();
81                 i++;
82                 System.out.println(elem);
83                 redZadaci.enqueue(elem);
84             }
85             else break;
86         }
87         t++;
88     }
89 }
90 t = 1;
91 System.out.println("Polagaat zadaci:");
92 if (redEtestZadaci.isEmpty())
93 while (!redZadaci.isEmpty())
94 {
95     System.out.println("termin "+t);
96     for (i = 0; i < 20;)
97     {
98         if (!redZadaci.isEmpty())
99         {
100            elem = redZadaci.dequeue();
101            i++;
102            System.out.println(elem);
103        }
104        else break;
105    }
106    t++;
107 }
108 }
109 }
```

4.3.7 Задачи за вежбање

Задача 1. Колоквиумска недела

Се организира колоквиумска недела на ФИНКИ и за таа цел асистентите се доделуваат за чување на испити. За таа цел се прави редица од асистентите во која на почеток се најмладите асистенти, а на крај се највозрасните. Потоа се даваат предметите и по колку асистенти се потребни за чување на секој предмет. Има некои асистенти кои се отсутни во тековната колоквиумска недела. Затоа дополнително се дава список кои од асистентите се отсутни. Асистентите се доделуваат на следниот начин: Прво се доделуваат најмладите, а на крај најстарите, со тоа што ако некој асистент е отсутен тој се игнорира во редицата. После секое доделување на предмет асистентот се сместува на крај на редицата (т.е. ако на секој асистент му се доделил за чување предмет, а има потреба од уште асистенти, повторно се започнува од најмладите).

Влез: Во влезот е даден прво бројот на асистенти и имињата на асистентите од најмлад до најстар. Следно се дава бројот на предмети за кои се потребни асистенти, па се наведуваат предметите и по колку асистенти се потребни за секој предмет. Потоа се дава бројот на асистенти кои се отсутни и списокот на тековно отсутните асистенти.

Излез: На излез треба да се испечати предмет, па асистенти задолжени за чување на тој предмет (за секој од дадените предмети).

Пример:

Влез:

4

IlinkaIvanoska

IgorKulev

MagdalenaKostoska

HristinaMihajloska

3

APS 3

MIS 1

OOS 2

1

HristinaMihajloska

Излез:

APS

3

IlinkaIvanoska

IgorKulev

MagdalenaKostoska

MIS

1

IlinkaIvanoska

OOS

2

IgorKulev

MagdalenaKostoska

Задача 2. Консултации

Пред да започне колоквиумската недела на ФИНКИ се организираат консултации по предметот Алгоритми и структури на податоци. Бидејќи има голем број на заинтересирани студенти за консултации се објавува анкета на курсот за да се пријават студентите и тоа има два избора на анкетата (може да се изберат и двета):

- 1) Имам кратки прашања
- 2) Ми треба објаснување за некои задачи
- 3) И кратки прашања и објаснување за задачи

Асистентката Магдалена ги држи консултациите. Студентите се примаат на консултации според следниот редослед: прво се примаат по еден студент од оние кои имаат кратки прашања според редоследот по кој се пријавиле. Ако нема еден од овие се пушта студенти кои кои имаат прашања и за задачи за да се исполнит квотата од 1 студент, ама овој прашува само кратки прашања. Ако се пуштил студент кој има и прашања за задачи тој се преместува на крај на редицата за задачи. Откако ќе се заврши овој студент со кратки прашања, се продолжува со оние кои имаат нејасни задачи. Од овие студенти се прима 1. Ако нема еден од овие се пушта студент кој има прашања и за задачи за да се исполнит квотата од 1 студент за задачи, ама овој прашува само за задачи, и потоа се преместува на крајот на редицата за кратки прашања. Понатаму се продолжува на истиот начин со тоа што за студентите кои се пријавиле и за задачи и за прашања влегуваат откако ќе се испразни редот со само кратки прашања или само со задачи. Студентите кои се пријавиле и за прашања и задачи, откако ќе завршат со прашањата се преместуваат на крај на редицата за задачи, и обратно. Кој ќе биде конечниот редослед на влегување?

Влез: Во влезот е даден прво бројот на студенти кои се пријавиле за кратки прашања, а потоа се наведуваат студентите според редоследот на пријавување за кратки прашања, потоа истото за студентите кои се пријавиле само за задачи, па на крај студентите кои се пријавиле и за двете.

Излез: На излез треба да се испечатат студентите според редоследот по кој влегле на консултации.

Пример:

Влез:

4

IlinkaIvanoska

IgorKulev

MagdalenaKostoska

HristinaMihajloska

2

AnastasMishev

VladimirTrajkovik

1

SlobodanKalajdziski

Излез:

IlinkaIvanoska

AnastasMishev

IgorKulev

VladimirTrajkovik

MagdalenaKostoska

SlobodanKalajdziski

HristinaMihajloska

SlobodanKalajdziski

Задача 3. Испит

За да се организира августовскиот испит по предметот Алгоритми и податочни структури, треба да се спроведе анкета. Во анкетата се дадени 2 избори:

- 1) Полагам само АПС
- 2) Полагам и АПС и Математика

Сите студенти сакаат да полагаат колку е можно порано, па затоа дел од студентите лажат и во анкетата наведуваат дека истиот ден полагаат и АПС и Математика. Асистентите го имаат вистинскиот список на студенти кои во истиот ден полагаат и АПС и Математика. Распределбата на студентите во термините се прави редоследно според следните чекори:

1. Најнапред се распределуваат студентите кои во анкетата се пријавиле дека полагаат и АПС и Математика и тоа е навистина така (студентите не лажеле). Редоследот по кој овие студенти се додаваат во термините е идентичен со редоследот по кој ја пополните анкетата.

2. Следно во термините се распределуваат студентите кои избрале дека по-лагают само АПС, по редослед идентичен со редоследот по кој ја пополните анкетата.

3. На крај се додаваат оние студенти кои лажеле дека полага и АПС и Математика, по редослед идентичен со редоследот по кој ја пополните анкетата.

Влез: Во влезот е даден прво капацитетот на студенти по термин (т.е. бројот на студенти што може да полагаат во еден термин). Следно се дава бројот и списокот на студенти кои на анкетата пополните дека истиот ден полагаат и АПС и Математика (според редоследот по кој се пријавиле). Потоа се дава бројот и списокот на останатите студенти (според редоследот по кој се пријавиле). На крај се дава број и список на студенти кои во истиот ден, навистина полагаат и АПС и Математика.

Излез: На излез треба да се испечати број на термин, па студентите кои полагаат во тој термин.

Пример:

Влез:

2

4

IlinkaIvanoska

IgorKulev

MagdalenaKostoska

HristinaMihajloska

3

VladimirTrajkovik

SlobodanKalajdziski

AnastasMisev

1

IlinkaIvanoska

Излез:

1

IlinkaIvanoska

VladimirTrajkovik

2

SlobodanKalajdziski

AnastasMisev

3

IgorKulev

MagdalenaKostoska

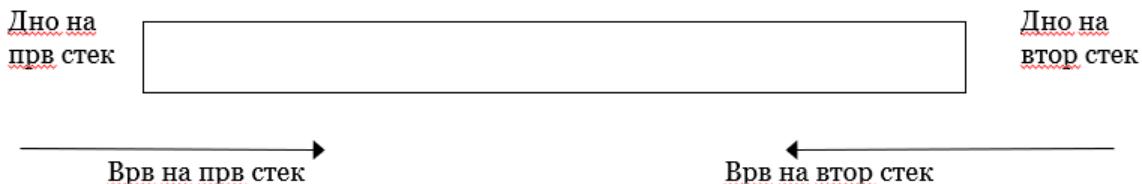
4

HristinaMihajloska

4.3.8 Проблеми со комбинирање на стек и редица

Задача 1. Два стека во еден

Да се изменат функциите за стек (магазин) така да овозможат користење на два стека кои делат заеднички простор, односно поле. Следната слика 4-10 ја покажува логичката структура на ваквите стекови.



Слика 4-10: Приказ на структурата.

Влез: Во влезот е дадена големината на структурата.

Излез: На излез треба да се испечати изгледот на структурата по дадена операција.

Решение

Во дадената задача треба да се имплементира структурата за двоен стек (DoubleArrayStack) користејќи низа за зачувување на елементите. Двоен стек е структура која овозможува зачувување на два стека во една низа. Оваа имплементација на двоен стек е корисна за ситуации кога треба да се употребат два стека, а треба да се заптеди меморија. Секој стек има своја независна длабочина и операции за додавање и отстранување на елементи. Двојниот стек содржи методи за додавање и отстранување на елементи од првиот и вториот стек, како и методи за проверка на состојбата на стековите: isEmptyFirst() е метод кој проверува дали првиот стек е празен; isEmptySecond() е метод кој проверува дали вториот стек е празен; isFull() е метод кој проверува дали целата низа (обединети прв и втор стек заедно) е полна; peekFirst() е метод кој го враќа елементот на врвот на првиот стек без да го отстрани; peekSecond() е метод кој го враќа елементот на врвот на вториот стек без да го отстрани; clearFirst() е метод кој ја брише содржината на првиот стек; clearSecond() е метод кој ја брише содржината на вториот стек; pushFirst(E x) е метод за додавање на елемент во првиот стек; pushSecond(E x) е метод за додавање на елемент во вториот стек; popFirst() е метод кој го отстрани и враќа елементот на врвот на првиот стек; popSecond() е метод кој го отстрани и враќа елементот на врвот на вториот

стек; и `recatiNizata()` е метод кој ги печати сите елементи во низата (обединети прв и втор стек заедно).

Имплементацијата на решението е дадена подолу. Резултатот од примерен влез се печати на излез во главниот `main` метод.

```
1 import java.util.NoSuchElementException;
2
3 public class DoubleArrayStack<E> {
4     //Стекот е претставен на следниот начин:
5     //depth1 е длабочината на првиот стек,
6     //depth2 е длабочината на вториот стек,
7     //elems[0...depth1-1] се елементи на првиот стек,
8     //elems[maxDepth-depth2...maxDepth-1] се елементи на вториот стек,
9     private E[] elems;
10    private int depth1;
11    private int depth2;
12
13    @SuppressWarnings("unchecked")
14    public DoubleArrayStack(int maxDepth) {
15        //Конструкција на нов, празен споделен стек.
16        elems = (E[]) new Object[maxDepth];
17        depth1 = 0;
18        depth2 = 0;
19    }
20
21    public boolean isEmptyFirst() {
22        //Враќа true ако и само ако првиот стек е празен.
23        return (depth1 == 0);
24    }
25
26    public boolean isEmptySecond() {
27        //Враќа true ако и само ако вториот стек е празен.
28        return (depth2 == 0);
29    }
30
31    public boolean isFull() {
32        //Враќа true ако и само ако целата низа е полна.
33        return (depth1 + depth2 == elems.length);
34    }
35
36    public E peekFirst() {
37        //Го враќа елементот на врвот од првиот стек.
```

```
38     if (depth1 == 0)
39         throw new NoSuchElementException();
40     return elems[depth1 - 1];
41 }
42
43 public E peekSecond() {
44     //Го враќа елементот на врвот од вториот стек.
45     if (depth2 == 0)
46         throw new NoSuchElementException();
47     return elems[elems.length - depth2];
48 }
49
50 public void clearFirst() {
51     //Го празни првиот стек.
52     for (int i = 0; i < depth1; i++)
53         elems[i] = null;
54     depth1 = 0;
55 }
56
57 public void clearSecond() {
58     //Го празни вториот стек.
59     for (int i = elems.length - 1; i >= elems.length - depth2; i--)
60         elems[i] = null;
61     depth2 = 0;
62 }
63
64 public void pushFirst(E x) {
65     //Го додава x на врвот на првиот стек.
66     if (!this.isFull())
67         elems[depth1++] = x;
68     else
69         System.out.println("Error, the array is full");
70 }
71
72 public void pushSecond(E x) {
73     //Го додава x на врвот на вториот стек.
74     if (!this.isFull())
75         elems[elems.length - (++depth2)] = x;
76     else
77         System.out.println("Error, the array is full");
78 }
```

```
79
80     public E popFirst() {
81         //Го отстранува и враќа елементот што е на врвот на првиот стек.
82         if (depth1 == 0)
83             throw new NoSuchElementException();
84         E topmost = elems[--depth1];
85         elems[depth1] = null;
86         return topmost;
87     }
88
89     public E popSecond() {
90         //Го отстранува и враќа елементот што е на врвот на вториот стек.
91         if (depth2 == 0)
92             throw new NoSuchElementException();
93         E topmost = elems[elems.length - depth2];
94         elems[depth2--] = null;
95         return topmost;
96     }
97
98     public String pecatiNizata() {
99         StringBuilder ret = new StringBuilder("Elementite se: ");
100        for (E elem : elems) ret.append(elem).append(" ");
101        return ret.toString();
102    }
103
104    public static void main(String[] args) {
105        DoubleArrayStack<Integer> d = new DoubleArrayStack<Integer>(6);
106        d.pushFirst(1);
107        d.pushFirst(2);
108        d.pushFirst(3);
109        d.pushSecond(-1);
110        d.pushSecond(-2);
111        d.pushSecond(-3);
112        System.out.println("Vrv na prv: " + d.peekFirst() + ", dolzina na
113                           prv: " + d.depth1);
114        System.out.println("Vrv na vtor: " + d.peekSecond() + ", dolzina na
115                           vtor: " + d.depth2);
116        d.pushFirst(4);
117        d.popFirst();
118        d.pushFirst(4);
119        System.out.println("Vrv na prv: " + d.peekFirst() + ", dolzina na
```

```

118     prv: " + d.depth1);
119     d.pecatiNizata();
120 }

```

Задача 2. РедоСтек

Да се направи имплементација за нова податочна структура RedoStek која што ќе биде ограничена структура. Новата структура треба да може да ги задоволи следните операции:

- вметнување елемент – Првиот елемент секогаш се става во редицата, а вториот на стекот. За останатите елементи важи следното: вметнувањето елемент треба да се прави со вметнување елемент во редицата или стекот, во зависност од тоа каде има помал елемент на почетокот (врвот) во моментот на вметнувањето. Доколку структурата во која треба да се вметне новиот елемент е веќе полна, тогаш елементот се вметнува во другата структура (Пр. Ако елементот треба да се вметне во редицата, а таа е полна, се вметнува во стекот). Оваа операција треба да се имплементира со сложеност $O(1)$.
- вадење елемент – Вадењето елемент од структурата се врши на тој начин што прво се вадат елементите од редицата, па после тоа од стекот. Оваа операција треба да се имплементира со сложеност $O(1)$.
- да се провери дали структурата е празна
- да се провери кој е елементот на врв на структурата – Оваа операција дава информација кој елемент е на ред за вадење.

За да се имплементираат соодветните операции може да се користат веќе постоечките имплементации за операциите на стек и редица. Поради тоа што структурата РедоСтек е ограничена, при нејзиното креирање треба да се наведе големината на стекот и големината на редицата од кои ќе биде креирана. Елементите се внесуваат во структурата во истиот редослед како што се читаат од стандардниот влез. Имате право на користење на само една ваква структура. Немате право на користење други дополнителни структури. Ваша задача е да го испечатите редоследот на вадење на сите елементи од структурата РедоСтек.

Влез: Во влезот во првиот ред е дадена големината на редицата, во вториот ред големината на стекот, во третиот ред бројот на карактери кои ќе се внесуваат, за потоа да се читаат карактерите.

Излез: На излез треба да се испечати прво елементот на врвот на структурата РедоСтек, а после тоа во нов ред се печатат сите елементи од структурата

РедоСтек во редослед како што се вадат од неа, одделени со празно место.

Пример:

Влез:

5

3

8

abcfgehd

Излез:

a

a c f g e d h b

Решение

Во дадената задача треба да се имплементира нова податочна ограничена структура RedoStek која комбинира функционалности од стек и редица. Со помош на оваа структура, може да се додаваат и отстрануваат елементи од две различни структури (редица и стек) и да се користи како една целосна структура со одредени правила за додавање на елементи во редицата или стекот во зависност од нивната содржина. За да се одговори на барањата од задачата мора да се искористат две дополнителни структури: ArrayQueue (низа-редица) и ArrayStack (низа-стек). Се искористат имплементациите на стек и редица со низа, затоа што структурата мора да е ограничена.

Имплементацијата на решението е дадена подолу. Во оваа имплементација во класата ArrayRedoStek за да се задоволат барањата на задачата се дадени: конструктор кој креира инстанца од структурата со максимален капацитет за редицата и стекот; isEmpty() метод кој проверува дали структурата е празна и враќа true ако и само ако и двата дела (редицата и стекот) се празни; peek() метод кој го враќа елементот на врвот од структурата без да го отстранува, а ако редицата е празна, ќе врати го врвот на стекот; clear() метод кој ја празни структурата; push(E x) метод кој додава елемент x кој додава елемент на структурата, и тоа ако редицата е празна, елементите се додаваат во редицата, а во спротивно, ако стекот е празен, елементите се додаваат во стекот, и ако и двата дела на структурата имаат елементи, елементите се додаваат во редицата или стекот во зависност од тоа каде има помал елемент на врвот; и pop() кој го отстранува и враќа елемент од структурата и тоа ако редицата не е празна, ќе врати и отстрани елемент од редицата, а во спротивно, ако стекот не е празен, ќе врати и отстрани елемент од стекот. Резултатот од примерен влез се печати на излез во главниот main метод.

¹ `import java.util.Collection;`

```
2 import java.util.NoSuchElementException;
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6
7 //Вметни класа Queue
8 //Вметни класа Stack
9 //Вметни класа ArrayQueue
10 //Вметни класа ArrayStack
11
12 interface RedoStek {
13     public boolean isEmpty ();
14     //Враќа true ако и само ако стекот е празен.
15
16     public Character peek ();
17     //Го враќа елементот на врвот од структурата
18
19     //Методи за трансформација:
20     public void clear ();
21     //Ја празни структурата.
22
23     public void push (Character x);
24     //Го додава x на врвот на структурата.
25
26     public Character pop ();
27     //Го отстранува и враќа елементот што е на ред за вадење.
28 };
29
30 class ArrayRedoStek implements RedoStek{
31     private ArrayQueue<Character> q;
32     private ArrayStack<Character> s;
33     private int lnRed, lnStek;
34
35     public ArrayRedoStek (int maxRed, int maxStek ) {
36         s = new ArrayStack<Character>(maxStek);
37         q = new ArrayQueue<Character>(maxRed);
38         lnStek = maxStek;
39         lnRed = maxRed;
40     }
41
42     @Override
```

```
43     public boolean isEmpty() {
44         if(q.isEmpty()&&s.isEmpty())
45             return true;
46         return false;
47     }
48
49     @Override
50     public Character peek() {
51         if(q.isEmpty())
52             return s.peek();
53         return q.peek();
54     }
55
56     @Override
57     public void clear() {
58
59     }
60
61     @Override
62     public void push(Character x) {
63         if(q.isEmpty())
64             q.enqueue(x);
65         else if(s.isEmpty())
66             s.push(x);
67         else{
68             if(s.size() < lnStek&&q.size() < lnRed){
69                 if(s.peek() < q.peek())
70                     s.push(x);
71                 else
72                     q.enqueue(x);
73             }
74             else if(s.size() < lnStek)
75                 s.push(x);
76             else if(q.size() < lnRed)
77                 q.enqueue(x);
78             else
79                 System.out.println("RedoStek is full");
80         }
81     }
82
83     @Override
```

```

84     public Character pop() {
85         while(!q.isEmpty())
86             return q.dequeue();
87         while(!s.isEmpty())
88             return s.pop();
89         return null;
90     }
91 };
92
93 public class TestRedoStek {
94     public static void main (String[] args) throws IOException {
95         BufferedReader br = new BufferedReader(new
96             InputStreamReader(System.in));
97
98         int maxRed = Integer.parseInt(br.readLine());
99         int maxStek = Integer.parseInt(br.readLine());
100
101         //ArrayStack<Integer> s = new ArrayStack(maxStek);
102         ArrayRedoStek sq = new ArrayRedoStek(maxRed, maxStek);
103
104         int brElementi = Integer.parseInt(br.readLine());
105
106         for(int i=0; i<brElementi; i++){
107             char x = (char)br.read();
108             sq.push(x);
109         }
110         System.out.println(sq.peek());
111
112         while(!sq.isEmpty()){
113             System.out.print(sq.pop()+" ");
114         }
115     }
116 }
```

Задача 3. Moj стек

Да се имплементира податочната структура стек MyStack само со користење на податочната структура ред.

Влез: Во првиот ред од влезот е даден бројот на елементи кои ќе се внесуваат во податочната структура, потоа дадени се елементите секој во посебен ред. По-

натаму се дадени редоследно акциите кои треба да се направат врз податочната структура се додека не се внесе KRAJ. Акциите се следни:

- 1 - значи извади елемент од стекот и испечати го
- 2 – додади елемент на стекот. Во наредната линија се проследува и елементот кој треба да се додаде.

Излез: На излез треба да се испечатат елементите според барањата над податочната структура. На крај треба да се испечати големината на податочната структура.

Забелешка: При реализација на задачата дозволено е да се користи само податочната структура ред. Не е дозволено да се користат дополнителни низи или листи. Да се коментираат сложеностите на методите со кои е имплементиран стекот.

Пример:

Влез:

5
1
2
3
4
5
1
1
1
1
1
KRAJ

Излез:

5
4
3
2
1

Goleminata na stekot e: 0

Решение

Во дадената задача треба да се имплементира нова податочна структура стек MyStack која ќе симулира работа на стек само со користење на податочна структура редица. За да се одговори на барањата од задачата мора да се искористат две

дополнителни структури за редица (ограничена редица во случајов ArrayQueue) и дополнителен бројач size за следење на големината на стекот.

Имплементацијата на решението е дадена подолу. Во оваа имплементација во класата MyStack за да се задоволат барањата на задачата се дадени: конструктор кој креира инстанца на структурата со иницијализација на две редици и бројач size на почетна вредност 0; isEmpty() метод кој проверува дали стекот е празен и враќа true ако и само ако и двете редици се празни; peek() метод кој го враќа елементот на врвот од стекот без да го отстранува, притоа го враќа првиот елемент од првата редица, а ако таа е празна, враќа null; getSize() метод кој ја враќа големината (бројот на елементи) на стекот; clear() метод кој ја брише содржината на стекот; push(E x) метод кој додава елемент на врвот на стекот, т.е. го додава елементот x во празната втора редица и потоа ги трансферира сите елементи од првата во втората, со што првата и втората редица се заменуваат, така што втората станува основната редица за стекот; pop() метод кој отстранува и враќа елемент од врвот на стекот, т.е. од првиот елемент од првата редицата. Резултатот од примерен влез се печати на излез во главниот main метод.

```

1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 import java.util.NoSuchElementException;
5
6 //Вметни класа Queue
7 //Вметни класа ArrayQueue
8
9 class MyStack<E> implements Stack<E>{
10     ArrayQueue<E> q1, q2;
11     int size;
12
13     public MyStack() {
14         q1 = new ArrayQueue<E>(1000);
15         q2 = new ArrayQueue<E>(1000);
16         size = 0;
17     }
18     @Override
19     public boolean isEmpty() {
20         if(q1.isEmpty()&&q2.isEmpty())
21             return true;
22         else
23             return false;
24     }

```

```
25
26     @Override
27     public E peek() {
28         if (q1.isEmpty())
29             return null;
30         return q1.peek();
31     }
32
33     public int getSize(){
34         return size;
35     }
36
37     @Override
38     public void clear() {
39
40     }
41
42     @Override
43     public void push(E x) {
44         size++;
45
46         //Push x прво во празната q2
47         q2.enqueue(x);
48
49         //Push на сите останати
50         //елементи од q1 во q2.
51         while (!q1.isEmpty())
52         {
53             q2.enqueue(q1.peek());
54             q1.dequeue();
55         }
56
57         //промена на двете редици
58         ArrayQueue<E> q = q1;
59         q1 = q2;
60         q2 = q;
61     }
62
63     @Override
64     public E pop() {
65         //ако нема елементи во q1
```

```
66     if (q1.isEmpty())
67         return null;
68     E tmp = q1.dequeue();
69     size--;
70     return tmp;
71 }
72 }
73
74 public class MyStackTest {
75     public static void main(String[] args) throws NumberFormatException,
76             IOException {
77         BufferedReader br = new BufferedReader(new
78             InputStreamReader(System.in));
79         MyStack ob = new MyStack();
80
81         int N = Integer.parseInt(br.readLine());
82
83         for(int i=0; i<N; i++){
84             int x = Integer.parseInt(br.readLine());
85             ob.push(x);
86         }
87         //System.out.println(ob.toString());
88         String str = br.readLine();
89         while(!str.equals("KRAJ")){
90             int x = Integer.parseInt(str);
91             if(x==1){
92                 if(ob.peek()==null)
93                     System.out.println("Prazen stek");
94                 else
95                     System.out.println(ob.pop());
96             }
97             if(x==2){
98                 str = br.readLine();
99                 x = Integer.parseInt(str);
100                ob.push(x);
101            }
102            str = br.readLine();
103        }
104    }
105 }
```

4.3.9 Задачи за вежбање

Задача 1. СтекоРед

Да се направи имплементација за нова податочна структура StekoRed која што ќе биде ограничена структура. Новата структура треба да може да ги задоволи следните операции:

- вметнување елемент – Вметнувањето елемент треба да се прави со вметнување елемент во стек. Доколку стекот е полн и нема повеќе место, се продолжува со вметнување на елементите во редицата. Оваа операција треба да се имплементира со сложеност $O(1)$.
- вадење елемент – Вадењето елемент од структурата може да биде вадење на елемент од стек или од редица. Треба да се извади поголемиот елемент, при споредба на елементите на врвот на стекот и почетокот на редицата. Доколку една од структурите се испразни, вадењето продолжува по редослед на елементите од другата структура. Оваа операција треба да се имплементира со сложеност $O(1)$.
- да се провери дали структурата е празна
- да се провери кој е елементот на врв на структурата – Оваа операција дава информација кој елемент е на ред за вадење.

За да се имплементираат соодветните операции може да се користат веќе постоечките имплементации за операциите на стек и редица. Поради тоа што структурата СтекоРед е ограничена, при нејзиното креирање треба да се наведе големината на стекот и големината на редицата од кои ќе биде креирана. Елементите се внесуваат во структурата во истиот редослед како што се читаат од стандардниот влез. Имате право на користење на само една ваква структура. Немате право на користење други дополнителни структури. Ваша задача е да го испечатите редоследот на вадење на сите елементи од структурата СтекоРед.

Влез: Во првиот ред од влезот е дадена големината на стекот, во вториот ред големината на редицата, во третиот ред бројот на елементи кои ќе се внесуваат, за потоа во секој нов ред да се елементите кои се читаат.

Излез: На излез треба да се испечати елементот на врвот на структурата СтекоРед, а после тоа во нов ред се печатат сите елементи од структурата СтекоРед во редослед како што се вадат од неа, одделени со празно место.

Пример:

Влез:

3 (големина на стекот)

5 (големина на редицата)

8 (број на елементи кои треба да се прочитаат)

0 (елемент 0)

2 (елемент 1) итн.

4

1

3

5

7

9

Излез:

4 (елементот на врвот на структурата)

4 2 1 3 5 7 9 0 (сите елементи од структурата СтекоРед)

Задача 2. Сортирана редица

Дадена е влезна редица чии елементи се првите n природни броеви (генериирани по случаен редослед). Ваши задача е да проверите дали елементите од влезната редица може да се сортираат во растечки редослед во друга излезна редица, притоа користејќи ја податочната структура стек.

Дозволени операции се само следниве:

1. Кај влезната редица може да се применуваат само операциите `dequeue` и `peek`.
2. Кај магацинот може да се применуваат стандардните операции за магацин (`push`, `pop`, `peek`).
3. Кај излезната редица може да се применуваат само операциите `enqueue` и `peek`.

Влез: Во влезот во првиот ред е даден бројот n . Во вториот ред се дадени елементите по редослед како што треба да се додадат во влезната редица.

Излез: На излез треба да се испечати “да” доколку е можно да се сортираат броевите или “не” доколку не е можно да се сортираат.

Забелешка: При реализација на задачата е дозволено да се користат само две редици и еден стек. Не е дозволено да се користат дополнителни структури како низи и листи.

Пример 1:

Влез:

5

5 1 2 3 4

Излез:

Да

Пример 2:

Влез:

6

5 1 2 6 3 4

Излез:

Не

Задача 3. Мој ред

Да се имплементира податочната структура редот MyQueue само со користење на податочната структура стек.

Влез: Во првиот ред од влезот е даден бројот на елементи кои ќе се внесуваат во податочната структура, потоа дадени се елементите секој во посебен ред. Понатака се дадени редоследно акциите кои треба да се направат врз податочната структура се додека не се внесе KRAJ. Акциите се следни:

1 - значи извади елемент од редот и испечати го

2 – додади елемент во редот. Во наредната линија се проследува и елементот кој треба да се додаде.

Излез: На излез треба да се испечатат елементите според барањата над податочната структура. На крај треба да се испечати големината на податочната структура.

Забелешка: При реализација на задачата дозволено е да се користи само податочната структура стек. Не е дозволено да се користат дополнителни низи или листи. Да се коментираат сложеностите на методите со кои е имплементиран редот.

Пример 1:

Влез: 5

1

2

3

4

5

1

1

1

1

KRAJ

Излез:

1
2
3
4
5

Goleminata na redicata e: 0

Пример 2:

Влез: 5

1
2
3
4
5
2
10

KRAJ

Излез:

Goleminata na redicata e: 6

4.4 Приоритетна редица

Приоритетната редица преставува еднодимензионална линеарна секвенца од елементи, каде секој елемент има свој приоритет. Карактеристично за неа е што елементот со најголем приоритет секогаш прв се вади од листата. Должина на приоритетна редица е бројот на елементи што ги содржи, при што празната приоритетна редица има должина 0.

Приоритетната редица има примена секаде каде што е потребно да се имплементира ред на чекање, но каде постојат разлики во приоритетот на оние кои чекаат во тој ред. Карактеристичен пример е претходно споменатиот распределувач на процеси во оперативните системи. Не сите процеси во еден оперативен систем имаат исто значење. Постојат процеси на таканареченото јадро на оперативниот систем за кои е посебно важно веднаш да добијат пристап до процесорот, за разлика од корисничките процеси, кои не се суштински за функционирањето на системот. Токму овој принцип се имплементира со приоритетна редица.

Уште еден пример на примена на приоритетната редица е подредувањето.

Основите операции за приоритетна редица се:

- Празнење на целата приоритетна редица
- Проверка дали приоритетната редица е празна

- Додавање на елемент во приоритетната редица
- Вадење на елементот со најголем приоритет од редицата
- Дополнително, проверка на елементот со најголем приоритет во редицата без негово вадење.

При изборот за имплементација на приоритетната редица, независно од тоа кој фундаментален тип ќе се користи, постојат 2 алтернативи:

1. подредена редица
2. неподредена редица

која ќе се имплементира како неограничена редица – со листа, или ограничена редица – со низа.

Првата варијанта е имплементација на подредена редица, имплементирана со подредена низа или подредена поврзана листа. Независно од тоа што ќе се користи за имплементација, во оваа варијанта додавањето на нов елемент ќе значи наоѓање на позицијата каде тој треба да се смести и негово вметнување. При имплементација со низа: пребарувањето ќе биде со сложеност $O(n)$ ако се пребарува линеарно. Ако се земе во предвид подреденоста, може да се пребарува и поинтелигентно, на пример бинарно, при што сложеноста е $O(\log n)$. Но на ова мора да се додаде сложеноста на вметнување на елемент во низа, која е $O(n)$.

При имплементација со листа: пребарувањето мора да биде од почеток и линеарно, значи $O(n)$. Самото вметнување е $O(1)$.

Втората варијанта е имплементација со неподредена низа или листа. При тоа, додавањето на елементот е на првата позиција (кога станува збор за поврзана листа), односно на последната позиција (кај имплементацијата со низа). И во двете имплементации, се додава онаму каде тоа е најефтино: кај низа на крај, кај поврзана листа на почеток, со сложеност $O(1)$. Проблемот овде настанува при вадењето на елементот. Тогаш е потребно да се измине целата структура за да се најде елементот со најголем приоритет. И во двете имплементации, наоѓањето на елементот со најголем приоритет значи поминување на сите елементи, од почеток до крај, што значи $O(n)$.

Глава 5

Хеш табели

Хеш табелите се податочни структури кои обезбедуваат ефикасно складирање и пронаоѓање на елементи дефинирани во форма на парови (клуч, вредност). Елементите во хеш табелата не се подредени.

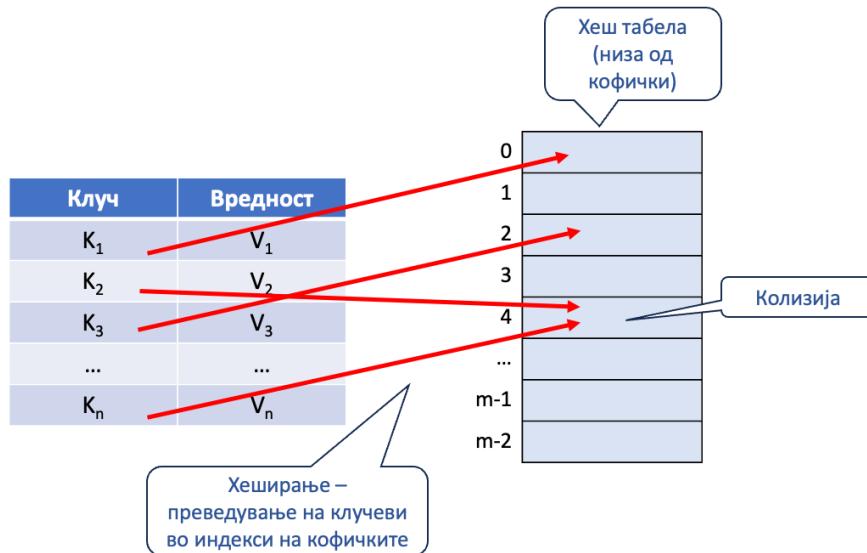
Каде што е важно ефикасно пребарување и додавање на елементи, но не и бришење, единствено исплатлива имплементација би била базирана на низа каде што операциите на додавање на нов елемент и пристапување на елемент на дадена позиција според индексот имаат сложеност $O(1)$. Динамички поврзаните листи обработени во глава 3 не дозволуваат истовремено пристап и пребарување на елементи со сложеност $O(1)$.

Доколку клучевите од паровите (клуч, вредност) се мали цели броеви, тогаш хеш табелата може да ја претставиме со низа каде клучевите се индексите на низата. Ако имаме произволни клучеви, тогаш може да направиме преведување на клучевите во индекси на низата. Ова преведување на произволен клуч во мал цел број кој претставува индекс во низата се нарекува **хеширање**. Преведувањето се врши со помош на хеш функција $\text{hash}(k)$ која секој клуч k го пресликува во вредност (индекс на низата) 0 до $m - 1$, ако m е должината на низата.

Хеш табелата може да ја дефинираме како низа од m „кофички”, заедно со хеш функција $\text{hash}(k)$ која секој клуч k го пресликува во индексот на неговата „кофичка”.

Секој клуч има своја домашна „кофичка”, т.е. „кофичка” со индекс $\text{hash}(k)$. Додавање, пребарување и бришење на елемент со клуч k се врши во неговата домашна „кофичка”. Ова значи дека хеш функцијата $\text{hash}(k)$ мора да биде конзистентна, т.е. доколку $k_1 = k_2$ следи $\text{hash}(k_1) = \text{hash}(k_2)$. Но, може да се случи за различни клучеви да се добие иста домашна „кофичка”, т.е $k_1 \neq k_2$, но $\text{hash}(k_1) = \text{hash}(k_2)$. Ова се нарекува колизија и секогаш треба да се тежнее да се дефинираат/избираат хеш функции кои ретко доведуваат до колизии. На слика 5-1

е прикажан процесот на хеширање, т.е. транслација на клучевите во индекси на „кофичките”.



Слика 5-1: Визуелен приказ на процесот на хеширање.

Генеричката Java класа со која моделира една „кофичка” е `MapEntry`. Во неа има дефинирано два параметри: клучот на „кофичката” K и вредноста E . Делот `<K extends Comparable<K>` означува ограничување на клучот K , т.е. означува дека клучот мора да биде тип на податок кој што може да се споредува со други клучеви. Вредноста E може да биде од било кој тип.

Делот `implements Comparable<K>` означува дека класата `MapEntry` го имплементира интерфејсот `Comparable` со тип на параметар K , т.е. објект од класата може да се споредува со друг објект од истата класа преку клучот K . Класата `MapEntry` мора да го имплементира методот `compareTo()`.

```

1 class MapEntry<K extends Comparable<K>,E> implements Comparable<K> {
2     //Секој MapEntry објект е пар од клуч и вредност.
3     K key;
4     E value;
5
6     public MapEntry (K key, E val) {
7         this.key = key;
8         this.value = val;
9     }
10
11    public int compareTo (K that) {
12        //Спореди ја тековната „кофичка“ this со „кофичката“ that.
13        @SuppressWarnings("unchecked")

```

```

14     MapEntry<K,E> other = (MapEntry<K,E>) that;
15     return this.key.compareTo(other.key);
16 }
17
18 public String toString () {
19     return "<" + key + "," + value + ">";
20 }
21 }
```

Постојат два начини на имплементација на хеш табели:

- Хеш табела со затворени „кофички“ (Closed-bucket hash table - **СВНТ**);
- Хеш табела со отворени „кофички“ (Open-bucket hash table - **ОВНТ**).

5.1 Хеш табела со затворени „кофички“

Каде хеш табелите со затворени „кофички“ важат следните правила:

- Во секоја „кофичка“ може да бидат сместени повеќе елементи;
- Секоја „кофичка“ е потполно одвоена од останатите, т.е. нема „претекување“ од една во друга „кофичка“.

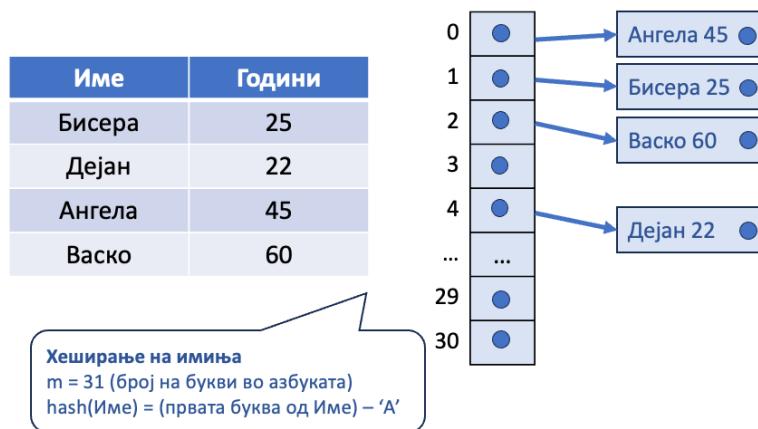
Наједноставна имплементација на секоја „кофичка“ во овој случај би било со еднострано поврзана листа, а хеш табелата би се имплементирала како низа од еднострано поврзани листи.

На слика 5-2 е прикажана илустрација на хеш табела со затворени „кофички“ без колизии, додека пак на слика 5-3 е прикажана илустрација на хеш табела со затворени „кофички“ со колизии. Хеш табелата има 31 „кофичка“, колку што има букви во македонската азбука. Хеширањето се прави така што се зема првата буква од името и се одредува нејзината позиција. Па така, имињата кои што почнуваат со А ќе бидат сместени во „кофичката“ 0, имињата кои што почнуваат со Б во „кофичката“ 1, и сл.

За да се избегнат/намалат колизиите и да се направи колку е можно подобра распределба на елементите во хеш табела со затворени кофички, при нејзинот дизајн треба да се внимава на следните два аспекти:

- Избор на број на „кофички“;
- Избор на хеш функција.

Доколку со n го означиме бројот на елементи кои треба да се додадат во хеш табелата, а со m бројот на „кофички“ во хеш табелата, во тој случај ќе дефинираме **фактор на пополнетост** (load factor) како n/m . Доколку n е однапред



Слика 5-2: Хеш табела со затворени „кофички” без колизии.



Слика 5-3: Хеш табела со затворени „кофички” со колизии.

познат број или може да се предвиди, вредноста на факторот на пополнетост треба да биде помеѓу 0.5 и 0.75. Дополнително, бидејќи при хеширањето често се прави аритметика по модул m , за да се постигне подобра дистрибуција на елементите се препорачува m да биде прост број.

Од друга страна, хеш функцијата треба да биде ефикасна, што значи да извршува неколку основи и брзи операции. Таа треба да обезбеди рамномерна распределба на елементите низ „кофичките”.

Хеш табела со затворени „кофички” е имплементирана со класата СВНТ. Хеш табелата е претставена со низа од „кофички” кои се вклучност SLLNode јазли (класа

опишана во глава 3) кои пак во себе чуваат MapEntry објекти.

```

1
2 public class CBHT<K extends Comparable<K>, E> {
3     //Табелата (со кофички) се состои од низа од јазли (SLLNode јазли) кои во
4     //себе чуваат MapEntry објекти.
5     private SLLNode<MapEntry<K,E>>[] buckets;
6
7     @SuppressWarnings("unchecked")
8     public CBHT(int m) {
9         //Креира празна хеш табела со m „кофички”.
10        buckets = (SLLNode<MapEntry<K,E>>[]) new SLLNode[m];
11    }
12
13    ...
14 }
```

5.1.1 Операции со хеш табела со затворени „кофички”

Основни операции со хеш табели со затворени „кофички” вклучуваат:

- хеш функција за пресметување на индекс на домашна кофичка;
- пребарување на елемент;
- додавање на елемент;
- бришење на елемент;
- печатење на хеш табела.

Методот `hash(K key)` се користи за пресметување на индексот на домашната „кофичка” на клучот `key`. Оваа функција се користи во методите за додавање, пребарување и бришење на елемент од хеш табелата. Методот `hashCode()` е метод наследен од класата `Object` во Java и се користи за генерирање на целобројна (`integer`) вредност (хеш код) за даден објект. Остатокот добиен од делењето со бројот на „кофички” обезбедува дека пресметаниот индекс ќе биде во границите на низата од „кофички”.

```

1
2 private int hash(K key) {
3     //Го преведува клучот во индекс на низата buckets.
4     return Math.abs(key.hashCode()) % buckets.length;
5 }
```

Имплементацијата на **пребарување на елемент** во хеш табела со затворени „кофички” е дадена со методот `search(K targetKey)`. Во овој метод со повикување на методот `hash(targetKey)` најпрвин се пресметува индексот на домашната „кофичка” на клучот `targetKey - b`, а потоа со `for` циклус се изминува листата од `SLL` јазли во домашната „кофичка” и се бара елемент чиј клуч е еднаков со `targetkey`, ако таков постои.

```

1 public SLLNode<MapEntry<K,E>> search(K targetKey) {
2     //Го наоѓа јазолот од СВНТ кој содржи елемент чиј клуч е еднаков на
3     //targetKey. Враќа врска до тој јазол (или null ако нема таков
4     //јазол).
5     int b = hash(targetKey);
6     for (SLLNode<MapEntry<K,E>> curr = buckets[b]; curr != null; curr =
7         curr.succ) {
8         if (targetKey.equals(((MapEntry<K, E>) curr.element).key))
9             return curr;
10    }
11    return null;
12 }
```

Операцијата **додавање на нов елемент** се имплементира со методот `insert(K key, E val)`. Во овој метод на почетокот се дефинира нов објект `newEntry` од класата `MapEntry` со клуч `key` и вредност `val`. Потоа на ист начин како и во методот `search(K targetKey)` се наоѓа домашната „кофичка” каде што треба да се додаде објектот `newEntry`. Следно, елементот се вметнува во листата на домашната кофичката, заменувајќи ја со `val` претходната вредност за клучот `key`, ако таков елемент претходно постоел. Доколку не постои елемент со клуч `key`, во тој случај елементот се дава на почетокот на листата во домашната „кофичка” бидејќи тоа е најисплатливо.

```

1 public void insert(K key, E val) {
2     //Вметнување на парот <key, val> во СВНТ.
3     MapEntry<K, E> newEntry = new MapEntry<K, E>(key, val);
4     int b = hash(key);
5     for (SLLNode<MapEntry<K,E>> curr = buckets[b]; curr != null; curr =
6         curr.succ) {
7         if (key.equals(((MapEntry<K, E>) curr.element).key)) {
8             //newEntry го заменува постоечкиот елемент со клуч key.
9             curr.element = newEntry;
10            return;
11        }
12    }
13    //Додавање на newEntry на почетокот на листата во домашната „кофичка”
```

```

    со индекс b.
13   buckets[b] = new SLLNode<MapEntry<K,E>>(newEntry, buckets[b]);
14 }
```

Бришењето на елемент од хеш табела со затворени „кофички” е имплементирано со методот `delete(K key)`. Слично, како и во претходните два методи, потребно е да се најде домашната „кофичка” на елементот со клуч `key`, а потоа се врши отстранување на елементот од листата. Ова се изведува со помош на два покажувачи: `pred` кој што покажува на позиција пред елементот кој што треба да биде избришан и `curr` кој покажува кон елементот што треба да се избрише.

```

1 public void delete(K key) {
2     int b = hash(key);
3     for (SLLNode<MapEntry<K,E>> pred = null, curr = buckets[b]; curr != null; pred = curr, curr = curr.succ) {
4         if (key.equals(((MapEntry<K,E>) curr.element).key)) {
5             if (pred == null)
6                 buckets[b] = curr.succ;
7             else
8                 pred.succ = curr.succ;
9             return;
10        }
11    }
12 }
```

Доколку ги анализираме методите за додавање, пребарување и бришење на елемент од хеш табела со затворени „кофички” и при тоа го земеме предвид бројот на споредби кои се вршат, за n елементи можеме да го заклучиме следното:

- Во најдобар случај не постои „кофичка” со повеќе од два елементи и бројот на споредби е две, така што сложеноста во најдобар случај е $O(1)$.
- Во најлош случај сите елементи се во една „кофичка” и бројот на споредби е n , така што сложеноста во најлош случај е $O(n)$.

Методот за печатење на хеш табела се имплементира со препокривање на методата `toString()` во Java која враќа `String`. Започнува со дефинирање на резултантен стринг `temp` и потоа се изминува низата на кофички, така што за секоја „кофичка” се изминуваат елементите од листата во таа „кофичка”. При изминување на листата за секој елемент се повикува методот `toString()` од класата `SLLNode`. Сложеноста на овој метод во најлош случај доколку што сите елементи се сместени во една „кофичка” е $O(n^2)$.

```

1 public String toString() {
```

```

2     String temp = "";
3     for (int i = 0; i < buckets.length; i++) {
4         temp += i + ":";
5         for (SLLNode<MapEntry<K,E>> curr = buckets[i]; curr != null; curr =
6             curr.succ) {
7             temp += curr.element.toString() + " ";
8         }
9     }
10    return temp;
11 }
```

5.1.2 Едноставни проблеми со хеш табели со затворени „кофички“

Задача 1. Роденденi

Во заводот на статистика се прави ново истражување каде што се открива бројот на луѓе родени во секој месец. Ваша задача е за даден месец да прикажете колку луѓе се родени во тој месец.

Влез: Во првиот ред од влезот е даден бројот на луѓе N , а во секој нареден ред е даден датумот на раѓање. Во последниот ред е даден месецот за кој треба да се прикаже бројот на луѓе родени во тој месец.

Излез: Број на луѓе кои се родени во тој месец. Доколку нема луѓе родени во тој месец да се испечати „Empty“.

Пример:

Влез:

4
20.7.1976
16.7.1988
18.7.1966
5.6.1988
7

Излез: 3

Решение

Во дадената задача треба да се избери колку луѓе се родени во даден месец. Според тоа, бидејќи пребарувањето се врши според месецот, тој ќе биде клуч

во елементите во хеш табелата. При додавање на секој нов елемент во хеш табелата, доколку таков месец не постои, т.е. истиот се додава прв пат, вредноста на елементот ќе биде 1 и ќе се добие пар (месец, 1). Во случај кога веќе постои елемент со дадениот месец како клуч, треба да се земе постоечката вредност на тој елемент и елементот да се замени со клуч месец и со вредност зголемена за 1.

Бидејќи постојат 12 месеци за да се добие фактор на пополнетост меѓу 0.5 и 0.75, бројот на „кофички” ќе биде поставен на 23. За да се најде месецот, потребно е стрингот да се подели по знакот „.”. Во тој случај, месецот се наоѓа на втората позиција во резултантната низа, т.е. на позиција со индекс 1.

Најпрвин се чита N (бројот на елементи), а потоа се дефинира СВНТ хеш табела. Во `for` циклус се чита линија по линија од влезот и за секоја линија се прави поделба на стрингот според знакот „.”. Во овој пример од секоја линија ќе се создаде низа со три елементи, така што на позиција 0 ќе биде денот, на позиција 1 месецот и на позиција 2 годината. Доколку елемент со ист клуч не постои во хеш табелата, истиот се додава првпат. Доколку постои, се пребарува елементот со дадениот клуч, се зема неговата вредност и елементот се заменува со новата вредност зголемена за 1.

Потоа се чита клучот и се пребарува елементот. Доколку елементот со таков клуч постои, се печати вредноста на елементот (бројот на луѓе родени во дадениот месец), а доколку нема елемент со таков клуч се печати „Empty”.

Во дадениот пример се бара бројот на луѓе кои се родени во седмиот месец, па според тоа резултатот е 3.

Имплементацијата на решението е дадена подолу.

```

1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;

4
5 //Почеток на класа SLL Node
6 class SLLNode<E> {
7     protected E element;
8     protected SLLNode<E> succ;

9
10    public SLLNode(E elem, SLLNode<E> succ) {
11        this.element = elem;
12        this.succ = succ;
13    }
14
15    @Override
16    public String toString() {

```

```
17     return element.toString();
18 }
19 }
20 //Крај на класа SLL Node
21
22 //Почеток на класа MapEntry
23 class MapEntry<K extends Comparable<K>, E> implements Comparable<K> {
24
25     K key;
26     E value;
27
28     public MapEntry (K key, E val) {
29         this.key = key;
30         this.value = val;
31     }
32
33     public int compareTo (K that) {
34         @SuppressWarnings("unchecked")
35         MapEntry<K,E> other = (MapEntry<K,E>) that;
36         return this.key.compareTo(other.key);
37     }
38
39     public String toString () {
40         return "<" + key + "," + value + ">";
41     }
42 }
43 //Крај на класа MapEntry
44
45 //Почеток на класа CBHT
46 class CBHT<K extends Comparable<K>, E> {
47
48     private SLLNode<MapEntry<K,E>>[] buckets;
49
50     @SuppressWarnings("unchecked")
51     public CBHT(int m) {
52         buckets = (SLLNode<MapEntry<K,E>>[]) new SLLNode[m];
53     }
54
55     private int hash(K key) {
56         return Math.abs(key.hashCode()) % buckets.length;
57     }
```

```
58
59     public SLLNode<MapEntry<K,E>> search(K targetKey) {
60         int b = hash(targetKey);
61         for (SLLNode<MapEntry<K,E>> curr = buckets[b]; curr != null; curr =
62             curr.succ) {
63             if (targetKey.equals(((MapEntry<K, E>) curr.element).key))
64                 return curr;
65         }
66         return null;
67     }
68
69     public void insert(K key, E val) {
70         MapEntry<K, E> newEntry = new MapEntry<K, E>(key, val);
71         int b = hash(key);
72         for (SLLNode<MapEntry<K,E>> curr = buckets[b]; curr != null; curr =
73             curr.succ) {
74             if (key.equals(((MapEntry<K, E>) curr.element).key)) {
75                 curr.element = newEntry;
76                 return;
77             }
78         }
79         buckets[b] = new SLLNode<MapEntry<K,E>>(newEntry, buckets[b]);
80     }
81
82     public void delete(K key) {
83
84         int b = hash(key);
85         for (SLLNode<MapEntry<K,E>> pred = null, curr = buckets[b]; curr !=
86             null; pred = curr, curr = curr.succ) {
87             if (key.equals(((MapEntry<K,E>) curr.element).key)) {
88                 if (pred == null)
89                     buckets[b] = curr.succ;
90                 else
91                     pred.succ = curr.succ;
92                 return;
93             }
94         }
95     } //Крај на класа СВНТ
```

```
96
97 public class Birthdays {
98
99     public static void main(String[] args) throws NumberFormatException,
100        IOException {
101
102         BufferedReader bf = new BufferedReader(new
103             InputStreamReader(System.in));
104         int N = Integer.parseInt(bf.readLine());
105         CBHT<String, Integer> birthdays = new CBHT<>(23);
106
107         for(int i = 0; i<N; i++){
108
109             String p [] = bf.readLine().split("\\.");
110
111             //доколку елементот со клуч p[1] не постои во хеш табелата
112             if(birthdays.search(p[1])==null)
113             {
114                 birthdays.insert(p[1], 1);
115             }
116             else
117             {
118                 //доколку елементот со клуч p[1] постои во хеш табелата
119                 SLLNode<MapEntry<String, Integer>> br = birthdays.search(p[1]);
120                 birthdays.insert(p[1], br.element.value+1);
121             }
122
123             String month = bf.readLine();
124             SLLNode<MapEntry<String, Integer>> result = birthdays.search(month);
125
126             //доколку не постои елемент со клуч mesec
127             if(result==null)
128             {
129                 System.out.println("Empty");
130             }
131             else
132             {
133                 System.out.println(result.element.value);
134             }
135         }
136     }
137 }
```

135 }
136 }

Задача 2. Најдобра понуда

На еден светски познат предавач секојдневно му пристигнуваат понуди да држи предавања. За секоја понуда се дадени датуми, време на почеток, градот и износот на хонорарот за предавањето (во долари). Ваша задача е за даден датум да го прикажете предавањето кое би му донело најголема заработка на предавачот. Доколку нема понуди за дадениот датум да се испечати „No offers”.

Влез: Во првиот ред од влезот е даден бројот на понуди, а во секој нареден ред се дадени: датумот и времето на предавањето (формат dd/mm/yyyyhh:mm), градот во кој ќе се одржува предавањето и износот на хонорарот. Во последниот ред е даден датумот за кој треба да испечатите која понуда е најдобра за тој датум.

Излез: Деталите на понудата за тој датум.

Пример:

Влез:

7

27/01/2016 14:00 NewYork 6000

28/01/2016 08:00 Paris 3000

28/01/2016 14:00 Munich 5000

27/01/2016 09:00 Beijing 8000

27/01/2016 08:00 Seattle 4000

28/01/2016 09:00 SaltLakeCity 10000

28/01/2016 09:00 Lagos 12000

27/01/2016

Излез:

09:00 Beijing 8000

Решение

Во оваа задача ќе се креира хеш табела со затворени „кофички” со парови од клуч - датум во форма на текст, а за вредностите ќе се креира класа `Lecture` во која ќе се зачуваат датумот, часот, местото и хонорарот за предавањето. За даден датум ќе треба да се зачуваат повеќе објекти од класата `Lecture`, па затоа вредностите во табелата ќе бидат зачувани во низи од објекти од класата `Lecture`. За да може при пребарување да се добие предавањето со највисок хонорар со сложеност $O(1)$, т.е. да се прочита само првиот елемент од низата, при

додавање на вредностите во низата најдобро ќе биде тие постојано да бидат сортирани според хонорарот во опаѓачки редослед. За таа цел во класата `Lecture` ќе се додаде метод `CompareTo(Lecture obj)` во кој ќе се дефинира дека споредбата помеѓу два објекти од класата `Lecture` ќе се извршува според хонорарот.

Имплементацијата на класата `Lecture` е дадена подолу.

```
1 class Lecture implements Comparable<Lecture>
2 {
3     String date;
4     String time;
5     String place;
6     Integer fee;
7
8     public Lecture(String date, String time, String place, Integer fee) {
9         this.date = date;
10        this.time = time;
11        this.place = place;
12        this.fee = fee;
13    }
14
15    public String getTime() {
16        return time;
17    }
18    public void setTime(String time) {
19        this.time = time;
20    }
21    public String getDate() {
22        return date;
23    }
24    public void setDate(String date) {
25        this.date = date;
26    }
27    public String getPlace() {
28        return place;
29    }
30    public void setPlace(String place) {
31        this.place = place;
32    }
33    public Integer getFee() {
34        return fee;
35    }
36    public void setFee(Integer fee) {
```

```

37     this.fee = fee;
38 }
39 @Override
40 public int compareTo(Lecture obj) {
41     if(this.fee > obj.fee)
42         return 1;
43     else if(this.fee < obj.fee)
44         return -1;
45     else
46         return 0;
47 }
48
49 }
```

При додавање на елементи во хеш табелата, постојат две опции. Првата е кога елементот со даден клуч не постои во табелата, а втората е кога елемент со таков клуч постои. И во двата случаи при обработка на влезот се креира нов објект од класата `Lecture`. Во првиот случај, се креира празна низа од тип `Lecture` и во табелата се додава елемент со клуч датум и вредност низа со објектот од класата `Lecture` креиран претходно. Во вториот случај, се пребарува објектот со истиот клуч и неговата вредност (низата од предавања) се менува, така што се додава новокреираниот објект. Потоа се сортира низата и во хеш табелата се додава елементот со истиот клуч - датум и новата вредност - низата од предавања. Со тоа се пребришува стариот елемент со истиот клуч, но бидејќи низата од предавања е сортирана во однос на хонорарот по опагачки редослед, секогаш предавањето со најголем хонорар ќе биде на прва позиција (0). Сортирањето се прави со функцијата `Collections.sort()` бидејќи се користи класата `ArrayList` која е дел од Java API.

Потоа се пребарува во хеш табелата според дадениот датум и резултатот се добива преку пристапување на првиот објект од низата со предавања. Доколку нема понуда за дадениот датум, се печати „No offers”.

Во дадениот пример се пребарува по датум „27/01/2016” и според тоа како најдобра понуда се добива „09:00 Beijing 8000”.

Имплементацијата на решението е дадена подолу.

```

1
2 //Вметни класа MapEntry
3 //Вметни класа SLLNode
4 //Вметни класа СВНТ
5 //Вметни класа Lecture
6
```

```
7 public class BestOffer{
8     public static void main(String[] args) throws IOException {
9
10    BufferedReader br = new BufferedReader(new
11        InputStreamReader(System.in));
12    int N = Integer.parseInt(br.readLine());
13
14    CBHT<String, ArrayList<Lecture>> hashtable = new
15        CBHT<String,ArrayList<Lecture>>(2*N);
16
17    for(int i=0;i<N;i++)
18    {
19        String[] input=br.readLine().split(" ");
20        Lecture p = new
21            Lecture(input[0],input[1],input[2],Integer.parseInt(input[3]));
22
23        if(hashtable.search(input[0])==null)
24        {
25            ArrayList<Lecture> lectures = new ArrayList<Lecture>();
26            lectures.add(p);
27            hashtable.insert(input[0], lectures);
28        }
29        else
30        {
31            SLLNode<MapEntry<String, ArrayList<Lecture>>> result =
32                hashtable.search(input[0]);
33            ArrayList<Lecture> lectures = result.element.value;
34            lectures.add(p);
35            Collections.sort(lectures,Collections.reverseOrder());
36            hashtable.insert(input[0], lectures);
37        }
38    }
39
40    String date=br.readLine();
41    SLLNode<MapEntry<String, ArrayList<Lecture>>> tosearch =
42        hashtable.search(date); //Сложеноста на оваа операција е O(1)
43        бидејќи низата е секогаш сортирана според хонорарот во опаѓачки
44        редослед
45
46    if(tosearch!=null) {
47        System.out.println(tosearch.element.value.get(0).getTime()+""
48            "+tosearch.element.value.get(0).getPlace()+""
49            "+tosearch.element.value.get(0).getPrice());
50    }
51}
```

```

        "+tosearch.element.value.get(0).getFee());
41    }
42    else
43    {
44        System.out.println("No offers");
45    }
46
47 }
48 }
```

Задача 3. Анаграми

Користејќи хеш табела да се групираат сите анаграми од дадена листа со зборови. Анаграми се зборови кои се добиваат со преуредување на буквите од зборот. На пример spar е анаграм на rasp.

Влез: Во првиот ред е даден бројот на зборови N . Во наредните N реда се дадени зборовите кои треба да се додадат во табелата. Во следниот ред е даден зборот за кој треба да се испечати бројот на анаграми во табелата.

Излез: Бројот на анаграми во табелата за дадениот збор.

Пример:

Влез:

```

6
eat
tea
tan
ate
nat
bat
ant
```

Излез:

```
2
```

Решение

Идејата на оваа задача е да се дефинира клучот како текст кој што ќе претставува сортирана низа од карактерите на зборот. На тој начин, сите анаграми кога ќе се сортираат ќе претставуваат ист збор.

Се креира хеш табела со текстуални клучеви и целобројни вредности. За секоја линија од влезот се сортира низата од карактери и се пребарува дали веќе има

таков клуч во хеш табелата. Доколку нема, се додава нов елемент со клуч сортирана низа од карактери и вредност 1. Доколку постои елемент со таков клуч, се зема вредноста (бројот на анаграми) и се зголемува за 1, па потоа се додава елементот во хеш табелата. На тој начин за сите анаграми ќе има ист клуч, а нивниот број ќе се зачува со тоа што ќе се зголемува вредноста на соодветниот елемент во хеш табелата.

Потоа, пред да се започне со пребарување на зборот, треба да се повтори постапката, т.е. да се сортираат карактерите од зборот.

Во примерот, за ant има два анаграми: tan и nat.

Имплементацијата на решението е дадена подолу.

```

1 //Вметни класа MapEntry
2 //Вметни класа SLLNode
3 //Вметни класа CBHT
4
5 public class Anagrams {
6
7     public static void main(String[] args) throws NumberFormatException,
8         IOException {
9
10        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
11        int N = Integer.parseInt(br.readLine());
12        CBHT<String, Integer> hashtable = new CBHT<String, Integer>(2*N);
13        String input;
14
15        for(int i=1;i<=N;i++){
16
17            input = br.readLine();
18            char [] letters = input.toCharArray();
19            Arrays.sort(letters);
20            String sortedLetters = new String(letters);
21
22            if(hashtable.search(sortedLetters)==null)
23            {
24                hashtable.insert(sortedLetters,1);
25            }
26            else
27            {
28                SLLNode<MapEntry<String, Integer>> result =
29                    hashtable.search(sortedLetters);
30                hashtable.insert(sortedLetters, result.element.value+1);
31            }
32        }
33    }
34}
```

```

29     }
30
31     }
32
33     String word = br.readLine();
34     char [] letters = word.toCharArray();
35     Arrays.sort(letters);
36     String sortedLetters = new String(letters);
37     SLLNode<MapEntry<String, Integer>> result =
38         hashtable.search(sortedLetters);
39     System.out.println(result.element.value);
40
41 }

```

5.1.3 Напредни проблеми со хеш табели со затворени „кофички“

Задача 1. Епидемија

Поради епидемија на сезонски грип при секое тестирање на даден пациент се зачувува општината во која живее, неговото презиме и информација дали е позитивен или негативен на вирусот. Потребни се статистички податоци за да се одреди ризик факторот за дадена општина. Ваша задача е за дадена општина на излез да го испечатите ризик факторот во дадената општина. Ризик факторот се пресметува на следниот начин:

$$\text{Ризик фактор} = \frac{\text{број на позитивни пациенти}}{\text{број на негативни пациенти} + \text{број на позитивни пациенти}}$$

Забелешка: Можно е да се појават пациенти со исто презиме. Истите треба да се земат како посебни вредности во статистиката.

Влез: На влез најпрво е даден бројот на пациенти N , а потоа секој пациент е даден во нов ред во форматот: „*Општина во која живее*”, „*Презиме на пациент*” „*Резултати од тестот(positive/negative)*”. На крај е дадена општината за која треба да се пресмета ризик факторот.

Излез: Децимален број заокружен на две децимали кој го претставува ризик факторот за дадената општина.

Пример:

Влез:

6

Centar Stojanoski negative

Centar Trajkovski positive

Centar Petkovski positive

Karpos Stojanoski positive

Karpos Trajkovski negative

Centar Trajkovski positive

Centar

Излез:

0.75

Решение

Доколку се разгледа примерот со шест пациенти според последниот збор во секоја линија може да се забележи дека двајца се негативни на вирусот и четворица се позитивни. Бидејќи треба да се избројат посебно позитивните и негативните пациенти наједноставно е да се креираат две хеш табели. Според примерот, во првата хеш табела со позитивни пациенти би имало четворица пациенти, а во втората со негативни, двајца.

Пребарувањето се врши по име на општина, па затоа името на општината ќе биде клуч, а бројот на пациенти од таа општина ќе биде вредност. Така, во хеш табелата со позитивни пациенти, во „кофичката“ со клуч „Centar“ треба да има елемент (Centar,3), а во „кофичката“ со клуч „Karpos“ треба да има елемент (Karpos,1).

Во хеш табелата со негативни пациенти во една „кофичка“ ќе биде елементот („Centar“, 1), а во друга „кофичка“ („Karpos“, 1).

Во хеш табелата со позитивни пациенти има тројца пациенти кои живеат во општина Центар, а во хеш табелата со негативни пациенти има еден. Според тоа, ризик факторот би се пресметал како $3/(1+3) = 0.75$.

Доколку клучот по кој се пребарува е „Karpos“, во тој случај ќе се добие $1/(1+1) = 0.5$.

Читањето на влезот и додавањето на елементите во хеш табелите може да се реализира на следниот начин: прво се чита N (бројот на елементи), а потоа се дефинираат две хеш табели, за позитивни и за негативни пациенти, соодветно.

Во **for** циклус се чита линија по линија од влезот и за секоја линија се прави поделба на стрингот според празно место. Во овој пример од секоја линија ќе се создаде низа со три елементи, така што на позиција 0 ќе биде името на општината, на позиција 1 презимето на пациентот и на позиција 2 неговиот статус од тестирањето. Според статусот, за секој пациент се проверува дали веќе постои

елемент со клуч Општина во соодветната хеш табела. Доколку не постои, во избраната хеш табела се додава елемент во формат (Општина, 1). Доколку постои елемент со клуч Општина, во тој случај се зема моменталната вредност на тој елемент и на неа се додава 1. Така променетиот елемент се додава во соодветната хеш табела.

Пребарувањето според клуч и обработката на резултатите се реализира на следниот начин: прво се чита клучот по кој ќе се пребарува и се дефинира променлива `positiveCount` во која се смесува вредноста на позитивните пациенти за дадениот клуч. Истата постапка се повторува и за хеш табелата со негативни пациенти со променливата `negativeCount`. На крајот се печати ризик факторот за дадената општина.

Имплементацијата на решението е дадена подолу.

```

1 //Вметни класа MapEntry
2 //Вметни класа SLLNode
3 //Вметни класа CBHT
4
5 public class InfluenzaRiskFactor {
6
7     public static void main(String[] args) throws NumberFormatException,
8         IOException {
9
10        BufferedReader bf = new BufferedReader(new
11            InputStreamReader(System.in));
12        int N = Integer.parseInt(bf.readLine());
13        CBHT<String, Integer> positivePatients = new CBHT<>(100);
14        CBHT<String, Integer> negativePatients = new CBHT<>(100);
15
16        for(int i = 0; i<N; i++){
17            String input []= bf.readLine().split(" ");
18            if(input[2].equals("positive"))
19            {
20                SLLNode<MapEntry<String, Integer>>
21                    positiveRes=positivePatients.search(input[0]);
22
23                if(positiveRes==null) {
24                    positivePatients.insert(input[0], 1);
25                }
26                else
27                {
28                    Integer numPositive=positiveRes.element.value+1;
29
30                    positiveRes.element.value=numPositive;
31
32                    if(negativeRes==null) {
33                        negativePatients.insert(input[0], 1);
34                    }
35                    else
36                    {
37                        Integer numNegative=negativeRes.element.value+1;
38
39                        negativeRes.element.value=numNegative;
40
41                        if(positiveRes.value>negativeRes.value)
42                            System.out.println("Positive");
43                        else
44                            System.out.println("Negative");
45
46                    }
47
48                }
49            }
50        }
51    }
52}
```

```

26         positivePatients.insert(input[0], numPositive);
27     }
28
29     }
30     else
31     {
32         SLLNode<MapEntry<String, Integer>>
33         negativeRes=negativePatients.search(input[0]);
34
35         if(negativeRes==null) {
36             negativePatients.insert(input[0], 1);
37         }
38         else
39         {
40             Integer numNegative=negativeRes.element.value+1;
41             positivePatients.insert(input[0], numNegative);
42         }
43     }
44
45     String municipality = bf.readLine();
46     SLLNode<MapEntry<String, Integer>> positiveRes =
47         positivePatients.search(municipality);
48     SLLNode<MapEntry<String, Integer>> negativeRes =
49         negativePatients.search(municipality);
50
51     Integer positiveCount = positiveRes.element.value;
52     Integer negativeCount = negativeRes.element.value;
53
54     System.out.println(String.format("%.2f",
55         positiveCount*1.00/(negativeCount+positiveCount)));
56 }

```

Задача 2. Застапеност на имиња

Потребно е да се направи компјутерска апликација која ќе му овозможи на корисникот брзо да пребарува низ базата на податоци на Заводот за статистика за застапеност на имиња (машки и женски). Начинот на кој се пребарува е следен: прво се внесува какво име ќе пребарува, машко или женско, после тоа доволно е

да се внесат првите 2 букви од името за да може да се излистаат женските/машките имена кои ги има во системот. Како резултат од пребарувањето треба да се врати честотата на појавување за даденото име. При секое евидентирање на новороденче неговото име се запишува во базата на заводот за статистика. Доколку името постои, се менува само бројот на застапеност, а доколку не постои, се додава како ново.

Влез: Од стандарден влез прво се чита број N кој претставува број на имена кои ќе бидат внесени во системот. Во наредните N редови се дадени имената на новороденчињата и од кој пол се (**M** за машки и **F** за женски) разделени со празно место. Во наредниот ред е даден полот по кој ќе се пребарува, а потоа се дадени имената кои се пребаруваат, секое во нов ред. За означување на крај е даден зборот „END”.

Излез: На стандарден излез треба да се испечати за секој од влезовите следната информација: Листа на имена кои ги дава начинот на пребарување како сугестија, секое во нов ред. Доколку постои името во хеш табелата се печати во нов ред ПОЛ ИМЕ ЗАСТАПЕНОСТ разделени со по едно празно место. Доколку името не е пронајдено се печати „No such name”.

Забелешка: Функцијата со која се врши мапирање на имената во број е следна:

$$h(w) = (100 * \text{ASCII}(c1) + \text{ASCII}(c2)) \% 9091,$$
 каде зборот $w = c1c2c3c4c5\dots$ е составен само од големи букви.

Пример:

Влез:

7

Hristina F

Magdalena F

Ivana F

Ivan M

Elena F

Ana F

Makedonka F

F

MARIJA

Ivana

Kristina

Anastasija

END

Излез:

MAKEDONKA

MAGDALENA

No such name

IVANA

F IVANA 1

No such name

ANA

No such name

Решение

Во оваа задача бидат дефинирани две хеш табели со затворени „кофички” со големина 9091 (број наведен во хеш функцијата), една за лицата од женски пол, другата за лицата од машки пол. И во двете хеш табели клучеви ќе бидат имињата, а вредности броевите на појавувања на имињата (застапеноста).

Бидејќи треба да дефинира корисничка хеш функција, ќе се дефинира нова класа Name со еден атрибут `name` (име) во која ќе се преоптовари функцијата `hashCode()`. Бидејќи класата го имплементира интерфејсот Comparable, во истата мора да се преоптовари методот `compareTo(Name arg0)`.

```

1 class Name implements Comparable<Name>{
2     String name;
3
4     public String getIme() {
5         return name;
6     }
7     public void setIme(String name) {
8         this.name = name;
9     }
10
11    public Name(String name) {
12        this.name = name.toUpperCase();
13    }
14    @Override
15    public boolean equals(Object obj) {
16        Name temp = (Name) obj;
17        return this.name.equals(temp.name);
18    }
19    @Override
20    public int hashCode() {
21        int hash = (100* name.charAt(0) + name.charAt(1));
22        return hash;
23    }

```

```

24     @Override
25     public String toString() {
26         return name;
27     }
28     @Override
29     public int compareTo(Name arg0) {
30         return name.compareTo(arg0.name);
31     }
32 }
```

Потоа се обработува влезот и во зависност од полот, се додава елемент во една од табелите. Доколку елемент со таков клуч не постои во табелата се додава парот (име, 1), а доколку постои, се пребарува во соодветната табела, е зема тековната вредност (застапеност на името) и се зголемува за 1.

Хеш функцијата ги вклучува ASCII кодовите на првите две букви од името, па тоа значи дека во иста „кофичка” ќе се сместат сите имиња кои што имаат исти први две букви. За да се испечатат сите имиња кои почнуваат на првите две букви од името дадено на влез, треба да се измине едностррано поврзаната листа на домашната „кофичка” на елементот. За да се стигне до првиот јазол од листата се користи методот `getFirst()` кој што враќа покажувач кон првиот елемент од листата во домашната „кофичка”.

```

1  public SLLNode<MapEntry<K,E>> getFirst(K targetKey) {
2      int b = hash(targetKey);
3      return buckets[b];
4 }
```

Потоа се пребарува елементот според името и доколку постои се печати полот, името и застапеноста, а доколку не постои се печати „No such name”.

Во дадениот пример се пребаруваат имиња од женски пол и затоа се пребарува само во табелата со женски пол. Бидејќи прво се пребарува „MARIJA”, се печатат сите имиња кои почнуваат на MA, а тоа се „MAKEDONKA” и „MAGDALENA” (сите се внесуваат со големи букви во табелата). Името „MARIJA” не постои во хеш табелата и затоа се печати „No such name”. Потоа, за „Ivana” се пребарува според првите две букви и се печати „IVANA”. Бидејќи постои елемент со клуч „IVANA”, се печати полот, името и застапеноста. Треба да се забележи дека „Ivan” не се печати во овој случај бидејќи тоа е елемент во табелата со машки пол. Потоа се пребарува елемент со клуч „Kristina” и се печати „No such name”. За „Anastasija”, според првите две букви се печати „ANA” и бидејќи елемент со клуч „Anastasija” не постои во табелата се печати „No such name”.

Во продолжение е дадена имплементацијата на решението.

```

1 //Вметни класа MapEntry
2 //Вметни класа SLLNode
3 //Вметни класа CBHT со додаден метод getFirst()
4 //Вметни класа Name
5
6 public class Statistics {
7
8     public static void main(String[] args) throws Exception, IOException {
9
10         CBHT<Name, Integer> tableM, tableF;
11         BufferedReader br = new BufferedReader(new
12             InputStreamReader(System.in));
13         int N = Integer.parseInt(br.readLine());
14         tableM = new CBHT<Name, Integer>(9091);
15         tableF = new CBHT<Name, Integer>(9091);
16
17         for(int i=1;i<=N;i++){
18             String line = br.readLine();
19             String[] input = line.split(" ");
20             Name nameUpper=new Name(input[0].toUpperCase());
21
22             if (input[1].compareTo("M") == 0){
23                 SLLNode<MapEntry<Name, Integer>> resM = tableM.search(nameUpper);
24                 if(resM==null){
25                     tableM.insert(nameUpper,1);
26                 }
27                 else{
28                     int oldValue = resM.element.value;
29                     tableM.insert(nameUpper,oldValue+1);
30                 }
31             }
32             if (input[1].compareTo("F") == 0){
33                 SLLNode<MapEntry<Name, Integer>> resF = tableF.search(nameUpper);
34                 if(resF==null){
35                     tableF.insert(nameUpper,1);
36                 }
37                 else {
38                     int oldValue = resF.element.value;
39                     tableF.insert(nameUpper,oldValue+1);

```

```
40         }
41     }
42 }
43
44 String sex = (br.readLine()).toUpperCase();
45 String names = (br.readLine()).toUpperCase();
46
47 while(names.compareTo("END")!=0){
48     if (sex.compareTo("M")==0){
49
50         SLLNode<MapEntry<Name, Integer>> resM1 = tableM.getFirst(new
51             Name(names));
52         SLLNode<MapEntry<Name, Integer>> curr;
53
54         for (curr = resM1; curr != null; curr = curr.succ) {
55             System.out.println(curr.element.key.getIme());
56         }
57
58         SLLNode<MapEntry<Name, Integer>> resM2 = tableM.search(new
59             Name(names));
60
61         if(resM2==null){
62             System.out.println("No such name");
63             names = (br.readLine()).toUpperCase();
64         }
65         else{
66             System.out.println(sex+" "+resM2.element.key.toString()+""
67                 "+resM2.element.value.toString());
68             names = (br.readLine()).toUpperCase();
69         }
70
71         if (sex.compareTo("F")==0){
72
73             SLLNode<MapEntry<Name, Integer>> resF1 = tableF.getFirst(new
74                 Name(names));
75             SLLNode<MapEntry<Name, Integer>> curr1;
76
77             for (curr1 = resF1; curr1 != null; curr1 = curr1.succ) {
78                 System.out.println(curr1.element.key.getIme());
79             }
80
81         }
82     }
83 }
```

```

77
78     SLLNode<MapEntry<Name, Integer>> resF2 = tableF.search(new
79         Name(names));
80         if(resF2==null){
81             System.out.println("No such name");
82             names = (br.readLine()).toUpperCase();
83         }
84         else{
85             System.out.println(sex+" "+resF1.element.key.toString()+""
86                 "+resF1.element.value.toString());
87             names = (br.readLine()).toUpperCase();
88         }
89     }
90 }
91 }
```

Задача 3. Аптека

Потребно е да се направи компјутерска апликација со која ќе се забрза работењето на една аптека. Притоа апликацијата треба да му овозможи на корисникот (фармацеввтот) брзо да пребарува низ огромното множество со лекови кои се внесени во системот. Начинот на кој тој треба да пребарува е следен: доволно е да ги внесе првите 3 букви од името на лекот за да може да се прикаже листа од лекови кои ги има во системот.

Работата на фармацеввтот е да провери дали внесениот лек го има во системот и да му даде информација на клиентот. Информацијата што треба да му ја даде на клиентот е дали лекот се наоѓа на позитивната листа на лекови, која е цената и колку парчиња од лекот има на залиха. Доколку лекот постои, клиентот го нарачува со што кажува колку парчиња ќе купи. Оваа акција фармацеввтот треба да ја евидентира во системот (односно да ја намали залихата на лекови за онолку парчиња колку што му издал на клиентот). Доколку нарачката на клиентот е поголема од залихата на лекот што ја има во системот, не се презема никаква акција.

Влез: Од стандарден влез прво се чита број N кој претставува број на лекови кои ќе бидат внесени во системот. Во наредните N реда се дадени имињата на лековите, дали ги има на позитивната листа (1/0), цената и број на парчиња, сите разделени со по едно празно место. Потоа се дадени редови со имиња на лекови и број на парчиња нарачани од клиентот. За означување на крај се наведува

зборот „END”.

Излез: На стандарден излез треба да се испечати за секој од влезовите следната информација: ИМЕ POS/NEG ЦЕНА КОЛИЧИНА. Доколку лекот не е најден се печати „No such drug”. Доколку нарачката на клиентот е поголема од залихата се печати „No drugs available”, инаку „Order made”.

Забелешка: Функцијата со која се врши мапирање на имињата на лековите во број е следна:

$$h(w) = (100 * (100 * (100 * 0 + \text{ASCII}(c_3)) + \text{ASCII}(c_2)) + \text{ASCII}(c_1)) \% 656565,$$

каде зборот $w = c1c2c3c4c5\dots$ е составен само од големи букви.

Исто така, за лековите да се направи посебна класа која како атрибути ќе ги има наведените карактеристики на лекот во системот.

Пример:

Влез:

5

ACEROLA 0 100 1000

ACIKLOVIR 1 1650 87

HYDROCYKLIN 0 55 10

GENTAMICIN 1 152 90

HYDROCYKLIN20 0 113 20

hydroCyklinn

2

hydroCyklin20

2

END

Излез: No such drug

HYDROCYKLIN20 NEG 113 20

Order made

Решение

Најпрвин ќе биде прикажана класата која треба да се креира за лековите (`Drug`). Оваа класа ги содржи атрибутите дадени во текстот со кои се описува лекот: име, цена, количина и информација дали лекот е на позитивна листа (1/0). Во класата се додадени `set` и `get` методи за сите атрибути и е дефиниран конструктор со параметри. Во конструкторот, името на лекот секогаш се додава со големи букви.

Во класата `Drug` е дефиниран метод `toString()` кој печати информации за лекот според барањето во текстот: име на лекот POS/NEG цена количина.

Во продолжение е дадена имплементацијата на класата `Drug`.

```
1 class Drug{
2     String name;
3     int posList;
4     int price;
5     int quantity;
6
7     public String getName() {
8         return name;
9     }
10    public void setName(String name) {
11        this.name = name;
12    }
13    public int getPrice() {
14        return price;
15    }
16    public void setPrice(int price) {
17        this.price = price;
18    }
19    public int getQuantity() {
20        return quantity;
21    }
22    public void setQuantity(int quantity) {
23        this.quantity = quantity;
24    }
25    public int getPosList() {
26        return posList;
27    }
28
29    public Drug(String name, int posList, int price, int quantity) {
30        this.name = name.toUpperCase();
31        this.posList = posList;
32        this.price = price;
33        this.quantity = quantity;
34    }
35
36    @Override
37    public boolean equals(Object obj) {
38        Drug temp = (Drug) obj;
39        return this.name.equals(temp.name);
40    }
41
```

```

42     @Override
43     public String toString() {
44         if(posList==1)
45             return name+" "+POS+" "+price+" "+quantity;
46         else
47             return name+" "+NEG+" "+price+" "+quantity;
48     }
49
50 }
```

Во оваа задача пребарувањето се врши по името на лекот и се бара да се дефинира корисничка хеш функција. Најдобро решение е да не прават промени во функцијата `hash(K key)` во класата СВНТ, туку да се дефинира нова класа `Name` со еден атрибут `name` (име на лек) каде што ќе се препокрие функцијата `hashCode()`. Во овој случај, кога ќе се повика функцијата `hash(K key)`, наместо вградената `hashCode()` функција, за имињата на лековите ќе се повикува препокриената. Хеш функцијата која ги вклучува ASCII кодовите на првите три букви од името, па тоа значи дека сите имиња на лекови кои што имаат исти први три букви ќе се сместат во иста „кофичка”.

Класата `Name` го имплементира интерфесот `Comparable`, па мора да се препокрие функцијата `compareTo(Name arg0)`, а дополнително и функцијата `equals(Object obj)`.

```

1  class Name implements Comparable<Name>{
2      String name;
3
4      public String getName() {
5          return name;
6      }
7      public void setName(String name) {
8          this.name = name;
9      }
10
11     public Name(String name) {
12         this.name = name.toUpperCase();
13     }
14     @Override
15     public boolean equals(Object obj) {
16         Name temp = (Name) obj;
17         return this.name.equals(temp.name);
18     }
19     @Override
```

```

20   public int hashCode() {
21     int hash = (100*(100*(100*0 + (name).charAt(2)) + (name).charAt(1)) +
22               (name).charAt(0));
23     return hash;
24   }
25   @Override
26   public String toString() {
27     return name;
28   }
29   @Override
30   public int compareTo(Name arg0) {
31     return name.compareTo(arg0.name);
32   }

```

Во оваа задача се користи хеш табела за зачувување и пребарување на лекови. Имено, бидејќи пребарувањето се врши по име на лекот, објекти од класата `Name` ќе бидат клучеви, а вредностите ќе бидат објекти од класата `Drug`. Се креира нова табела со број на „кофички” како што е наведено во дадената хеш функција, т.е. 656565.

Потоа се обработува влезот. Се чита бројот на лекови N кои треба да се внесат во табелата и се започнува со обработка на следните N линии. Од секоја линија кога ќе се направи поделба според празно место користејќи го методот `split("")` се добива низа од четири елементи: име на лек, информација дали е на позитивна или негативна листа (1/0), цена и моментална количина на залиха. Се креираат објекти од класите `Name` и `Drug` и во хеш табелата се додава елемент со клуч објект од класата `Name` (со име на лекот со големи букви) и со вредност објект од класата `Drug`.

Следно, треба да се обработи нарачката. Во примерот има две нарачки, една `hydroCyklinn` со количина 2 и втората `hydroCyklin20`, исто така со количина 2. Се чита името на лекот (се прави конверзија во големи букви) и се проверува дали постои елемент со таков клуч во табелата. Доколку не постои, се печати „No such drug”. Во спротивно, се печатат информациите за лекот според методот `toString()` и се проверува количината. Ако нема доволно количина од лекот, се печати „No drugs available”, а доколку има, се прави следново: се зема тековната количина на лекот со помош на методот `getQuantity()` методот од класата `Drug`, се менува вредноста со помош на методот `setQuantity()` така што од постоечката количина се одзема бараната количина. Во хеш табелата се внесува елементот со истиот клуч и новата количина. На крајот се печати „Order made”.

Процесот го повторуваме додека не се прочита зборот „END”.

Имплементацијата на решението е дадена подолу.

```

1 //Вметни класа MapEntry
2 //Вметни класа SLLNode
3 //Вметни класа CBHT
4 //Вметни класа Name
5 //Вметни класа Drug
6
7 public class Pharmacy {
8
9     public static void main(String[] args) throws Exception, IOException {
10
11         CBHT<Name,Drug> hashtable= new CBHT<Name,Drug>(656565);
12         BufferedReader br = new BufferedReader(new
13             InputStreamReader(System.in));
14         int N = Integer.parseInt(br.readLine());
15
16         for(int i=1;i<=N;i++){
17             String line = br.readLine();
18             String[] input = line.split(" ");
19             Drug drug = new Drug(input[0], Integer.parseInt(input[1]),
20                 Integer.parseInt(input[2]), Integer.parseInt(input[3]));
21             hashtable.insert(new Name(input[0].toUpperCase()),drug);
22         }
23
24         String order = (br.readLine()).toUpperCase();
25
26         while(order.compareTo("END")!=0){
27
28             int quantity = Integer.parseInt(br.readLine());
29             SLLNode<MapEntry<Name,Drug>> result = hashtable.search(new
30                 Name(order));
31
32             if(result==null){
33
34                 System.out.println("No such drug");
35                 order = (br.readLine()).toUpperCase();
36             }
37             else if(result.element.value.getName().equals(order)){
38
39                 System.out.println(result.element.value.toString());
40             }
41         }
42     }
43 }
```

```

38     if(result.element.value.getQuantity() < quantity)
39     {
40         System.out.println("No drugs available");
41     }
42     else {
43
44         int oldQuantity = result.element.value.getQuantity();
45         result.element.value.setQuantity(oldQuantity - quantity);
46         hashtable.insert(new Name(order), result.element.value);
47         System.out.println("Order made");
48     }
49     order = (br.readLine()).toUpperCase();
50 }
51 else{
52     order = br.readLine();
53 }
54 }
55 }
56 }
```

5.1.4 Задачи за вежбање

Задача 1. Телефонски именик

Даден е телефонски именик од вашиот мобилен телефон. Во тој телефонски именик за секој број е дадено името на сопственикот. Некои телефони при повик го даваат целосниот број (заедно со повикувачкиот број на земјата, на пример за Македонија +389 X XXX XXX), а некои само локалниот број (0XX XXX XXX). Ова значи дека ако во некои телефони бројот започнува со +389, во други би започнувал само со 0. Ваша задача е независно од тоа како вашиот мобилен број го прикажува бројот да го пронајдете сопственикот на бројот. Доколку го нема, треба да испечатите: „Unknown number”.

Влез: Во првиот ред е даден број на мобилни броеви N . Во следните N редови се дадени броеви (во формат 0XXXXXXXXX) и нивните сопственици разделени со празно место. Во последниот ред е даден бројот за кој треба да го одредите сопственикот.

Излез: Се печати сопственикот за дадениот број или „Unknown number” доколку го нема дадениот број во именикот.

Пример 1:

Влез:

3

070111222 IvanIvanoski

071222333 PetrePetrevski

022333444 TrajceTrajkovski

+38970111222

Излез: IvanIvanoski

Пример 2:

Влез:

3

070111222 IvanIvanoski

071222333 PetrePetrevski

022333444 TrajceTrajkovski

+38977111222

Излез: Unknown number

Задача 2. Роденди

Во заводот на статистика се прави ново истражување каде што се открива зависноста на месецот на раѓање со имињата на луѓето родени во тој месец. Ваша задача е за даден месец да ги прикажете сите различни имиња на луѓе родени во тој месец.

Влез: Во првиот ред од влезот е даден бројот на луѓе N ($N \leq 10000$), а во секој нареден ред се дадени името на човекот и датумот на неговото раѓање, разделени со празно место. Во последниот ред е даден месецот за кој треба да се прикажат сите различни имиња на луѓето родени во тој месец.

Излез: Листа со различни имиња на луѓе родени во дадениот месец. Доколку нема луѓе родени во тој месец да се испечати „Empty”.

Пример 1:

Влез:

4

Ivan 20.7.1976

Ivan 16.7.1988

Ana 18.7.1966

Ivan 5.6.1988

7

Излез:

Ivan Ana

Пример 2:

Влез:

4

Vesna 13.4.2015

Katerina 12.3.1945

Milan 14.2.1996

Olja 13.2.1988

1

Излез: Empty

Задача 3. Температура

За различни градови дадени се мерењата на температурата (степени Целзиусови) во одредени временски интервали. Ваша задача е за даден град да се најде најтоплиот период од денот.

Влез: Во првиот ред од влезот е даден бројот на мерења N ($N \leq 10000$), а во секој нареден ред е даден прво градот, потоа почеток на интервал, крај на интервал и температурата разделени со празно место. Во последниот ред е даден градот за кој треба да најдете најтопол период од денот и истиот период да се испечати. Сложеноста на оваа операција треба да биде $O(1)$.

Излез: Најтоплиот период од денот за даден град. Да се испечати во следниов формат: $G: HH:MM - XX:YY Z$, каде што G е градот, $HH:MM$ е почетокот на интервалот, $XX:YY$ е крајот на интервалот, а Z е температурата во степени Целзиусови.

Пример:

Влез:

4

Ohrid,Macedonia 10:00 12:00 23.1

Skopje,Macedonia 09:00 10:30 24

Ohrid,Macedonia 12:00 13:00 25

Skopje,Macedonia 10:00 11:00 26.2

Ohrid,Macedonia

Излез:

Ohrid,Macedonia: 12:00 - 13:00 25.0

Задача 4. Временска прогноза

За различни градови дадени се мерењата на температурата (степени Целзиусови) во одредени временски интервали. Ваша задача е за даден град да се прикаже листа од сите температурни мерења. Притоа, ако се случи при внес два интервали да се исти, тогаш се зема средната вредност од измерените температури во интервалите.

Влез: Во првиот ред од влезот е даден бројот на мерења N ($N \leq 10000$), а во секој нареден ред е даден прво градот, потоа почеток на интервал, крај на интервал и температурата разделени со празно место. Во последниот ред е даден градот за кој треба да се испечати соодветната информација.

Излез: Листата од мерењата за даден град. Да се испечати во следниов формат:

G:

HH:MM – XX:YY Z

HH:MM – XX:YY Z

...

каде што G е градот, $HH:MM$ е почетокот на интервалот, $XX:YY$ е крајот на интервалот, а Z е температурата во степени Целзиусови.

Пример 1:

Влез:

4

Ohrid 10:00 12:00 23.1

Skopje 09:00 10:30 24

Ohrid 12:00 13:00 25

Skopje 10:00 11:00 26.2

Ohrid

Излез:

Ohrid:

10:00 - 12:00 23.10

12:00 - 13:00 25.00

Пример 2:

Влез:

4

Ohrid 10:00 12:00 23.1

Skopje 09:00 10:30 24

Ohrid 12:00 13:00 25

Skopje 10:00 11:00 26.2

Strumica

Излез:

Strumica: does not exist

Задача 5. Граница

Донесен е нов закон со кој не се дозволува да ја напуштите државата со стар пасош ако во меѓувреме сте го промениле името или презимето (односно мора да имате пасош со новото име и презиме). За таа цел постојат два регистри: еден на издадени патни дозволи (пасоши) и еден на лица кои го промениле своето име или презиме. Ваша задача е за даден број на пасош (лице кое дошло на граница) да проверите дали смее да ја напушти државата (т.е. дали има промена во своето име или презиме).

Влез: Во првата линија е даден број на лица N на кои им е издадена патна дозвола. Во наредните N линии се дадени броевите на пасош и имињата и презимињата на лицата. Потоа е даден број M на лица кои го промениле своето име или презиме. Во наредните M линии дадени се прво старите, па новите имиња на лицата. Во последниот ред е даден број на пасош кој треба да се провери.

Излез: Да се испечати дали лицето со дадениот број на пасош смее („Allowed“) или не смее („Not allowed“) да на ја напушти државата.

Пример 1:

Влез:

4

A112233 IvanaIvanovska

B345680 AleksandarPetreski

A878999 ElenaTrajkovska

B783789 IvanIvanov

2

PetrankaJanevska PetrankaPetrovska

AleksandarPetreski AleksandarKocevski

B345680

Излез:

Not Allowed

Пример 1:

Влез:

4

A112233 IvanaIvanovska
B345680 AleksandarPetreski
A878999 ElenaTrajkovska
B783789 IvanIvanov
2
PetrankaJanevska PetrankaPetrovska
AleksandarPetreski AleksandarKocevski
B783789
Излез:
Allowed

Задача 6. Магацин

Во магацинот на една фармацевтска компанија се чуваат најразлични видови лекови. За секој лек потребно е да се чуваат податоци за името на лекот, цената во денари и намената на лекот. За поефикасен пристап до податоците за лековите, фармацевтската компанија одлучила податоците да ги чува во една хеш табела каде се сместуваат соодветните податоци.

Хеш табелата е достапна до крајните клиенти и истите може да пребаруваат низ внесените податоци. Бидејќи на пазарот постојат повеќе лекови кои таргетираат иста болест, најчесто клиентите го бараат овој лек кој има најниска цена. Па вашата задача е со користење на хеш табелата, за дадена намена (болест), да го испечатите лекот кој има најниска цена на пазарот.

Влез: Најпрво е даден бројот на лекови - N , а потоа секој лек е даден во нов ред во форматот:

Име на лек@Намена@Цена во денари

На крај е дадена намената за која треба да се пронајде лекот со најниска цена.

Излез: Името на лекот со најмала цена.

Пример:

Влез:

5

Analgin@Headache@80

Daleron@Headache@90

Spazmek@Stomachache@120

Lineks@Stomachache@150

Loratadin@Allergy@150

Headache

Излез:

Analgin

Задача 7. Датотеки

Информациите за организацијата на датотечниот систем на серверот на ФИНКИ се зачувани во хеш табела. За секоја датотека се знае нејзината содржина и патеката на која се наоѓа. Ваша задача е да ги најдете сите датотеки кои имаат идентична содржина.

Влез: Во првиот ред е даден бројот на датотеки N . Во следните N редици се дадени податоци за секоја датотека во формат „path file (content)”, каде што path е патеката на директориумот во кој се наоѓа датотеката, file е називот на датотеката заедно со наставката и content е содржината на датотеката. Во следниот ред е даден број на команди M . Во следните M редици се дадени команди во формат „cmd path file (content)” каде што cmd може да биде add, delete или find. Командата add треба да ја додаде датотеката file со содржина content во директориумот кој се наоѓа на патеката path. Командата delete треба да ја избрише датотеката file со содржина content од директориумот кој се наоѓа на патеката path. Командата find треба да провери дали постои датотеката file со содржина content во директориумот со патека path и да испечати на екран „true” или „false”. Во последната редица од влезот е дадена содржината content.

Излез: Листа од сите патеки на сите датотеките со содржина content.

Пример:

Влез:

```
2
root/a/ 1.txt (abcd)
root/a/ 2.txt (efgh)
3
add root/c/d/ 4.txt (efgh)
delete root/a/ 1.txt (abcd)
find root/a/ 1.txt (abcd)
efgh
```

Излез:

```
false
root/a/2.txt root/c/d 4.txt
```

5.2 Хеш табела со отворени „кофички”

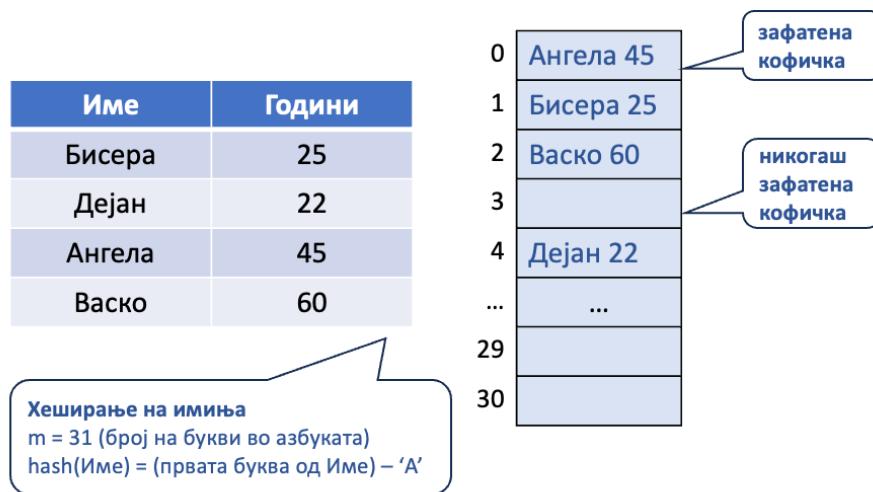
Кај хеш табелите со отворени „кофички” (OBHT) важат следните правила:

- Секоја „кофичка” содржи најмногу еден елемент;

- Во случај на колизија, новиот елемент се преместува во друга кофичка, односно „претекува“.
- „Кофичката“ може да се најде во една од трите состојби:
 - **никогаш зафатена** - никогаш не содржела елемент;
 - **засфатена** - во моментов содржи елемент;
 - **претходно зафатена** - содржела елемент, но тој е избришан и во моментов нема нов.

Во овој случај хеш табелата би се имплементирала како низа од елементи - парови (клуч, вредност).

На слика 5-4 е прикажана илустрација на хеш табела со отворени „кофички“ без колизии, додека пак на слика 5-5 е прикажана илустрација на хеш табела со отворени „кофички“ со колизии. Во случај кога има колизии, тогаш елементот се сместува во првата слободна „кофичка“. Во случајот со елементот со клуч „Ана“, бидејќи „кофичката“ 0 е зафатена, елементот се сместува во првата слободна „кофичка“, а тоа е „кофичката“ 3. Истото се случува и со елементот со клуч „Бојан“. По додавањето на сите елементи во хеш табелата се создава кластер од три елементи.



Слика 5-4: Хеш табела со отворени „кофички“ без колизии.

За да се избегнат/намалат колизиите и да се направи колку е можно подобра распределба на елементите во хеш табела со отворени кофички, при нејзиниот дизајн, покрај двата аспекти кај СВНТ (фактор на пополнетост и избор на хеш функција), треба да се внимава и на трет аспект: **Избор на должина на чекор**. Чекорот претставува број за кој се зголемува индексот на следната разгледана



Слика 5-5: Хеш табела со отворени „кофички“ со колизии.

„кофичка“, доколку домашната „кофичка“ е зафатена во моментот. Крајната цел е да нема кластери со повеќе од четири елементи.

За избегнување на колизии, алгоритмите за додавање, пребарување и бришење на елементи, доколку должината на чекорот $s = 1$ го зголемуваат индексот на „кофичката“ за еден и се обидуваат повторно. Алтернативно, може да се користи и фиксно $s > 1$. Вообичаено, бројот на „кофички“ m се бира да биде прост број, а s да биде во рангот $2 \dots m - 1$. Во тој случај, s и m немаат заеднички фактори.

Друга можност е чекорот s да се пресметува од клучот на елементот. Во тој случај, за различните клучеви ќе се добијат различни должини на чекорот, а за ист клуч секогаш ќе се добие иста вредност на чекорот. Овој принцип се нарекува двојно хеширање. За додавање, пребарување или бришење на елемент со клуч k , се пресметува s од k , користејќи втора хеш функција $s = \text{step}(k)$.

Имплементацијата на хеш табела со отворени „кофички“ е дадена со класата OBHT. Хеш табелата е претставена со низата buckets која е всушност низа од објекти од класата MapEntry.

```

1 public class OBHT<K extends Comparable<K>, E> {
2
3     //Табелата се состои од MapEntry објекти.
4     private MapEntry<K,E>[] buckets;
5
6     //buckets[b] е null ако „кофичката“ b не била никогаш зафатена.
7     //buckets[b] е претходно зафатена ако во „кофичката“ b имало претходно

```

елемент кој е избришан и моментално нема елемент во оваа „кофичка”.

```

8
9     static final int NONE = -1; //... различно од било кој индекс на
10    „кофичка”.
11
12    private static final MapEntry former = new MapEntry(null, null);
13    //Ова гарантира дека за било кој елемент е e.key.equals(former.key) е
14    false.
15
16
17    @SuppressWarnings("unchecked")
18    public OBHT (int m) {
19        //Се креира празна OBHT со m „кофички”.
20        buckets = (MapEntry<K,E>[] ) new MapEntry[m];
21    }
22
23    //методи на OBHT
24    ...
25 }
```

5.2.1 Операции со хеш табела со отворени „кофички”

Основни операции со хеш табели со отворени „кофички” вклучуваат:

- хеш функција за пресметување на индекс на домашна кофичка;
- пребарување на елемент;
- метод get што ја враќа содржината според бројот на „кофичка”.
- додавање на елемент;
- бришење на елемент;
- печатење на хеш табела.

Методот `hash(K key)`, идентично како и кај СВТ се користи за пресметување на индексот на домашната „кофичка” на клучот `key`. Оваа функција се користи во методите за додавање, пребарување и бришење на елемент од хеш табелата. Методот `hashCode()` е метод наследен од класата `Object` во Java и се користи за генерирање на целобројна (`integer`) вредност (хеш код) за даден објект. Остатокот добиен од делењето со бројот на „кофички” обезбедува дека пресметаниот индекс ќе биде во границите на низата од „кофички”.

```

1
2  private int hash(K key) {
```

```

3 //Го преведува клучот во индекс на низата buckets.
4     return Math.abs(key.hashCode()) % buckets.length;
5 }
```

Имплементацијата на **пребарување на елемент** во хеш табела со отворени „кофички“ е дадена во методот `search(K targetKey)`. Во овој метод со повикување на методот `hash(targetKey)` најпрвин се пресметува индексот на „кофичката“ `b` на клучот `targetKey`. Во `n_search` се чува бројот на изминати „кофички“ (на почеток 0). Потоа со `for` циклус се започнува пребарувањето. Се разгледуваат повеќе сценарија:

- Ако „кофичката“ `b` е никогаш зафатена, пребарувањето завршува со одговор -1 (NONE).
- Ако „кофичката“ `b` е зафатена со клуч кој е еднаков со `targetKey`, пребарувањето завршува со одговор `b`.
- Ако „кофичката“ `b` е претходно зафатена, или е зафатена со клуч кој не е еднаков со `targetKey`, `b` се зголемува за 1 по модул m (`buckets.length`). Ова се користи за да не се излезе од границите на низата. Потоа, се проверува дали `n_search` е еднаков на должината на низата. Доколку да, тоа значи дека елементот не е пронајден и се враќа -1.

```

1 public int search (K targetKey) {
2     int b = hash(targetKey);
3     int n_search=0;
4     for (;;) {
5         MapEntry<K,E> oldEntry = buckets[b];
6         if (oldEntry == null)
7             return NONE;
8         else if (targetKey.equals(oldEntry.key))
9             return b;
10        else
11        {
12            b = (b + 1) % buckets.length;
13            n_search++;
14            if(n_search==buckets.length)
15                return NONE;
16        }
17    }
18 }
19 }
```

На слика 5-6 е прикажана илустрација на пребарување во хеш табела со отворени „кофички”. Пребарувањето на елементот со клуч „Ангела” е едноставно, бидејќи тој се сместен во „кофичката” што се добива како резултат од функцијата `hash(targetKey)`. Пребарувањето на елементот со клуч „Бојан” започнува со „кофичката” 1, и бидејќи таму не е пронајден, пребарувањето продолжува во следната „кофичка” и така сè до „кофичката” број 4. Пребарувањето на елементот со клуч „Цена” започнува од „кофичката” 29, но бидејќи не е пронајден се пребарува во следната „кофичка” 30. Бидејќи таа „кофичка” има состојба на никогаш зафатена, тоа значи дека во таа „кофичка” никогаш немало елемент. Според тоа, елементот со клуч „Цена” не постои во хеш табелата.



Слика 5-6: Пребарување во ОВНТ.

Методот `getBucket(int i)` враќа објект од класата `MapEntry<K, E>`.

```

1  public MapEntry<K,E> getBucket(int i){
2      return buckets[i];
3  }

```

Операцијата **додавање на нов елемент** се имплементира со методот `insert(K key, E val)`. Во овој метод на почетокот се дефинира нов објект `newEntry` од класата `MapEntry` со клуч `key` и вредност `val`. Потоа на ист начин како и во методот `search(K key)` се наоѓа „кофичката” ѕ каде што треба да се започне со пребарување и да се провери дали може да се додаде објектот `newEntry`. Со `for` циклус започнува пребарувањето за да се најде соодветната „кофичка” каде што ќе се додаде новиот елемент. Се разгледуваат повеќе сценарија:

- Ако „кофичката” ѕ е никогаш зафатена, прво се проверува дали „кофичката” ѕ е последната никогаш зафатена „кофичка” и ако одговорот е да тогаш

ОВХТ е преполнета. Се додава елементот во „кофичката“ b и таа станува зафатена.

- Ако „кофичката“ b е претходно-зрафатена, или е зафатена од клуч кој е еднаков со key , се додава елементот во „кофичката“ b и таа станува зафатена.
- Ако „кофичката“ b е зафатена од клуч кој е различен од key , b се зголемува за 1 по модул m ($buckets.length$). Ова се користи за да не се излезе од границите на низата.

```

1  public void insert (K key, E val) {
2      MapEntry<K,E> newEntry = new MapEntry<K,E>(key, val);
3      int b = hash(key);
4      int n_search=0;
5      for (;;) {
6          MapEntry<K,E> oldEntry = buckets[b];
7          if (oldEntry == null) {
8              if (++occupancy == buckets.length) {
9                  System.out.println("Hash table is full!!!!");
10             }
11             buckets[b] = newEntry;
12             return;
13         }
14         else if (oldEntry == former || key.equals(oldEntry.key)) {
15             buckets[b] = newEntry;
16             return;
17         }
18         else
19         {
20             b = (b + 1) % buckets.length;
21             n_search++;
22             if(n_search==buckets.length)
23                 return;
24         }
25     }
26 }
27 }
```

На слика 5-7 е прикажана илустрација на додавање на елементи во хеш табела со отворени „кофички“. Додавањето на елементот со клуч „Ана“ започнува со проверка на „кофичката“ 0, но таа е зафатена. Потоа алгоритмот продолжува со проверки во следните кофички, сè додека не ја најде првата слободна кофичка, а тоа е „кофичката“ број 3. Додавањето на елементот со клуч „Дејан“ е поеднос-

тавно бидејќи „кофичката” 4 е слободна.



Слика 5-7: Додавање во ОВНТ.

Бришењето на елемент од хеш табела со отворени „кофички” е имплементирано со методот `delete(K key)`. Слично, како и во претходните два методи, потребно е да се најде домашната „кофичка” на елементот со клуч `key`. Најпрвин се наоѓа „кофичката” `b` каде што треба да се започне со пребарување. Потоа со `for` циклус се започнува пребарувањето за да се најде елементот што треба да се избрише. Се разгледуваат повеќе сценарија:

- Ако „кофичката” `b` е никогаш зафатена, тоа значи дека елементот не постои и пребарувањето завршува;
- Ако „кофичката” `b` е зафатена со клуч еднаков со `key`, „кофичката” станува претходно зафатена и пребарувањето завршува;
- Ако „кофичката” `b` е претходно зафатена, или е зафатена со клуч кој е различен од `key`, `b` се зголемува за 1 по модул m (`buckets.length`). Ова се користи за да не се излезе од границите на низата.

```

1  @SuppressWarnings("unchecked")
2  public void delete (K key) {
3      int b = hash(key); int n_search=0;
4      for (;;) {
5          MapEntry<K,E> oldEntry = buckets[b];
6
7          if (oldEntry == null)
8              return;
9          else if (key.equals(oldEntry.key)) {
10              buckets[b] = former; //MapEntry<K,E>)former;
11              return;
12      }

```

```

13     else{
14         b = (b + 1) % buckets.length;
15         n_search++;
16         if(n_search==buckets.length)
17             return;
18     }
19 }
20 }
21 }
```

На слика 5-8 е прикажана илустрација на бришење на елементи во хеш табела со отворени „кофички”. Бришењето на елементот со клуч „Бисера” е едноставно бидејќи тој се наоѓа во „кофичката” пресметана со функцијата `hash(key)`, а тоа е „кофичката” 1. По бришењето на овој елемент „кофичката” 1 добива состојба на претходно зафатена. Бришењето на елементот со клуч „Ана” започнува со пребарување на клучот во „кофичката” 0. Бидејќи таму има елемент со различен клуч, алгоритмот продолжува да пребарува во следната „кофичка”. Во „кофичката” 1 нема елемент, но таа има состојба на претходно зафатена, па тоа значи дека алгоритмот ќе продолжи да пребарува во следната „кофичка”. Доколку „кофичката” 1 немаше состојба на претходно зафатена ќе се добиеше погрешен одговор дека елементот не постои во хеш табелата. Елементот е пронајден во „кофичката” 3 и истиот е избришан од табелата.



Слика 5-8: Бришење во ОВНТ.

Доколку се анализираат методите за додавање, пребарување и бришење на елемент од хеш табела со отворени „кофички” и при тоа се земе предвид бројот на споредби кои се вршат, за n елементи може да се заклучи следното:

- Во најдобар случај нека не постои кластер со повеќе од 4 соседни пополнени

ти „кофички”. Според тоа, може да се смета дека сложеноста во најдобар случај е $O(1)$.

- Во најлош случај нека има еден кластер од сите елементи и максималниот број на споредби е n . Сложеноста во најлош случај е $O(n)$.

Методот за печатење на хеш табела се имплементира со препокривање на методата `toString()` во Java која враќа `String`. Започнува со дефинирање на резултантен стринг `temp` и потоа се изминува низата на „кофички”. Доколку „кофичката” е претходно зафатена се додава зборот „former”, а доколку е никогаш зафатена не се додава текст. Бидејќи методот линерно ги изминува сите елементи во низата, неговата сложеност е $O(n)$.

```

1 public String toString () {
2     String temp = "";
3     for (int i = 0; i < buckets.length; i++) {
4         temp += i + ":";
5         if (buckets[i] == null)
6             temp += "\n";
7         else if (buckets[i] == former)
8             temp += "former\n";
9         else
10            temp += buckets[i] + "\n";
11    }
12    return temp;
13 }
```

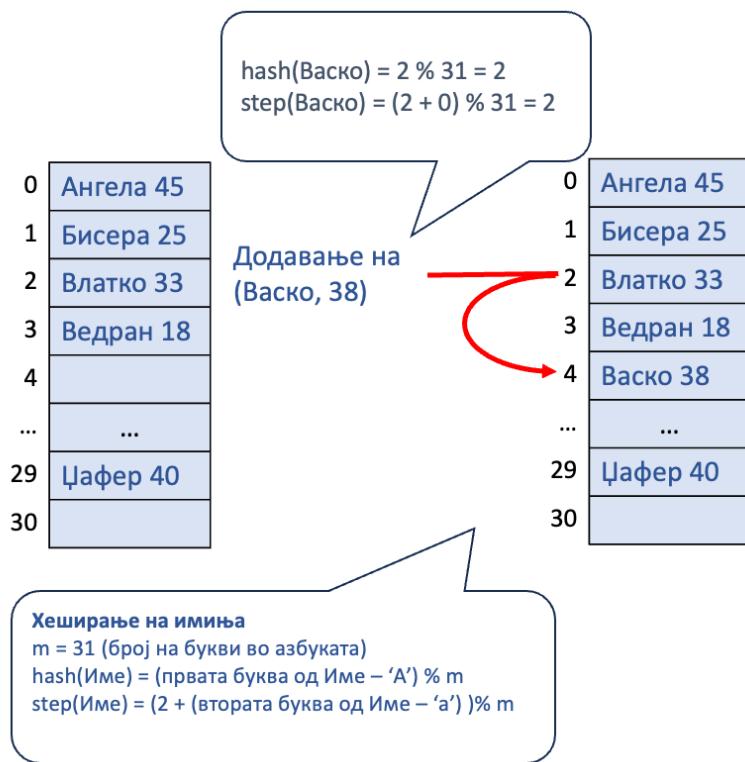
5.2.2 Двојно хеширање

Доколку чекорот s за кој се зголемува индексот на следната разгледана „кофичка” во хеш табела со отворени „кофички” при пребарување, додавање или бришење на елемент се пресметува од клучот k користејќи втора хеш функција $s = \text{step}(k)$, станува збор за **двојно хеширање**. Тоа значи дека за различни клучеви ќе се добијат различни должини на чекорот, а за ист клуч секогаш ќе се добие иста вредност за чекорот. На пример, при додавање на нов елемент, доколку веќе постои друг елемент на позицијата $\text{hash}(k)$, т.е. се добил ист хеш код за друг елемент во табелата, тогаш новиот елемент ќе биде додаден на позиција $\text{hash}(k) + \text{step}(k)$ (по модул m).

Операцијата додавање на нов елемент со двојно хеширање кај хеш табели со отворени „кофички” е иста како додавањето со единствено хеширање, освен во сценариото каде што „кофичката” b е зафатена со клуч чија вредност е различна

од клучот key на елементот што се додава. Во тој случај, наместо за $1, b$ се зголемува за $\text{step}(k)$ по модул m .

На слика 5-9 е прикажана илустрација на двојно хеширање при додавање на елемент во хеш табела со отворени „кофички”. Според алгоритмот за додавање, прво се проверува дали елементот со клуч „Васко” може да биде додаден во „кофичката” 2. Бидејќи таа е зафатена, а резултатот од функцијата $\text{step}(k)$ е 2, следната „кофичка” која ќе се провери дали е слободна е „кофичката” 4 ($2+2$). Тоа значи дека наместо да се проверува „кофичката” 3, па потоа „кофичката” 4, веднаш се преминува со проверка на „кофичката” 4. „Кофичката” 4 е слободна, па елементот со клуч „Васко” се додава во истата.



Слика 5-9: Двојно хеширање во ОВНТ.

Во продолжение е прикажан интерфејс кој треба да се имплементира при двојно хеширање. Покрај веќе описаната функцијата `hashCode()`, постои и функција `stepCode()` која што на идентичен начин како и `hashCode()` ја пресметува должината на чекорот.

```

1 public interface DoublyHashable<K> extends Comparable<K> {
2
3     public int hashCode ();
4     //Враќа хеш код за клучот.

```

```

5
6     public int stepCode ();
7     //Враќа должина на чекор за клучот.
8
9 }
```

Имплементацијата на хеш табела со отворени „кофички” и двојно хеширање е дадена со класата DoublyHashedOBHT.

```

1  public class DoublyHashedOBHT <K extends DoublyHashable<K>,E> {
2
3     //Табелата се состои од MapEntry објекти.
4     private MapEntry<K,E>[] buckets;
5
6     //buckets[b] е null ако „кофичката” b не била никогаш зафатена.
7     //buckets[b] е претходно зафатена ако во „кофичката” b имало претходно
8     //елемент кој е избришен и моментално нема елемент во оваа „кофичка”.
9
10    static final int NONE = -1; //... различно од било кој индекс на
11        //„кофичка”.
12
13    private static final MapEntry former = new MapEntry(null, null);
14    //Ова гарантира дека за било кој елемент е e.key.equals(former.key) е
15        //false.
16
17    @SuppressWarnings("unchecked")
18    public DoublyHashedOBHT (int m) {
19        //Се креира празна DoublyHashedOBHT со m „кофички”.
20        buckets = (MapEntry<K,E>[] ) new MapEntry[m];
21    }
22
23    //методи на DoublyHashedOBHT
24    ...
25 }
```

Покрај дефинираните методи за OBHT, кај хеш табелата со двојно хеширање се додава и методот `step(K key)`. Во продолжение е дадена имплементацијата на функцијата `step(K key)` која ја повикува функцијата `stepCode()`.

```

1  private int step (K key) {
2      return Math.abs(key.stepCode()) % buckets.length;
3  }
```

Имплементацијата на методот за додавање на елемент кај хеш табела со двојно хеширање е прикажан подолу. Промените во споредба со ОВТ се означени со портокалова боја. Истите промени се прават и кај методите за пребарување и бришење на елемент.

```
1 public void insert (K key, E val) {
2     MapEntry<K,E> newEntry = new MapEntry<K,E>(key, val);
3     int b = hash(key);
4     int s = step(key);
5     int n_search = 0;
6     for (;;) {
7         MapEntry<K,E> oldEntry = buckets[b];
8         if (oldEntry == null) {
9             if (++occupancy == buckets.length) {
10                 System.out.println("Hash table is full!!!!");
11             }
12             buckets[b] = newEntry;
13             return;
14         } else if (oldEntry == former || key.equals(oldEntry.key)) {
15             buckets[b] = newEntry;
16             return;
17         }
18     else {
19         b = (b + s) % buckets.length;
20         n_search++;
21         if (n_search == buckets.length)
22             return;
23     }
24 }
25 }
26 }
```

5.2.3 Едноставни проблеми со хеш табели со отворени „кофички“

Задача 1. Црвен крст

Во рамки на една хуманитарна организација, потребно е да се направи статистика за крвните групи кои се на располагање за донација, и од кои донатори.

Подгрупите A1+, A2+ припаѓаат на крвна група A+, додека A1-, A2- припаѓаат на група A-.

Влез: Во првиот ред од влезот е даден бројот на парови N , а во секој нареден ред се дадени паровите (донатор, крвна група).

Излез: Да се испечати по колку донатори има од секоја крвна група согласно внесените податоци.

Пример:

Влез:

5

Alek A1+

Dejan B-

Sandra A+

Trajce 0+

Rebeka A1-

Излез:

A+=2

B-=1

0+=1

A-=1

Решение

Во дадениот пример, A+=2 е пресметано од крвните групи на Алек и Сандрата, B-=1 од Дејан, 0+=1 е од Трајче, а A-=1 доаѓа од крвната група на Ребека.

Бидејќи треба да изброиме колку луѓе припаѓаат на дадените крвни групи, клучот во хеш табелата ќе биде крвната група. Притоа, доколку има подгрупа (1 или 2), цифрата треба да се отстрани од клучот со функцијата `ReplaceAll("[1-2]", "")` која се применува на објект од класата `String`.

При првото додавање на одредена крвна група како клуч во хеш табелата ќе се внесе парот (крвна група, 1), а при секое наредно додавање на елемент со клуч од истата крвна група, треба да се зголеми вредноста за 1. Процедурата би била следна: се зема тековната вредност на елементот за дадениот клуч и се заменува елементот со истиот клуч со вредност зголемена за 1. Функцијата за додавање `insert (K key, E val)` прави замена на елементот, доколку се внесува елемент со постоечки клуч.

Во оваа задача потребно е да се направи и промена на методот `toString()` во класата `OBHT`, за да се добие излезот во посакуваниот формат.

Имплементацијата на решението е дадена подолу.

¹ `import java.io.BufferedReader;`

```
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4
5 class MapEntry<K extends Comparable<K>,E> implements Comparable<K> {
6     K key;
7     E value;
8
9     public MapEntry (K key, E val) {
10         this.key = key;
11         this.value = val;
12     }
13     public int compareTo (K that) {
14         @SuppressWarnings("unchecked")
15         MapEntry<K,E> other = (MapEntry<K,E>) that;
16         return this.key.compareTo(other.key);
17     }
18     public String toString () {
19         return "(" + key + "," + value + ")";
20     }
21 }
22
23 class OBHT<K extends Comparable<K>,E> {
24
25     private MapEntry<K,E>[] buckets;
26     static final int NONE = -1;
27     @SuppressWarnings({ "rawtypes", "unchecked" })
28     private static final MapEntry former = new MapEntry(null, null);
29     private int occupancy = 0;
30
31     @SuppressWarnings("unchecked")
32     public OBHT (int m) {
33         buckets = (MapEntry<K,E>[] ) new MapEntry[m];
34     }
35
36     private int hash (K key) {
37         return Math.abs(key.hashCode()) % buckets.length;
38     }
39
40     public MapEntry<K,E> getBucket(int i){
41         return buckets[i];
42     }
```

```

43
44     public int search (K targetKey) {
45         int b = hash(targetKey); int n_search=0;
46         for (;;) {
47             MapEntry<K,E> oldEntry = buckets[b];
48             if (oldEntry == null)
49                 return NONE;
50             else if (targetKey.equals(oldEntry.key))
51                 return b;
52             else{
53                 b = (b + 1) % buckets.length;
54                 n_search++;
55                 if(n_search==buckets.length)
56                     return NONE;
57             }
58         }
59     }
60
61     public void insert (K key, E val) {
62         MapEntry<K,E> newEntry = new MapEntry<K,E>(key, val);
63         int b = hash(key); int n_search=0;
64
65         for (;;) {
66             MapEntry<K,E> oldEntry = buckets[b];
67             if (oldEntry == null) {
68                 if (++occupancy == buckets.length) {
69                     System.out.println("Hash table is full!!!!");
70                 }
71                 buckets[b] = newEntry;
72                 return;
73             } else if (oldEntry == former
74                   || key.equals(oldEntry.key)) {
75                 buckets[b] = newEntry;
76                 return;
77             } else{
78                 b = (b + 1) % buckets.length;
79                 n_search++;
80                 if(n_search==buckets.length)
81                     return;
82             }
83         }

```

```
84         }
85     }
86
87     @SuppressWarnings("unchecked")
88     public void delete (K key) {
89         int b = hash(key); int n_search=0;
90         for (;;) {
91             MapEntry<K,E> oldEntry = buckets[b];
92
93             if (oldEntry == null)
94                 return;
95             else if (key.equals(oldEntry.key)) {
96                 buckets[b] = former;
97                 return;
98             } else{
99                 b = (b + 1) % buckets.length;
100                n_search++;
101                if(n_search==buckets.length)
102                    return;
103
104            }
105        }
106    }
107
108    public String toString () {
109        String temp = "";
110        for (int i = 0; i < buckets.length; i++) {
111            if (buckets[i] == null)
112                temp += "";
113            else
114                temp += buckets[i].key + "=" + buckets[i].value + "\n";
115        }
116        return temp;
117    }
118 }
119
120
121    public class RedCross {
122
123        public static void main (String[] args) throws IOException {
124
```

```

125     OBHT<String, Integer> hashtable = new OBHT<String, Integer>(11);
126     BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
127     int N = Integer.parseInt(br.readLine());
128     String input;
129
130     for(int i=1;i<=N;i++){
131
132         input = br.readLine();
133         String[] row = input.split(" ");
134         String key = row[1];
135         key = key.replaceAll("[1-2]", "");
136
137         if(hashtable.search(key)==-1)
138         {
139
140             hashtable.insert(key, 1);
141
142         }
143         else
144         {
145
146             hashtable.insert(key,
147                         hashtable.getBucket(hashtable.search(key)).value+1);
148
149         }
150     }
151
152     System.out.println(hashtable);
153 }
154 }
```

Задача 2. Квалитет на воздух

Дадени се мерења на PM10 честички за населбите во Скопје. Ваша задача е за дадена населба да ја најдете просечната концентрација на PM10 честички.

Влез: Во првиот ред од влезот е даден бројот на мерења, а во секој нареден ред се дадени населбата и концентрацијата на PM10 честички разделени со празно место. Во последниот ред е дадена населбата за која треба да најдете просечна концентрација на PM10 честички.

Излез: Просечната концентрација на PM10 честички за дадената населба

(заокружена на 2 децимали, притоа прво наоѓате просечна концентрација, па заокружувате).

Пример:

Влез:

8
Centar 319.61
Karposh 296.74
Centar 531.98
Karposh 316.44
GaziBaba 384.05
GaziBaba 319.3
Karposh 393.37
GaziBaba 326.42
Karposh

Излез:

355.52

Решение

Во примерот се пребарува по населба, па тоа значи дека името на населбата треба да биде клуч. Бидејќи треба да се пресмета просек од вредности за PM10 честички, една опција е да се зачуваат вредностите во низа или листа. Така, во хеш табелата се внесуваат парови во следната форма (име на населба, низа/листа од PM10 честички за населбата). Во конкретниот пример, за креирање на низа се користи класата `ArrayList` од Java.

Доколку во хеш табелата нема елемент со клучот на елементот кој се внесува, тогаш се креира нов објект од класата `ArrayList` и се додава вредноста на PM10 честичките.

Во случај ако постои елемент со таков клуч, се зема елементот од хеш табелата, во низата (вредноста на елементот) се додава новата вредност на PM10 честички и повторно се внесува елементот во хеш табелата.

Потоа се пребарува по дадената населба и доколку постои во хеш табелата се наоѓа сума на сите PM10 вредности од низата за тој клуч, се пресметува просек и се печати заокружен на две децимали. Ако не постои елемент со клучот по кој се пребраува, тогаш се печати „No info”.

Во случајов, бидејќи се пребарува по населбата „Karposh”, просечната концентрација на PM10 честички се добива како просек од вредностите за населбата „Karposh” $(296.74+316.44+393.37)/3 = 355.52$.

Имплементацијата на решението е дадена подолу.

```
1 //Вметни класа MapEntry
2 //Вметни класа OBHT
3
4 public class PM10 {
5
6     public static void main (String[] args) throws IOException {
7
8
9         BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
10        int N = Integer.parseInt(br.readLine());
11        OBHT<String, ArrayList<Double>> hashtable = new
12            OBHT<String,ArrayList<Double>>(2*N);
13        String input;
14
15        for(int i=1;i<=N;i++){
16
17            input = br.readLine();
18            String[] row = input.split(" ");
19            String neighbourhood = row[0];
20            String pm10 = row[1];
21            ArrayList<Double> list=new ArrayList<Double>();
22
23            if(hashtable.search(neighbourhood)==-1)
24            {
25                list.add(Double.parseDouble(pm10));
26                hashtable.insert(neighbourhood,list);
27            }
28
29            else
30            {
31                list=hashtable.getBucket(hashtable.search(neighbourhood)).value;
32                list.add(Double.parseDouble(pm10));
33                hashtable.insert(neighbourhood, list);
34            }
35        }
36
37        String neighbourhoodSearch=br.readLine();
38        ArrayList<Double>result =
39            hashtable.getBucket(hashtable.search(neighbourhoodSearch)).value;
40        double sum=0;
```

```

40
41     if(result.size()>0)
42     {
43         for(int i=0;i<result.size();i++)
44         {
45             sum+=result.get(i);
46         }
47
48         System.out.printf("%.2f", sum/result.size());
49     }
50     else
51     {
52         System.out.println("No info");
53     }
54
55 }
56 }
```

Задача 3. Роденденi

Дадена е листа од датуми на раѓање на сите вработени во една организација. Ваша задача е за даден датум да испечатите кои вработени на организацијата ќе слават роденден тој ден и колку години ќе полнат (сортирани според името на вработениот). Доколку за дадениот датум нема роденден да се испечати „Empty”.

Влез: Во првиот ред од влезот е даден бројот на вработени N , а во следните N редови се дадени името и презимето на вработениот и датата на раѓање (формат dd/mm/yyyy) одделени со едно празно место. Во последниот ред е даден датумот за кој треба да испечатите кои луѓе слават роденден на тој датум.

Излез: Името на вработениот и колку години полни, сортирани според името на вработениот.

Пример 1:

Влез:

3

Ivana Ivanovska 15/05/1982
 Elena Todorovska 30/05/1984
 Maja Petrevska 15/05/1986
 15/05/2023

Излез:

Ivana Ivanovska 41

Maja Petrevska 37

Пример 2:

Влез:

4

Ivana Ivanovska 15/05/1982

Elena Todorovska 30/05/1984

Maja Petrevska 15/05/1986

Stefan Stefanovski 30/05/1975

15/04/2023

Излез:

Empty

Решение

Во оваа задача се креира хеш табела со отворени „кофички” со парови од клуч датум (ден и месец) во форма на текст, а за вредностите се креира класа `Employee` во која ќе се зачуваат името, презимето и датумот на раѓање на вработениот.

За даден датум ќе треба да се зачуваат повеќе објекти од класата `Employee`, па затоа вредностите во табелата ќе бидат зачувани во низи од објекти од класата `Employee`. За да може да се сортираат објектите од класата `Employee` според нивното име, во класата ќе се додаде метод `CompareTo(Employee obj)` каде што ќе се дефинира дека споредбата помеѓу два објекти од класата `Employee` ќе се извршува според името.

Имплементацијата на класата `Employee` е дадена подолу.

```

1
2 class Employee implements Comparable<Employee>{
3
4     String name;
5     String surname;
6     String dateB;
7
8     public Employee(String name, String surname, String dateB) {
9         super();
10        this.name = name;
11        this.surname = surname;
12        this.dateB = dateB;
13    }
14    public String getName() {

```

```

15     return name;
16 }
17 public void setName(String name) {
18     this.name = name;
19 }
20 public String getSurname() {
21     return surname;
22 }
23 public void setSurname(String surname) {
24     this.surname = surname;
25 }
26 public String getDateB() {
27     return dateB;
28 }
29 public void setDateB(String dateB) {
30     this.dateB = dateB;
31 }
32 @Override
33 public int compareTo(Employee o) {
34     return this.name.compareTo(o.name);
35 }
36 @Override
37 public String toString() {
38     return this.name+" "+this.surname;
39 }
40 }

```

При додавање на елементи во хеш табелата, постојат две опции. Првата е доколку елемент со таков клуч постои, а втората доколку елементот со даден клуч не постои во табелата. Клучот се добива од датумот со земање само на првите 5 карактери, со методот `substring(0,5)`. И во двата случаи при обработка на влезот се креира нов објект од класата `Employee`.

Во првиот случај, се пребарува објектот со истиот клуч и неговата вредност (низата од вработени) се менува, така што се додава новокреираниот објект од класата `Employee`. Елементот се додава повторно во хеш табелата и со тоа се пребришнува стариот елемент со истиот клуч. Во вториот случај, се креира празна низа од тип `Employee` и во табелата се додава елемент со клуч датум и вредност низа со објектот од класата `Employee` креiran претходно.

Потоа се пребарува во хеш табелата според дадениот датум (само денот и месецот). Доколку елементот постои, кога ќе се пристапи вредноста се добива низа (`ArrayList`) од објекти од класата `Employee`. Потоа се сортира низата со

функцијата `Collections.sort()`. Сортирањето се врши по име на вработениот бидејќи така е дефиниран методот `CompareTo()` во класата `Employee`. Годините се пресметуваат така што од внесената година „2023” се одзема годината кога е роден вработениот. Годината кога е роден вработениот се добива со земање на карактерите од позиција 6 до позиција 10 во датумот со помош на методот `substring(6, 10)`.

Доколку елементот не постои, се печати „Empty”.

Во дадениот пример 1 се бараат сите луѓе кои што се родени на 15 мај. Такви има двајца, „Ivana Ivanovska” и „Maja Petrevska”. Во вториот пример не постои вработен со роденден на 15 април.

Имплементацијата на решението е дадена подолу.

```

1 //Вметни класа MapEntry
2 //Вметни класа OBHT
3 //Вметни класа Vraboten
4
5 public class Birthdays {
6
7     public static void main(String[] args) throws NumberFormatException,
8         IOException {
9
10        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
11        int N = Integer.parseInt(br.readLine());
12        OBHT<String,ArrayList<Employee>> hashtable = new
13            OBHT<String,ArrayList<Employee>>(2*N);
14        String input;
15
16        for(int i=1;i<=N;i++){
17
18            input = br.readLine();
19            String[] elems = input.split(" ");
20            Employee emp = new Employee(elems[0],elems[1],elems[2]);
21            String key = elems[2].substring(0, 5);
22
23            if(hashtable.search(key)!=-1)
24            {
25                MapEntry<String,ArrayList<Employee>> result =
26                    hashtable.getBucket(hashtable.search(key));
27                ArrayList<Employee> array = result.value;
28                array.add(emp);
29                hashtable.insert(key, array);
30            }
31        }
32    }
33}
```

```
27
28     }
29     else
30     {
31         ArrayList<Employee> a = new ArrayList<Employee>();
32         a.add(emp);
33         hashtable.insert(key, a);
34     }
35 }
36 String dateIn = br.readLine();
37 String date = dateIn.substring(0,5);
38 int yearIn = Integer.parseInt(dateIn.substring(6,10));
39
40 if(hashtable.search(date)!=-1)
41 {
42     MapEntry<String,ArrayList<Employee>> result =
43         hashtable.getBucket(hashtable.search(date));
44     ArrayList<Employee> niza = result.value;
45     Employee[] p = new Employee[niza.size()];
46     for(int i = 0;i<p.length;i++)
47         p[i] = niza.get(i);
48     Arrays.sort(p);
49     for(int i=0;i<p.length;i++){
50         int year = Integer.parseInt(p[i].getDateB().substring(6, 10));
51         System.out.println(p[i].toString()+" "+(yearIn-year));
52     }
53     else
54     {
55         System.out.println("Empty");
56     }
57
58 }
59 }
```

5.2.4 Напредни проблеми со хеш табели со отворени „кофички“

Задача 1. Проверка на спелување

За даден текст на английски јазик, потребно е да се направи проверка дали е правилно напишан, односно дали правилно се напишани зборовите. За таа цел прво се внесува речник на зборови (односно листа на зборови кои ги содржи английскиот јазик), а потоа е даден текст. Како резултат треба да се испечатат сите зборови кои се неправилно напишани или ги нема во речникот.

Влез: Прво е даден број N на поими кои ќе ги содржи речникот, а во наредните N реда се дадени зборовите кои ги содржи английскиот јазик. Потоа е даден текст, кој треба да се провери дали е правилно напишан.

Излез: Се печати листа на зборови кои се неправилно напишани (секој во посебен ред). Доколку сите зборови се добро напишани се печати: Bravo.

Забелешка: Треба да се игнорираат интерпункциски знаци како точка (.), запирка (,), извичник (!) и прашалник (?). Исто така, да се внимава на голема и мала буква, односно иако зборовите во речникот се со мали букви, во реченица може да појават со голема почетна буква и притоа се сметаат за точни. Треба да се одреди бројот на „кофички“ и хеш функцијата.

Пример:

Влез:

4

where

is

my

cat

Where is my Ccat?

Излез:

Ccat

Решение

Бидејќи пребарувањето се врши по зборови, во хеш табелата клучеви ќе бидат зборовите. Најдобро е клучевите да бидат внесени со мали букви, и притоа да се отстранат интерпункциските знаци „?“, „!“, „.“ и „““. Вредности ќе бидат оригиналните зборови кои може да содржат и големи букви, секако со отстранети интерпункциски знаци.

За конвертирање на зборот во мали букви се користи функцијата `toLowerCase()` од `String` класата, а за отстранување на интерпункциските знаци

се користи функцијата `replaceAll(String regex, String replacement)`.

Во оваа задача е потребно да се дефинира хеш функција. Ќе дефинираме посебна класа `Word` со атрибут `word` како `String` и ќе ја препокриеме функцијата `hashCode()`. Така, кога ќе се повика функцијата `hashCode()` во `hash(K key)`, за зборовите ќе се повикува препокриената `hashCode()` функција. Во препокриената `hashCode()` функција, најпрвин се пресметува сума од сите ASCII карактери во зборот, а потоа се додава должината на зборот.

Во продолжение е дадена имплементацијата на класата `Word`.

```

1  class Word implements Comparable<Word>{
2      String word;
3
4      public Word(String word) {
5          this.word = word;
6      }
7      @Override
8      public boolean equals(Object obj) {
9          Word temp = (Word) obj;
10         return this.word.equals(temp.word);
11     }
12     @Override
13     public int hashCode() {
14         int hash=0;
15         for (int i = 0; i < word.length(); i++) {
16             hash += word.charAt(i); //Збир на ASCII вредностите на сите
17             //карактери
18         }
19         hash+=word.length(); //Должина на зборот
20
21         return hash;
22     }
23     @Override
24     public String toString() {
25         return word;
26     }
27     @Override
28     public int compareTo(Word arg0) {
29         return word.compareTo(arg0.word);
30     }

```

При пребарувањето во хеш табелата доколку се добие резултат -1, тоа значи

дека тој збор не постои и во тој случај се печати истиот. Во случај кога сите зборови постојат, се печати „Bravo”. За таа цел има дефинирано бројач m , кој се зголемува при секој точно напишан збор што го има во речникот. Во дадениот пример, погрешно е напишан зборот „Seat” и истиот се печати на излез.

Имплементацијата на решението е дадена подолу.

```

1 //Вметни класа MapEntry
2 //Вметни класа OBHT
3 //Вметни класа Word
4
5 public class Spelling {
6     public static void main(String[] args) throws IOException {
7
8         BufferedReader br = new BufferedReader(new
9             InputStreamReader(System.in));
10        int N = Integer.parseInt(br.readLine());
11        OBHT<Word, String> hashtable = new OBHT<Word, String>(2*N);
12
13        for(int i=0;i<N;i++)
14        {
15            Word word = new Word(br.readLine());
16            String newWord = word.word.toLowerCase().replaceAll("\\"?,
17                "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
18                .replaceAll("\\", " ");
19            hashtable.insert(new Word(newWord), word.word.replaceAll("\\"?,
20                "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
21                .replaceAll("\\", " ));
22
23        }
24
25        String text = br.readLine();
26        String [] p = text.split(" ");
27        int m = 0;
28
29        for(int i=0; i<p.length; i++)
30        {
31            String original = p[i];
32            p[i] = p[i].toLowerCase().replaceAll("\\"?, " ")
33                .replaceAll("\\"!, " ").replaceAll("\\".", " ")
34                .replaceAll("\\", " );
35
36            if(hashtable.search(new Word(p[i]))==-1)
37            {
38                System.out.println(original.replaceAll("\\"?,
39                ""));
40            }
41        }
42    }
43
44    public static void main(String[] args) throws IOException {
45        BufferedReader br = new BufferedReader(new
46            InputStreamReader(System.in));
47        int N = Integer.parseInt(br.readLine());
48        OBHT<Word, String> hashtable = new OBHT<Word, String>(2*N);
49
50        for(int i=0;i<N;i++)
51        {
52            Word word = new Word(br.readLine());
53            String newWord = word.word.toLowerCase().replaceAll("\\"?,
54                "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
55                .replaceAll("\\", " ");
56            hashtable.insert(new Word(newWord), word.word.replaceAll("\\"?,
57                "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
58                .replaceAll("\\", " ));
59
60        }
61
62        String text = br.readLine();
63        String [] p = text.split(" ");
64        int m = 0;
65
66        for(int i=0; i<p.length; i++)
67        {
68            String original = p[i];
69            p[i] = p[i].toLowerCase().replaceAll("\\"?, " ")
70                .replaceAll("\\"!, " ").replaceAll("\\".", " ")
71                .replaceAll("\\", " );
72
73            if(hashtable.search(new Word(p[i]))==-1)
74            {
75                System.out.println(original.replaceAll("\\"?,
76                ""));
77            }
78        }
79    }
80
81    public static void main(String[] args) throws IOException {
82        BufferedReader br = new BufferedReader(new
83            InputStreamReader(System.in));
84        int N = Integer.parseInt(br.readLine());
85        OBHT<Word, String> hashtable = new OBHT<Word, String>(2*N);
86
87        for(int i=0;i<N;i++)
88        {
89            Word word = new Word(br.readLine());
90            String newWord = word.word.toLowerCase().replaceAll("\\"?,
91                "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
92                .replaceAll("\\", " ");
93            hashtable.insert(new Word(newWord), word.word.replaceAll("\\"?,
94                "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
95                .replaceAll("\\", " ));
96
97        }
98
99        String text = br.readLine();
100       String [] p = text.split(" ");
101      int m = 0;
102
103      for(int i=0; i<p.length; i++)
104      {
105          String original = p[i];
106          p[i] = p[i].toLowerCase().replaceAll("\\"?, " ")
107              .replaceAll("\\"!, " ").replaceAll("\\".", " ")
108              .replaceAll("\\", " );
109
110          if(hashtable.search(new Word(p[i]))==-1)
111          {
112              System.out.println(original.replaceAll("\\"?,
113                  ""));
114          }
115      }
116  }
117
118  public static void main(String[] args) throws IOException {
119      BufferedReader br = new BufferedReader(new
120          InputStreamReader(System.in));
121      int N = Integer.parseInt(br.readLine());
122      OBHT<Word, String> hashtable = new OBHT<Word, String>(2*N);
123
124      for(int i=0;i<N;i++)
125      {
126          Word word = new Word(br.readLine());
127          String newWord = word.word.toLowerCase().replaceAll("\\"?,
128              "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
129              .replaceAll("\\", " ");
130          hashtable.insert(new Word(newWord), word.word.replaceAll("\\"?,
131              "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
132              .replaceAll("\\", " ));
133
134      }
135
136      String text = br.readLine();
137      String [] p = text.split(" ");
138      int m = 0;
139
140      for(int i=0; i<p.length; i++)
141      {
142          String original = p[i];
143          p[i] = p[i].toLowerCase().replaceAll("\\"?, " ")
144              .replaceAll("\\"!, " ").replaceAll("\\".", " ")
145              .replaceAll("\\", " );
146
147          if(hashtable.search(new Word(p[i]))==-1)
148          {
149              System.out.println(original.replaceAll("\\"?,
150                  ""));
151          }
152      }
153  }
154
155  public static void main(String[] args) throws IOException {
156      BufferedReader br = new BufferedReader(new
157          InputStreamReader(System.in));
158      int N = Integer.parseInt(br.readLine());
159      OBHT<Word, String> hashtable = new OBHT<Word, String>(2*N);
160
161      for(int i=0;i<N;i++)
162      {
163          Word word = new Word(br.readLine());
164          String newWord = word.word.toLowerCase().replaceAll("\\"?,
165              "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
166              .replaceAll("\\", " ");
167          hashtable.insert(new Word(newWord), word.word.replaceAll("\\"?,
168              "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
169              .replaceAll("\\", " ));
170
171      }
172
173      String text = br.readLine();
174      String [] p = text.split(" ");
175      int m = 0;
176
177      for(int i=0; i<p.length; i++)
178      {
179          String original = p[i];
180          p[i] = p[i].toLowerCase().replaceAll("\\"?, " ")
181              .replaceAll("\\"!, " ").replaceAll("\\".", " ")
182              .replaceAll("\\", " );
183
184          if(hashtable.search(new Word(p[i]))==-1)
185          {
186              System.out.println(original.replaceAll("\\"?,
187                  ""));
188          }
189      }
190  }
191
192  public static void main(String[] args) throws IOException {
193      BufferedReader br = new BufferedReader(new
194          InputStreamReader(System.in));
195      int N = Integer.parseInt(br.readLine());
196      OBHT<Word, String> hashtable = new OBHT<Word, String>(2*N);
197
198      for(int i=0;i<N;i++)
199      {
200          Word word = new Word(br.readLine());
201          String newWord = word.word.toLowerCase().replaceAll("\\"?,
202              "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
203              .replaceAll("\\", " ");
204          hashtable.insert(new Word(newWord), word.word.replaceAll("\\"?,
205              "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
206              .replaceAll("\\", " ));
207
208      }
209
210      String text = br.readLine();
211      String [] p = text.split(" ");
212      int m = 0;
213
214      for(int i=0; i<p.length; i++)
215      {
216          String original = p[i];
217          p[i] = p[i].toLowerCase().replaceAll("\\"?, " ")
218              .replaceAll("\\"!, " ").replaceAll("\\".", " ")
219              .replaceAll("\\", " );
220
221          if(hashtable.search(new Word(p[i]))==-1)
222          {
223              System.out.println(original.replaceAll("\\"?,
224                  ""));
225          }
226      }
227  }
228
229  public static void main(String[] args) throws IOException {
230      BufferedReader br = new BufferedReader(new
231          InputStreamReader(System.in));
232      int N = Integer.parseInt(br.readLine());
233      OBHT<Word, String> hashtable = new OBHT<Word, String>(2*N);
234
235      for(int i=0;i<N;i++)
236      {
237          Word word = new Word(br.readLine());
238          String newWord = word.word.toLowerCase().replaceAll("\\"?,
239              "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
240              .replaceAll("\\", " ");
241          hashtable.insert(new Word(newWord), word.word.replaceAll("\\"?,
242              "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
243              .replaceAll("\\", " ));
244
245      }
246
247      String text = br.readLine();
248      String [] p = text.split(" ");
249      int m = 0;
250
251      for(int i=0; i<p.length; i++)
252      {
253          String original = p[i];
254          p[i] = p[i].toLowerCase().replaceAll("\\"?, " ")
255              .replaceAll("\\"!, " ").replaceAll("\\".", " ")
256              .replaceAll("\\", " );
257
258          if(hashtable.search(new Word(p[i]))==-1)
259          {
260              System.out.println(original.replaceAll("\\"?,
261                  ""));
262          }
263      }
264  }
265
266  public static void main(String[] args) throws IOException {
267      BufferedReader br = new BufferedReader(new
268          InputStreamReader(System.in));
269      int N = Integer.parseInt(br.readLine());
270      OBHT<Word, String> hashtable = new OBHT<Word, String>(2*N);
271
272      for(int i=0;i<N;i++)
273      {
274          Word word = new Word(br.readLine());
275          String newWord = word.word.toLowerCase().replaceAll("\\"?,
276              "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
277              .replaceAll("\\", " ");
278          hashtable.insert(new Word(newWord), word.word.replaceAll("\\"?,
279              "").replaceAll("\\"!, " ").replaceAll("\\".", " ")
280              .replaceAll("\\", " ));
281
282      }
283
284      String text = br.readLine();
285      String [] p = text.split(" ");
286      int m = 0;
287
288      for(int i=0; i<p.length; i++)
289      {
289          String original = p[i];
290          p[i] = p[i].toLowerCase().replaceAll("\\"?, " ")
291              .replaceAll("\\"!, " ").replaceAll("\\".", " ")
292              .replaceAll("\\", " );
293
294          if(hashtable.search(new Word(p[i]))==-1)
295          {
296              System.out.println(original.replaceAll("\\"?,
297                  ""));
298          }
299      }
300  }

```

```

        "").replaceAll("\\!", "").replaceAll("\\.", "",
        "").replaceAll("\\", "");

31     }
32     else
33     {
34         m++;
35     }
36
37 }
38
39 if(m==p.length)
40     System.out.println("Bravo");
41 }
42 }
```

Задача 2. Уписи на факултет

Секој кандидат кој сака да се запише на факултет поднесува електронска пријава на системот за упис на Универзитетот. Потоа комисијата за упис ги проверува еден по еден кандидатите и нивните внесени податоци, а посебно го проверува успехот на кандидатот од средно во апликацијата Е-Дневник (која содржи податоци за сите ученици во сите средни училишта во Македонија). Ваша задача е за даден кандидат да проверите дали е валиден внесениот просек од средно училиште во апликацијата за уписи.

Влез: Во првата линија е даден број N на кандидати кои сакаат да се запишат на факултет. Во наредните N линии се дадени матичните броеви на кандидатите и просек од средно образование кој го внеле во апликацијата за уписи. Потоа е даден број M на податоци во Е-Дневник. Во наредните M линии се дадени матичните броеви на средношколците и нивниот вистински просек од средно образование. Во последниот ред е даден матичниот број на кандидатот чиј просек треба да се провери.

Излез: Да се испечати дали кандидатот со дадениот матичен број го внел точниот просек од средно образование („OK”), дали просекот е погрешно внесен („Error”) или пак кандидатот воопшто го нема во Е-Дневник („Empty”).

Пример 1:

Влез:

```

2
0610992333666 5.0
0901993222233 4.78
```

4
2205990121212 2.45
0901993222233 4.68
0610992333666 5.0
1511989984256 3.45
0901993222233

Излез:

Error

Пример 2:

Влез:

2
0610992333666 5.0
0901993222233 4.78
4
2205990121212 2.45
0901993222233 4.68
0610992333666 5.0
1511989984256 3.45
0610992333666

Излез:

OK

Пример 3:

Влез:

2
0610992333666 5.0
0901993222233 4.78
4
2205990121212 2.45
0901993222233 4.68
0610992333663 5.0
1511989984256 3.45
0610992333666

Излез:

Empty

Решение

Во оваа задача се креираат две посебни хеш табели, една за складирање на кандидатите кои сакаат да се запишат на факултет и втората за складирање на податоците од Е-Дневник.

И двете табели имаат ист тип на клучеви (матичен број како текст) и вредности (просек како децимална вредност).

Кога ќе се прочита матичниот број кој треба да се пребара, се проверува дали постои во двете хеш табели и доколку постои, се проверува дали просекот од двете табели за дадениот клуч е еднаков. Во тој случај се печати „OK”. Доколку постои клучот, но просеците се различни се печати „Error”. Ако клучот не постои во некоја од табелите се печати „Empty”.

Во пример 1, се пребарува матичниот број „0901993222233”. Како што може да се забележи овој број го има и во двете табели, но просекот во првата е „4.78”, а во втората „4.68”. Затоа се печати „Error”. Во пример 2, просекот на матичниот број кој се пребарува „0610992333666” е еднаков во двете табели и затоа се печати „OK”. Во пример 3, матичниот број по кој се пребарува не постои во втората табела и се печати „Empty”.

Имплементацијата на решението е прикажана подолу.

```

1 //Вметни класа MapEntry
2 //Вметни класа OBHT
3
4 public class FacultyEnrollment {
5     public static void main(String[] args) throws
6         NumberFormatException, IOException {
7
8         BufferedReader br = new BufferedReader(new
9             InputStreamReader(System.in));
10        int N = Integer.parseInt(br.readLine());
11        OBHT<String, Double> candidates = new OBHT<String, Double>(2*N);
12        String input;
13
14        for (int i = 0; i < N; i++) {
15            input = br.readLine();
16            String[] p = input.split(" ");
17            candidates.insert(p[0], Double.parseDouble(p[1]));
18        }
19
20        int M = Integer.parseInt(br.readLine());
21        OBHT<String, Double> gradebook = new OBHT<String, Double>(2*M);

```

```

20
21     for (int i = 0; i < M; i++) {
22         input = br.readLine();
23         String[] p = input.split(" ");
24         gradebook.insert(p[0], Double.parseDouble(p[1]));
25     }
26
27     String PIN = br.readLine();
28
29     if(candidates.search(PIN)!=-1){
30         if(gradebook.search(PIN)!=-1){
31             if(candidates.getBucket(candidates.search(PIN)).value.equals(gradebook.getBucket(PIN).value))
32                 System.out.println("OK");
33             else
34                 System.out.println("Error");
35         }
36         else
37             System.out.println("Empty");
38     }
39     else
40         System.out.println("Empty");
41 }
42 }
```

Задача 3. Семафори

На семафорите низ градот, планирано е да се постават радари со камери кои ќе ги забележуваат возилата што возат над дозволената брзина. За таа цел, потребно е да се чуваат информации за регистарските таблички со информациите на сопственикот на возилото (име и презиме), така што во моментот кога ќе биде забележана регистарската табличка на возило што ја надминал дозволената брзина, треба да се добијат информациите за сопственикот на возилото со сложеност $O(1)$.

Влез: Во првиот ред е даден бројот на регистарски таблички N . Секоја од следните N линии содржи информација за регистарската табличка и името и презимето на сопственикот на возилото. Во следниот ред е дадена максималната дозволена брзина, а потоа во нов ред е даден дневниот извештај од радарот, односно, листа од возила со регистарски таблички што биле забележани од радарот, со која брзина поминале и точното време, одвоени со празно место (редоследот не е хронолошки).

Излез: Дневниот извештај од радарот треба да се преработи, така што треба да се добијат информации за возачите кои направиле престап. Излезот треба да биде во формат: име и презиме на сопствениците на возила кои ја надминале дозволената брзина, одвоени со празно место и подредени според времето кога радарот ги забележал.

Забелешка: За времето може да искористите некоја од следните класи на Java: `SimpleDateFormat` или `Date`.

Пример:

Влез:

5

SK1234AA Anita Angelovska
 OH1212BE Aleksandar Antov
 ST0989OO Ognen Spirovski
 ST0000AB Sara Spasovska
 SK8888KD Dino Ackov

50

SK8888KD 48 14:00:00 ST0000AB 55 12:00:02 ST0989OO 60 08:10:00 SK1234AA
 65 20:00:10 OH1212BE 50 22:00:21

Излез:

Ognen Spirovski
 Sara Spasovska
 Anita Angelovska

Решение

Хеш табелата во оваа задача ќе се користи за зачувување на регистрациите и податоците за возачите (име и презиме). Клуч е регистрацијата, а вредност е името и презимето на возачите.

Креирана е класа `Driver` со атрибути име, презиме и време (објект од класата `Date` од Java). Во класата `Driver` е имплементиран и методот `CompareTo(Driver o)` каде што е дефиниран начинот на споредување на двајца возачи, по нивното време кога радарот ги забележал. Ова е потребно бидејќи на излез листата со возачи треба да биде сортирана по времето.

Во продолжение е дадена имплементацијата на класата `Driver`.

```

1 class Driver implements Comparable<Driver>{
2     String name;
3     String surname;
4     Date time;
5

```

```

6   public Driver(String name, String surname, Date time) {
7       this.name = name;
8       this.surname = surname;
9       this.time = time;
10  }
11
12  public Date getTime() {
13      return time;
14  }
15
16  @Override
17  public String toString() {
18      return name + " " + surname;
19  }
20
21  @Override
22  public int compareTo(Driver o) {
23
24      return this.getTime().compareTo(o.getTime());
25  }
26 }
```

По дефинирането на хеш табелата, истата се полни со податоците за регистрациите и имињата на возачите. Во хеш табелата се внесуваат парови во формат (регистрација, име и презиме).

Покомплексниот дел во оваа задача е обработувањето на влезните податоци за дневниот извештај. За да може да се прочита датумот и да се зачува во објект од класата `Date` се користи објект од класата `SimpleDateFormat("HH:mm:ss")`. Обработувањето на дневниот извештај се врши на следниот начин: во еден циклус се читаат три податоци од дневниот извештај (регистрација, брзина и час). Доколку прочитаната брзина е поголема од дозволената брзина, во хеш табелата за дадената регистрација се бара името и презимето на возачот. Возачот се додава во листата `drivers` (објект од класата `LinkedList` од Java).

Откако ќе се додадат сите возачи кои ја надминале дневната брзина, листата се сортира. Сортирањето се врши со методот `Collections.sort(drivers)`. За споредба на два објекти од класата `Driver` се користи методот `CompareTo()`.

На крај, се печатат сите возачи од листата која се изминува со помош на итератор (`ListIterator`).

Во дадениот пример дозволената брзина е 50 km/h, а според извештајот возачите со регистерски таблички „ST0000AB”, „ST0989OO” и „SK1234AA” ја надминале брзината. Доколку се сортираат по време, би се добило „ST0989OO” во

, „08:10:0”, „ST0000AB” во „12:00:02” и „SK1234AA” во „20:00:10” часот. Доколку се пребраат сопствениците на возилата со дадените регистрации по истиот редослед, се добива „Ognen Spirovski”, „Sara Spasovska” и „Anita Angelovska”.

Имплементацијата на решението е дадена подолу.

```

1 //Вметни класа MapEntry
2 //Вметни класа OBHT
3 //Вметни класа Driver
4
5 public class TrafficLights {
6     public static void main(String[] args) throws IOException,
7         ParseException {
8
9         BufferedReader bf = new BufferedReader(new
10            InputStreamReader(System.in));
11         int N = Integer.parseInt(bf.readLine());
12         OBHT<String, String> hashtable = new OBHT<String, String>(2*N);
13
14         for(int i = 0;i<N;i++){
15             String p [] = bf.readLine().split(" ");
16             hashtable.insert(p[0],p[1]+" "+p[2]);
17         }
18         SimpleDateFormat formatter=new SimpleDateFormat("HH:mm:ss");
19
20         int speed = Integer.parseInt(bf.readLine());
21         String traffic [] = bf.readLine().split(" ");
22         LinkedList<Driver> drivers = new LinkedList<Driver>();
23
24         for(int i = 0;i<(traffic.length-2);i+=3){
25
26             String plateDriver = traffic[i];
27             int speedDriver = Integer.parseInt(traffic[i+1]);
28             String timeDriver = traffic[i+2];
29
30             if(speedDriver>speed){
31                 String pom[] =
32                     hashtable.getBucket(hashtable.search((plateDriver))).value.split(" ");
33                 drivers.add(new
34                     Driver(pom[0],pom[1],formatter.parse(timeDriver)));
35             }
36         }
37     }
38 }
```

```

33
34     Collections.sort(drivers);
35     ListIterator<Driver> listIterator = drivers.listIterator();
36
37     while (listIterator.hasNext()) {
38         System.out.println(listIterator.next().toString());
39     }
40 }
41 }
```

5.2.5 Задачи за вежбање

Задача 1. Повикувачки број на земја

Ваша задача е при повик од даден број да испечатите од која земја потекнува бројот. Секоја земја има свој повикувачки код (за Македонија е 389), но кога го гледате бројот на екран напред има + (ако ве бара број од Македонија +389XXXXXXX). Повикувачките кодови на земјите можат да бидат едноцифрени, двоцифрени или троцифрени. Нека бројот на цифри на кодовите на земјите се одредува според следните правила: Ако првата цифра на повикувачкиот код е 1, тогаш кодот е едноцифрен и се работи за САД. Ако првата цифра е 2 тогаш кодот е двоцифрен и ако првата цифра е 3 тогаш кодот е троцифрен.

Влез: Во првата линија од влезот даден е бројот на повикувачки кодови за земји - N . Во следните N линии се дадени повикувачките кодови и нивните земји соодветно (разделени со празно место). Во последната линија е даден бројот за кој треба да одредите од која земја доаѓа според неговиот повикувачки код.

Излез: На излез се печати земјата за дадениот број.

Забелешка: Сами треба да ја имплементирате хеш функцијата, како и да ја дефинирате големината на хеш табелата.

Пример:

Влез:

```

12
1 SoedinetiAmerikanskiDrzavi
20 Egipet
21 Maroko
26 Zambia
351 Portugalija
355 Albanija
359 Bugarija
```

372 Estonija
381 Srbija
385 Hrvatska
387 BosnaiHercegovina
389 Makedonija
+2611332345678
Излез: Zambija

Задача 2. Најди го криминалецот

Министерството за внатрешни работи добило модерна опрема со која може да врши анализа на ДНК доказите од местата на злосторствата. Со оваа опрема МВР изградиле база на податоци и за секој криминалец се чуваат 2 карактеристични примероци (нишки од карактеристични делови од ДНК), а секој од овие примероци се состои од низа од 4 можни нуклеотиди: А, С, Г или Т. Ваша задача е за дадените примероци (нишки) од ДНК најдени на местото на злосторството да проверите дали веќе ги имате во базата на податоци и кој е сопственикот на најдените примероци. Доколку и двата извадени примероци припаѓаат на еден сопственик, се испишува името на сопственикот, во спротивно се испишува: „Unknown”.

Влез: Во првиот ред е даден бројот N на криминалци кои ги содржи базата на податоци. Во наредните редови за секој криминалец прво е дадено неговото име и презиме, а потоа во двата наредни реда се дадени неговите примероци од ДНК. На крај се дадени двата примероци кои се најдени на местото на злосторството.

Излез: Се печати името и презимето на криминалецот доколку и двата примероци на ДНК се совпаѓаат. Во спротивно се печати: „Unknown”.

Пример 1:

Влез:

3

IvanIvanovski

ACGTGTACCATGATAG

GGTACGATCCT

ElenaPetrevska

GTACCGATGCTAGGATC

ACGTAGCTCCGGATCG

KostaKostovski

CGCTAATTAAAGC

TAGACTCGATCGCT

ACGTGTACCATGATAG

GGTACGATCCT

Излез: IvanIvanovski

Пример 2:

Влез:

3

IvanIvanovski

ACGTGTACCATGATAG

GGTACGATCCT

ElenaPetrevska

GTACCGATGCTAGGATC

ACGTAGCTCCGGATCG

KostaKostovski

CGCTAATTAAAGC

TAGACTCGATCGCT

ACGTACCATGATA

A

Излез: Unknown

Задача 3. Пермутации

Користејќи хеш табела да се направи групирање на зборовите кои се пермутации еден на друг од дадена листа со зборови. Пермутација на даден збор е збор кој се добива со преуредување на буквите од зборот. На пример pots е пермутација на stop.

Влез: Во првиот ред е даден бројот на зборови N . Во наредните N реда се дадени зборовите кои треба да се додадат во табелата. Во следниот ред е даден зборот за кој треба да се испечати бројот на зборови во табелата кои се негови пермутации.

Излез: Бројот на зборови во табелата кои се пермутации на дадениот збор.

Пример:

Влез:

4

stop

tsop

ooos

toos

tosp

Излез:

2

Задача 4. Испитна сесија

Објавен е распоред на испитна сесија на ФИНКИ. Во него се дадени датуми, време на почеток, имиња на предмети и простории. Ваша задача е за даден датум да испечатите кои испити се полагаат на тој датум (со сите нивни детали).

Влез: Во првиот ред од влезот е даден бројот на испити N , а во следните N реда се дадени датум и време на полагањето (формат dd/mm/yyyyhh:mm), просторијата и името на испитот. Во последниот ред е даден датумот за кој треба да испечатите кои испити се полагаат тој датум.

Излез: Деталите на испитите за тој датум, сортирани според времето на почеток на испитот.

Пример:

Влез:

5

27/01/2016 14:00 Rooms Kalkulus 1/Matematika 1

27/01/2016 08:00 Laboratories Napredno programiranje

28/01/2016 08:00 Laboratories Algoritmi i podatochni strukturi

28/01/2016 14:00 Rooms Diskretna matematika 1

28/01/2016 09:00 315 Kalkulus 3

28/01/2016

Излез:

08:00 Laboratories Algoritmi i podatochni strukturi

09:00 315 Kalkulus 3

14:00 Rooms Diskretna matematika 1

Задача 5. Компанија

Во една компанија организацијата на работните позиции е дадена како пар (вработен, менаџер) со што се прикажува кој менаџер е раководител на кој вработен. Потребно е да се направи извештај во компанијата за тоа колку вработени има секој менаџер под негово раководство.

Влез: Во првиот ред од влезот е даден бројот на парови кои ќе се внесуваат N ($N \leq 10000$), а во секој нареден ред се дадени паровите (име_вработен, име_менаџер).

Излез: Листа со имиња на менаџерите и тоа колку вработени имаат под нивно раководство, сортирана според имињата на менаџерите.

Пример:

Влез:

6

Aleksandra,Marko

Beti,Marko

Marko,Filip

Darko,Elena

Elena,Filip

Filip,Filip

Излез:

Elena: 1

Filip: 5

Marko: 2

Задача 6. Подароците на Дедо Мраз

Дедо Мраз преку целата година води список на деца кои биле добри, а ги има и нивните адреси за да им достави подароци. Така е и со децата од Скопје, но градот Скопје решил да менува имиња на улици и Дедо Мраз во последен момент добил листа со изменети имиња на улици. Проверете за дадено дете дали Дедо Мраз треба да му достави подарок (дали го има детето во списокот на добри деца) и ако треба, на која адреса ќе му го достави.

Влез: Во првата линија е даден број N на деца кои биле добри. Во наредните N линии се дадени името на детето и неговата адреса (адресата е во формат Име На Улица Број). Потоа е даден број M на улици од Скопје кои го промениле своето име. Во наредните M линии дадени се прво старите, па новите имиња на улиците. Во последниот ред е дадено името на детето кое треба да се провери.

Излез: Ако даденото дете не било добро (т.е. го нема во списокот на добри деца) да се испечати „No gift”, а ако било добро да се испечати валидната адреса на која ќе се достави подарокот (т.е. ако името на улицата се променило, да се испечати адресата со новото име на улицата).

Пример:

Влез:

3

Ivana Vodnjanska 4

Marko Leninova 18/2

Elena StivNaumov 10

4

Vodnjanska MajkaTereza

Leninova AmintaTreti

StivNaumov SlavkoJanevski

AdolfCiborovski GoranStefanovski

Elena

Излез:

SlavkoJanevski 10

Задача 7. Веб домени

Еден веб домен како „courses.finki.ukim.mk” се состои од повеќе поддомени. На највисокото ниво е „mk”, потоа на следното ниво е „ukim.mk”, па „finki.ukim.mk”, а на најниското ниво е „courses.finki.ukim.mk”. Кога ја посетуваме страната „courses.finki.ukim.mk”, имплицитно ги посетуваме и „finki.ukim.mk”, „ukim.mk” и „mk”.

Треба да се избројат посетите на дадените домени. Еден пример на број на посети за даден домен може да биде: „**5043 courses.finki.ukim.mk**” каде 5043 е бројот на посети за веб доменот „courses.finki.ukim.mk”.

Влез: На влез се добива листа (со големина N) во која се чуваат домените и бројот на нивните посети. Потоа се читаат M зборови за кои сакаме да го добијеме бројот на посетите при што експлицитно ќе се бројат посетите на сите поддомени.

Излез: За секој збор што се пребарува да се испечати бројот на посети од сите поддомени. Доколку поддоменот не постои да се испечати „Not found”.

Пример 1:

Влез:

1

5043 courses.finki.ukim.mk

3

courses.finki.ukim.mk

finki.ukim.mk

ukim.mk

mk

Излез:

5043

5043

5043

5043

Појаснување:

Имаме еден веб домен: „courses.finki.ukim.mk”. Како што е објаснето погоре, поддоменот „finki.ukim.mk”, „ukim.mk”, „mk” исто така ќе бидат посетени. Затоа сите тие ќе бидат посетени 5043 пати.

Пример 2:**Влез:**

```
4
900 google.mail.com
50 yahoo.com
1 intel.mail.com
5 wiki.org
5
mail.com
com
org
yahoo.com
test
```

Излез:

```
901
951
5
50
Not found
```

Појаснување: Го посетуваме „google.mail.com” 900 пати, „yahoo.com” 50 пати, „intel.mail.com” еднаш и „wiki.org” 5 пати. За поддомените, ќе го посетиме „mail.com” $900 + 1 = 901$ пати, „com” ќе го посетиме $900 + 50 + 1 = 951$ пати, и „org” 5 пати. Поддоменот „test” не постои па затоа се печати „Not found”.

5.3 Hashmap и Hashtable во Java

Класите `HashMap<K,V>` и `Hashtable<K,V>` се готови класи од Java за хеш табели со затворени кофички, каде што првиот параметар K е клуч, а вториот V вредност.

Постојат два параметри `int initialCapacity` (почетен капацитет на хеш табелата) и `float loadFactor` (фактор на пополнетост) кои што во стандардните конструктори имаат предефинирана вредност, но исто така може да се дефинираат и од корисниците.

Конструкторот `Hashtable()` креира празна хеш табела со иницијален капацитет 11 и фактор на пополнетост 0.75, додека конструкторот `HashMap()` креира празна хеш табела со иницијален капацитет 16 и фактор на пополнетост 0.75. Доколку се зголеми бројот на елементи и факторот на пополнетост излезе од границите, тогаш табелата динамички се зголемува и се прави повторно хеширање на елементите, т.е. тие добиваат нови домашни „кофички”.

Единствената разлика помеѓу класите `HashMap<K, V>` и `Hashtable<K, V>` е во тоа што `HashMap<K, V>` дозволува `null` за клуч и вредност.

Класите `HashMap<K, V>` и `Hashtable<K, V>` во Java обезбедуваат неколку методи за работа со паровите (клуч, вредност) и самата хеш табела. Во прилог се дадени описи на неколку од методите:

- `containsKey(Object key)` и `containsValue(Object value)` - враќа `true` ако клучот/вредноста постои во табелата;
- `get(Object key)` - го враќа објектот (од тип `V`) за дадениот клуч `key`. Доколку нема клуч `key` во табелата, враќа `null`;
- `put(K key, V value)` - поставува вредност `value`, за дадениот клуч `key`;
- `remove(Object key)` - го брише елементот со клуч `key` (ако постои) и го враќа.

Во продолжение е даден решен пример за хеш табела во која се внесуваат буквите од англиската азбука како клучеви (`Character`) и нивната позиција како вредност (`Integer`). Напрвин е прикажано решението со користење на `Hashtable<K, V>`, а потоа со `HashMap<K, V>`.

```

1 import java.util.Hashtable;
2
3 public class TestHashtable {
4     public static void main(String[] args) {
5         Hashtable<Character, Integer> m = new Hashtable<Character, Integer>();
6         String s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
7         for (int i = 0; i < s.length(); i++) {
8             m.put(s.charAt(i), i); //се полни хеш табелата
9         }
10
11         System.out.println(m);
12         //ја земаме вредноста за буквата Z
13         System.out.println("The value of letter Z is " + m.get('Z'));
14
15     }
16 }
```

```

1 import java.util.HashMap;
2 import java.util.Map;
3
4 public class TestHashMap {
5     public static void main(String[] args) {
6         Map<Character, Integer> m = new HashMap<Character, Integer>();

```

```

7     String s = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
8     for (int i = 0; i < s.length(); i++) {
9         m.put(s.charAt(i), i); //се полни хеш табелата
10    }
11
12    System.out.println(m);
13    //ја земаме вредноста за буквата Z
14    System.out.println("The value of letter Z is " + m.get('Z'));
15
16 }
17 }
```

Доколку корисникот сака да дефинира сопствена класа за клуч во класите `HashMap<K,V>` и `Hashtable<K,V>` мора да ги имплементира методите `int hashCode()` и `boolean equals(Object obj)`.

Да се потсетиме на **Задача 1. Проверка на спелување** од секцијата 5.2.4.

Во оваа задача за даден текст на англиски јазик, потребно е да се направи проверка дали е правилно напишан, односно дали правилно се напишани зборовите. За таа цел прво се внесува речник на зборови (односно листа на зборови кои ги содржи англискиот јазик), а потоа е даден текст. Како резултат треба да се испечатат сите зборови кои се неправилно напишани или ги нема во речникот.

Влез: Прво е даден број N на поими кои ќе ги содржи речникот, а во наредните N реда се дадени зборовите кои ги содржи англискиот јазик. Потоа е даден текст, кој треба да се провери дали е правилно напишан.

Излез: Се печати листа на зборови кои се неправилно напишани (секој во посебен ред). Доколку сите зборови се добро напишани се печати: Bravo.

Забелешка: Треба да се игнорираат интерпункциски знаци како точка (.), запирка (,), извичник (!) и прашалник (?). Исто така, да се внимава на голема и мала буква, односно иако зборовите во речникот се со мали букви, во реченица може да појават со голема почетна буква и притоа се сметаат за точни. Треба да се одреди бројот на „кофички“ и хеш функцијата.

Пример:

Влез:

```

4
where
is
my
cat
```

Where is my Ccat?

Излез:

Ccat

Во оваа задача беше направена нова класа Word со еден атрибут `word` (збор) во која се преоптовари функцијата `int hashCode()`. Бидејќи класата го имплементира интерфејсот Comparable, во истата мораше да се преоптовари методот `int compareTo(Word arg0)`. Дополнително, имплементиран е и методот `boolean equals(Object obj)` со кој се споредуваат два збора.

Имплементацијата на класата Word е дадена подолу.

```
1 class Word implements Comparable<Word>{
2     String word;
3
4     public Word(String word) {
5         this.word = word;
6     }
7     @Override
8     public boolean equals(Object obj) {
9         Word temp = (Word) obj;
10        return this.word.equals(temp.word);
11    }
12    @Override
13    public int hashCode() {
14        int hash=0;
15        for (int i = 0; i < word.length(); i++) {
16            hash += word.charAt(i);
17        }
18        hash+=word.length();
19
20        return hash;
21    }
22    @Override
23    public String toString() {
24        return word;
25    }
26    @Override
27    public int compareTo(Word arg0) {
28        return word.compareTo(arg0.word);
29    }
30 }
```

Претходно, во главата 5.2.4 беше прикажано решение со користење на хеш табела со отворени „кофички”, користејќи ја класата `OBHT`. Во продолжение е прикажано решение со користење на класата `Hashtable`.

```

1 //Вметни класа Word
2 public class SpellingHashtable {
3
4 public static void main(String[] args) throws IOException {
5
6     BufferedReader br = new BufferedReader(new
7         InputStreamReader(System.in));
8     int N = Integer.parseInt(br.readLine());
9     Hashtable<Word, String> hashtable = new Hashtable<Word, String>(2*N);
10
11    for(int i=0;i<N;i++)
12    {
13        Word word = new Word(br.readLine());
14        String newWord = word.word.toLowerCase().replaceAll("\\"?,
15            "").replaceAll("\\"!, ""),
16            "").replaceAll("\\", "");
17        hashtable.put(new Word(newWord), word.word.replaceAll("\\"?,
18            "").replaceAll("\\"!, "").replaceAll("\\".,
19            "").replaceAll("\\", ""));
20    }
21
22    String text = br.readLine();
23    String [] p = text.split(" ");
24    int m = 0;
25
26    for(int i=0; i<p.length; i++)
27    {
28
29        String original = p[i];
30        p[i] = p[i].toLowerCase().replaceAll("\\"?, ""),
31            "").replaceAll("\\"., "").replaceAll("\\", "");
32
33        if(hashtable.get(new Word(p[i]))==null)
34        {
35            System.out.println(original.replaceAll("\\"?,
36                "").replaceAll("\\"!, "").replaceAll("\\".,
37                "").replaceAll("\\", ""));
38        }
39        else
40        {
41            m++;
42        }
43    }
44
45    System.out.println(m);
46
47 }

```

```
34     }
35
36     }
37
38     if(m==p.length)
39         System.out.println("Bravo");
40
41     }
42
43 }
```

Глава 6

Дрва

6.1 Теоретски основи за дрва

6.1.1 Вовед

Дрвата во најопшта форма претставуваат хиерархиска колекција на елементи. Со нив се претставуваат хиерархиски структури и како такви се една од најважните нелинеарни структури што се сретнуваат во компјутерските алгоритми. Не секоја хиерархиска структура е дрво, но секоја хиерархиска структура може да биде претставена во облик на дрво.

Интуитивно, под нелинеарна структура „дрво“ се подразбира групирање на објекти преку гранење слично на дрвата во природата. Дрво е колекција од елементи наречени јазли (анг. nodes), од кои еден јазел се нарекува корен (анг. root). Во јазлите може да се чуваат произволни податочни структури, односно кориснички-дефинирани класи на различни објекти. Јазлите се поврзани со линии ребра. За јазлите од оваа колекција е дефинирана релацијата „е родител“. Во секој од јазлите на дрвото може да се сместат податоци од кој бил податочен тип.

Причината зошто се употребуваат дрва е бидејќи ги комбинираат предностите од две други структури: подредена низа и поврзана листа. Дрвото може да се пребарува брзо, слично како и подредена низа, но може да се додаваат и бришат елементи брзо, слично како што може кај поврзаните листи. Генерално ќе не интересираат бинарните дрва, но на почеток ќе се запознаеме со дрвата воопшто, а потоа и со процесот за нивна трансформација во бинарни дрва.

6.1.2 Терминологија

Поформално, едно дрво може да се опише со следната дефиниција:

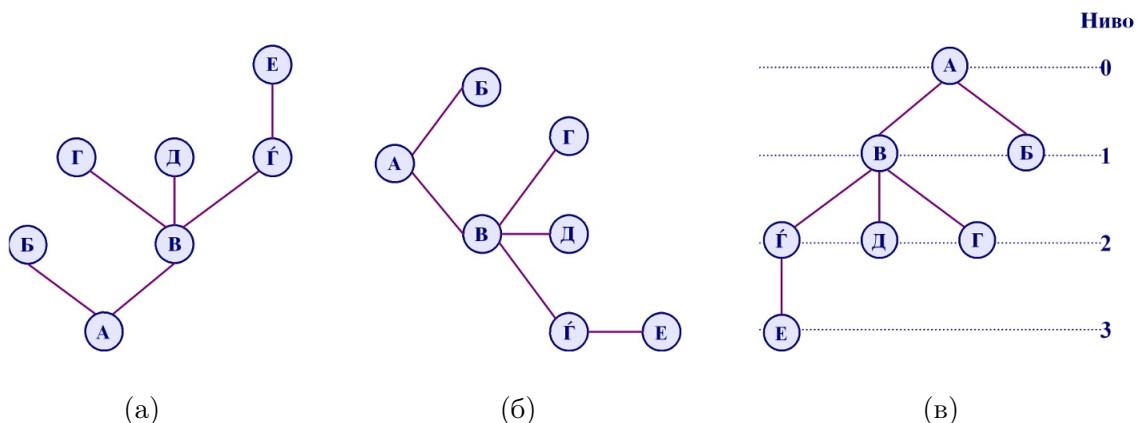
1. Јазел сам за себе претставува дрво. Тогаш, јазелот е и корен на дрвото.
2. Нека n е јазел и T_1, T_2, \dots, T_k се дрва со корени n_1, n_2, \dots, n_k соодветно. Тогаш дрво може да се конструира ако јазелот n го направиме корен на дрвото што ги содржи поддрвата T_1, T_2, \dots, T_k . Јазлите n_1, n_2, \dots, n_k ги нарекуваме деца на јазелот n .

Коренот на секое дрво е единствен јазел во дрвото што нема родител. Јазел кој има барем еден наследник се нарекува нетерминален (внатрешен) јазел. Ако еден јазел нема наследници, тогаш истиот се нарекува терминален јазел или лист.

Дрвата можат да се дефинираат рекурзивно на следниот начин. Дрво е конечно множество со еден или повеќе елементи наречени јазли што ги задоволува следниве правила:

1. Постои еден јазел наречен корен на дрвото;
2. Останатите јазли (без коренот) се групирани во $k \geq 0$ дисјунктни множества T_1, T_2, \dots, T_k , од кои секое е дрво. Овие дрва се нарекуваат поддрва на дрвото T .

Постојат и други начини на дефинирање на дрва кои обично се базираат на теоријата на вгнездени множества или графови.



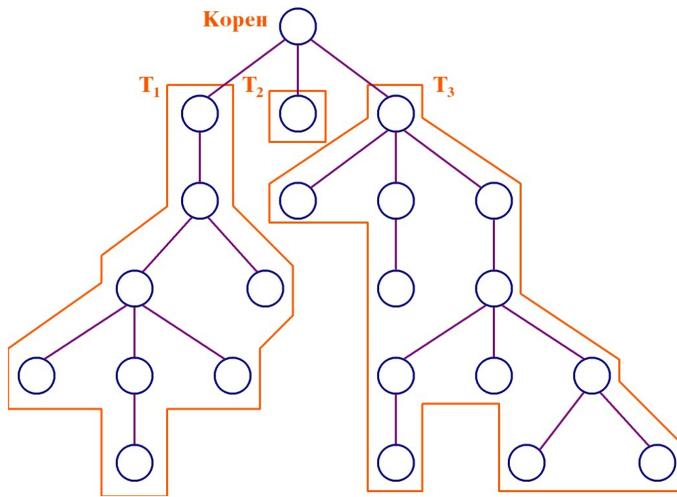
Слика 6-1: Три еквивалентни претстави на едно дрво

На слика 6-1 се прикажани три начини на претставување на едно дрво.

Јазелот **A** е **корен** (анг. root) на дрвото. Јазелот **B** е корен на поддрвото **Г, Ѓ, Г, Е**. Трите скици претставуваат исто дрво и се добиени со едноставни ротации. Во пракса најчесто се користи третиот дијаграм, каде што дрвото се претставува на тој начин што коренот е горе, а листовите се долу.

Од дефиницијата за дрво следува дека секој **внатрешен јазел** во дрвото е корен на некое поддрво (види слика 6-2). Бројот на поддрва на еден јазел се

нарекува *степен на јазелот*. Кога овој број е 0, јазелот се нарекува *краен* (терминален) јазел или *лист*. Сите јазли (освен коренот) имаат свој родител.



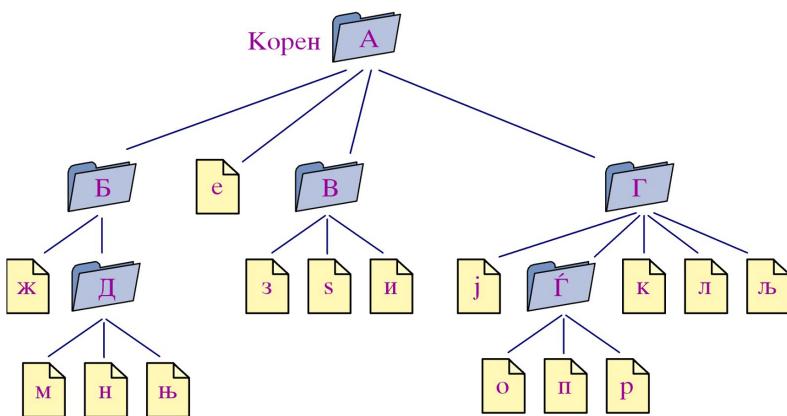
Слика 6-2: Корен на дрво со поддрва

Јазел што има најмалку едно дете се нарекува *нетерминален* или *внатрешен јазел*. Нивото на секој јазел е бројот на поддрва преку кои се доаѓа до него, односно бројот на јазли низ кои треба да се помине за од коренот да се стигне до јазелот, имајќи предвид дека нивото на коренот е 0. Стандардната терминологија за јазлите во дрвата е земена од фамилијарните дрва. Така, секој корен е татко (или родител). Јазлите непосредно под таткото се негови деца (понекогаш се користат и термините синови или ќерки), а меѓу себе се браќа (сестри). Во литературата се користат и термините наследници и следбеници за да го претстават односот на родителството.

На слика 6-3 е прикажано дрво, во кое за поедноставно објаснување јазлите се означени со букви од кириличната азбука. Според сликата јазелот означен со **Г** е родител на јазлите **Ѓ**, **ј**, **к**, **л**, **љ**. Јазелот означен со **Ѓ** е родител на јазлите **о**, **п**, **р**. Јазлите **Д** и **ж** се деца на јазелот **Б**. Сите јазли означени со буквите меѓу **е** и **р** се листови на дрвото.

Множество (обично подредено) на различни (дисјунктни) дрва се нарекува *шума*. Ако од едно дрво го избришеме коренот, се добива шума. Ако пак во една шума додадеме само еден јазел и го поврземе со корените на дрвата, од шумата добиваме едно дрво.

Ако n_1, n_2, \dots, n_k е низа на јазли во дрво така што n_i е родител на n_{i+1} , $1 \leq i < k$, тогаш низата се нарекува *патека* од јазелот n_1 до n_k . За слика 6-3, патеката од јазелот **А** до **м** гласи: **А**, **Б**, **Д**, **м**. *Должина на патека* претставува број на врски меѓу два јазли, односно таа е за еден помала од бројот на јазли во патеката. Должина на патека од еден јазел до самиот себе изнесува 0. Ако постои патека



Слика 6-3: Пример на дрво

од јазелот означен со **A** во некое дрво до јазелот означен со **B**, тогаш велиме дека **A** е *предок* на **B**, а **B** е *наследник* на **A**.

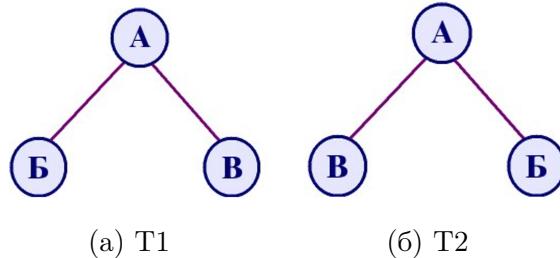
За слика 6-3, наследници на јазелот **B** се јазлите **ж**, **Д**, **м**, **н**, **њ**. Секој јазел е предок и наследник на самиот себе. *Вистински наследник* (предок) на даден јазел **A** е јазел кој е различен од јазелот **A**. *Предци* на еден јазел се сите јазли долж патеката од коренот до самиот јазел. За дрвото на слика 6-3, предци на јазелот **м** се јазлите **A**, **Б**, **Д**. Јазел без вистински наследник се нарекува лист (терминален јазел). За дрвото на слика 6-3 терминалните јазли се означени со малите букви од кириличната азбука.

Поддрво на даден јазел во дрво е јазелот дете со сите негови наследници. Бројот на поддрвата на еден јазел се нарекува *степен на јазелот*. За дрвото на слика 6-3, јазелот **A** има степен 4, јазелот **Г** степен 5 итн. *Степен на дрвото* е максималниот степен на јазлите во дрвото. За дрвото на слика 6-3, степенот на дрвото изнесува 5. *Висина на јазел* во дрво е долнината на најдлгата патека од јазелот до неговите листови. *Висина на дрвото* е висина на коренот на дрвото. *Длабочина на јазел* е долнината на единствената патека од коренот до јазелот. *Длабочината на дрвото* е дефинирана со максималната длабочина на било кој јазел од дрвото.

Кога редоследот на поддрвата во едно дрво е важен, дрвото се нарекува *подредено дрво*. На примерот од слика 6-4, постојат две различни подредени дрва **T1** и **T2**. Притоа важи: ако **T1** и **T2** се подредени дрва тогаш, $T1 \neq T2$, инаку $T1 = T2$.

6.1.3 Репрезентација на дрвата

Најприродно претставување на дрво е едноставно родителот да ги содржи покажувачите кон своите деца. Ваквото „идеално“ претставување содржи проблеми



Слика 6-4: Примери на подредени дрва

бидејќи еден јазел може да има произволен број деца. Така, бројот на покажувачи во јазлите би требало да биде променлив. Ваквото усложнување на потребната структура значително би го искомплицирало имплементирањето на алгоритмите за обработка на дрва. Алтернативно решение би било да се предвиди некој максимален број на деца и секој јазел да има простор за толков број на покажувачи. Но, ова решение повторно има ограничувања, прво во бројот на можни деца, а второ кога некој јазел ќе има малку деца, ќе се заземе непотребен мемориски простор за остатокот од предвидените деца. Поради ваквите предизвици кај дрвата со произволен степен, тие се непрактични за имплементација.

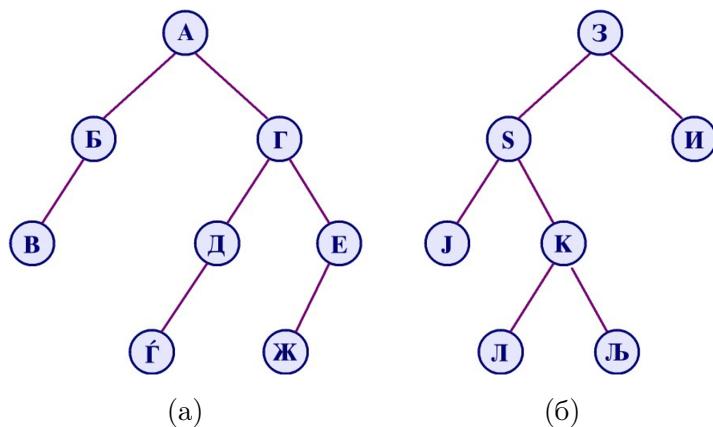
Наместо тоа, обично дрвата се препрезентираат со структура каде што бројот на покажувачи е најмногу два. Ова претставување на дрвата се нарекува претставување со помош на бинарни дрва. Бинарното дрво е погодно за најголемиот дел од алгоритмите кои работат со хиерархии на податоци. Единствен недостаток е значителниот број на празни покажувачи (со нулева вредност) [1].

6.1.4 Бинарни дрва

Бинарните дрва се дрва кај кои секој јазел може да има најмногу две поддрва, при што ако постои само едно од нив ние разликуваме дали е левото или десното. Бинарно дрво е дрво чиј степен изнесува два, односно сите негови јазли може да немаат наследници, да имаат еден или најмногу два наследници. Кај бинарните дрва разликуваме лево и десно поддрво. Поформална дефиниција на бинарното дрва е следната: Бинарно дрво е конечно множество на јазли кое е или празно или се состои од корен и две дисјунктни поддрва наречени лево и десно поддрво.

На слика 6-5 се прикажани две бинарни дрва. Дрвото на левата страна (слика 6-5а) репрезентира нецелосно бинарно дрво. Целосно бинарно дрво е дрво чии нетерминални јазли секогаш имаат два потомци. На примерот на слика 6-5б, јазлите **Z**, **S** и **K** имаат по два потомци. Од друга страна, дрвото на слика 6-5а е нецелосно бидејќи јазлите **B**, **D** и **E** немаат по два потомци.

Во општ случај, бинарните дрва не се поткласа на класата дрва туку нов



Слика 6-5: Нецелосно и целосно бинарно дрво

поим иако двете класи се со многу сличности. Во суштина двете клучни работи што ги прават бинарните дрва посебна класа се, правењето разлика помеѓу лево и десно поддрво и можноста бинарното дрво да биде празно. Важноста на бинарните дрва не потекнува само од нивната практична применливост туку и од фактот што било кое произволно дрво на едноставен начин може да се претстави (трансформира) во бинарно дрво [1].

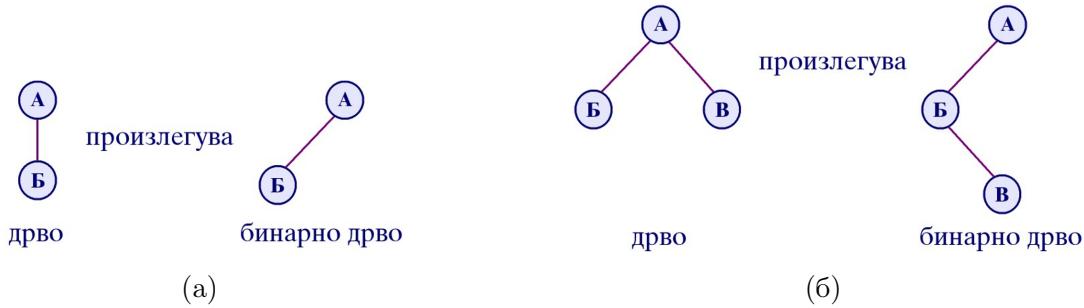
6.1.5 Трансформација на шума од произволни дрва во бинарно дрво

Откако поимите дрво и бинарно дрво се дефинирани, може да се опише и алгоритамот за претставување на произволно дрво (односно шума од дрва) со помош на бинарно дрво. Произволно дрво може да се измине така што се изминува коренот, па сите негови деца, при што таа постапка рекурзивно се повторува. Поточно, првото дете на секој јазел станува прво лево дете, додека секое следно дете станува десна гранка на претходното дете. Оваа трансформација важи за произволни дрва. На слика 6-6 е претставена примената на таа трансформација за две едноставни дрва.

Конкретно, на слика 6-ба е претставен тривијален случај на обично дрво кое има само една гранка. Според дефиницијата изнесена претходно, кај бинарното се разликуваат левата и десната гранка (која во случајов не постои). Според алгоритамот од претходниот пасус, првото дете на **A** станува лева гранка на бинарното дрво.

Продолжуваме кај малку посложениот пример од слика 6-6б. Јазелот **A** е корен и на бинарното дрво. Овде првото дете на јазелот **A** кај обичното дрво е јазелот **B**. Според алгоритамот, **B** станува лева гранка во бинарното дрво. Потоа,

второто дете на јазелот **A** во обичното дрво е **B**. Бидејќи **B** има друг јазел на исто ниво пред него (јазелот **B**), тогаш во трансформираното бинарно дрво, **B** ќе стане десно дете на **B**.

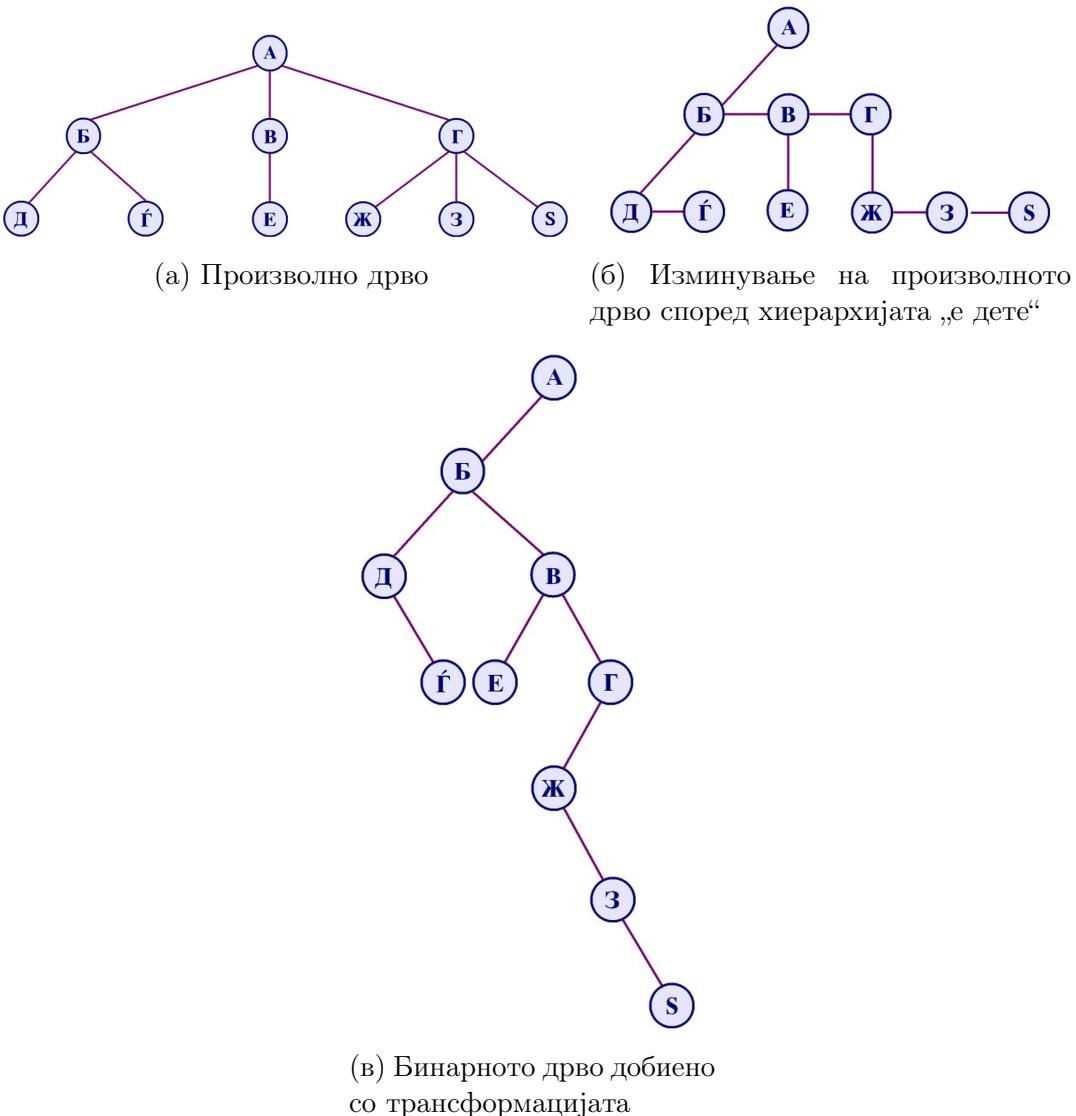


Слика 6-6: Репрезентација на едноставни дрва со бинарно дрво

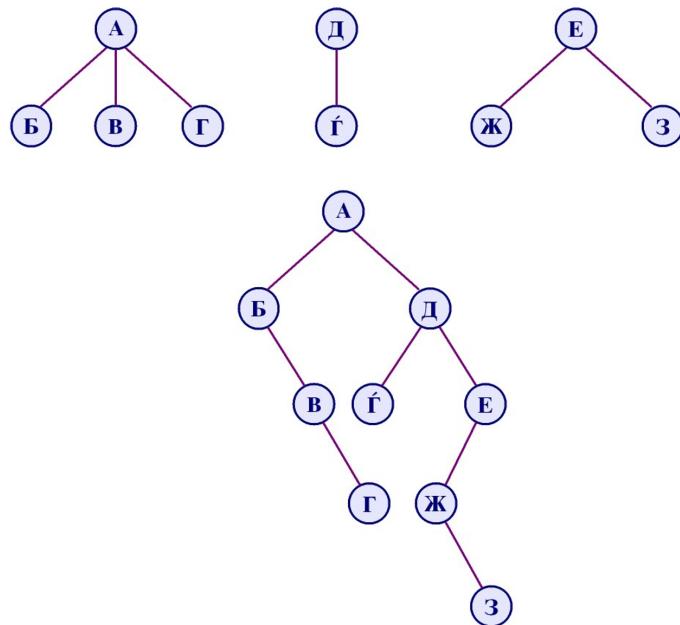
Претходно илустрираната трансформација на произволно дрво во бинарно дрво може да се опише поформално на следниот начин: Нека T_1, \dots, T_n е шума на дрва, тогаш бинарното дрво кое се добива со трансформацијата на оваа шума може да се означи со $B(T_1, \dots, T_n)$ и за неа важи:

- $B(T_1, \dots, T_n)$ е празно ако $n = 0$;
- $B(T_1, \dots, T_n)$ има корен еднаков со коренот на T_1 ; има лево поддрво $B(T_1^1, T_1^2, \dots, T_1^m)$ каде $T_1^1, T_1^2, \dots, T_1^m$ се поддрва на коренот (T_1); има десно поддрво $B(T_2, \dots, T_n)$.

Да разгледаме и уште посложен пример даден на слика 6-7а. Таквото изменување е прикажано на сликата 6-7б. Конкретно, првото дете на **A** е **B** и станува лева гранка на **A** во бинарното дрво. Потоа, второто дете на **A** е **B** и станува десна гранка на претходниот јазел на исто ниво (**B**) во бинарното дрво. Слично, и **C** станува десна гранка на претходниот јазел на исто ниво (**B**). Со тоа завршува трансформацијата на јазлите од второто ниво. Јазелот **D** е прво дете на **B** и поради тоа станува негова лева гранка во бинарното дрво. **E** е второ дете, и соодветно станува десна гранка на првото дете (**D**). Следно, **F** е прво дете на **B** и заради тоа е негова лева гранка после трансформацијата. **G** е второ дете на **F**, и соодветно станува негово лево дете во бинарното дрво. **H** е трето дете на **F** и станува десна гранка на претходното дете на **G** - јазелот **J**. Конечно, јазелот **I** не е прво дете, соодветно станува десна гранка на претходното дете на **G** - јазелот **H**. Со тоа завршува трансформацијата на ова произволно дрво. Ако сликата 6-7б се заврти за 45 степени се добива сликата 6-7в со која е представено бинарното дрво кое соодветствува на дрвото од слика 6-7а.



Слика 6-7: Процес на трансформација на произволно дрво во бинарно дрво



Слика 6-8: Репрезентација на шума од дрва со бинарно дрво

Репрезентацијата на шума од дрва со помош на бинарно дрво е тривијална ако се претпостави дека корените на сите дрва од шумата се браќа. Тогаш согласно претходно кажаното за даден пример се добива резултатот од слика 6-8

6.1.6 Претставување на бинарните дрва со Java код

Како и кај другите структури, постојат повеќе пристапи за претставување на дрво со код. Најчестиот пристап е да се чуваат јазли на произволни неповрзани мемориски локации и да се поврзат со покажувачи од секој јазел до неговите деца.

Класата `TreeNode`

Прво, ни треба класа за претставување на јазлите. Оваа класа ќе содржи податоци што ги претставуваат објектите што се чуваат (на пример броеви, стрингови, посложени објекти како вработени во база на вработени, итн.). Со цел да овозможиме употреба на оваа класа без разлика на податочниот тип на објектите што ќе ги чуваме, се користат генерици, слично како и кај претходните структури [2]. Исто така, се проширува соодветниот `Comparable` интерфејс со цел да можеме понатаму да ги споредуваме и подредуваме јазлите. Дополнително ни се потребни и референци до секое од двете деца на јазелот. Еве како изгледа тоа:

```

1
2 public class TreeNode<T extends Comparable<T>> {
3     public T data;
4     public TreeNode<T> left;
5     public TreeNode<T> right;
6
7     public TreeNode(T data) {
8         this.data = data;
9         left = null;
10        right = null;
11    }
12
13    // other methods will be shown later
14 }
```

Понекогаш може да се чува референца и до родителот на јазелот. Ова поеноставува некои операции, но некои други операции се усложнуваат со него, па затоа ние нема да го вклучиме.

Класата `BinaryTree`

Исто така, ни треба класа која ќе го опишува целото бинарно дрво заедно со сите јазли кои ги содржи. Ќе ја наречеме `BinaryTree` бидејќи се однесува на бинарни дрва. Во неа е потребно само едно поле - покажувач кон коренот на дрвото. Сите

останати јазли ќе ги пристапуваме тргнувајќи од него. Понатаму ќе дефинираме други дополнителни методи за оваа класа.

```

1 public class BinaryTree<T extends Comparable<T>> {
2     private TreeNode<T> root;
3
4     public BinaryTree() {
5         root = null;
6     }
7     // create a root element with the provided data
8     public void makeRoot(T data) {
9         root = new TreeNode<T>(data);
10    }
11    // other methods will be shown later
12 }
```

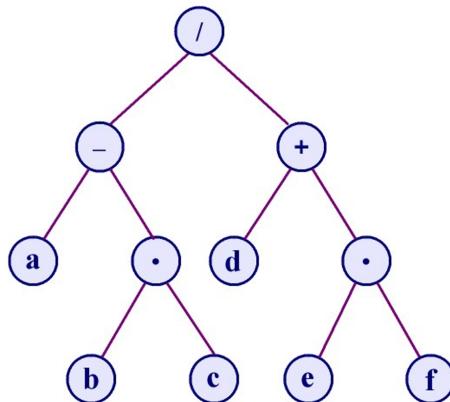
6.1.7 Изминување на бинарните дрва

Постојат три основни начини на кои може да се изминат сите елементи на едно бинарно дрво: приордер (анг. preorder), инордер (анг. inorder) и постордер (анг. postorder) изминување. Ако T е празно дрво, тогаш празна листа се добива при приордер, инордер и постордер изминување на дрвото. Ако T содржи еден јазел, тогаш тој јазел претставува приордер, инордер, и постордер изминување на дрвото T . Нека ја имаме структурата на дрвото чиј корен е јазелот K и ги содржи подстеблата T_1 и T_2 . Тогаш:

- Приордер изминување на јазлите на T е низа која ја сочинува информацијата од коренот K на кој му следи низата од информациите од јазлите на поддрвото T_1 изминато во приордер, по кое следи низата од информациите од јазлите од поддрвото T_2 изминато во приордер.
- Инордер изминување на јазлите на T е низата од информациите од јазлите на поддрвото T_1 изминато во инордер, потоа информацијата од коренот K , на кои им следи низата од информациите од јазлите од T_2 изминато во инордер.
- Постордер изминување на јазлите на T е низа која ја сочинува низата од информациите од јазлите на поддрвото T_1 изминато во постордер, потоа низата од информациите од јазлите од поддрвото T_2 изминато во постордер и на крајот информацијата од коренот.

Една од најчестите примени на овие изминувања е во пресметувањето на аритметичките изрази со помош на компјутер (или дигитрон). Аритметичките

изрази обично имаат еден или два операнди, со што претставуваат добар пример за можност на користење на бинарно дрво за нивна репрезентација. Во компјутерската анализа на аритметичките (и логичките) изрази важна улога играат начините на изминувањето на дрвото со кои се претставени тие изрази [1]. На слика 6-9 е дадено бинарно дрво за еден аритметички израз и соодветните изминувања на дрвото. Дрвото го претставува изразот $(a - b \times c) / (d + e \times f)$.



Слика 6-9: Репрезентација на шума од дрва со бинарно дрво

Изминувањето на дрвото ги дава следните низи:

- приордер: $/ - a \times b c + d \times e f$
- инордер: $a - b \times c / d + e \times f$
- постордер: $a b c \times - d e f \times + /$

Во литературата овие изминувања на дрвото претставени во соодветните низи на информации од јазлите во англиската терминологија се нарекуваат: prefix, infix и postfix нотација, коишто се користат во реализацијата на преведувачите на програмските јазици. Имплементациите на овие алгоритми за основните начини на изминување на дрвото во својата рекурзивна верзија се речиси тривијални [3]. Во прилог се кодовите за соодветните изминувања.

Во приордер варијантата, прво се печати информацијата во тековниот јазел, па потоа се печати левото поддрво со рекурзивен повик на истата функција, па на крај се печати десното поддрво, повторно со рекурзивен повик.

```

1 // Preorder traversal
2 private void preorder(TreeNode<T> node) {
3     if (node != null) {
4         System.out.print(node.data.toString() + " ");
5         preorder(node.left);
6         preorder(node.right);
    }
}
```

```

7         }
8     }
9     public void preorder() {
10        System.out.println("Preorder traversal:");
11        preorder(root);
12        System.out.println();
13    }

```

Притоа, првата функција е приватна и има аргумент кој кажува кој е тековниот јазел што се изминува. Како што се оди на следно ниво на рекурзија, се менува тековниот јазел што се изминува. Од друга страна, втората функција е јавна и може да се повика без аргументи, така што ќе тргне од коренот и ќе го изминува дрвото соодветно.

Во варијантата на инордер изминување, единствената разлика е тоа што прво се изминува рекурзивно левото поддрво, па се печати јазолот, па потоа се изминува десното поддрво.

```

1  private void inorder(TreeNode<T> node) {
2      if (node != null) {
3          inorder(node.left);
4          System.out.print(node.data + " ");
5          inorder(node.right);
6      }
7  }
8  public void inorder() {
9      System.out.println("Inorder traversal:");
10     inorder(root);
11     System.out.println();
12 }

```

Слично, во варијантата на постордер изминување, единствената разлика е тоа што печатењето на јазолот е по рекурзивното изминување на двете подрва.

```

1  private void postorder(TreeNode<T> node) {
2      if (node != null) {
3          inorder(node.left);
4          inorder(node.right);
5          System.out.print(node.data + " ");
6      }
7  }
8
9  public void postorder() {
10     System.out.println("Postorder traversal:");

```

```

11     postorder(root);
12     System.out.println();
13 }
14 }
```

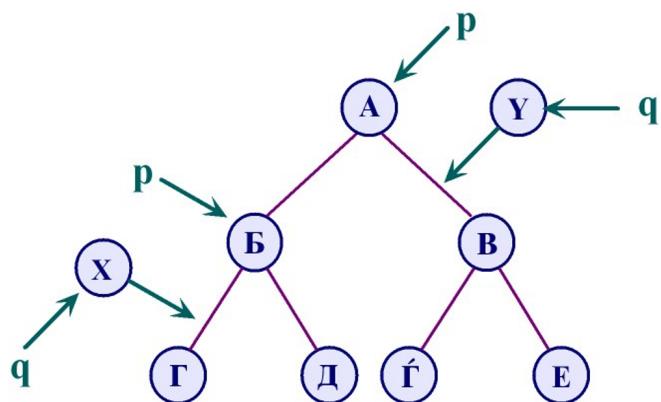
Пред да покажеме како изгледа извршувањето на овие функции за изминување, прво ќе покажеме основни манипулации кај бинарните дрва и ќе креираме пример дрво.

6.1.8 Основни манипулации со јазлите во бинарните дрва

Вистинската вредност на динамичките структури е во можноста за едноставно менување на структурата во смисла на вметнување нови елементи или бришење на постоечки. За бинарните дрва динамичноста е особено важна бидејќи нивното користење во практичните алгоритми најчесто е базирано на нивно постојано менување. Понатаму ќе ги разгледаме двете основни операции на работа со бинарно дрво, вметнување и бришење јазел [1].

Вметнување на нови елементи во дрвото

На слика 6-9 е дадена илустрација на вметнување јазел во бинарно дрво.



Слика 6-10: Вметнување јазел со покажувач q под јазел со покажувач p во лево и десно поддрво од едно бинарно дрво

Соодветниот метод `addLeftChild(T x, TreeNode<T> parent)` за додавање на нов јазел со вредност x како лево дете на родителот $parent$ во бинарно дрво е даден подолу. Заради практични причини, методот го враќа новиот јазел од тип `TreeNode<T>`. Тоа понатаму ќе го искористиме за креирање на дрво со потребна структура. Прво се иницијализира новиот јазел `newNode` со податокот x . Понатака се проверува дали родителот $parent$ е празен и во тој случај новиот

јазел се поставува како корен. Во спротивно, методот го поставува левото дете на `parent` да биде `newNode` и потоа го враќа новиот јазел.

```
1 // Function to add a new node with data field x to the left of the given
2     parent node p
3 public TreeNode<T> addLeftChild(T x, TreeNode<T> parent) {
4     TreeNode<T> newNode = new TreeNode<>(x);
5     if (parent == null) {
6         this.root = newNode;
7         System.out.println("Added " + x + " as the root.");
8     } else {
9         newNode.left = parent.left;
10        parent.left = newNode;
11        System.out.println("Added " + x + " as the left child of " +
12            parent.data);
13    }
14    return newNode;
15 }
```

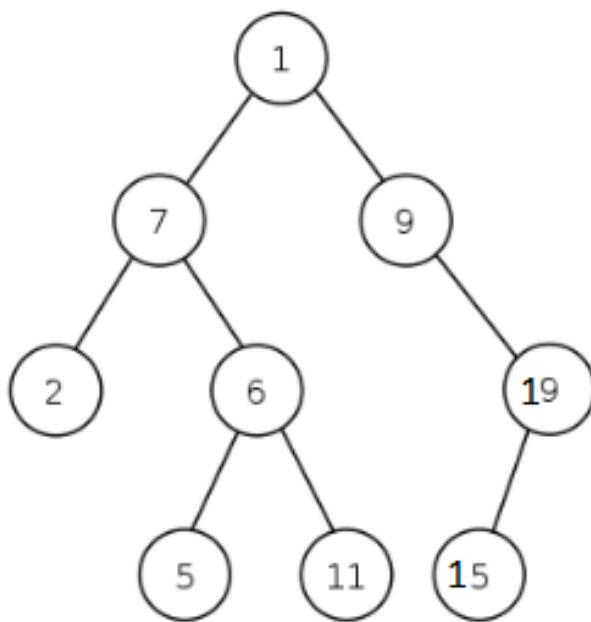
Методот за додавање на десно дете во бинарно дрво е многу сличен и даден во продолжение.

```
1 // Function to add a new node with data field x to the right of the given
2     parent node p
3 public TreeNode<T> addRightChild(T x, TreeNode<T> parent) {
4     TreeNode<T> newNode = new TreeNode<>(x);
5     if (parent == null) {
6         this.root = newNode;
7         System.out.println("Added " + x + " as the root.");
8     } else {
9         newNode.right = parent.right;
10        parent.right = newNode;
11        System.out.println("Added " + x + " as the right child of " +
12            parent.data);
13    }
14    return newNode;
15 }
```

Бришење на јазли од дрво

Бришење јазел во дрвото е посложена операција, особено во случаите кога тој има две деца. Проблемот е во тоа што со еден влезен покажувач во јазелот не можеме да ги покриеме двата излезни покажувачи кон неговите деца. Во таквите случаи,

наместо вистинско бришење, ние правиме замена на јазелот што треба да биде избришан со друг јазел (најчесто некој од неговото поддрво). Оваа операција кај бинарните дрва за пребарување, со кои ќе се запознаеме понатаму во оваа глава, е попредвидлива и добро дефинирана. Така, кај нив, при бришењето избришаниот јазел се заменува со најдесниот јазел од левото поддрво или најлевиот јазел од десното поддрво. Кога јазелот што се брише нема деца (е лист) или пак има само едно дете, алгоритамот е тривијален. На оваа операција заедно со сите нејзини потслучай ќе се навратиме кога ќе зборуваме за бинарни пребарувачки дрва понатаму во ова поглавје.



Слика 6-11: Пример на бинарно дрво

Полнење на бинарното дрво

Нека е дадено дрво како на сликата 6-11. За да го представиме ова дрво со помош на структурата `BinaryTree`, треба да се користат операциите `addLeftChild` и `addRightChild` соодветно. Притоа важно е да се напомене дека за да почне да се креира дрвото потребно е првиот јазел **1** да го дефинираме како корен на дрвото со помош на функцијата `makeRoot`. Откако коренот е додаден, соодветно онака како што е нацртано дрвото ги даваме последователно јазлите. На пример, за да се внесе јазелот **7**, тој треба да се додаде како лево дете на коренот. Соодветно, за да се внесе јазелот **2** тој треба да се додаде како лево дете на веќе додадениот

јазел 7, а јазелот **6** како десно дете на јазелот 7. За таа цел пред да го додадеме јазелот **7** креираме помошен објект `TreeNode` кој го иницијализираме со вредноста што ја враќа функцијата `addLeftChild` за додавање на јазелот **7** итн. Комплетниот код за внесување на дрвото од слика е даден подолу.

```

1 //main function for creating a binary tree
2 public static void main(String[] args) {
3     TreeNode<Integer> tmp1, tmp2, tmp3;
4     BinaryTree<Integer> tree = new BinaryTree<>();
5     tree.makeRoot(1);
6     tmp1 = tree.addLeftChild(7, tree.root);
7     tmp2 = tree.addLeftChild(2, tmp1);
8     tmp2 = tree.addRightChild(6, tmp1);
9     tmp3 = tree.addLeftChild(5, tmp2);
10    tmp3 = tree.addRightChild(11, tmp2);
11
12    tmp1 = tree.addRightChild(9, tree.root);
13    tmp2 = tree.addRightChild(19, tmp1);
14    tmp3 = tree.addLeftChild(15, tmp2);
15
16    tree.inorder();
17    tree.preorder();
18    tree.postorder();
19 }
```

Последните три линии прават изминување на дрвото во инордер, приордер и постордер, соодветно, со што се добива следниот излез:

```

1 Inorder traversal:
2 2 7 5 6 11 1 9 15 19
3 Preorder traversal:
4 1 7 2 6 5 11 9 19 15
5 Postorder traversal:
6 7 2 6 5 11 9 19 15 1
```

Печатење на дрвото

Иако ова изминување што го покажавме дава претстава како би било изминувањето на одредено дрво со одреден алгоритам, сепак тоа не ја прикажува соодветно структурата на дрвото. Со други зборови, повеќе различни дрва може да дадат исто изминување во било кој од начините (инордер, приордер, постордер). За да дадеме еднозначна претстава на дрвото, ќе направиме дополнителна

функција во класата `TreeNode` која дрвото ќе го печати во приордер начин каде појасно ќе бидат испечатени јазлите. Печатењето на дрвото може да се направи со следниот код:

```
1  @Override
2      public String toString() {
3          // Use a StringBuilder to build the output string.
4          StringBuilder sb = new StringBuilder();
5
6          // Call the helper function to build the output string.
7          toStringHelper(sb, this.root, 0, 7);
8
9          // Convert the StringBuilder to String and return it.
10         return sb.toString();
11     }
12
13     private void toStringHelper(StringBuilder sb, TreeNode<T> node, int
14         space, int count) {
15
16         if (node == null)
17             return;
18
19         // to increase the distance between levels
20         space += count;
21
22         // print the right child first
23         toStringHelper(sb, node.right, space, count);
24
25         // print the current node after adding the spaces
26         sb.append("\n");
27         for (int i = count; i < space; i++)
28             sb.append(" ");
29         sb.append(node.data + "\n");
30
31         //print the left child
32         toStringHelper(sb, node.left, space, count);
33     }
```

Излезот од извршувањето на овој код би бил следниот што одговара на дрвото прикажано на слика 6-11:

15

9

1

11

6

5

7

2

6.1.9 Задачи

Сите задачи во оваа глава ќе го користат истиот код за класите `TreeNode` и `BinaryTree`. Се очекува дека со решавањето на овие задачи ќе се додаде некој нов метод во една од овие класи и соодветно ќе се искористи `main` функција која го повикува новиот метод, како и која прави вчитување и печатење на потребните работи. За да не се повторува целокупниот код во секоја функција посебно, во продолжение е кодот што го пишувавме до сега за основните класи, а потоа само ќе го дополнуваме со нови методи.

```
1 // Define a generic class called TreeNode. It holds data of a type that
  implements Comparable.
2 public class TreeNode<T extends Comparable<T>> {
3
4     // Declare a public variable 'data' to hold the node's data.
5     public T data;
6
7     // Declare pointers for the left and right children of the TreeNode.
8     public TreeNode<T> left;
9     public TreeNode<T> right;
10
11    // Constructor for initializing the TreeNode with data.
12    public TreeNode(T data) {
13        // Store the given data in this node.
```

```
14     this.data = data;
15
16     // Initialize the left and right children to null.
17     left = null;
18     right = null;
19 }
20
21 // Override the toString method to convert the TreeNode into a String
22 // representation.
22 @Override
23 public String toString() {
24     // Use a StringBuilder to build the output string.
25     StringBuilder sb = new StringBuilder();
26
27     // Call the helper function to build the output string.
28     toStringHelper_sb, this, 0, 7);
29
30     // Convert the StringBuilder to String and return it.
31     return sb.toString();
32 }
33
34 // Helper function that builds the output recursively
35 private void toStringHelper(StringBuilder sb, TreeNode<T> node, int
36     space, int count) {
37
38     if (node == null)
39         return;
40
41     // to increase the distance between levels
42     space += count;
43
44     // print the right child first
45     toStringHelper(sb, node.right, space, count);
46
47     // print the current node after adding the spaces
48     sb.append("\n");
49     for (int i = count; i < space; i++)
50         sb.append(" ");
51     sb.append(node.data + "\n");
52
53     //print the left child
```

```

53         toStringHelper(sb, node.left, space, count);
54     }
55 }



---


1 // Define a generic class BinaryTree with elements of type T, where T
2 // extends Comparable.
3
4 public class BinaryTree<T extends Comparable<T>> {
5
6     // Declare the root node of the binary tree.
7     public TreeNode<T> root;
8
9
10    // Default constructor to initialize the binary tree with a null root.
11    public BinaryTree() {
12        root = null;
13    }
14
15    // Function to add a left child node with the data 'x' to a given parent
16    // node 'parent'.
17    public TreeNode<T> addLeftChild(T x, TreeNode<T> parent) {
18        // Create a new node with the data 'x'.
19        TreeNode<T> newNode = new TreeNode<>(x);
20
21        // If the parent node is null, the tree is empty. Make the new node
22        // the root.
23        if (parent == null) {
24            this.root = newNode;
25            System.out.println("Added " + x + " as the root.");
26        } else {
27            // Otherwise, add the new node as the left child of the parent.
28            // First, link any existing left child of the parent to the new
29            // node.
30            newNode.left = parent.left;
31
32            // Now, link the new node to the parent as its left child.
33            parent.left = newNode;
34            System.out.println("Added " + x + " as the left child of " +
35                parent.data);
36        }
37        return newNode;
38    }
39
40    // Function to add a right child node with the data 'x' to a given

```

```
    parent node 'parent'.
34  public TreeNode<T> addRightChild(T x, TreeNode<T> parent) {
35      // Create a new node with the data 'x'.
36      TreeNode<T> newNode = new TreeNode<>(x);
37
38      // If the parent node is null, the tree is empty. Make the new node
39      // the root.
40      if (parent == null) {
41          this.root = newNode;
42          System.out.println("Added " + x + " as the root.");
43      } else {
44          // Otherwise, add the new node as the right child of the parent.
45          // First, link any existing right child of the parent to the new
46          // node.
47          newNode.right = parent.right;
48
49          // Now, link the new node to the parent as its right child.
50          parent.right = newNode;
51          System.out.println("Added " + x + " as the right child of " +
52                           parent.data);
53      }
54      return newNode;
55  }
56
57  // Function to make a node with the data 'data' the root of the tree.
58  public void makeRoot(T data) {
59      root = new TreeNode<T>(data);
60  }
61
62  // Public function to perform inorder traversal and print the tree.
63  public void inorder() {
64      System.out.println("Inorder traversal:");
65      inorder(root);
66      System.out.println();
67  }
68
69  // Public function to perform preorder traversal and print the tree.
70  public void preorder() {
71      System.out.println("Preorder traversal:");
72      preorder(root);
73      System.out.println();
```

```
71     }
72
73     // Public function to perform postorder traversal and print the tree.
74     public void postorder() {
75         System.out.println("Postorder traversal:");
76         postorder(root);
77         System.out.println();
78     }
79
80     // Private function for inorder traversal starting at node 'node'.
81     private void inorder(TreeNode<T> node) {
82         if (node != null) {
83             inorder(node.left);
84             System.out.print(node.data + " ");
85             inorder(node.right);
86         }
87     }
88
89     // Private function for preorder traversal starting at node 'node'.
90     private void preorder(TreeNode<T> node) {
91         if (node != null) {
92             System.out.print(node.data + " ");
93             preorder(node.left);
94             preorder(node.right);
95         }
96     }
97
98     // Private function for postorder traversal starting at node 'node'.
99     private void postorder(TreeNode<T> node) {
100        if (node != null) {
101            postorder(node.left);
102            postorder(node.right);
103            System.out.print(node.data + " ");
104        }
105    }
106
107    // Override toString method to convert the BinaryTree to a String
108    // representation.
109    public String toString() {
110        // If the root is not null, return its String representation.
111        // Otherwise, say the tree is empty.
```

```

110         return (root != null) ? root.toString() : "The tree is empty.";
111     }
112 }
```

Во следниот код, кој е дел од новата класа `BinaryTreeTest` ќе видиме како ќе се тестираат функциите. Воедно дадена е функцијата за полнење на бинарно дрво со целоброжни вредности.

```

1 public class BinaryTreeTest {
2
3     // Method to create an example integer-based binary tree.
4     public static BinaryTree<Integer> GetExampleIntTree() {
5         // Declare temporary TreeNode variables.
6         TreeNode<Integer> tmp1, tmp2, tmp3;
7
8         // Create a new BinaryTree of Integer type.
9         BinaryTree<Integer> intTree = new BinaryTree<>();
10
11        // Make the root of the tree with the value 1.
12        intTree.makeRoot(1);
13
14        // Add 7 as the left child of the root and keep a reference to this
15        // new node in tmp1.
16        tmp1 = intTree.addLeftChild(7, intTree.root);
17
18        // Add 2 as the left child of node 7 (tmp1) and keep a reference in
19        // tmp2.
20        tmp2 = intTree.addLeftChild(2, tmp1);
21
22        // Add 6 as the right child of node 7 (tmp1) and update the
23        // reference in tmp2.
24        tmp2 = intTree.addRightChild(6, tmp1);
25
26        // Add 5 as the left child of node 6 (tmp2) and keep a reference in
27        // tmp3.
28        tmp3 = intTree.addLeftChild(5, tmp2);
29
30        // Add 11 as the right child of node 6 (tmp2) and update the
31        // reference in tmp3.
32        tmp3 = intTree.addRightChild(11, tmp2);
33
34        // Add 9 as the right child of the root and keep a reference in tmp1.
35 }
```

```

30     tmp1 = intTree.addRightChild(9, intTree.root);
31
32     // Add 19 as the right child of node 9 (tmp1) and keep a reference
33     // in tmp2.
34     tmp2 = intTree.addRightChild(19, tmp1);
35
36     // Add 15 as the left child of node 19 (tmp2).
37     tmp3 = intTree.addLeftChild(15, tmp2);
38
39     // Return the example integer tree.
40     return intTree;
41
42
43     // Main method to test the BinaryTree class.
44     public static void main(String[] args) {
45         // Create an example integer tree and keep a reference in intTree.
46         BinaryTree<Integer> intTree = GetExampleIntTree();
47
48         // Perform and print inorder, preorder, and postorder traversals of
49         // intTree.
50         intTree.inorder();
51         intTree.preorder();
52         intTree.postorder();
53
54         // Print the string representation of intTree.
55         System.out.print(intTree);
56     }

```

Задача 1. Креирање бинарно дрво со текстуални јазли

За дадено бинарно дрво, да се напише функција која ќе го креира, така што елементи на неговите јазли ќе бидат текстуални низи. Потоа да се прикаже неговото изминување во инордер и истото да се испечати.

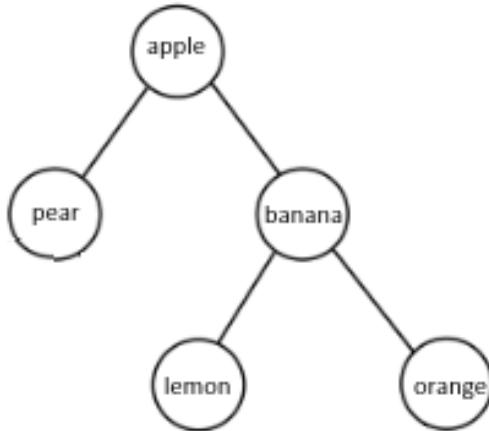
Влез:

Бинарно дрво од слика 6-12.

Излез:

Inorder traversal:

peach pear apple lemon banana orange



Слика 6-12: Пример на бинарно дрво

orange

banana

lemon

apple

pear

```

1 public class BinaryTreeTest {
2     // Method to create an example string-based binary tree.
3     public static BinaryTree<String> GetExampleStringTree() {
4         // Create a new BinaryTree of String type.
5         BinaryTree<String> stringTree = new BinaryTree<>();
6
7         // Set "apple" as the root of the tree.
8         stringTree.makeRoot("apple");
9
10        // Add "pear" as the left child of the root node.
11        TreeNode<String> node1 = stringTree.addLeftChild("pear",
12                                stringTree.root); // root
13
14        // Add "peach" as the left child of the "pear" node.
15        TreeNode<String> node2 = stringTree.addLeftChild("peach", node1);
  
```

```

15
16     // Add "banana" as the right child of the root node.
17     TreeNode<String> node3 = stringTree.addRightChild("banana",
18         stringTree.root);
19
20     // Add "orange" as the right child of the "banana" node.
21     node2 = stringTree.addRightChild("orange", node3);
22
23     // Add "lemon" as the left child of the "banana" node.
24     node2 = stringTree.addLeftChild("lemon", node3);
25
26     // Return the example string tree.
27     return stringTree;
28 }
29 // Main method to test the BinaryTree class.
30 public static void main(String[] args) {
31     // Create an example of string tree and keep a reference in strTree.
32     BinaryTree<String> strTree = GetExampleStringTree();
33
34     // Perform and print inorder of the tree
35     strTree.inorder();
36
37     // Print the string representation of the tree.
38     strTree.print();
39 }
```

Задача 2. Проверка дали постои даден јазел во бинарно дрво

За дадено бинарното дрво и јазел за кој се вчитува податочниот елемент од тастатура, да се провери дали вредноста на јазелот постои во бинарното дрво.

Влез:

Бинарно дрво од слика 6-11.

Вредност на јазел: 6

Излез:

Node with value 6 exists in the given binary tree

Идејата за имплементација на оваа функција е следна: тргни од почетен јазел и провери дали тој ја содржи бараната вредност, доколку не ја содржи пребарувај во неговото лево поддрво. Доколку се најде вредноста ќе се врати цел јазел како објект од бинарното дрво, а доколку не се најде, се продолжува со пребарувањето

во десното поддрво. Исто така, доколку вредноста се најде, ќе се врати цел јазел како објект од бинарното дрво, доколу вредноста не е најдена се враќа вредност null. Имплементацијата на оваа функција е дадена како дел од приватната метода `findNode(TreeNode<T>, T)`. За да може комплетно да функционира алгоритамот оваа метода се повикува во друга јавна метода со истото име, но сега само со еден аргумент, а тоа е вредноста на јазелот (како што и гласи барањето во задачата). Јавната метода `findNode(T)` прави повик на приватната метода `findNode`, со тоа што пребарувањето стартува директно од коренот на дрвото. Сите овие методи се дел од класата `BinaryTree`. Во продолжение е даден кодот за нив:

```
1 // private method to find a node with a specific value in a binary tree.
2
3 private TreeNode<T> findNode(TreeNode<T> n, T key) {
4     // If the current node is null, the key is not found in the tree.
5     if (n == null)
6         return null;
7
8     // Check if the current node's data is equal to the key.
9     if (n.data.equals(key))
10        return n; // Return the current node if it matches the key.
11
12    // Recursively search for the key in the left subtree.
13    TreeNode<T> foundLeft = findNode(n.left, key);
14
15    // If the key is found in the left subtree, return the result.
16    if (foundLeft != null)
17        return foundLeft;
18
19    // If the key is not found in the left subtree, search in the right
20    // subtree.
21    TreeNode<T> foundRight = findNode(n.right, key);
22
23    // Return the result of the search in the right subtree.
24    return foundRight;
25 }
26
27 //public function to check if a node with a specific value exists in the
28 //binary tree
29 public void findNode(T data){
30     // Call the findNode method to search for the node with the given
31     // data starting from the root node
32     if(findNode(root, data) != null)
33         System.out.println("Node with value " + data + " exists in the
34         binary tree");
35 }
```

```
30         given binary tree");
31     else
32         System.out.println("Node with value " + data + " does NOT exist
33                             in the given binary tree");
34     }
35 }
```

За да се истестираат соодветните функции ќе направиме соодветна тестирачка метода `exampleFindNode()` во класата `BinaryTreeTest` која само ќе ја повикаме во главната функција `main`.

```
1 public class BinaryTreeNode {
2     // method to test findNode()
3     public static void exampleFindNode() {
4         // Create an example integer tree and keep a reference in intTree.
5         BinaryTree<Integer> intTree = GetExampleIntTree();
6
7         // Create a Scanner to get user input.
8         Scanner input = new Scanner(System.in);
9
10        // Prompt the user to enter an integer value.
11        System.out.print("Enter an integer value to search in the binary
12          tree: ");
13
14        // Read the user's input as an integer.
15        int value = Integer.parseInt(input.next());
16
17        // Call the findNode method to check if the value exists in the
18          binary tree.
19        if(intTree.findNode(value) != null)
20            System.out.println("Node with value " + value + " exists in the
21              given binary tree");
22        else
23            System.out.println("Node with value " + value + " does NOT exist
24              in the given binary tree");
25    }
26
27    public static void main(String[] args) {
28        //Call the exampleFindNode method
29        exampleFindNode();
30    }
31}
```

 }

Задача 3. Наоѓање ниво на јазел во бинарно дрво

Да се напише функција која што го враќа нивото на соодветен јазел во дрвото (сметајќи дека коренот се наоѓа на ниво 1) или 0 ако не постои во дрвото. Потоа, да се тестира истата за бинарното дрво на слика 6-11, и јазол за кој се вчитува податочниот елемент од тастатура.

Влез:

Бинарно дрво од слика 6-11.

Вредност на јазел: 6

Излез:

Level of 6 is 3

Во дадената задача треба да се одреди нивото на даден јазел – во случајов за јазелот 6 во бинарното дрво. За таа цел, се почнува соодветно од коренот на кој му се доделува ниво 1. Ако бараниот јазел соодветствува со коренот, се враќа нивото, ако не, рекурзивно се изминуваат неговото лево и десно поддрво со вредност на нивото за еден поголема од претходно. Имплементацијата на оваа функција е дадена во приватниот методот `getLevel(TreeNode<T>, T, int)`. За да може алгоритамот комплетно да функционира се прави повик на оваа метода во ново дефинирана јавна функција `getLevel(T)` која на влез прима само информација за јазелот за кој треба да се врати нивото, а таа интерно ја повикува другата - приватна метода `getLevel` која стартира директно од коренот на дрвото и почетна вредност на ниво 1. Сите овие методи се дел од класата `BinaryTree`.

Во продолжение е даден кодот за нив:

```

1 // This method calculates the level (depth) of a node with a specific
2 // 'data' value in a binary tree.
3 private int getLevel(TreeNode<T> node, T data, int level) {
4     // Base case: If the current node is null, return 0.
5     if (node == null) {
6         return 0;
7     }
8     // Check if the current node's data matches the specified 'data'.
9     if (node.data.equals(data)) {
10        return level; // The level of the node with 'data' has been
11        found.
12    }

```

```

13     // Recursively search for 'data' in the left subtree.
14     int tmpLevel = getLevel(node.left, data, level + 1);
15
16     if (tmpLevel != 0) {
17         return tmpLevel; // Return the level if found in the left
18         subtree.
19     }
20
21     // If 'data' is not found in the left subtree, search in the right
22     // subtree.
23     tmpLevel = getLevel(node.right, data, level + 1);
24     return tmpLevel;
25 }
26
27 // This public method calculates the level (depth) of a node with a
28 // specific 'data' value in a binary tree.
29 public int getLevel(T data) {
30     // Start the level calculation from the root of the tree.
31     int level = getLevel(root, data, 1);
32
33     // Return the calculated level.
34     return level;
35 }
```

За да се истестираат соодветните функции ќе направиме соодветна тестирачка метода `exampleGetLevel()` во класата `BinaryTreeTest` која само ќе ја повикаме во главната функција `main`.

```

1  public class BinaryTreeTest {
2     // Method to test the getLevel() method.
3     public static void exampleGetLevel() {
4         // Create an example integer tree and keep a reference in intTree.
5         BinaryTree<Integer> intTree = GetExampleIntTree();
6
7         // Create a Scanner to get user input.
8         Scanner input = new Scanner(System.in);
9
10        // Prompt the user to enter an integer value.
11        System.out.print("Enter an integer value to get its level in the
12                                binary tree: ");
13
14        // Read the user's input as an integer.
```

```

14     int value = Integer.parseInt(input.next());
15
16     // Call the getLevel method to find the level of the node in the
17     // binary tree.
17     int level = intTree.getLevel(value);
18
19     // Check if the level is not 0
20     if (level != 0) {
21         System.out.println("Level of " + value + " is " + level);
22     } else {
23         System.out.println(value + " is not present in the tree");
24     }
25 }
26
27 public static void main(String[] args) {
28     //Call the exampleGetLevel method
29     exampleGetLevel();
30 }
31 }
```

Задача 4. Сума на елементи во бинарно дрво

Нека е дадено бинарно дрво со цели броеви. Да се напише функција која што ќе го пресмета збирот на елементите од целото дрво.

Влез:

Бинарно дрво од слика 6-11.

Излез:

Sum of all the elements in the tree is: 75

За да се пресмета сумата на елементите на бинарното дрво, потребно е да се измине целото дрво. Оваа функција е карактеристична за бинарни дрва кои како вредност на јазел имаат вредности кои се од нумерички податочен тип. Токму затоа, оваа функција нема да важи општо за сите бинарни дрва, туку само за примерот кој што ни е даден. На пример, за бинарни дрва од стринг податочен тип нема смисол да се пресметува сума. Поради тоа, имплементацијата ќе ја ставиме да биде дел од тестирачката класа `BinaryTreeTest`. Сумата на елементите во бинарно дрво ја правиме на тој начин што почнуваме од коренот на дрвото и рекурзивно се движиме во неговото лево и десно поддрво. Така движејќи се низ дрвото, ние ја калкулураме сумата на елементите кои ги посетуваме. Рекурзивната функција е дадена под името `sumBT(TreeNode<Integer>)` и истата е повикана во методата `exampleSumBT()` каде соодветно се тестира.

```
1  public class BinaryTreeTest {
2      // Method to calculate the sum of all elements in a binary tree.
3      public static int sumBT(TreeNode<Integer> node){
4          // If the current node is null, return 0 (base case).
5          if (node == null)
6              return 0;
7
8          // Recursively calculate the sum of the current node's data,
9          // left subtree, and right subtree, and return the total sum.
10         return (node.data + sumBT(node.left) + sumBT(node.right));
11     }
12
13     // Method to test the sumBT() method.
14     public static void exampleSumBT(){
15         // Create an example integer tree and keep a reference in intTree.
16         BinaryTree<Integer> intTree = GetExampleIntTree();
17
18         // Calculate the sum of all elements in the tree starting from the
19         // root.
20         int sum = sumBT(intTree.root);
21
22         // Print the result.
23         System.out.println("Sum of all the elements in the tree is: " + sum);
24     }
25
26     public static void main(String[] args) {
27         //Call the exampleSumBT method
28         exampleSumBT();
29     }
30
31 }
```

Задача 5. Сума на поддрва на даден елемент во бинарно дрво

Нека е дадено бинарно дрво со цели броеви и вредноста на еден јазел во дрвото. Да се напише функција која што ќе го пресмета збирот на елементите во левото поддрво на дадениот јазел кои се помали од него и функција која што ќе го пресмета збирот на елементите во десното поддрво на дадениот јазел кои се поголеми од него.

Пример:

Влез:

Бинарно дрво од слика 6-11.

Вредност на јазел: 7

Излез:

The sum of the left subtree is 2

The sum of the right subtree is 11

Во дадената задача треба да се одреди збирот на елементите од левото/десното поддрво на даден јазол кои се помали/поголеми од него. За таа цел, прво што треба да направиме е да ја најдеме позицијата во дрвото на дадениот јазел. Според тоа, соодветно треба да ја искористиме веќе креираната функција `findNode(T key)` како дел од постоечката класа `BinaryTree`. Откако соодветниот јазел е најден, може да се пристапи кон имплементација на методите за пресметување на сумите. За да ја имплементираме функцијата која ќе го пресметува збирот на елементите во бинарното дрво помали од вредноста на внесениот јазел, употребуваме рекурзивана функција која пребарува во левото поддрво на најдениот јазел и доколку вредноста на јазлите е помала од неговата вредност, сумата се зголемува. Рекурзивната функција завршува тогаш кога ќе стигнеме до листовите на поддрвото. Слична е и имплементацијата на функцијата за пресметување на сумата на елементите во бинарното дрво, поголеми од вредноста на внесениот јазел. Комплетниот код на функциите е даден во продолжение:

```

1 public class BinaryTreeTest {
2     public static int sumMinLeftSubtree(TreeNode<Integer> current, int
3         value) {
4         // Base case: If the current node is null, return 0.
5         if (current == null)
6             return 0;
7         // recursively summing the values in its subtrees
8         int tmp = sumMinLeftSubtree(current.left, value) +
9             sumMinLeftSubtree(current.right, value);
10        // Check if the value of the current node is less than the specified
11        // 'value'.
12        if (current.data < value) {
13            // Include the current node's value in the sum and recursively
14            // sum
15            return tmp + current.data;
16        } else {
17            // If the current node's value is not less than 'value', just
18            // return the subtree's values
19            return tmp;
20        }
21    }
22 }
```

```
15         }
16     }
17
18     public static int sumMaxRightSubtree(TreeNode<Integer> current, int
19         value) {
20
21         // Base case: If the current node is null, return 0.
22         if (current == null)
23             return 0;
24
25         // recursively summing the values in its subtrees
26         int tmp = sumMaxRightSubtree(current.left, value) +
27             sumMaxRightSubtree(current.right, value);
28
29         // Check if the value of the current node is greater than the
30         // specified 'value'.
31         if (current.data > value) {
32
33             // Include the current node's value in the sum and recursively
34             // sum
35             return current.data + tmp;
36         } else {
37
38             // If the current node's value is not less than 'value', just
39             // return the subtree's values
40             return tmp;
41         }
42     }
43
44
45     // Method to test the exampleSumSubtrees() method.
46     public static void exampleSumSubtrees() {
47
48         // Create an example integer tree and keep a reference in intTree.
49         BinaryTree<Integer> intTree = GetExampleIntTree();
50
51
52         // Create a Scanner to get user input.
53         Scanner input = new Scanner(System.in);
54
55
56         // Prompt the user to enter an integer value.
57         System.out.print("Enter an integer value to search in the binary
58             tree: ");
59
60         // Read the user's input as an integer.
61         int value = Integer.parseInt(input.next());
62
63
64         // Find the node with the entered value in the binary tree.
65         TreeNode<Integer> node = intTree.findNode(value);
```

```

50
51     // Check if the node exists in the tree.
52     if (node != null) {
53         // Calculate and display the sum of the left subtree of the
54         // found node.
55         System.out.println("The sum of the left subtree is " +
56             sumMinLeftSubtree(node.left, value));
57
58         // Calculate and display the sum of the right subtree of the
59         // found node.
60         System.out.println("The sum of the right subtree is " +
61             sumMaxRightSubtree(node.right, value));
62     } else {
63         // Display a message indicating that the node with the entered
64         // value doesn't exist in the tree.
65         System.out.println("Node with value " + value + " does NOT exist
66             in the given binary tree");
67     }
68 }
```

Задача 6. Растојание помеѓу јазли

Нека е дадено бинарно дрво со цели броеви. Да се напише функција која што ќе го пресмета растојанието помеѓу два јазли. Вредностите на јазлите за кои треба да се пресмета растојанието се соодветно дадени.

Пример:

Влез:

Бинарно дрво од слика 6-11

Вредноста на првиот јазел: 7

Вредноста на вториот јазел: 15

Излез:

$\text{Dist}(7, 15) = 4$

Под растојание помеѓу два јазли се подразбира најмалиот број на ребра кои треба да се изминат за да се стигне од едниот до другиот јазел во дрвото. За да

се определи ова растојание имаме неколку случаи кои треба да се разгледаат и тоа:

- едниот јазел се наоѓа во левото другиот во десното поддрво на даденото дрво. Во овој случај пресметувањето на растојание се сведува на наоѓање растојание од коренот до едниот јазел + растојанието од коренот до другиот јазел. Во дадениот пример, тоа ни се јазлите, на пример, 2 и 9.
- двата јазли се наоѓаат во исто поддрво на влезното дрво. Во овој случај за да се пресмета растојанието треба да се земат во предвид најмалиот број на ребра, а тоа ќе се постигне со тоа што ќе се најде најдолниот заеднички предок на дадените јазли. Во дадениот пример тоа ни се јазлите на пример 2 и 11. Така што прво треба да се најде соодветниот најдолен заеднички предок (lowest common ancestor - LCA), а тоа е 7. Според тоа растојанието ќе се пресмета како растојание од LCA до едниот јазел + растојанието од LCA до другиот јазел.

Како што веќе напоменавме за да го имплементираме ова решение потребно е прво да ја имплементираме методата за наоѓање на најдолен заеднички предок на два јазли дадени со соодветните вредности. Во продолжение е кодот за оваа функција која е дел од класата `BinaryTree`.

```

1 // This method finds the Lowest Common Ancestor (LCA) of two nodes with
2 // values 'n1' and 'n2' in a binary tree.
3 private TreeNode<T> findLCA(TreeNode<T> node, T n1, T n2) {
4     // Base case: If the current node is null, return null.
5     if (node == null) {
6         return null;
7     }
8
9     // Check if the current node's data matches either of the two values
10    // 'n1' or 'n2'.
11    if (node.data.equals(n1) || node.data.equals(n2)) {
12        return node; // This node is LCA.
13    }
14
15    // Recursively search for the LCA in the left and right subtrees.
16    TreeNode<T> leftLCA = findLCA(node.left, n1, n2);
17    TreeNode<T> rightLCA = findLCA(node.right, n1, n2);
18
19    // Determine the LCA based on the results from left and right
20    // subtrees.
21    if (leftLCA != null && rightLCA != null) {
22        return node;
23    }
24
25    return leftLCA != null ? leftLCA : rightLCA;
26}
```

```

19         return node; // LCA is the root of the subtrees
20     }
21     if (leftLCA == null && rightLCA == null) {
22         return null; // not found in either subtree.
23     }
24     if (leftLCA != null) {
25         return leftLCA; // LCA found in the left subtree.
26     } else {
27         return rightLCA; // LCA found in the right subtree.
28     }
29 }
```

Откако LCA е најден како јазел во дрвото, многу лесно може да се пресмета растојанието. Односно тута ќе го искористиме веќе креираниот метод за наоѓање на ниво на јазел во дрво од претходна задача, методот `getLevel(TreeNode<T>, T, int)` со кој ќе ги најдеме растојанието од LCA до вредноста на првиот јазел и растојанието од LCA до вредноста на вториот јазел. Со ова го покриваме и случајот кога јазлите се наоѓаат во различни поддрва на почетното дрво, каде улогата на LCA ја игра коренот на дрвото. Во продолжение е даден кодот:

```

1 // Returns the distance between 'n1' and 'n2' in the binary tree.
2 public int getDist(T n1, T n2) {
3     // Find the Lowest Common Ancestor (LCA) of 'n1' and 'n2'.
4     TreeNode<T> lca = findLCA(root, n1, n2);
5
6     // Calculate the level (depth) of 'n1' and 'n2' with respect to the
7     // LCA.
8     int d1 = getLevel(lca, n1, 0);
9     int d2 = getLevel(lca, n2, 0);
10
11    // Return the sum of the distances from LCA to 'n1' and LCA to 'n2'.
12    return d1 + d2;
13 }
```

За да се истестираат соодветните функции ќе направиме соодветна тестирачка метода `exampleGetDist()` во класата `BinaryTreeTest` која само ќе ја повикаме во главната функција `main`.

```

1 public class BinaryTreeTest {
2     // Method to test the getDist() method.
3     public static void exampleGetDist() {
4         // Create an example integer tree and keep a reference in intTree.
5         BinaryTree<Integer> intTree = GetExampleIntTree();
```

```
6
7      // Create a Scanner to get user input.
8      Scanner input = new Scanner(System.in);
9
10     // Prompt the user to enter two integer values to find their
11     // distance in the binary tree.
12     System.out.print("Enter two integer values to get their distance in
13     // the binary tree: ");
14
15     // Read the user's input as two integers.
16     int value1 = Integer.parseInt(input.next());
17     int value2 = Integer.parseInt(input.next());
18
19     // Call the getDist method to find the distance between the two
20     // values in the binary tree.
21     int dist = intTree.getDist(value1, value2);
22
23     // Check if the distance is not 0, indicating both values were found
24     // in the tree.
25     if (dist != 0) {
26         System.out.println("Dist(" + value1 + ", " + value2 + ") = " +
27             dist);
28     } else {
29         // Display a message indicating that one or both of the values
30         // are not present in the tree.
31         System.out.println("Some of the values are not present in the
32         // tree");
33     }
34 }
```

6.1.10 Задачи за вежбање

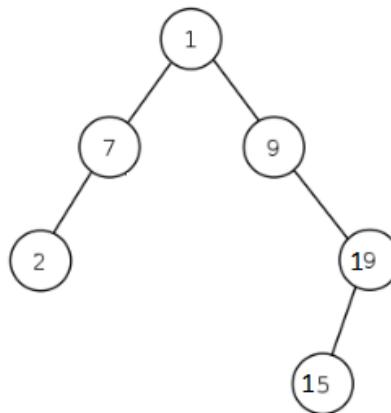
Задача 1. Сума на леви деца

Нека е дадено бинарно дрво со цели броеви. Да се напише функција која што ќе го пресмета збирот на елементите чии јазли имаат само леви деца.

Пример:

Влез:

Бинарно дрво од слика 6-13.



Слика 6-13: Пример на бинарно дрво

Излез:

The sum is 26

Задача 2. Збирно дрво

Нека е дадено бинарно дрво со цели броеви. Да се напише функција која што ќе го провери дали даденото бинарно дрво е збирно дрво. Збирно дрво е бинарно дрво каде вредноста на еден јазел е еднаква со збирот од вредностите на јазлите во неговото лево и неговото десно поддрво. Се смета дека празно дрво е збирно дрво и збирот е 0.

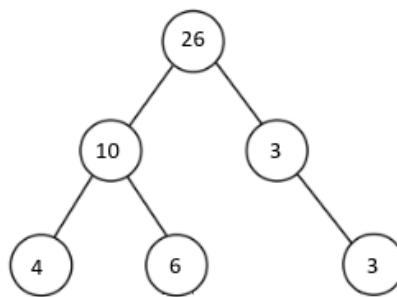
Пример:

Влез:

Бинарно дрво од слика 6-14

Излез:

The given tree is a SumTree



Слика 6-14: Пример на бинарно дрво

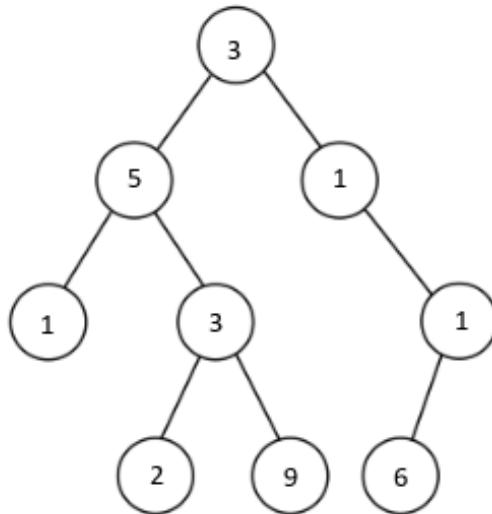
Задача 3. Сума на сите патишта од корен до лист

Нека е дадено бинарно дрво кое се состои од цифрите 1-9. Да се напише функција која што ќе ја пресмета сумата на сите броеви кои се формираат на патот од коренот до соодветниот лист на дрвото.

Пример:

Влез:

Бинарно дрво од слика 6-15



Слика 6-15: Пример на бинарно дрво

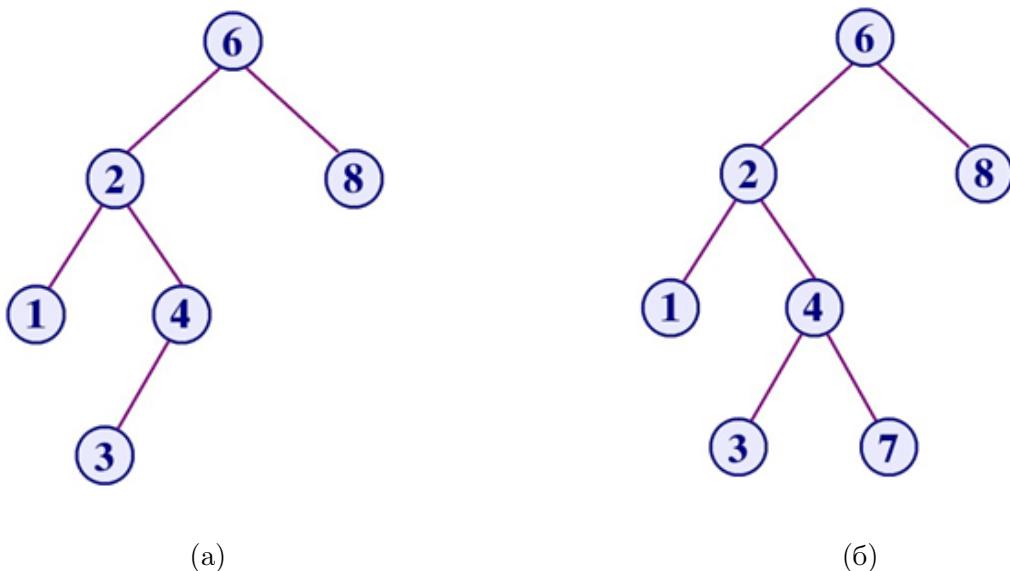
Излез:

$$351 + 3532 + 3539 + 3116 = 10538$$

6.2 Пребарувачки дрва

6.2.1 Бинарни пребарувачки дрва

Една важна примена на бинарните дрва е во областа на пребарувањето. Да претпоставиме дека на секој јазел од бинарното дрво е доделена една вредност наречена клуч. Бинарното дрво се нарекува пребарувачко доколку за секој јазел од левото поддрво на јазелот со клуч X содржи вредности (клучеви) помали од X и секој јазел од десното поддрво на јазелот со клучот X , содржи јазли со клучеви поголеми од X (за сега се претпоставува дека не постојат дупликати на клучевите). На слика 6-16а е прикажано бинарно пребарувачко дрво, а на слика 6-16б обично бинарно дрво. Второто дрво не е пребарувачко поради тоа што јазелот со клуч 7 не се наоѓа во десното поддрво на јазелот со клуч 6.



Слика 6-16: Бинарно пребарувачко дрво (а) и обично бинарно дрво (б)

Интересно е да се забележат неколку својства на бинарните пребарувачки дрва. Јазелот со најмал клуч е најлевиот лист на целото дрво, односно јазелот до кој се доаѓа доколку од коренот се оди само по левите врски се додека не се дојде до листот на дрвото. Јазелот со најголем клуч е најдесниот лист на целото дрво, односно листот до кој се доаѓа со одењето по само десните врски почнувајќи од коренот на дрвото. Доколку дрвото се измине во инордер, се добива подредена низа од клучеви и тоа во растечки редослед. Дефиницијата за јазел од пребарувачко дрво е во принцип иста како и дефиницијата на јазел за обично бинарно дрво. Од тука следи дека апликативно мора да се води грижа за дефиницијата на пребарувачко бинарно дрво инаку во суштина тоа е обично

бинарно дрво. При имплементација, односно при креирањето на самото бинарно дрво мора да се исполнат сите проверки кои ќе гарантираат дека е задоволена дефиницијата на бинарно пребарувачко дрво.

6.2.2 Основни манипулации со јазлите во бинарните пребарувачки дрва

Пребарување

Типична операција која се извршува над бинарното пребарувачко дрво е операцијата на пребарување. Оваа операција е лесна бидејќи јазлите во бинарното пребарувачко дрво се зачувани по специфичен редослед. Чекорите кои треба да ги преземеме кога пребаруваме се следни:

1. спореди го клучот кој се пребарува со коренот на дрвото
2. доколку коренот соодветствува на клучот, тогаш го враќаме јазелот од коренот
3. доколку коренот не соодветствува на клучот, тогаш споредуваме дали клучот е помал или поголем од коренот.
 - 3.1. доколку клучот е помал од коренот, продолжуваме во левото поддрво со пребарување
 - 3.2. доколку клучот е поголем од коренот, продолжуваме во десното поддрво
4. повторувај ги овие чекори рекурзивно се додека не се најде клучот. Доколку пребарувањето заврши (сме стигнале до листовите на левото и десното поддрво), а клучот не е најден, треба да се врати null.

Описаната рекурзивна процедура, може да се претстави со следниот јава код:

```

1 // Private function that recursively searches for a node containing a
2 // specific element 'x' in the BST
3 private TreeNode<T> find(TreeNode<T> node, T x) {
4     // If the current node is null, the element 'x' is not present in
5     // this branch of the tree.
6     if (node == null)
7         return null;
8
9     // Compare 'x' with the current node's data to decide whether to
10    // search in the left or right subtree.
11    if (x.compareTo(node.data) < 0) {
12        // 'x' is smaller than the current node's data, so continue
13        // searching in the left subtree.
14    }
15 }
```

```

10         return find(node.left, x);
11     } else if (x.compareTo(node.data) > 0) {
12         // 'x' is larger than the current node's data, so continue
13         // searching in the right subtree.
14         return find(node.right, x);
15     } else {
16         // 'x' matches the current node's data, so we've found a
17         // match.
18         return node;
19     }

```

```

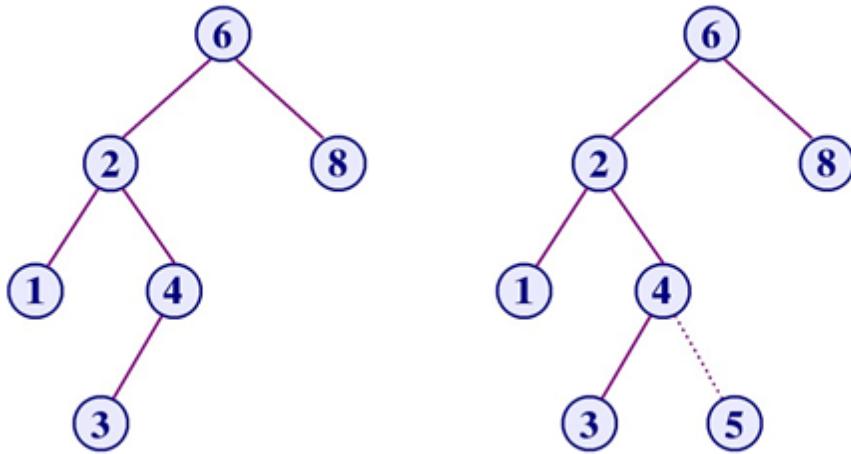
1 // Public function to search for an element 'x' in the BST
2     public TreeNode<T> find(T x) {
3         // Call the private 'find' method starting from the root of the tree.
4         return find(root, x);
5     }

```

Вметнување јазел

Операцијата за вметнување во бинарно пребарувачко дрво се сведува на вметнување терминален јазел на соодветната позиција. Секако дека треба да се внимава да биде задоволена дефиницијата на бинарното пребарувачко дрво и откако ќе се вметне новиот јазел. За да се вметне јазел со даден клуч, мора да се започне со пребарување од коренот на дрвото; ако клучот на јазелот што сакаме да го вметнеме е помал од коренот, тогаш пребаруваме слободна позиција (како терминален јазел) во левото поддрво. Инаку, бараме слободен терминален јазол во десното поддрво за да го направиме вметнувањето. Нека оваа операција ја разгледаме преку пример кој што е претставен на слика 6-17. За да го вметнеме јазелот со клуч 5 во бинарното пребарувачко дрво (лево на сликата 6-17), започнуваме од коренот, односно споредба на клучот 5 со коренот ($5 < 6$). Продолжуваме со пребарување во левото поддрво и тоа во следниот чекор правиме споредба со коренот 2. Со оглед на тоа што $5 > 2$, продолжуваме со пребарувањето на соодветната поизија за вметнување во десното поддрво, односно споредба со коренот 4. Повторно $5 > 4$, па продолжуваме во десното поддрво. Поради тоа што јазелот 4 нема десно поддрво, значи дека најдовме слободна терминалана локација на која може соодветно да го вметниме новиот јазел. Како резултат по вметнувањето го добивме бинарното пребарувачко дрво претставено на десната страна на слика 6-17.

Комплетниот код за оваа операција е даден во продолжение.



Слика 6-17: Бинарно пребарувачко дрво пред и по вметнување на јазел со клуч 5

```

1 // Private function to perform the insert operation, starting at node
2   'node'.
3
4   private TreeNode<T> insert(TreeNode<T> node, T x) {
5     // If node is null, create and return a new node with data 'data'.
6     if (node == null) {
7       return new TreeNode<>(x);
8     }
9
10    // Compare the new data with the node's data. If new data is less,
11    // insert into the left subtree.
12    if (x.compareTo(node.data) < 0) {
13      node.left = insert(node.left, x);
14    }
15    // If new data is greater, insert into the right subtree.
16    else if (x.compareTo(node.data) > 0) {
17      node.right = insert(node.right, x);
18    }
19
20    // Return the node after insertion.
21    return node;
22  }
23
24
25 // Public insert function to add a new node with data 'x'.
26 public void insert(T x) {
27
28 }
```

```

3     // Internal function call to perform the insert operation.
4         root = insert(root, x);
5

```

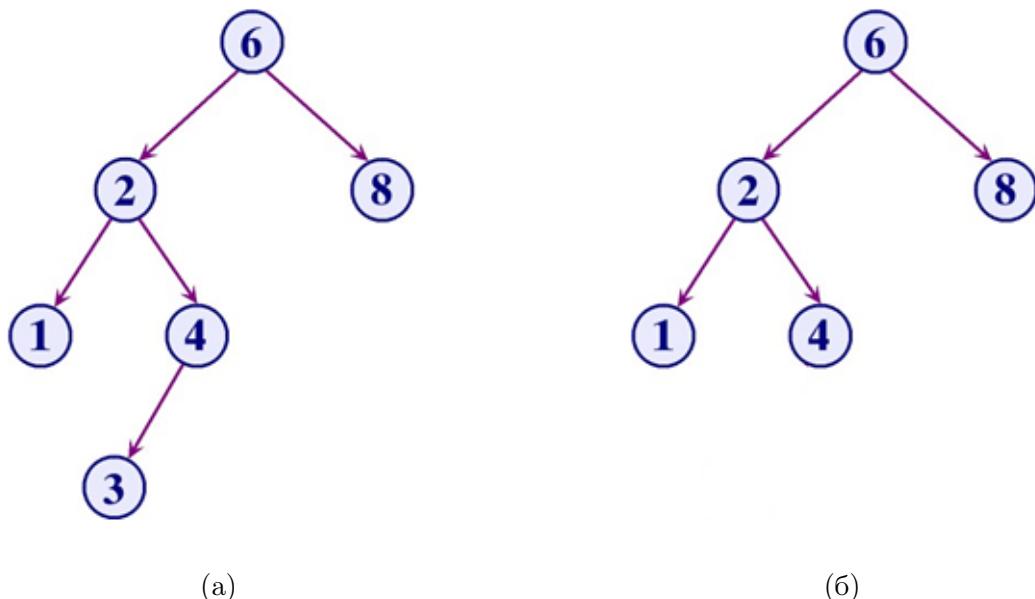
Бришење јазел

Операцијата на бришење клуч (односно јазел што го содржи клучот) од бинарно пребарувачко дрво е покомплексна операција од операцијата вметнување. Тоа се должи на фактот што и по бришењето на саканиот јазел, дрвото мора да ги задоволува условите кои се барани од него за тоа да биде пребарувачко. Поради тоа, при бришењето на јазел од бинарно дрво ќе бидат разгледани неколку случаи:

- јазелот што треба да се избрише е лист (терминален јазел)
- јазелот што треба да се избрише има само едно дете и
- јазелот што треба да се избрише има две деца.

Јазелот што треба да се избрише е лист (терминален јазел)

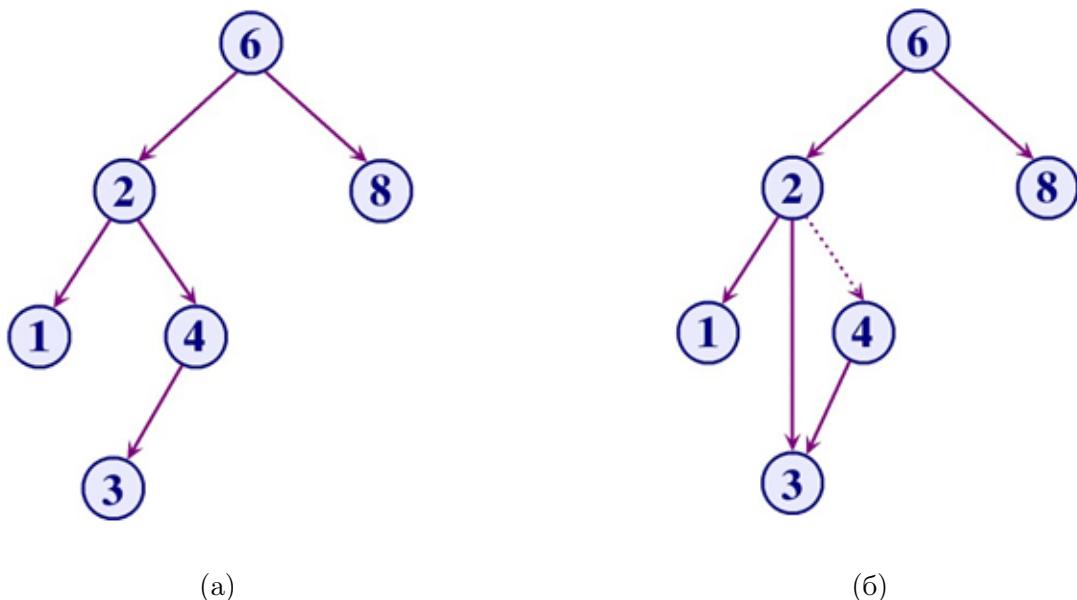
Ова е наједноставниот начин да се избрише јазел од бинарно пребарувачко дрво. Во овој случај потребно е само да го замениме терминалниот јазел со null. Операцијата бришење јазел од бинарно пребарувачко дрво кој всушност е терминален јазел е претставена на слика 6-18.



Слика 6-18: Бришење на терминален јазел пред (а) и после операцијата (б)

Јазелот што треба да се избрише има само едно дете

Во овој случај, треба да се направи замена на јазелот што го бришеме со неговото дете. Тоа значи дека врската од родителот на јазелот што сакаме да го избришеме кон него сега станува врска од неговиот родител кон неговото дете. Операцијата бришење јазел од бинарно пребарувачко дрво кој всуност има само едно дете е претставена на слика 6-19.



Слика 6-19: Бришење на јазел со едно дете (клуч 4) пред (а) и после операцијата (б)

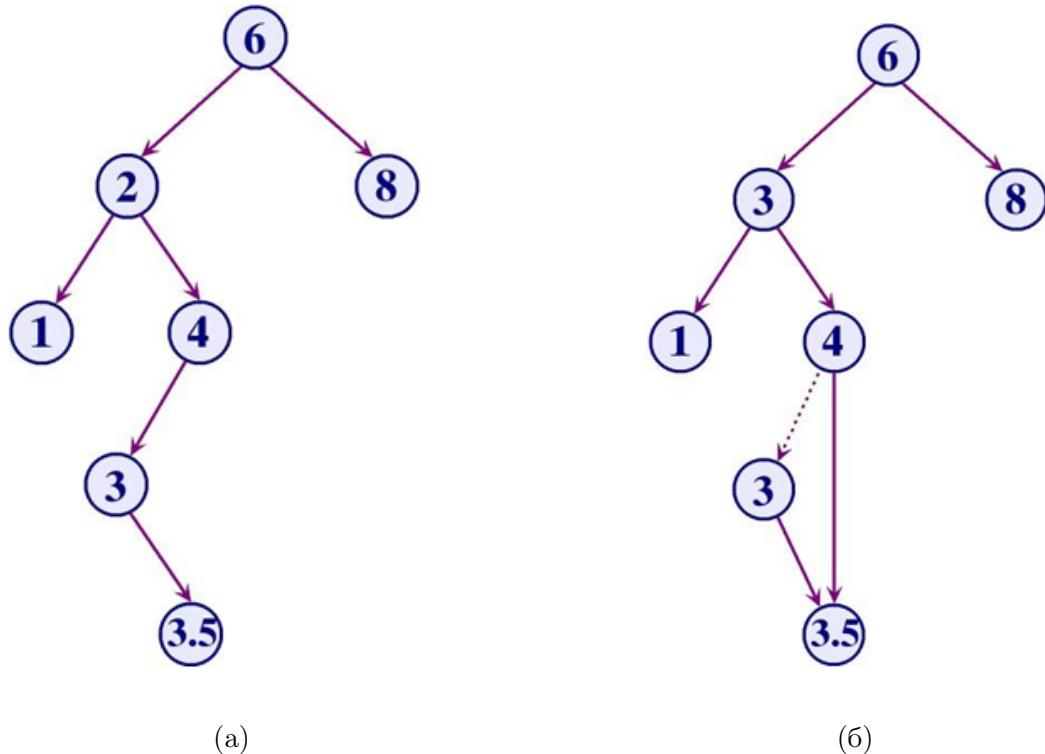
Јазелот што треба да се избрише има две деца

Најкомплексен е случајот кога јазелот што треба да се избрише има две деца. Алгоритамот за бришење кој треба да се реализира се состои од три чекори:

1. Во првиот чекор се наоѓа најмалиот клуч од десното поддрво на јазелот што треба да се избрише.
2. Во вториот чекор тој се заменува со клучот од јазелот кој треба да се избрише.
3. Во третиот чекор се брише јазелот од кој се искористил клучот во предходниот чекор. Овој случај на бришење е едноставен затоа што јазелот што го содржи најмалиот клуч во некое поддрво мора да има нулева лева врска, па согласно предходната анализа бришењето се сведува на премостување врски. Важно е да се разбере дека со оваа замена дрвото останува пребарувачко, поради тоа што сите јазли од десното поддрво се поголеми од јазелот

што сакаме да го избришеме, па согласно тоа, по неговото бришење тие ќе бидат поголеми од најмалиот јазел во тоа поддрво.

Илустрација на овој случај на бришење е дадена на слика 6-20.



Слика 6-20: Бришење на јазел со две деца (клуч 2) пред (а) и после операцијата (б)

Кодот за реализација на операцијата бришење јазел од бинарно пребарувачко дрво заедно со сите негови случаи е даден во продолжение.

```

1 // Private function to remove a node with element 'x' from the BST,
2 // starting at node 'node'
3 private TreeNode<T> remove(TreeNode<T> node, T x) {
4     // If the current node is null, the element 'x' is not present in
5     // this branch of the tree.
6     if (node == null)
7         return node; // Item not found; do nothing
8
9     // Compare 'x' with the current node's data to decide whether to
10    // search in the left or right subtree.
11    if (x.compareTo(node.data) < 0) {
12        // 'x' is smaller than the current node's data, so continue
13        // searching in the left subtree.
14    }
15 }
```

```

10         node.left = remove(node.left, x);
11     } else if (x.compareTo(node.data) > 0) {
12         // 'x' is larger than the current node's data, so continue
13         // searching in the right subtree.
14         node.right = remove(node.right, x);
15     } else if (node.left != null && node.right != null) { // Two children
16         // If the node to remove has two children, replace its data with
17         // the smallest element
18         // from its right subtree (in-order successor), and then remove
19         // the duplicate element.
20         node.data = findMin(node.right).data;
21         node.right = remove(node.right, node.data);
22     } else {
23         // If the node has only one child (or no children), return the
24         // non-null child,
25         // effectively removing the current node from the tree.
26         if (node.left != null)
27             return node.left;
28         else
29             return node.right;
30     }
31     return node; // Return the updated node.
32 }
```

```

1 // Private method to find the minimum element in a subtree rooted at
2 // 'node'.
3 private TreeNode<T> findMin(TreeNode<T> node) {
4     // If the current node is null, the subtree is empty, so return null.
5     if (node == null) {
6         return null;
7     } else if (node.left == null) {
8         // If there is no left child, this node contains the minimum
9         // element in the subtree.
10        return node;
11    } else {
12        // Continue searching for the minimum element in the left
13        // subtree.
14        return findMin(node.left);
15    }
16 }
```

```

1 // Public remove function to remove a node with data 'x'.
```

```

2   public void remove(T x) {
3       // Internal function call to perform the remove operation.
4       root = remove(root, x);
5   }

```

6.2.3 Сложеност на операциите кај бинарни дрва наспроти бинарни пребарувачки дрва

Кога заборуваме за податочна структура бинарно дрво заедно со неговиот подтип бинарно пребарувачко дрво, кажавме дека најупотребувани операции се операцијата на пребарување, вметнување и бришење на јазел. За секоја од овие операции ја прикажавме нивната соодветна имплементација, па сега останува да ги дефинираме временските сложености за секоја од нив претпоставувајќи дека станува збор за дрва кои имаат намногу N јазли [3].

- операција пребарување
 - кај бинарните дрва, кажавме дека оваа операција е најскапата операција поради фактот што нема правило како се подредени јазлите во дрвото и тие можат да се појават било каде. Според тоа за да пребарајме јазел потребно е да го изминеме целото дрво, со што се потврдува дека во најлош, а и просечен случај сложеноста е $O(N)$.
 - кај бинарните пребарувачки дрва, ситуацијата е поразлична, односно според начинот на кој што се распределени јазлите, пребаруваме само во левото или само во десното поддрво. Па според тоа, ако дрвото е балансирано, тоа значи дека најголемиот број проверки кои треба да ги направиме е колку што е висината на дрвото, односно $\log_2 N$. Значи во просечен случај, сложеноста е $\mathcal{O}(\log_2 N)$. Доколку дрвото не е балансирано, односно имаме случај кога дрвото има само леви односно само десни деца, тогаш мора да ги изминеме сите јазли на дрвото и сложеноста станува $O(N)$ како најлош случај.
- операција вметнување јазел
 - за да се изведе оваа операција потребно е прво да се стигне до местото на кое сакаме да направиме вметнување, а потоа да се додаде новиот јазел што во суштина е едноставна операција. Според тоа и за обично и за пребарувачко дрво објаснувањето е исто и се сведува на сложеноста на операцијата пребарување. Сложеноста во најлош случај е $O(N)$, а во просечен случај промена има само кај пребарувачките дрва, каде сложеноста се намалува на $\mathcal{O}(\log_2 N)$.

- операција бришење јазел
 - сложеноста за изведување на оваа операција е иста како и кај операцијата за вметнување јазел.

Сумарен преглед на сложеностите на операциите кај обични и пребарувачки бинарни дрва е даден во табела 6.1 и табела 6.2 .

Табела 6.1: Сложеност на најчестите операции кај бинарно дрво со N јазли

операции кај ВТ	просечен случај	најлош случај
пребарување јазел	$O(N)$	$O(N)$
вметнување јазел	$O(N)$	$O(N)$
бришење јазел	$O(N)$	$O(N)$

Табела 6.2: Сложеност на најчестите операции кај бинарно пребарувачко дрво со N јазли

операции кај BST	просечен случај	најлош случај
пребарување јазел	$O(\log_2 N)$	$O(N)$
вметнување јазел	$O(\log_2 N)$	$O(N)$
бришење јазел	$O(\log_2 N)$	$O(N)$

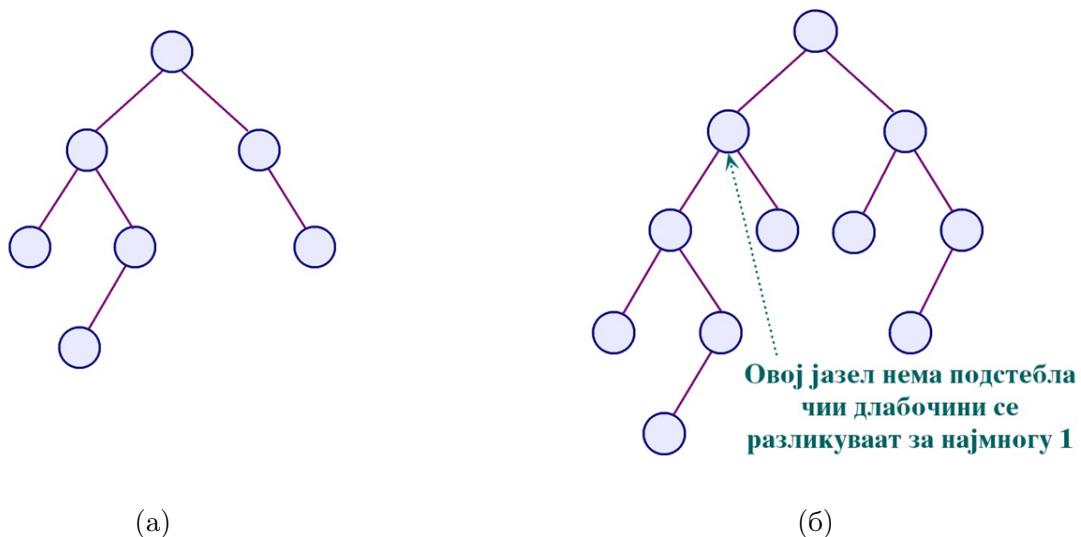
Според оваа анализа на сложеностите на најчестите операции кај бинарните дрва, може да се изведе заклучок дека доколу бинарното дрво не треба само да е пребарувачко, туку и да е балансирано, има најоптимална сложеност на операциите $O(\log_2 N)$. Токму затоа најупотребувани во пракса од бинарните дрва се пребарувачките и балансирани дрва кои во литературата се среќаваат под ново име, односно AVL дрва.

6.2.4 Балансирани бинарни дрва

Балансирано бинарно дрво е она бинарно дрво каде што за секое поддрво на произволен јазел важи дека висините не се разликуваат за повеќе од еден. На слика 6-21а, е претставено балансирано бинарно дрво. На слика 6-21б е прикажано небалансирано бинарно дрво кое е такво поради тоа што за посочениот јазел не важи дека висините на неговото лево и десно поддрво се еднакви или се разликуваат за еден. Во конкретниот случај висините на левото и десното поддрво на посочениот јазел се разликуваат за два [1].

Врска помеѓу бројот на јазли во бинарно дрво и неговата висина

Во балансирани дрва, висината на двете поддрва од секој јазол е приближно иста, што гарантира дека основните операции како пребарување, вметнување и



Слика 6-21: Балансирано бинарно дрво (а) и небалансирано бинарно дрво (б)

бришење ќе се извршуваат во логаритамско време. Во продолжение е описано како балансираноста на бројот на јазли во бинарното дрво и неговата висина резултираат со логаритамска ефикасност на многу од операциите..

Доколку бинарното дрво не е празно и е максимално пополнето, на ниво 0 тоа има еден јазел, на ниво 1, има два јазли, на ниво 2 има четири јазли или во општ случај, на ниво k има $2k - 1$ јазли. Доколку сите јазли се соберат добиваме дека за максимално пополнето бинарно дрво важи дека доколку има n јазли, тој број е еднаков на сумата $1 + 2^1 + 2^2 + \dots + 2^{d-1} = 2^d - 1$ каде d е длабочината на дрвото. Од тута може да заклучиме дека за секое дрво со n јазли важи $n \leq 2^{d-1}$.

Од ова следи дека минималната длабочина (висина) на бинарно дрво со n јазли (која се добива кога е тоа максимално пополнето) е: $d_{min} = \lceil \log_2(n+1) \rceil$

каде $\lceil A \rceil$ го означува најмалиот цел број поголем или еднаков на A . Од тута следи дека за $n = 20$, имаме $d_{min} = \lceil \log_2(21) \rceil = \lceil 4.39 \rceil = 5$; додека за $n = 1000$ се добива $d_{min} = \lceil \log_2(1001) \rceil = 10$. Ова е значајно затоа што најголемата висина на дрво со 1000 јазли е 1000 (во случај кога е секогаш активна само една врска).

Доколку претпоставиме дека дрвото како податочна структура се користи за складирање на некоја информација и под претпоставка дека бараната информација се наоѓа во листовите на дрвото (на пример при градење на индекс), од интерес ќе биде нивото на секој јазел во дрвото да биде еднаквао или приближно еднакво.

Примена на балансирани бинарни дрва

Балансираните бинарни дрва се важни бидејќи обезбедуваат ефикасен начин на пребарување, вметнување и бришење на елементи. Тие гарантираат логаритамско време на извршување на основните операции, што е значително побрзо во споредба со линеарните структури на податоци како низи или поврзани листи. Ова ги прави балансираните бинарни дрва одличен избор за разни алгоритамски проблеми и апликации, како што се:

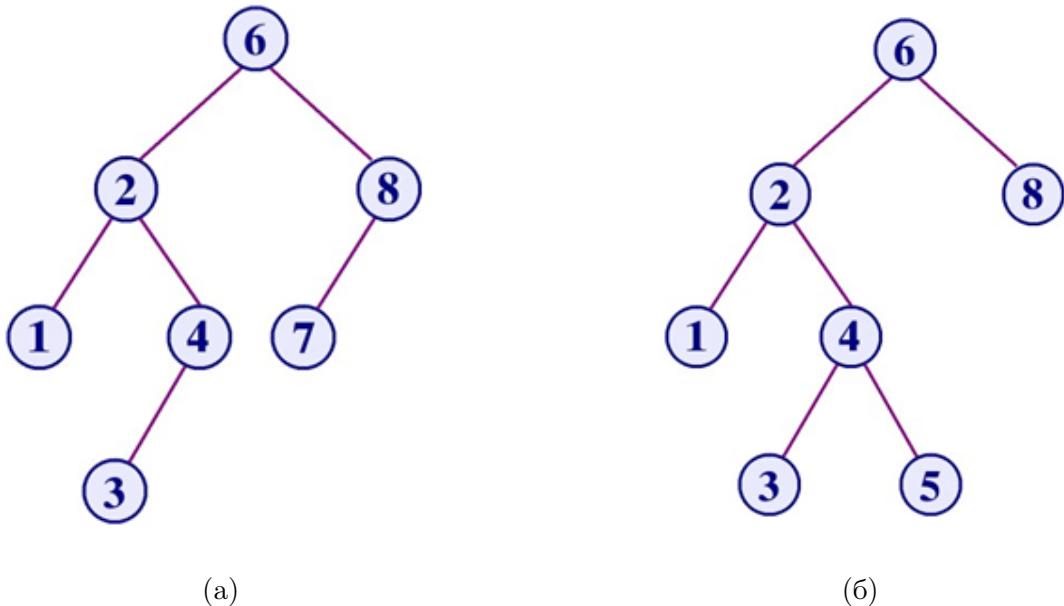
- **Бази на податоци:** Често се користат структури како B-дрва и B+ дрва за ефикасно чување и пребарување на записи.
- **Асоцијативни низи и мали:** Во програмски јазици како C++ и Java, структурите за податоци како `std::map` и `java.util.TreeMap` се базирани на балансираните бинарни дрва.
- **Приоритетни редици:** Алгоритми за наоѓање на најкраток пат како Dijkstra и A* користат приоритетни редици кои честопати се имплементирани со помош на балансираните бинарни дрва.
- **Статистички и научни анализи:** Во анализа на податоци, балансираните дрва можат да се користат за брзо наоѓање на медијана, перцентили и други статистички мерки.
- **Системи за пребарување:** Во системите за пребарување како Google и Bing, балансираните дрва може да се користат за брзо индексирање и пребарување на веб-страници.
- **Игри и симулации:** Во компјутерски игри, балансираните бинарни дрва се користат за чување на различни видови информации, како што се точки во простор или множества на можни позиции и стратегии.
- **Мрежни рутери и протоколи:** Во мрежни алгоритми и протоколи, балансираните дрва се користат за ефикасно управување со IP адреси и пакети.

За да резимираме, балансираните бинарни дрва, како AVL дрва за кои зборуваме во следната подглава, играат клучна улога во модерната компјутерска наука и се користат во широк спектар на апликации заради нивната ефикасност и универзалност.

6.2.5 AVL дрва

AVL (Adelson-Velskii & Landis) дрво е бинарно пребарувачко дрво кое е дополнително и балансирано дрво. Со тоа се осигурува дека длабочината на дрвото (а со тоа и комплексноста на најчестите операции) е од редот $O(\log_2 N)$. Разликата во висините на левото и десното поддрво на било кој јазел е позната како фактор на балансираност. На слика 6-22 е даден пример за AVL дрво (a) и бинарно

пребарувачко дрво (б). Дрвото на слика 6-22б не е AVL дрво, затоа што, иако е пребарувачко, факторот на балансираност на јазелот со клуч 6 (коренот) е 2.

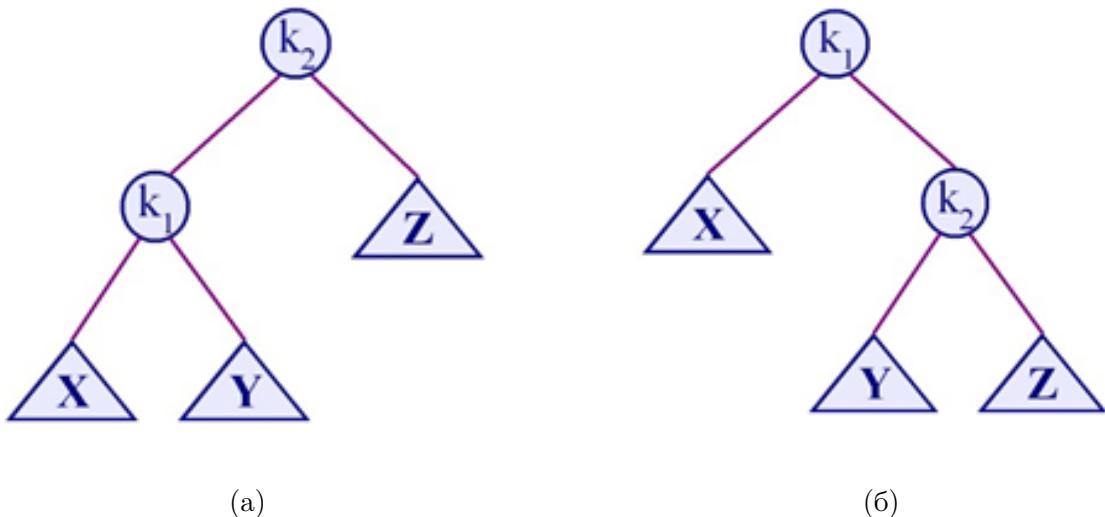


Слика 6-22: AVL дрво (а) и бинарно пребарувачко дрво (б)

Со користење на AVL дрвата се добиваат подобри перформанси. Реализацијата на овие дрва (како и во случајот на обичните пребарувачки дрва) повторно мора да се изврши програмски. Со ова се зголемува комплексноста на имплементацијата кај операциите на вметнување и бришење јазел. Поради природата на операциите, тие може да предизвикаат нарушување на балансираноста на дрвото, па во тој случај веднаш мора да се примени постапка за враќање на балансираноста на дрвото [1].

Двете дрва дадени на слика 6-23 се бинарни пребарувачки дрва доколку за нив важи дека $k_2 > k_1$. Трансформацијата со која се конвертира дрвото (а) во дрво (б) и обратно се нарекува еднократна ротација, и може да биде корисна доколку се примени за исполнување на условот за балансираност кај некоја операција со AVL дрвата.

Треба да се укаже дека еднократната ротација може да се изврши врз било кој јазел од дрвото. Изборот на јазел треба да биде оној во кој е утврдено нарушување на балансираноста. Постапката за проверка на балансираноста после некоја операција над даден јазел вообичаено почнува од тој јазел, при што се проверува факторот на балансираност и ако е потребно се балансира. Доколку балансираноста на тој јазел не е нарушена се проверува балансираноста на неговиот родител итн. Оваа постапка се повторува се додека не се дојде до коренот



Слика 6-23: Еднократна ротација на десно од (а) кон (б) и еднократна ротација на лево од (б) кон (а)

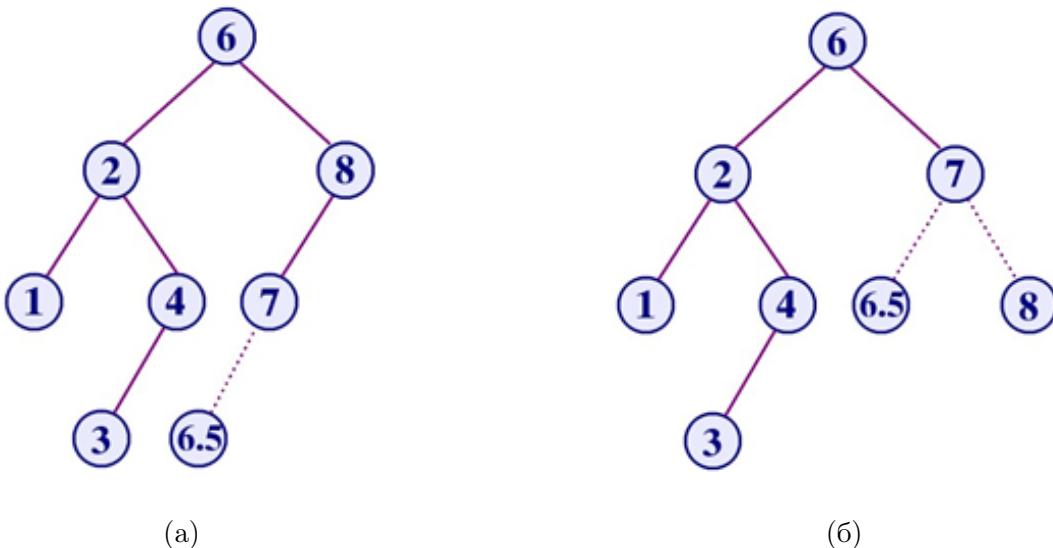
на дрвото.

Возможни се два типа еднократни ротации и тоа:

- еднократна ротација во десно
 - овој тип ротација позната е и како “left left case“ односно на лево дете од даден јазел k_2 , додадено е лево дете. Со тоа се нарушила балансираноста во k_2 , па потребно е да се направи ротација во десно за дрвото повторно да биде AVL. При еднократната ротација во десно, јазелот k_2 станува десно дете на неговото лево дете, јазелот k_1 , а десното дете на јазелот k_1 сега станува лево дете на јазелот k_2 . Види слика 9.9, каде на дрвото под (а) се применува ротација во десно за да се добие дрвото под (б).
- еднократна ротација во лево
 - овој тип ротација позната е и како “right right case“ односно на десно дете од даден јазел k_1 , додадено е десно дете. Со тоа се нарушила балансираноста во k_1 , па потребно е да се направи ротација во лево за дрвото повторно да биде AVL. При еднократната ротација во лево, јазелот k_1 станува лево дете на неговото десно дете, јазелот k_2 , а левото дете на јазелот k_2 сега станува десно дете на јазелот k_1 . Види слика 9.9, каде на дрвото под (б) се применува ротација во лево за да се добие дрвото под (а).

Ротациите најлесно може да се разберат преку примери, затоа во продолжение

Ќе дадеме неколку од нив. На примерот даден на слика 6-24 по вметувањето на јазелот 6.5, нарушен е балансираноста на јазелот 8 и таа се корегира со еднократна ротација во десно.



Слика 6-24: Пример за балансирање на дрво со помош на еднократна ротација во десно

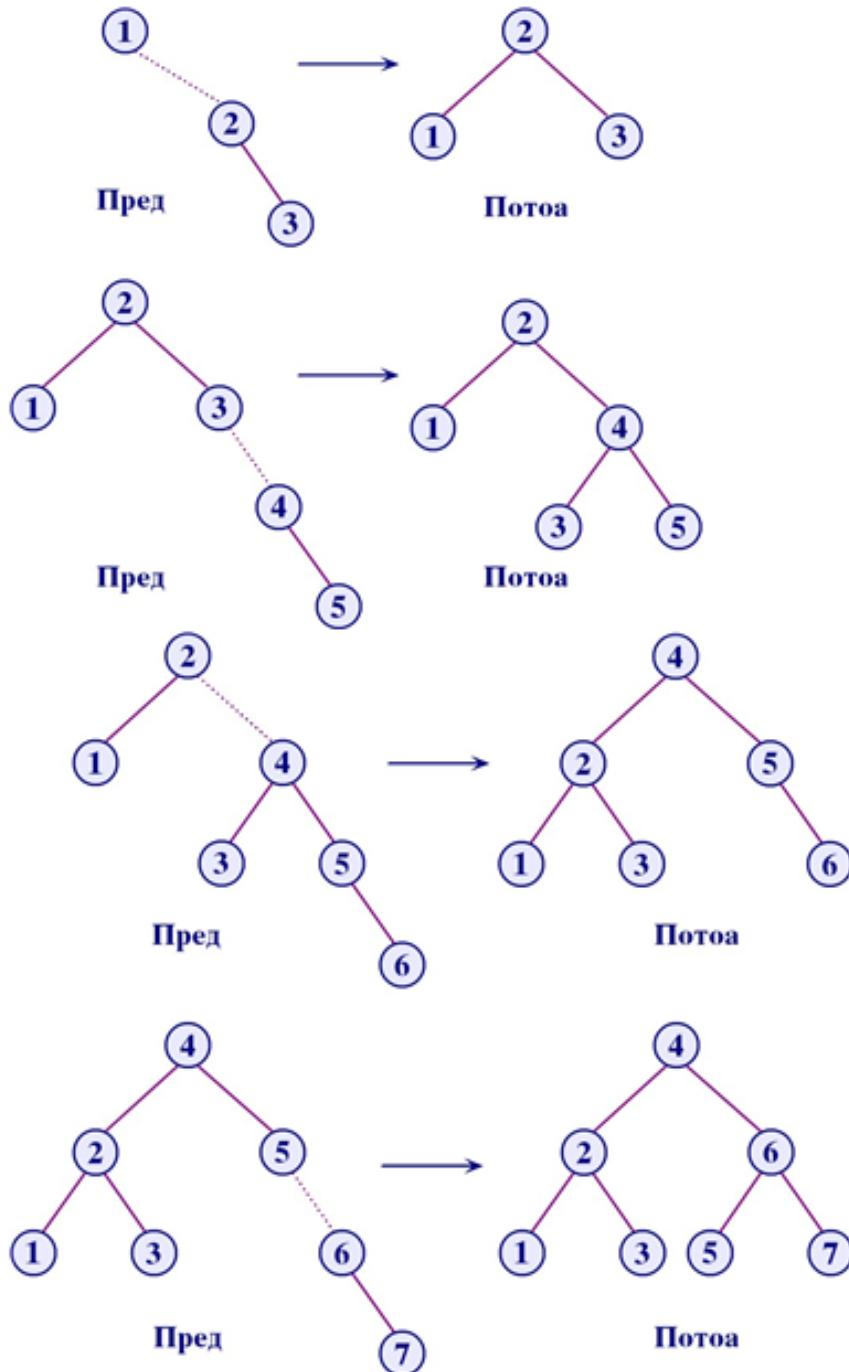
Во следниот пример (види слика 6-25) е покажана постапката на креирање на AVL дрво кога во него се внесуваат последователно клучеви со вредности: 1, 2, 3, 4, 5, 6 и 7. При тоа по потреба се вршат еднократни ротации во лево. Јазлите кај кои е утврдена нарушеност на балансираноста имаат една врска означена со испрекината линија.

Предходниот алгоритам односно едноставна ротација во десно или лево не дава секогаш задоволителни резултати. За илустрација да го продолжиме вметнувањето јазли во предходното AVL дрво со клучевите 15 и 14 (види слика 6-26).

Вметнувањето на клучот 14 предизвикува нарушување на балансираноста, но со примена на еднократна ротација во лево таа не се разрешува. Тоа е поради тоа што ново внесениот клуч (14) припаѓа на поддрвото Y од слика 6-23а. Тоа се елементи кои се вметнати како поголеми елементи од некое лево поддрво или помали елементи од некое десно поддрво (елементи кои визуелно одат кон средината на поддрвото). Во тој случај треба да се примени двојна или двократна ротација.

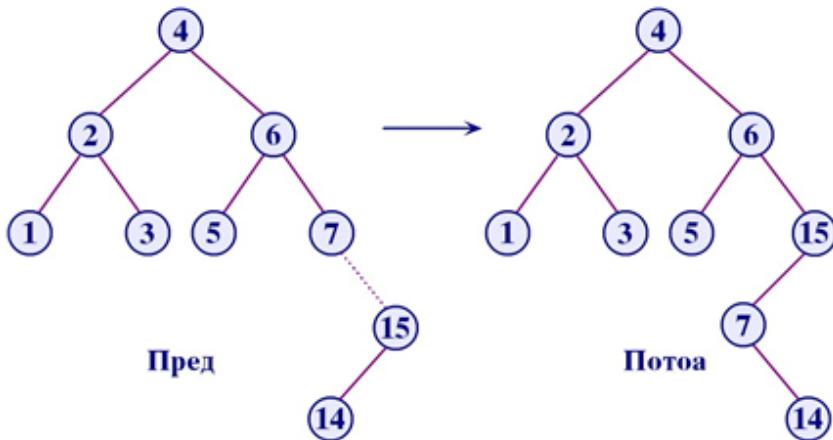
Двократната ротација е графички описана на слика 6-27 и слика 6-28. Постојат два симетрични типа на двократна ротација и тоа:

- двократна ротација десно-лево



Слика 6-25: Пример за балансирање на дрво со помош на еднократна ротација во лево при додавање на јазлите со клучеви 1, 2, 3, 4, 5, 6 и 7 последователно

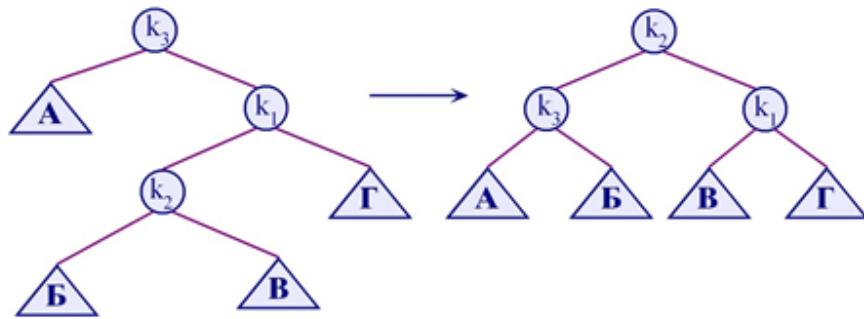
- овој тип ротација е позната и како “right left case“, односно на десно дете од даден јазел k_3 додадено е лево дете. Со тоа се нарушила балансираноста во јазелот k_3 , па потребно е да се направи двократ-



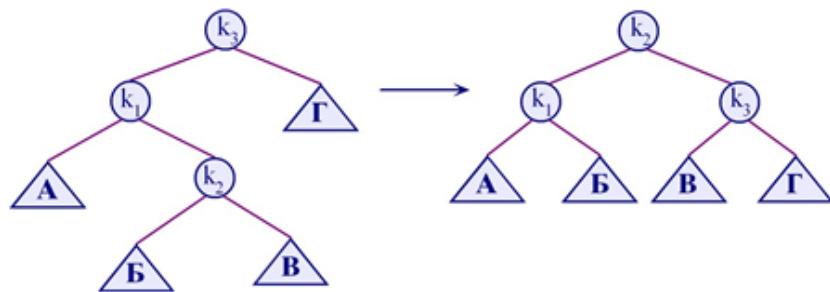
Слика 6-26: Пример кога еднократната ротација не помага во балансирање на дрвото

на ротација за дрвото повторно да биде AVL. Двократната ротација десно-лево едноставно е комбинација од еднократна ротација во десно, проследена со еднократна ротација во лево. Притоа јазелот k_2 станува десно дете на јазелот k_3 , додека пак k_1 станува неговото десно дете. При оваа прва ротација повторно балансираноста не е задоволена, па продолжуваме со ротацијата во лево. Понатака, јазелот k_3 станува лево дете на јазелот k_2 , а јазелот k_1 останува неговото десно дете. Види 6-27, каде на левото дрво на сликата се применува двократна ротација десно-лево за да се добие балансирано дрво на десната страна на сликата.

- двократна ротација лево-десно
 - овој тип ротација е позната и како “left right case“, односно на лево дете од даден јазел k_3 додадено е десно дете. Со тоа се нарушила балансираноста во јазелот k_3 , па потребно е да се направи двократна ротација за дрвото повторно да биде AVL. Двократната ротација лево-десно едноставно е комбинација од еднократна ротација во лево, проследена со еднократна ротација во десно. Притоа јазелот k_2 станува лево дете на јазелот k_3 , додека пак k_1 станува негово лево дете. При оваа прва ротација повторно балансираноста не е задоволена, па продолжуваме со ротацијата во десно. Понатака, јазелот k_3 станува десно дете на јазелот k_2 , а јазелот k_1 останува неговото лево дете. Види слика 6-28, каде на дрвото на левата страна на сликата се применува двократна ротација лево-десно за да се добие балансираното дрво на десната страна на сликата.

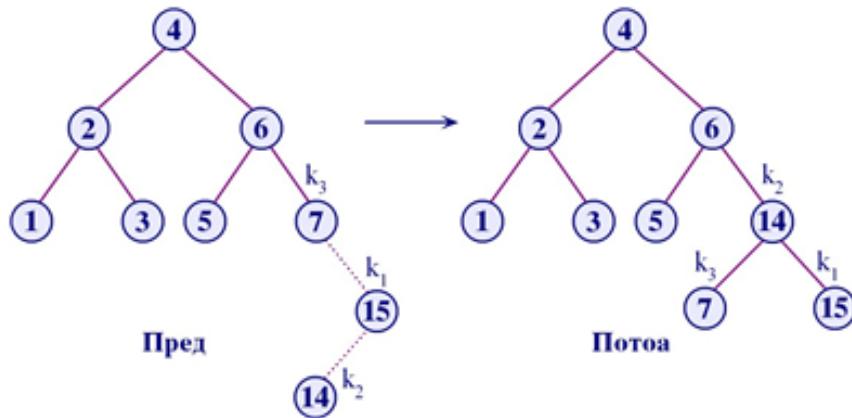


Слика 6-27: Двократна ротација десно-лево



Слика 6-28: Двократна ротација лево-десно

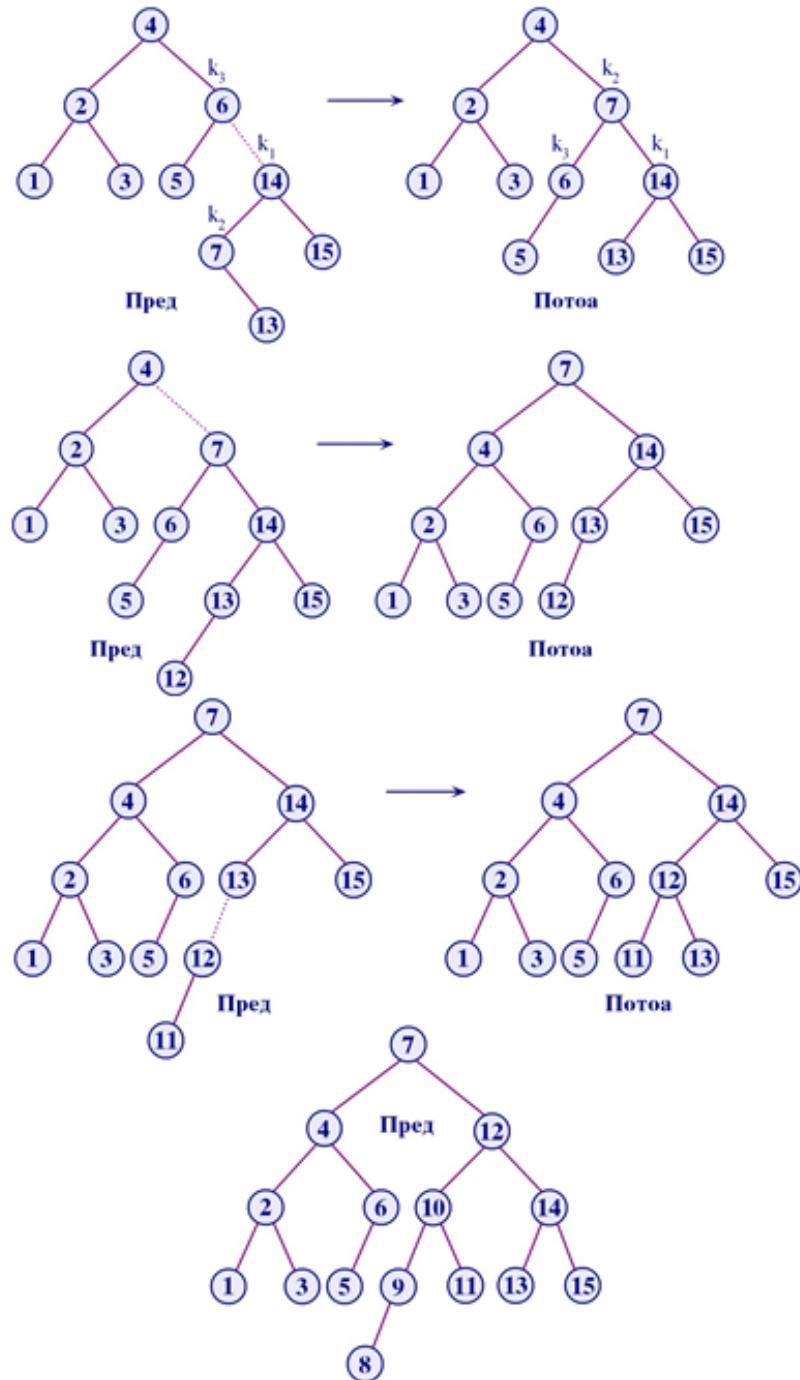
Решението на примерот со вметнување на клучевите 15 и 14 за балансирање на AVL дрвото со употреба на двократна ротација десно-лево, дадено е на слика 6-29.



Слика 6-29: Двократна ротација лево-десно

Да продолжиме со примерот внесувајќи ги последователно клучевите 13, 12, 11, 10, 9 и 8. Дел од овие клучеви ќе предизвикаат нарушување на балансираноста

кое треба да се реши со двојна ротација (за клучот 13), еднократна ротација (за клучевите 12, 11, 10 и 9), додека внесувањето на клучот 8 не ја нарушува балансираноста на дрвото. Постапката на градење на дрвото е дадена на слика 6-30.



Слика 6-30: Постапка на градење на AVL дрво

Бришењето јазли во AVL дрва се сведува на операцијата бришење како кај бинарните пребарувачки дрва (заедно со трите потслушаи), со тоа што сега после секое бришење на јазел мора да се применат некои од ротациите соодветно за дрвото да остане балансирано.

Кога зборуваме за AVL дрвата може да кажеме дека тие се само-балансирачки бинарни пребарувачки дрва каде најчестите операции како што се пребарување, вметнување и бришење се изведуваат со сложеност $O(\log N)$. Тоа е така поради фактот што операциите на ротации, како и одредувањето на факторот на балансираност на даден јазел се изведуваат во константно време. Сепак, AVL дрвата најголема практична примена имаат во сценарија каде операцијата пребарување е најфреќвентната операција, наспроти операциите вметнување и бришење кои се помалку барани. За ситуации каде операциите на вметнување и бришење се многу почести операции, се избегнува користење на AVL дрва, а на нивно место се земаат Red-Black дрвата. Red-Black дрвата нема детално да ги опишуваме во оваа глава. Остануваат тие да се истражат и проучат самоиницијативно од страна на читателот.

Претставување на AVL дрва со Java код

При претставувањето на AVL дрвата со код во најголема мера ќе се послужиме со кодот за бинарните пребарувачки дрва, со тоа што ќе додадеме атрибути и методи во соодветните класи. За да ги претставиме јазлите на AVL дрвото, од голема важност ни е да ја додадеме како атрибут висината на соодветниот јазел, со што ќе може лесно и во константно време да го определуваме факторот на балансираност. Во продолжение е даден кодот:

```

1 // Define an AVLNode class, extending TreeNode
2 public class AVLNode<T extends Comparable<T>> extends TreeNode<T>{
3     int height; // Additional field for AVLNode
4
5     public AVLNode(T data) {
6         super(data);
7         height = 1; // Initialize height as 1
8     }
9 }
```

Исто така потребно ни е да креираме и класа која ќе го опишува AVL дрвото заедно со сите јазли кои ги содржи. Оваа класа е иста како и класата `BinaryTree` со тоа што сега се грижиме плус за балансираноста на јазлите. Во прилог е кодот за неа:

```
1 // Define an AVLTree class, which extends the BinarySearchTree class.
```

```

2 // This class takes a generic type T that extends Comparable interface,
3 // meaning T can be compared.
4
5     // Function to get the height of a node.
6     // If the node is null, its height is 0.
7     int height(AVLNode<T> node) {
8         return (node == null) ? 0 : node.height;
9     }
10
11    // Function to insert a new node while keeping the tree balanced.
12    public AVLNode<T> insert(AVLNode<T> node, T data) {
13        // Standard BST insert operation.
14        if (node == null) {
15            return new AVLNode<T>(data);
16        }
17
18        // Insert in the left or right subtree based on the value of 'data'.
19        if (data.compareTo(node.data) < 0) {
20            // Type-casting is needed because node.left is of type
21            // BinaryNode,
22            // but we know it is actually AVLNode in this context.
23            node.left = insert((AVLNode<T>) node.left, data);
24        } else if (data.compareTo(node.data) > 0) {
25            node.right = insert((AVLNode<T>) node.right, data);
26        } else {
27            return node; // Duplicates not allowed
28        }
29
30        // Update the height of the current node.
31        node.height = 1 + Math.max(height((AVLNode<T>) node.left),
32                                    height((AVLNode<T>) node.right));
33
34        // Compute the balance factor to check for imbalance.
35        int balance = height((AVLNode<T>) node.left) - height((AVLNode<T>)
36                                            node.right);
37
38        // If left subtree is heavier
39        if (balance > 1) {
40            // If the data belongs to the left of left child, it's a
41            // straight left-left case.

```

```

38     if (data.compareTo(node.left.data) < 0) {
39         return rotateRight(node); // Single right rotation
40     } else {
41         // This is a left-right case.
42         node.left = rotateLeft((AVLNode<T>) node.left); // First
43             rotate the left child
44         return rotateRight(node); // Then rotate the unbalanced node
45     }
46 }
47
48 // If right subtree is heavier
49 if (balance < -1) {
50     // If the data belongs to the right of right child, it's a
51     // straight right-right case.
52     if (data.compareTo(node.right.data) > 0) {
53         return rotateLeft(node); // Single left rotation
54     } else {
55         // This is a right-left case.
56         node.right = rotateRight((AVLNode<T>) node.right); // First
57             rotate the right child
58         return rotateLeft(node); // Then rotate the unbalanced node
59     }
60 }
61
62 return node; // Return the (now balanced) node pointer
63 }
64
65 // Function to perform right rotation
66 // It handles the rotation and also updates the height.
67 private AVLNode<T> rotateRight(AVLNode<T> y) {
68     AVLNode<T> x = (AVLNode<T>) y.left;
69     AVLNode<T> T2 = (AVLNode<T>) x.right;
70
71     // Perform rotation
72     x.right = y;
73     y.left = T2;
74
75     // Update heights
76     y.height = Math.max(height((AVLNode<T>) y.left), height((AVLNode<T>)
77         y.right)) + 1;
78     x.height = Math.max(height((AVLNode<T>) x.left), height((AVLNode<T>)
79         x.right)) + 1;
80 }
81
82 }
83
84
85
86
87
88
89
90
91
92
93
94

```

```

    x.right)) + 1;

75
76     // Return new root
77     return x;
78 }
79
80     // Function to perform left rotation
81     // It handles the rotation and also updates the height.
82     private AVLNode<T> rotateLeft(AVLNode<T> x) {
83         AVLNode<T> y = (AVLNode<T>) x.right;
84         AVLNode<T> T2 = (AVLNode<T>) y.left;
85
86         // Perform rotation
87         y.left = x;
88         x.right = T2;
89
90         // Update heights
91         x.height = Math.max(height((AVLNode<T>) x.left), height((AVLNode<T>)
92             x.right)) + 1;
93         y.height = Math.max(height((AVLNode<T>) y.left), height((AVLNode<T>)
94             y.right)) + 1;
95
96         // Return new root
97         return y;
98     }
99 }
```

6.2.6 В дрва

Постојат и повеќе типови на небинарни пребарувачки дрва.

Во рамките на овој курс ќе бидат обработени само В-дрвата и накратко ќе бидат објаснети разликите повеќу нив и В+ дрвата. Останатите (B*, R итн) претставуваат подобрување на В дрвата но, во суштина се засноваат на истата идеја [3].

В-дрва се вид на балансирана структура на податоци кои се особено корисни за складирање и ракување со големи количества на податоци кои не можат целосно да се сместат во главната меморија. Тие се користат во бази на податоци, системи за датотеки и други системи каде што податоците се складирани на надворешни уреди за складирање како тврди дискови. Еве зошто В-дрва се важни во овие контексти:

- **Ефикасно запишување и читање од диск.** В-дрвата се дизајнирани да ги минимизираат бројот на читања и запишувања на диск. За разлика од структури на податоци како што се поврзани листи или низи, каде можеби треба да се читаат повеќе блокови на податоци за да се најде одреден елемент, В-дрвата овозможуваат поприфатлива употреба на диск операции. Секој јазол во В-дрвото може да има многу деца, овозможувајќи на дрвото да содржи повеќе клучеви по јазол. Оваа карактеристика ја минимизира височината на дрвото и со тоа го намалува бројот на пристапи до дискот потребни за повеќето операции.
- **Динамична природа.** В-дрвата се динамични, што значи дека ефикасно ја одржуваат својата балансирана структура додека се додаваат или отстрануваат податоци. Овој баланс е клучен за одржување на конзистентни и ефикасни перформанси при читање и запишување. Дрвото се прилагодува автоматски додека се вметнуваат или бришат елементи, па не мора да се води рачно грижа за ребалансирање на дрвото.
- **Поддршка на различни операции.** В-дрвата поддржуваат повеќе видови на операции вклучувајќи пребарување, секвенцијален пристап, вметнувања и бришења, сите во логаритамска временска сложеност. Структурата исто така ефикасно поддржува прашања кои бараат податоци во некој опсег, како и сортирано изминување на елементите.
- **Прилагодливост.** В-дрвата се лесно прилагодливи за да служат на различни типови на податоци и операции. Тие може да се прилагодат за работа со големи множества на податоци, различни типови на дискови системи и различни модели на пристап.
- **Подредени податоци.** Една предност на В-дрва над хеш-базирани структури е дека тие ги чуваат клучевите во сортиран редослед.
- **Ефикасно користење на просторот.** Секој јазол обично одговара на страница на дискот, па со одржувањето на јазлите да бидат полупополнети, дрвото може рамномерно да ги дистрибуира податоците на диск.

Дефиниција и примери

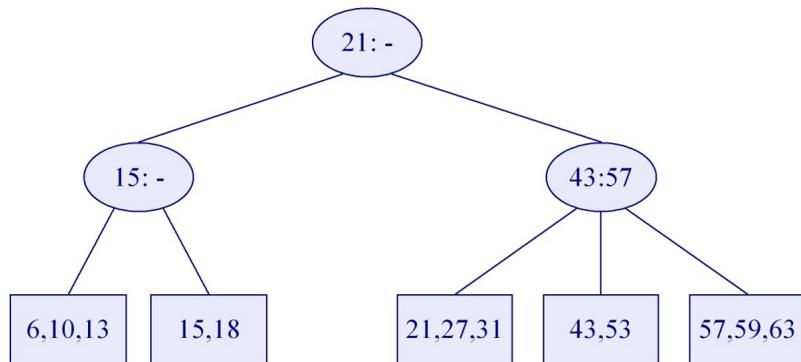
В-дрво од ред m е дрво со следните структурни својства:

- Коренот е или лист или има помеѓу 2 и m деца.
- Сите внатрешни јазли (освен коренот) имаат помеѓу $\lceil m/2 \rceil$ и m деца.
- Сите листови се на исто ниво.

Сите податоци се сместени во листовите. Внатрешните јазли содржат покажувачи кон децата (p_1, p_2, \dots, p_m) и вредности - клучеви $(k_1, k_2, \dots, k_{m-1})$ кои ги означуваат најмалите клучеви кои постојат во поддрвата одредени со p_2, p_3, \dots, p_m

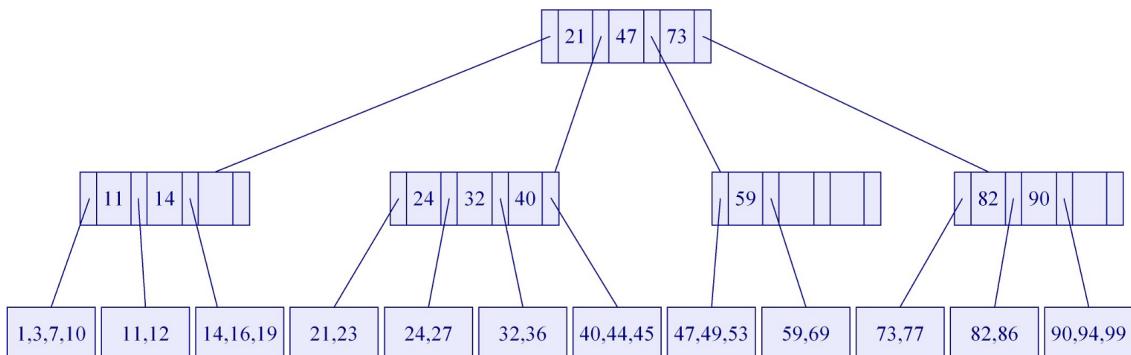
соодветно. Доколку не постојат вредности за клучевите во внатрешните јазли, соодветните покажувачи се нулеви. Нека (иако тоа не мора да биде случај) бројот на информации и во листовите се движи во опсегот од $\lceil m/2 \rceil$ и m .

На слика 6-31 е дадено В дрво од ред 3 (секој јазел освен коренот мора да има 2 или 3 покажувачи), каде што нетерминалните јазли се означени со елипси, додека терминалните јазли со квадрати. Нетерминалните јазли содржат еден или два клучка.



Слика 6-31: В дрво од ред 3

На слика 6-32 е прикажано В дрво од ред 4.



Слика 6-32: В дрво од ред 4

И во двата случаи на дрва прикажани на слика 6-31 и слика 6-32 се забележува дека информациите во листовите се подредени.

Пребарување

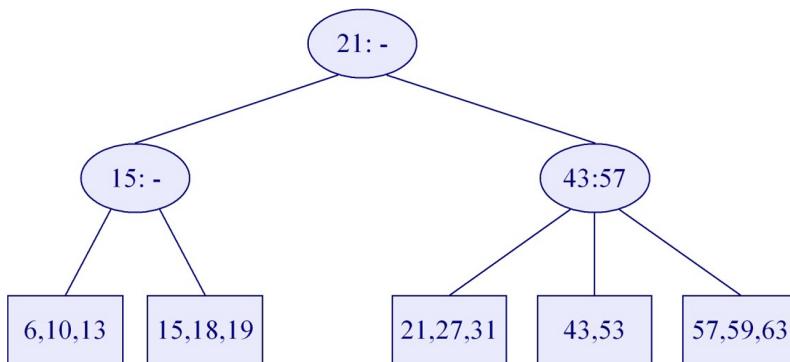
Операцијата на пребарување е релативно едноставна. Се поаѓа од коренот и се движиме по нетерминалните јазли споредувајќи ја вредноста на клучот што го бараме, на начин многу сличен со движењето по бинарните пребарувачки дрва.

(одиме по покажувачот лево или десно од клучот што го споредуваме со бараниот клуч, по правилото „сите клучеви на лево се помали“). Еднаш кога ќе дојдеме до лист се движиме секвенцијално (како во низа). Така, во споредба со бинарните дрва, во многу помалку чекори се доаѓа до бараниот јазол (или се заклучува дека таков јазол не постои).

Внесување на вредност

Внесувањето на вредност на клуч се сведува на внесување на вредност на клучот во еден терминален јазел и евентуална промена на клучевите во нетерминалните јазли. Да го разгледаме тој процес, на пример во кој на предходното В дрво прикажано на слика 6-31, ќе ги внесеме клучевите: 19, 2, 20 и 29.

Внесувањето на клучот со вредност 19, во првиот чекор одредува дека тој треба да се смести во непополнет краен јазел. Тогаш имаме едноставен случај на негово внесување на правото место (види слика 6-33).

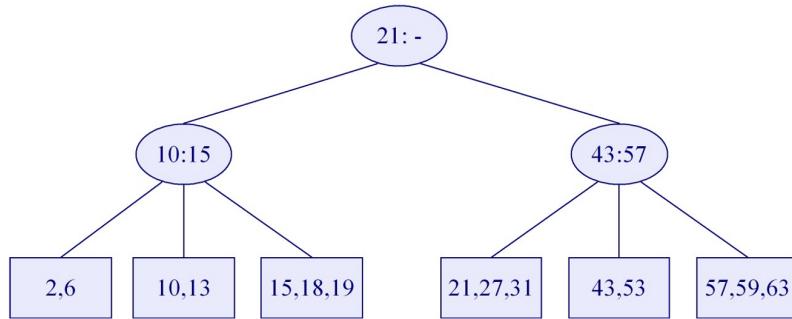


Слика 6-33: Внесување на клуч во непополнет јазел од В дрво од ред 3

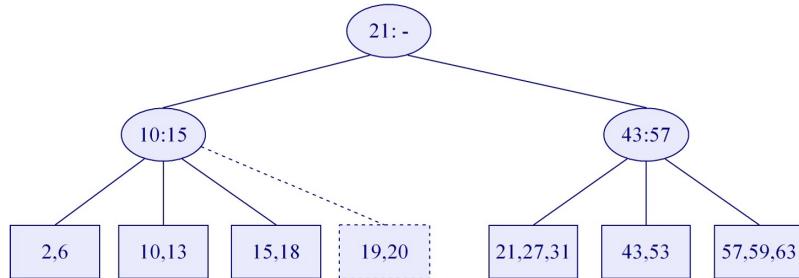
Внесувањето на клучот со вредност 2 открива дека тој треба да се смести во терминален јазел кој е веќе полн. Ставањето на уште еден клуч во полн јазел би предизвикало креирање на јазел со 4 елементи што не е дозволено со дефиницијата на В дрвото. Во конкретниот случај така добиениот јазел со 4 елементи може да го разделиме во два јазла со по 2 елементи, при што ќе ја ажурираме вредноста на клучевите и покажувачите во родителскиот јазел. Со тоа доаѓаме во ситуација претставена на слика 6-34. Оваа постапка може да се повтори и по останатите родители, доколку за тоа има потреба.

Доколку на сличен начин сакаме да го внесеме и клучот 20 ќе предизвикаме недозволена ситуација прикажана на слика 6-35. Иако терминалните јазли не се пополнети, нивниот родител содржи повеќе од дозволениот број на покажувачи.

Во тој случај внатрешниот јазел кој е „преоптоварен“ се дели на два јазли со по две деца (види слика 6-36). Тоа може да предизвика рекурзивно ажурирање

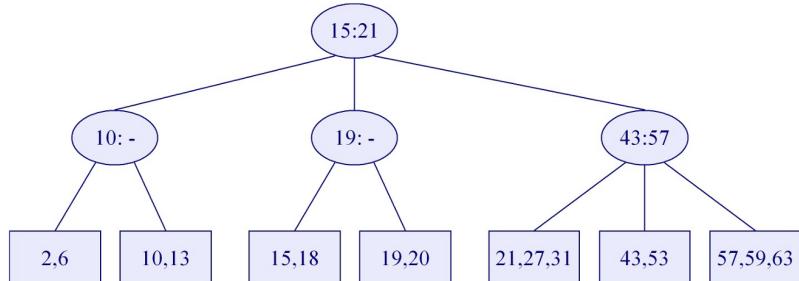


Слика 6-34: Внесување на клуч во пополнет јазел од В дрво од ред 3 со помош на делење на терминален јазел



Слика 6-35: Недозволена состојба за В дрво од ред 3

на неговиот родител и така сè до коренот.



Слика 6-36: Делење на внатрешен јазел кај В дрво од ред 3

Итеративното креирање на нови внатрешни јазли се добива при внесувањето на клучот 29. Во првата итерација родителот на листот се дели на два нови внатрешни јазли. Со тоа се добиваат четири јазли на ниво 1, односно од коренот треба да излегуваат четири покажувачи што не е дозволено. Тоа предизвикува делење на коренот на два нови јазли, кои пак се обединуваат со воведување на нов корен со што се зголемува вкупната висина на дрвото. Целата постапка на делење е претставена на слика 6-37.

Доколку сакаме да го спречиме брзото „растење“ на дрвото може да разгледа-

ме стратегии кои ќе проверат дали со прераспределба на клучевите во листовите (и ажурирање на клучевите на внатрешните јазли) може да се спречи растењето на дрвото. Така, на пример, доколку сакаме да внесеме клуч со вредност 70, наместо да воведеме нов терминален јазел, може да провериме дали соседниот терминален јазел има место (а во случајов има) и да извршиме поместување на клучот 57 во соседниот терминален јазел (на лево) со што ни се ослободува место во крајниот десен терминален јазел во кој можеме да го сместиме клучот 70. Притоа, очигледно, треба да направиме ажурирање на вредноста на клучот и кај родителот на двата јазли (57 да стане 59) што може каскадно да предизвика и други ажурирања во внатрешноста на дрвото. Очигледно, на овој начин запштедуваме мемориски простор, но ја правиме операцијата вметнување на клуч многу посложена.

Бришење

Операцијата на бришење се сведува на наоѓање на клучот и негово отстранување. Доколку со тоа отстранување терминалниот јазел остане непополнет (во нашиот пример само со еден клуч) можеме да извршиме спојување со соседен терминален јазел (брат) при што има два случаја: да се добие јазел со три или четири елементи. Во вториот случај, слично на предходно описаното, јазелот со четири клуча се дели на два јазла со по два клуча. Во првиот случај, бидејќи родителот губи едно дете тоа може да предизвика негова непополнетост, па постапаката итеративно се повторува одејќи нагоре низ дрвото. Во одреден случај може да се јави потреба од намалување на висината на дрвото. Примената на В дрвата (или т патните пребарувачки дрва) главно се наоѓа во градењето на индексите кај системите за управување со бази на податоци.

6.2.7 B+ дрва

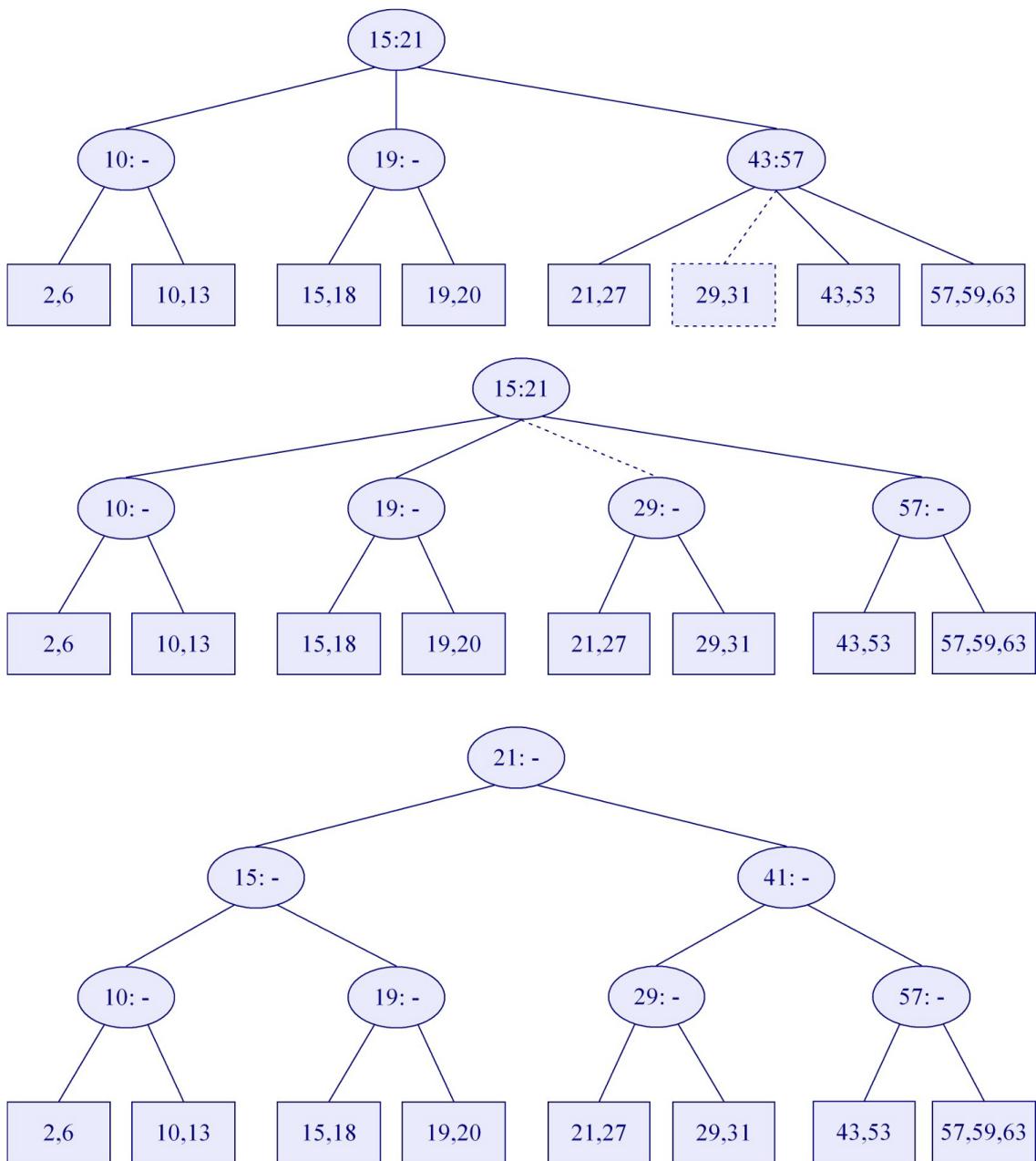
На слика 6-38 е даден пример на B+ дрво. На сликата со d_1, d_2, \dots, d_7 се означени покажувачи кон целите објекти чии клучеви се наоѓаат низ дрвото. На пример, ако се чуваат објекти кои ги опишуваат сутдентите на ФИНКИ, тие би имале повеќе својства (пр. име, презиме, итн.), но само еден клуч – број на индекс. Во тој случај, само клучот ќе се среќава низ целото дрво, додека во листовите ќе има покажувач до целиот објект за студентот.

B+ дрвата се многу слични на В дрвата, со тоа што сепак има неколку разлики дадени во табела 6.3.

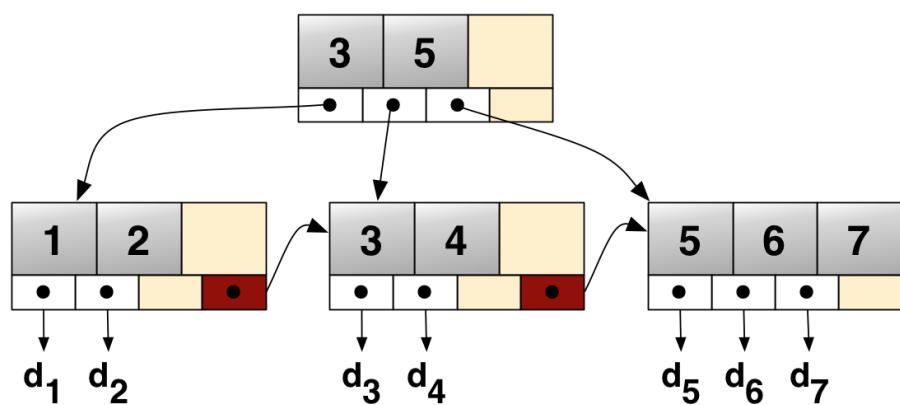
Други интерактивни примери за операции со В дрва и B+ дрва има на <https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BTree.html> и <https://www.javatpoint.com/b-tree-visualization>.

Критериум	B-дрва	B+ дрва
Структура	Клучевите и податоците може да се сместат во и внатрешните јазли и во листовите.	Клучевите се појавуваат само еднаш, а покажува-чите кон податоците се сместени само во листо-вите. Внатрешните јазли содржат само клучеви за да го водат пребарување-то.
Пребарување	Пребарувањето може да заврши во внатрешен ја-зол или во листови јазол.	Пребарувањето секогаш поминува од коренот до некој лист.
Бришење	Посложено поради тоа што клучевите може да бидат на било кое ниво.	Поедноставно бидејќи сите податоци се во листовите.
Операции на диск	Може да изведат повеќе операции на читање од диск за пребарување бидејќи податоците може да бидат на било кој ниво.	Потребни се помалку опе-рации на читање од диск, бидејќи сите податоци се во листовите.
Просторна ефикасност	Може да бидат помалку просторсно ефикасни бидејќи внатрешните јазли исто така може да содржат податоци.	Повеќе клучеви може да се сместат во секој внатре-штен јазол, што може да ре-зултира со помала висина на дрвото.
Пребарување во опсег	Не се оптимизирани за пребарување во опсег.	Добро се оптимизирани за пребарување во опсег; лис-товите често се поврзани.

Табела 6.3: Споредба помеѓу B дрва и B+ дрва



Слика 6-37: Делење на внатрешен јазел кај В дрво од ред 3 со што се предизвикува зголемување на висината на дрвото



Слика 6-38: Пример за B+ дрво

6.2.8 Задачи

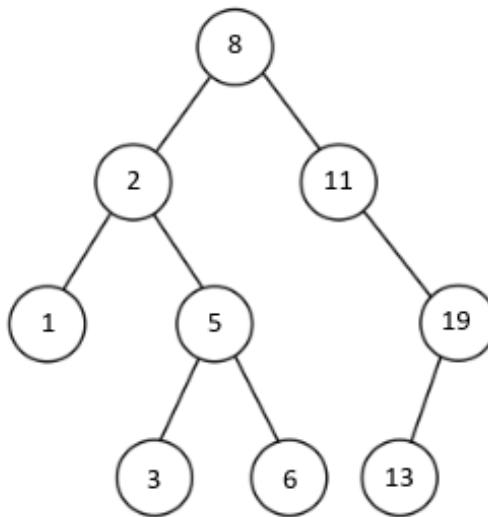
Задача 1. Балансирано дрво

Да се имплементира функција која ќе проверува дали дадено бинарно пребарувачко дрво е балансирано или не.

Пример:

Влез:

Бинарно дрво од слика 6-39



Слика 6-39: Пример на бинарно пребарувачко дрво

Излез:

The given binary search tree is also balanced.

За едно дрво викаме дека е балансирано ако висините на левите и десните поддрва на било кој негов јазел се разликуваат најмногу за 1, а и левото и десното поддрво сами по себе се балансирани. Според тоа прво треба да креираме метода `height(TreeNode<T>)` која ќе ни пресметува висина на јазел во дрвото (onoј што се задава како параметар), а тоа се добива како најдолгата патека од јазелот до неговите листови (лево и десно). Кодот за оваа метода е даден во продолжение:

```

1 // This method calculates the height (maximum depth) of a binary tree
2   rooted at 'node'.
3
4 public int height(TreeNode<T> node) {
5     // Base case: If the current node is null, the height is 0.
6     if (node == null) {
7         return 0;
8     }
9
10    int leftHeight = height(node.left);
11    int rightHeight = height(node.right);
12
13    // The height of the tree is the maximum of the heights of its children plus 1.
14    return Math.max(leftHeight, rightHeight) + 1;
15 }
```

```

6         }
7
8     // Recursively calculate the height of the left and right subtrees,
9     // and return the maximum height plus 1 (to account for the current
10    // node).
11    return 1 + Math.max(height(node.left), height(node.right));
12

```

Откако ја имаме функцијата која пресметува висина на јазел, лесно може да се имплементира методата која ќе одредува дали дадено дрво е балансирано или не. Според тоа за секој јазел од дрвото треба да се пресметаат висините на неговото лево и десно поддрво и да се определи нивната разлика со тоа што не треба да надминува 1. Доколку ова е исполнето за секој јазел во дрвото, следува дека тоа е балансирано. Во продолжение е даден кодот за оваа метода:

```

1  // This method checks whether a binary tree rooted at 'node' is balanced.
2  public boolean isBalanced(TreeNode<T> node) {
3      int left_h, right_h;
4
5      // Base case: If the current node is null, it's considered balanced.
6      if (node == null) {
7          return true;
8      }
9
10     // Calculate the heights of the left and right subtrees.
11     left_h = height(node.left);
12     right_h = height(node.right);
13
14     // Check if the difference in heights of left and right subtrees is
15     // at most 1,
16     // and both left and right subtrees are themselves balanced.
17     if (Math.abs(left_h - right_h) <= 1 && isBalanced(node.left) &&
18         isBalanced(node.right)) {
19         return true; // The tree rooted at 'node' is balanced.
20     }
21
22     return false; // The tree rooted at 'node' is not balanced.
23

```

Претходните две методи се дел од класата за бинарно пребарувачко дрво `BinarySearchTree`. Тестирањето на ново креираната метода ќе го извршиме во заедничката класа за тестирање од бинарните дрва `BinaryTreeTest`.

```

1  public class BinarySearchTreeTest {
2      // This method serves as an example for testing the 'isBalanced' method
3      // on a binary search tree (BST).
4      public static void exampleIsBalanced() {
5          // Create an example integer binary search tree (BST) and keep a
6          // reference in intTree.
7          BinarySearchTree<Integer> intTree = GetExampleBSTree();
8
9          // Print the contents of the binary search tree (BST).
10         intTree.print();
11
12         // Check if the binary search tree (BST) is balanced using the
13         // 'isBalanced' method.
14         if (intTree.isBalanced(intTree.root)) {
15             System.out.println("The given binary search tree is also
16                 balanced.");
17         } else {
18             System.out.println("The given binary search tree is NOT
19                 balanced.");
20         }
21     }
22 }
```

Задача 2. К-најмал или најголем елемент

Да се напише функција која ќе го најде k-от најмал/најголем елемент во бинарно пребарувачко дрво. Вредноста за k се внесува од тастатура.

Пример:

Влез:

Бинарно дрво од слика 6-39

Вредност за k: 4

Излез:

The 4-th smallest element in BST is 5

The 4-th biggest element in BST is 8

Идејата за имплементација на оваа функција е следна: Направи инордер из-

минување на дрвото, со што вредностите на јазлите ќе ги додаваш во една дополнителна динамичка низа. На крај од изминувањето добиваме низа во која елементите се сортирани во растечки редослед. Задачата бара да се испечати k -от најмал, односно најголем елемент од дрвото, па според тоа со пристапување на k -от елемент од низата од напред односно од назад ние ќе ги исполниме барањата на задачата.

Проблем за размислување: Пробај да го решиш дадениот проблем без користење на дополнителна меморија, односно без користење на динамичка низа.

Комплетниот код за оваа задача е даден во прилог како дел од класата `BinarySearchTree`.

```

1 // Private function for performing an inorder traversal and store the
2 // data in a list
3 private void inorder(TreeNode<T> node, List<T> sortedInorder) {
4     // Base case: If the current node is not null.
5     if (node != null) {
6         // Recursively traverse the left subtree.
7         inorder(node.left, sortedInorder);
8
9         // Add the data of the current node to the 'sortedInorder' list.
10        sortedInorder.add(node.data);
11
12        // Recursively traverse the right subtree.
13        inorder(node.right, sortedInorder);
14    }
15
16 // Public function to perform an inorder traversal starting from the
17 // root and store the result in a list.
18 public void inorder(List<T> sortedInorder) {
19     // Start the inorder traversal from the root of the tree.
20     inorder(root, sortedInorder);
}

```

Тестирањето на ново креираната метода ќе го извршиме во заедничката класа за тестирање од бинарните дрва `BinaryTreeTest`.

```

1 public class BinaryTreeTest {
2     // This method serves as an example for finding the kth smallest and kth
3     // largest elements in a binary search tree (BST).
4     public static void exampleKthSmallest() {
5         // Create an example integer binary search tree (BST) and keep a
6         // reference in intTree.

```

```

5      BinarySearchTree<Integer> intTree = GetExampleBSTree();
6
7      // Create a Scanner to get user input.
8      Scanner input = new Scanner(System.in);
9
10     // Read the user's input as an integer 'k' to find the kth smallest
11     // and kth largest elements.
12     int k = Integer.parseInt(input.next());
13
14     // Create a dynamic array to store the elements of the BST in sorted
15     // order.
16     List<Integer> sortedInorder = new ArrayList<Integer>();
17
18
19     // Perform an inorder traversal of the BST to populate
20     // 'sortedInorder' with sorted elements.
21     intTree.inorder(sortedInorder);
22
23     // Calculate the size of the 'sortedInorder' array.
24     int len = sortedInorder.size();
25
26     // Check if 'k' is within a valid range (1 <= k <= len).
27     if (k > 0 && k <= len) {
28         // Print the kth smallest and kth largest elements in the BST.
29         System.out.println("The " + k + "th smallest element in BST is "
30             + sortedInorder.get(k - 1));
31         System.out.println("The " + k + "th biggest element in BST is "
32             + sortedInorder.get(len - k));
33     }
34 }
```

Задача 3. Наоѓање на инордер претходник и следбеник

Да се напише функција која ќе го најде претходникот и следбеникот на даден јазел при inorder изминување на бинарно пребарувачко дрво. Вредноста на јазелот се внесува од тастатура.

Пример:

Влез:

Бинарно дрво од слика 6-39

Вредност за јазел: 3

Излез:

The inorder predecessor for 3 is 2

The inorder successor for 3 is 5

За да може да се определи претходник и следбеник на даден јазел при инордер изминување на бинарно пребарувачко дрво, прво треба да се стигне до јазелот кој ја содржи вредноста. Откако ќе се најде соодветниот јазел треба да се провери дали неговото лево односно десно поддрво не е празно. Доколку постојат уште јазли во поддрвата, тогаш следбеникот ќе го најдеме како најлевото дете од десното поддрво или самото десно дете, а претходникот ќе го најдеме како најдесното дете од левото поддрво или самото лево дете. Изминувањето на јазлите во дрвото се извршува во инордер редослед со користење на рекурзивни повици. Комплетниот код за оваа задача е даден во продолжение и е дел од заедничката класа за тестирање од бинарните дрва `BinaryTreeTest`.

```

1 public class BinaryTreeTest {
2     // Initialize the predecessor and successor nodes as global with null
3     // values.
4     static TreeNode<Integer> pred = new TreeNode<Integer>(null);
5     static TreeNode<Integer> succ = new TreeNode<Integer>(null);
6
7     // This function finds the inorder predecessor and successor nodes of a
8     // given 'key' in a BST.
9     static void findPredSucc(TreeNode<Integer> node, int key) {
10         // Base case: If the current node is null, return.
11         if (node == null) {
12             return;
13         }
14
15         // If the current node's data matches the 'key', we've found the
16         // node.
17         if (node.data == key) {
18             // If the node has a left subtree, find the predecessor.
19             if (node.left != null) {
20                 TreeNode<Integer> tmp = node.left;
21                 while (tmp.right != null)
22                     tmp = tmp.right;
23             }
24         }
25     }
26 }
```

```

21         pred = tmp;
22     }
23
24     // If the node has a right subtree, find the successor.
25     if (node.right != null) {
26         TreeNode<Integer> tmp = node.right;
27         while (tmp.left != null)
28             tmp = tmp.left;
29
30         succ = tmp;
31     }
32     return;
33 }
34
35 // If 'key' is smaller than the current node's data, search in the
36 // left subtree.
37 if (node.data > key) {
38     succ = node; // Update the successor.
39     findPredSucc(node.left, key);
40 }
41
42 // If 'key' is larger than the current node's data, search in the
43 // right subtree.
44 else {
45     pred = node; // Update the predecessor.
46     findPredSucc(node.right, key);
47 }
48
49 // This method serves as an example for finding the inorder predecessor
50 // and successor of a key in a binary search tree (BST).
51 public static void exampleInorderPredSucc() {
52     // Create an example integer binary search tree (BST) and keep a
53     // reference in intTree.
54     BinarySearchTree<Integer> intTree = GetExampleBSTree();
55
56     // Create a Scanner to get user input.
57     Scanner input = new Scanner(System.in);
58
59     // Read the user's input as an integer 'key'.
60     int key = Integer.parseInt(input.next());

```

```
58
59      // Find the inorder predecessor and successor for the 'key' in the
60      // BST.
61      findPredSucc(intTree.root, key);
62
63      // Print the inorder predecessor if it exists.
64      if (pred != null) {
65          System.out.println("Inorder predecessor for " + key + " is " +
66                           pred.data);
67
68      // Print the inorder successor if it exists.
69      if (succ != null) {
70          System.out.println("Inorder successor for " + key + " is " +
71                           succ.data);
72
73
74      public static void main(String[] args) {
75          //Call the exampleInorderPredSucc method
76          exampleInorderPredSucc();
77      }
78 }
```

6.2.9 Задачи за вежбање:

Задача 1. Сума на k најмали елементи

За дадено бинарно пребарувачко дрво да се напише функција која ќе пресметува сума на сите елементи помали или еднакви на k-от најмал елемент во дрвото. Вредноста за k се внесува од тастатура.

Задача 2. LCA во бинарно пребарувачко дрво

За дадено бинарно пребарувачко дрво да се напиše функција која ќе го најде соодветниот најдолен заеднички предок (lowest common ancestor - LCA) во дрвото.

Задача 3. Избриши листови

За дадено бинарно пребарувачко дрво да се напише функција која ќе направи бришење на сите терминирачки јазли - листови од дрвото.

Задача 4. Споредба на AVL со бинарни пребарувачки дрва

Да се напише метода за одредување разлика во висините на јазлите кај AVL дрво.
Да се спореди овој метод со методот за балансирано бинарно пребарувачко дрво од претходните примери под услов двете дрва да се исполнети со исти елементи.

Задача 5. Наогање на минимални и максимални елементи во AVL дрво

Да се напише функција за определување на минимален и максимален елемент во AVL дрво.

Глава 7

Графови

7.1 Што е тоа Граф?

Графот е една од фундаменталните структури на податоци во компјутерските науки. Во суштина, графот се состои од множество темиња и збир на работи. Темињата се нарекуваат и јазли. Работите поврзуваат парови темиња. Ова е слично на градовите поврзани со патишта, каде што градовите се темиња, а патиштата се работи.



Слика 7-1: Мостовите на паркот во Konigsburg (Калининград)

Графовите не се само теоретска математичка конструкција. Тие се длабоко вгнездени во апликациите од реалниот свет. Структурата на интернетот, социјалните мрежи како LinkedIn или Facebook, транспортните мрежи, па дури и молекуларните структури во биологијата може да се претстават како графови.

Во контекст на современите апликации, социјалната мрежа на Facebook може да се замисли како граф, каде што секој корисник е теме, а пријателствата меѓу корисниците се работи. На пример, во социјалните мрежи, алгоритмите за графови можат да помогнат да се препорачаат пријатели или содржини преку анализа на корисничките врски. Слично на тоа, транспортните мрежи користат графови за да ги идентификуваат најкратките или најмалку оптеретени рути. Дури и во биологијата, проучувањето на мрежите за интеракција помеѓу протеините или структурата на молекулите може да се гледа преку призмата на теоријата на графови. Графовите може да се користат за прикажување на речиси секаков вид на врска и на тој начин имаат широк опсег на апликации во различни домени.

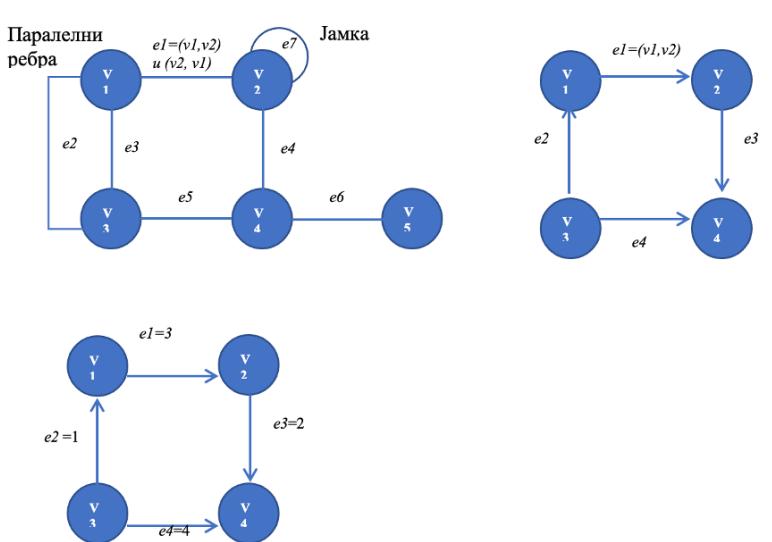
Графовите не се нов концепт. За почеток на развој на математичката теорија на графови се смета Ојлеровата дискусија за проблемот на поминување на 7-те мостови на реката Прегол во градот Кенигсберг (денешен Калининград, Слика 7-1¹), публикувана во 1736 година. Дискусијата го разгледувала проблемот „Дали е можно при прошетка низ паркот да се помине преку секој од мостовите точно по еднаш?“. После скоро еден век, Хамилтон и Киркман во 1856 година дискутирале сличен проблем на патувања во кои градовите би се поминале само еднаш. Овие навигациски проблеми овозможиле формирање на теоријата на графови.

7.2 Дефиниции и својства

Секој граф може да се престави како двојка од множества (V, E) , каде што V е множество од темиња а E е множество од работи. Секој раб, всушност претставува пар од темиња, односно, секој раб е дефиниран преку функција на соседство од темињата $E -> V \times V$. Темињата u и v се нарекуваат краеви на работ $e = (u, v) \in E$.

Постојат неколку начини за класификација на графовите. Најосновната класификација е врз основа на насоките на работите. Ако работите имаат насока, графот се нарекува насочен граф. На пример, Твитер може да се смета за насочен граф, каде што еден корисник може да следи друг корисник, меѓутоа обратното не мора да важи. Од друга страна, ако работите немаат насока, графот се нарекува ненасочен. Пријателствата на Фејсбук можат да бидат претставени со ненасочен граф бидејќи пријателството е взајмено. Во случај на ориентиран граф, темето u во работ (u, v) се нарекува почеток, а темето v крај на работ. По дефиниција, доколку за работите во еден граф сметаме дека $(u, v) = (v, u)$ тогаш графот

¹Merian-Erben - http://www.preussen-chronik.de/_/bild_jsp/key=bild_kathe2.html, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=1447648>



Слика 7-2: Илустрација на различни видови на графови

е неориентиран или ненасочен, додека доколку сметаме дека $(u, v) \neq (v, u)$ графот е ориентиран или насочен. Доколку графот е ориентиран или насочен, тогаш дозволено е меѓу две темиња да има повеќе работи кои ги сметаме за паралелни. На пример, доколку градовите ги сметаме за темиња, а патиштата за работи, постоењето на повеќе од еден директен пат помеѓу два града можеме да го сметаме за постоење на паралелни работи во графот. Во графот може да постојат и работи дефинирани како (v, v) кои ги нарекуваме јамки. Работите може да имаат вредност односно тежина. Во тој случај на секој раб се придржува број кој ја означува тежината на графот. На Сликата 7-2 се илустрирани ненасочен, насочен граф, паралелни работи, јамка и тежински насочен граф.

Темето што не е почеток или крај на ниту еден раб се нарекува изолирано теме. Темето соседно на само едно теме се нарекува висечко теме. Бројот на работи $d(v)$, чијшто крај е во темето v , се нарекува степен или валентност на v . Каде ориентираните графови бројот на работи чиј почеток е во v се нарекува степен на излез, а бројот на работи чиј крај во v се нарекува степен на влез. Ретки графови (Sparse graphs) се оние кои имаат многу помал број на работи во споредба со максималниот можноен број на работи за даден број на темиња. По обичај, ако бројот на работи E е многу помал од бројот на темиња V^2 (за ненасочени графови), графот може да се смета за редок.

Густи графови (Dense graphs) се оние кои имаат голем број на работи близок до максималниот можноен број за даден број на темиња. Ако бројот на работи E е близок до V^2 (за ненасочени графови), графот се смета за густ.

7.3 Репрезентација на графови

Графовите можат да се претстават во компјутерската меморија на повеќе начини. Пред да ги дефинираме начините на кои може да се дефинира граffот, ќе го дефинираме како абстрактен податочен тип кој ќе содржи информација за темињата и работите. Дополнително овој податочен тип треба да ги овозможува следните функции:

- Додавање раб
- Бришење раб
- Додавање теме
- Бришење теме
- Проверка на соседност

Најчесто графовите се претставуват со матрица на соседност, листа на соседи или листа на работи. Како абстрактен податочен тип или податочна структура, графовите дополнително, во своите темиња може да чуваат информација за темињата, односно, самите да бидат објекти од некоја произволна класа. Заради тоа во програмскиот јазик Јава, класите ги дефинираме користејќи генерици.

Во натамошниот текст е даден преглед на една можна имплементација на секоја од спомнатите репрезентации на графовите, во Јава, како и дел од предностите и недостатоците кои тие ги имаат. Во пракса, изборот на имплементацијата на граffот често зависи од специфичната употреба и природата на граffот (ретки наспроти густи) и операциите што треба често да се изведуваат. На пример, ако се прават многу проверки на соседството, матрицата за соседство е најдобар избор. Ако проблемот бара често поминување низ соседите или справување со ретки графови, листата со соседството е посоодветен пристап.

7.3.1 Матрица на соседност

Првата репрезентација користи дводимензионална матрица (2D низа) каде што секоја ќелија на позицијата (i, j) има вредност што покажува дали има раб од темето i до темето j .

Предности:

- Константно време за проверки на соседството: Директен пристап до матрицата $m[i][j]$ за да провериме дали темињата i и j се поврзани.
- Јасна структура: Особено корисно за густи графови, каде што врската помеѓу јазлите е посложена.

Недостатоци:

- Неефикасен простор за ретки графови: Секогаш користи V^2 простор, без оглед на бројот на работите.
- Помалку флексибилни: Прилагодувањето на големината (додавање или отстранување темиња) може да биде незгодно бидејќи бара промена на големината на матрицата што може да биде пресметковно скапа операција за оперативниот систем.

Во продолжение е дадена дефиницијата на класата `AdjacencyMatrixGraph` која дефинира граф, претставен со матрица на соседност.

```

1
2 public class AdjacencyMatrixGraph<T> {
3     private int numVertices;
4     private int[][] matrix;
5     private T[] vertices;
6
7     @SuppressWarnings("unchecked")
8     public AdjacencyMatrixGraph(int numVertices) {
9         this.numVertices = numVertices;
10        matrix = new int[numVertices][numVertices];
11        vertices = (T[]) new Object[numVertices];
12    }
13
14    public void addVertex(int index, T data) {
15        vertices[index] = data;
16    }
17
18    public T getVertex(int index) {
19        return vertices[index];
20    }
21
22    public void addEdge(int source, int destination) {
23        matrix[source][destination] = 1;
24        matrix[destination][source] = 1; // For undirected graph
25    }
26
27    public boolean isEdge(int source, int destination) {
28        return matrix[source][destination] == 1;
29    }
30
31    public void removeEdge(int source, int destination) {
32        matrix[source][destination] = 0;

```

```

33         matrix[destination][source] = 0; // For undirected graph
34     }
35
36     @SuppressWarnings("unchecked")
37     public void removeVertex(int vertexIndex) {
38
39         if (vertexIndex < 0 || vertexIndex >= size) {
40             throw new IndexOutOfBoundsException("Vertex index out of
41             bounds!");
42         }
43
44         int[][] newMatrix = new int[size-1][size-1];
45         T[] newVertices = (T[]) new Object[size-1];
46
47         // Copy the vertices and matrix excluding the given vertex
48         int ni = 0;
49         for (int i = 0; i < numVertices; i++) {
50             if (i == vertexIndex) continue;
51
52             int nj = 0;
53             for (int j = 0; j < numVertices; j++) {
54                 if (j == vertexIndex) continue;
55
56                 newMatrix[ni][nj] = matrix[i][j];
57                 nj++;
58             }
59
60             newVertices[ni] = vertices[i];
61             ni++;
62         }
63
64         // Replace the old matrix and vertices with the new ones
65         matrix = newMatrix;
66         vertices = newVertices;
67         numVertices--;
68     }
69
70     // Function to get all neighbors of a vertex
71     public List<T> getNeighbors(int vertexIndex) {
72         List<T> neighbors = new ArrayList<>();
73         for (int i = 0; i < matrix[vertexIndex].length; i++) {

```

```

73         if (matrix[vertexIndex][i] == 1) {
74             neighbors.add(vertices[i]);
75         }
76     }
77     return neighbors;
78 }
79 }
```

Во кодот, креираме класа за `AdjacencyMatrixGraph`. Делот `<T>` значи дека оваа класа може да работи со секаков вид податоци, како што се стрингови, цели броеви или други произволни класи. Матрицата е суштината на оваа класа. Замислете дека имате множество на градови и сакате да следите кои градови се директно поврзани со пат. „Матрицата за соседство“ е како табела што ја содржи оваа информација. Можеме да ја замислим како шаховска табла. Секој квадрат на таблота претставува можна патна врска помеѓу два града. Ако има 1 на квадратот, тоа значи дека двата града се поврзани. Ако има 0, тие не се. Темиња се градовите и низата од темиња е само список на градовите кои ги имаме.

Задача: Направете промена на класата `AdjacencyMatrixGraph` така што графот ќе биде насочен и тежински. Што треба да се промени во класата?

7.3.2 Листа на соседи

Листата на соседи, користи низа од листи. Секој индекс во низата претставува теме, а листата на секој индекс ги содржи соседите на темето.

Предности:

- Ефикасен простор за ретки графови: користи простор пропорционален на $V + E$, каде што V е бројот на темиња и E е бројот на работите.
- Брзо пребарување на соседите: Директно ги наведува сите соседи на темето.
- Флексибилен: Лесно може да претставува и насочени и ненасочени графови.

Недостатоци:

- Неефикасност на просторот за густи графови: за густи графови (каде што бројот на работите е близку до V^2), матрицата на соседството може да биде неефикасна за простор.
- Побавни проверки на соседството од матрицата: Проверката дали две темиња се директно поврзани бара $O(V)$ време во најлош случај.

Кодот во продолжение претставува имплементација на граф со користење на листа од соседи. Забележете дека оваа имплементација содржи мапа од поврзани листи како атрибут, додека индексот на мапата претставуваат јазлите. Вака дефинираната композиција овозможува лесно дефинирање на графот.

¹ `import java.util.HashMap;`

```
2 import java.util.HashSet;
3 import java.util.Set;
4 import java.util.Map;
5
6 public class AdjacencyListGraph<T> {
7     private Map<T, Set<T>> adjacencyList;
8
9     public AdjacencyListGraph() {
10         this.adjacencyList = new HashMap<>();
11     }
12
13     // Add a vertex to the graph
14     public void addVertex(T vertex) {
15         if (!adjacencyList.containsKey(vertex)) {
16             adjacencyList.put(vertex, new HashSet<>());
17         }
18     }
19
20     // Remove a vertex from the graph
21     public void removeVertex(T vertex) {
22         // Remove the vertex from all adjacency lists
23         for (Set<T> neighbors : adjacencyList.values()) {
24             neighbors.remove(vertex);
25         }
26
27         // Remove the vertex's own entry in the adjacency list
28         adjacencyList.remove(vertex);
29     }
30
31     // Add an edge to the graph
32     public void addEdge(T source, T destination) {
33         addVertex(source);
34         addVertex(destination);
35
36         adjacencyList.get(source).add(destination);
37         adjacencyList.get(destination).add(source); // for undirected graph
38     }
39
40     // Remove an edge from the graph
41     public void removeEdge(T source, T destination) {
42         if (adjacencyList.containsKey(source)) {
```

```

43         adjacencyList.get(source).remove(destination);
44     }
45     if (adjacencyList.containsKey(destination)) {
46         adjacencyList.get(destination).remove(source); // for undirected
47         graph
48     }
49
50     // Get all neighbors of a vertex
51     public Set<T> getNeighbors(T vertex) {
52         return adjacencyList.getOrDefault(vertex, new HashSet<>());
53     }
54 }
```

Задача: Направете ја истата промена како и претходно за да добиете класа која овозможува тежински граф.

7.3.3 Листа на работи

Оваа претстава се состои од листа на сите работи каде што секој раб е претставен како пар (u, v) . Во имплементацијата која ја имаме во Јава, дефинирана е помошна класа Edge која ги содржи темето почеток и темето крај на реброто.

Предности:

- Ефикасност на просторот за ретки графови: ако граffot има многу малку работи (редок граф), оваа претстава може да биде поефикасна за простор од матрицата за соседство.
- Едноставна структура: Тоа е само листа на парови или тројки (ако се вклучени тежините).
- Недостатоци:
 - Побавни проверки на соседството: за да проверите дали два јазли се соседни, треба да ја поминете целата листа.
 - Неefикасни за алгоритми за графови: Многу алгоритми на графови бараат проверка на соседите на темето. Во претставувањето на листата на работи, тоа не е едноставно и брзо.

Во продолжение е имплементација на класата ListOfEdgesGraph во Јава. Бидејќи се чуваат само работите, нема функција за додавање на темиња и бришење на темиња.

```

1
2 import java.util.ArrayList;
3 import java.util.List;
4 import java.util.Iterator;
5
6 public class ListOfEdgesGraph<T> {
```

```
7
8     private static class Edge<U> {
9         U source;
10        U destination;
11
12        public Edge(U source, U destination) {
13            this.source = source;
14            this.destination = destination;
15        }
16
17        public boolean involves(U vertex) {
18            return source.equals(vertex) || destination.equals(vertex);
19        }
20    }
21
22    private List<Edge<T>> edges;
23
24    public ListOfEdgesGraph() {
25        edges = new ArrayList<>();
26    }
27
28    public void addEdge(T source, T destination) {
29        edges.add(new Edge<>(source, destination));
30    }
31
32    public List<Edge<T>> getEdges() {
33        return edges;
34    }
35
36    public List<T> getNeighbors(T vertex) {
37        List<T> neighbors = new ArrayList<>();
38        for (Edge<T> edge : edges) {
39            if (edge.source.equals(vertex)) {
40                neighbors.add(edge.destination);
41            } else if (edge.destination.equals(vertex)) {
42                neighbors.add(edge.source);
43            }
44        }
45        return neighbors;
46    }
47
```

```

48     public void removeEdge(T source, T destination) {
49         Iterator<Edge<T>> iterator = edges.iterator();
50         while (iterator.hasNext()) {
51             Edge<T> edge = iterator.next();
52             if ((edge.source.equals(source) &&
53                 edge.destination.equals(destination)) ||
54                 (edge.destination.equals(source) &&
55                 edge.source.equals(destination))) {
56                 iterator.remove();
57             }
58         }
59     }
60
61     public void removeVertex(T vertex) {
62         Iterator<Edge<T>> iterator = edges.iterator();
63         while (iterator.hasNext()) {
64             Edge<T> edge = iterator.next();
65             if (edge.involves(vertex)) {
66                 iterator.remove();
67             }
68         }
69     }

```

Задача: Направете промена за графот да биде тежински.

7.4 Видови патеки кај графовите

Во теоријата на графови, секој а низа која содржи темиња меѓу кои има раб во графот, се нарекува патека. Постојат повеќе видови патеки, секоја со посебно значење и примена.

- **Едноставна патека** или патека: $P = (V_0, V_1 \dots V_k)$ во графот, така што секој од темињата се различни.

Пример: Едноставна патека е патека по која можеме да патуваме од град А до град С без да посетиме некој од попатните градови повеќе од еднаш.

- **Циклус** (затворена патека): $C = (V_0, V_1, \dots V_k)$ во која $V_0 = V_k$ и $k \geq 3$.
- Пример: Патување од град А до град В, С и.т.н. и на крајот враќање во А.
- **Хамилтонов пат**: Едноставна патека при која секое теме во графот се посетува точно по еднаш.

Пример: ако има градови A, B и C, посетувањето на овие градови само по еднаш би го направиле движејќи се по Хамилтонов пат.

- **Хамилтонов циклус:** Едноставен циклус во графот кој ги посетува сите темиња точно еднаш, со исклучок на првото теме кое е во исто време и последно.

Пример: Посетување на сите градови во една држава, по еднаш, почнувајќи од произволен град.

- **Ојлерова патека:** Патека во графот која минува низ секој раб точно по еднаш.

Пример: Ако има три града (A, B, C) и само три патишта што ги поврзуваат (A-B, B-C, A-C), патувањето по секој пат точно еднаш (на пример, A-B, B-C, C-A) ќе биде Ојлеровата патека.

- **Ојлеров циклус:** циклус во граф кој минува низ секој од работите точно по еднаш. Ако графот е поврзан и секое теме има парен степен тогаш содржи Ојлеров циклус.

Пример: Посетување на сите градови во една држава користејќи го секој пат само по еднаш.

- **Геодетска патека:** Доколку е даден граф G и функција на тежини $w : E \rightarrow R$, геодетскиот или најкраткиот пат помеѓу две темиња u и v претставува патеката со минимална сума од тежините на работите. За не-тежински граф, тоа е патеката која има најмал број на работи.

Пример: Доколку постои патека од град A до град B и друга патека преку град C тогаш директната патека е геодетска.

- **Изминување** е секвенца $W = \{v_0, e_1, v_1, e_2, \dots, v_k, e_k\}$ каде што секој раб e_i е раб кој поврзува две темиња v_{i-1} и v_i . Темињата и работите може да се повторуваат.

Пример: Почнувајќи од градот A, изминуваме преку градот B, се враќаме до градот A по истиот пат па потоа одиме до градот C.

- **Патека** е изминување во кое сите работи се различни, а темињата може да се повторуваат.

Пример: Почнувајќи од градот A, изминуваме преку градот B, потоа до градот C, а потоа назад во градот B користејќи различни патишта.

- **Јамка** е раб кој почнува и завршува во истото теме.

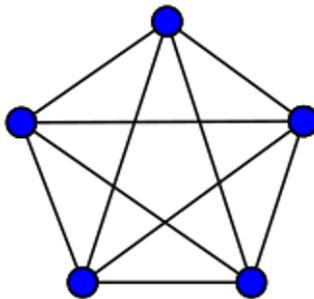
Пример: Кружна патека која тргнува од некој град и се враќа пак назад во градот.

7.5 Некои видови на графови

Постојат повеќе видови на графови, кои имаат различни карактеристики и својства. Во продолжение на овој текст ќе спомнеме дел од нив:

- Целосен или комплетен граф :

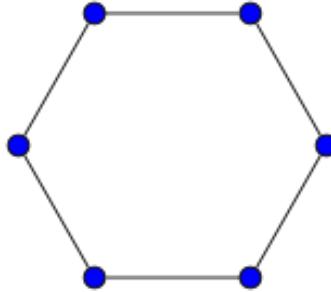
Целосен или комплетен граф $G(V, E)$ е граф во кој секој пар на различни темиња е поврзан со единствен раб. Пример за комплетен граф со 5 темиња е прикажан на Сликата 7-3².



Слика 7-3: Пример за комплетен граф со 5 темиња

- Циклус:

Граф $G(V, E)$ кој се состои од еден циклус, односно одреден број темиња (најмалку 3) поврзани во затворен синцир. Пример за ваков граф е прикажан на Сликата 7-4³



Слика 7-4: Пример за граф Циклус

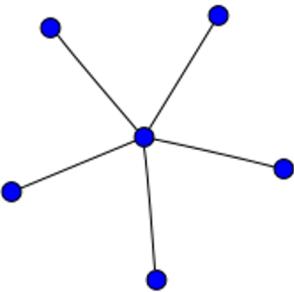
- Граф съвезда

График $G(V, E)$ во кој има централно теме поврзано со сите други темиња, а не постојат други работи. Пример за ваков граф е прикажан на сликата 7-5 ⁴.

²https://commons.wikimedia.org/wiki/File:Complete_graph_K5.svg

³https://commons.wikimedia.org/wiki/File:Cycle_graph.png

⁴https://commons.wikimedia.org/wiki/File:Star_graphs.svg



Слика 7-5: Пример за граф с вежда

- Граф патека

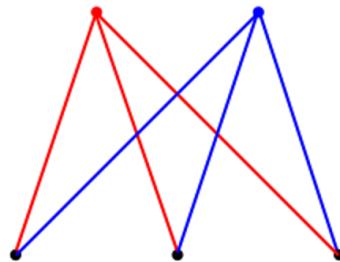
Граф $G(V, E)$ кој е составен од линеарно подредени темињата. Ако почнете од едниот крај, постои единствен пат до другиот крај (Слика 7-6⁵).



Слика 7-6: Пример за граф патека

- Бипартитен граф

Граф $G(V, E)$ чии темиња може да се поделат на две дисјунктни множества така што нема две темиња кои се од истото множество да бидат соседни. Пример за овој граф е прикажан на слика 7-7 ⁶.



Слика 7-7: пример за бипартитен граф

- Граф тркало

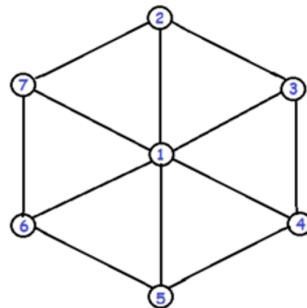
График $G(V, E)$ формиран со поврзување на едно универзално теме со сите темиња на еден циклус како на сликата 7-8 ⁷

- Дрво

⁵<https://commons.wikimedia.org/wiki/File:Path-graph.svg>

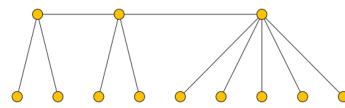
⁶https://commons.wikimedia.org/wiki/File:Complete_bipartite_graph_K32-001.svg

⁷https://commons.wikimedia.org/wiki/File:Harmonious_coloring_on_wheel_graph.png



Слика 7-8: Граф тркало

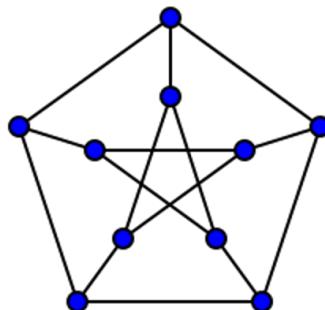
График $G(V, E)$ кој е поврзан и нема циклуси. Пример за ваков граф е даден на сликата 7-9⁸



Слика 7-9: Дрво

- Петерсен граф

Специфичен ненасочен график со 10 темиња и 15 работи (Слика 7-10⁹. Тоа е мал граф кој често се користи како контрапример во теоријата на графови за повеќе интуитивни својства.



Слика 7-10: Петерсен график

⁸https://commons.wikimedia.org/wiki/File:Tree_with_480_symmetries.svg

⁹https://commons.wikimedia.org/wiki/File:Petersen1_tiny.svg

7.6 Алгоритми за изминување и преbarување на графовите

Одењето теме по теме и посетување на секое од темињата на граffот се нарекува изминување на граffот. При изминувањето, се посетува секое теме, а потоа се посетуваат темињата од соседството. Според редоследот на изминување постојат два примарни начини на изминување на граffот. Изминување по длабочина (Depth-First Search - DFS) и изминување по широчина (Breadth-First Search - BFS). Покрај овие изминувања, постојат и други начини на изминување кои го забрзуваат процесот, како што е Бидирекционалното изминување. Со оглед на тоа што изминувањето често се користи за наоѓање на најкраткиот патека од едно до друго теме или наоѓање на најкраткиот пат помеѓу две темиња или од едно теме до сите останати темиња, во ова поглавје ќе ги разгледаме и алгоритмите Дијикстра и Флојд-Варшал алгоритмот.

7.6.1 Изминување по длабочина - DFS

Ова изминување наликува на одење во лавиринт се додека не се стигне до слеп пат по што се враќаме назад и избираме друг пат. Се нарекува изминување по длабочина бидејќи се движиме најдлабоко во граffот што можеме пред да продолжиме по друг раб. Имплементацијата на изминувањето може да биде со користење на стек или рекурзивна.

Изминувањето по длабочина функционира на следниот начин:

- Прво се избира почетното или изворното теме.
- Се избира првото соседно теме.
- Се посетува соседното теме.
- Повторно се избира соседно теме од тековно посетеното теме.
- ...
- Процесот продолжува се додека не се дојде до теме без соседи после што се враќа назад и се продолжува со следното соседно теме.

Во кодот кој следи е дадена рекурзивна имплементација со користење на класата AdjacencyListGraph.

```

1
2 public void DFS(T startVertex) {
3     Set<T> visited = new HashSet<>();
4     DFSUtil(startVertex, visited);
5 }
6

```

```

7  private void DFSUtil(T vertex, Set<T> visited) {
8      // Mark the current node as visited and print it
9      visited.add(vertex);
10     System.out.print(vertex + " ");
11
12     // Recur for all the vertices adjacent to this vertex
13     for (T neighbor : getNeighbors(vertex)) {
14         if (!visited.contains(neighbor)) {
15             DFSUtil(neighbor, visited);
16         }
17     }
18 }
```

Во кодот кој следи е дадена нерекурсивна имплементација со AdjacencyListGraph:

```

1  public void DFS(T startVertex) {
2      Set<T> visited = new HashSet<>();
3      Stack<T> stack = new Stack<>();
4
5      stack.push(startVertex);
6
7      while (!stack.isEmpty()) {
8          T vertex = stack.pop();
9
10         if (!visited.contains(vertex)) {
11             visited.add(vertex);
12             System.out.print(vertex + " ");
13
14             for (T neighbor : getNeighbors(vertex)) {
15                 if (!visited.contains(neighbor)) {
16                     stack.push(neighbor);
17                 }
18             }
19         }
20     }
21 }
```

DFS е корисен кога:

- сакаме да ги посетиме сите темиња во графот
- сакаме да провериме поврзаност во графот
- бараме решение во простор на состојби кое се наоѓа во листовите од графот

7.6.2 Изминување по широчина - BFS

BFS можеме да го замислим како кружни бранови во вода кога ќе фрлиме камен. Најпрво ги посетува најблиските соседи на темето, а потоа продолжува се подалеку и подалеку.

Изминувањето се одвива на следниот начин:

- Прво се избира почетното теме
- Се додаваат соседите на темето во листа на темиња кои треба да се посетат
- Се посетуваат соседите на тоа теме кои не се поестени и при секое посетување, соседите се додаваат во листата и се продолжува понатаму

Ваквото изминување може да се имплементира со користење на редица. Во продолжение е дадена имплементација која е компатибилна со класата `AdjacencyListGraph`.

```

1  public void BFS(T startVertex) {
2      Set<T> visited = new HashSet<>();
3      Queue<T> queue = new LinkedList<>();
4
5      visited.add(startVertex);
6      queue.add(startVertex);
7
8      while (!queue.isEmpty()) {
9          T vertex = queue.poll();
10         System.out.print(vertex + " ");
11
12         for (T neighbor : getNeighbors(vertex)) {
13             if (!visited.contains(neighbor)) {
14                 visited.add(neighbor);
15                 queue.add(neighbor);
16             }
17         }
18     }
19 }
```

BFS е корисен кога:

- сакаме да најдеме најкратка патека меѓу две темиња
- решаваме проблеми во кои сакаме да ги најдеме поврзаните компоненти во графот
- бараме решение во простор на состојби кое најблизу до почетокот

7.6.3 Двонасочно преbarување

Двонасочното преbarување е алгоритам кој извршува две истовремени преbarувања: едно од почетното теме или темето извор и едно од темето цел или крајното теме. Потрагата престанува кога ќе се сртнат двата фронта. Ова може да биде значително побрзо од извршувањето на еден BFS од почетното теме до крајното теме, особено во големи графици. Алгоритмот е сличен со BFS и има слична примена. Во продолжение е дадена пример имплементација на ова преbarување компатибилно со AdjacencyListGraph.

```

1   public boolean bidirectionalSearch(T startVertex, T endVertex) {
2       if (startVertex.equals(endVertex)) {
3           return true;
4       }
5
6       Set<T> visitedFromStart = new HashSet<>();
7       Set<T> visitedFromEnd = new HashSet<>();
8
9       Queue<T> queueFromStart = new LinkedList<>();
10      Queue<T> queueFromEnd = new LinkedList<>();
11
12      visitedFromStart.add(startVertex);
13      queueFromStart.add(startVertex);
14
15      visitedFromEnd.add(endVertex);
16      queueFromEnd.add(endVertex);
17
18      while (!queueFromStart.isEmpty() && !queueFromEnd.isEmpty()) {
19          if (pathExists(queueFromStart, visitedFromStart,
20                         visitedFromEnd)) {
21              return true;
22          }
23
24          if (pathExists(queueFromEnd, visitedFromEnd, visitedFromStart)) {
25              return true;
26          }
27
28          return false;
29      }
30
31      private boolean pathExists(Queue<T> queue, Set<T> visitedFromThisEnd,
```

```

    Set<T> visitedFromOtherEnd) {
32     T currentVertex = queue.poll();
33
34     for (T neighbor : getNeighbors(currentVertex)) {
35         if (visitedFromOtherEnd.contains(neighbor)) {
36             return true;
37         }
38         if (!visitedFromThisEnd.contains(neighbor)) {
39             visitedFromThisEnd.add(neighbor);
40             queue.add(neighbor);
41         }
42     }
43
44     return false;
45 }
```

7.6.4 Алгоритмот Дијикстра (Dijkstra)

Овој алгоритам за изминување на графови служи за наоѓање на најкраткиот пат од почетното или извортото теме до сите темиња во тежински граф. Сличен е на BFS но вклучува тежини во графот. Користи приоритетна редица за да ги избере најблиските темиња кои треба да се посетат.

За имплементација на овој алгоритам ни треба тежински граф. Тоа можеме да го постигнеме со промена на променливата adjacencyList во класата AdjacencyListGraph

```
1 private Map<T, Map<T, Integer>> adjacencyList;
```

Соодветната имплементација на ова пребарување е дадена во кодот подолу:

```

1 public Map<T, Integer> shortestPath(T startVertex) {
2     Map<T, Integer> distances = new HashMap<>();
3     PriorityQueue<T> queue = new
4         PriorityQueue<>(Comparator.comparingInt(distances::get));
5     Set<T> explored = new HashSet<>();
6
6     // Initialize distances
7     for (T vertex : adjacencyList.keySet()) {
8         distances.put(vertex, Integer.MAX_VALUE);
9     }
10    distances.put(startVertex, 0);
11 }
```

```

12     queue.add(startVertex);
13
14     while (!queue.isEmpty()) {
15         T current = queue.poll();
16         explored.add(current);
17
18         for (Map.Entry<T, Integer> neighborEntry :
19             adjacencyList.get(current).entrySet()) {
20             T neighbor = neighborEntry.getKey();
21             int newDist = distances.get(current) +
22                 neighborEntry.getValue();
23
24             if (newDist < distances.get(neighbor)) {
25                 distances.put(neighbor, newDist);
26
27                 // Update priority queue
28                 if (!explored.contains(neighbor)) {
29                     queue.add(neighbor);
30                 }
31             }
32         }
33     }
34
35     return distances;
36 }
```

7.6.5 Алгоријатм на Флојд-Варшал (Floyd-Warshall)

Алгоритмот Флојд-Варшал се користи за пронаоѓање на најкратките патеки помеѓу сите парови темиња во тежински граф. Овој алгоритам работи и за насочени и за ненасочени графови и се справува со негативни тежини, но не работи со графови кои содржат негативни циклуси.

За алгоритмот Флојд-Воршал, обично се користи матрична претстава на граѓот бидејќи треба да пресметаме растојанија помеѓу сите парови темиња. За да ја искористиме класата која претходно ја дефиниравме, мора да дозволиме класата да има тежини наместо само врски (единици и нули), и соодветно да се променат сите функции во класата. Во продолжение е даден модифициран код на дел од класата заедно со имплементацијата на алгоритмот. Гледаме дека во промената во тежински граф кај класата, наместо да има 0 кај неповрзаните темиња во конструкторот при иницијализација, се става INF како многу голема

ПОЗИТИВНА ВРЕДНОСТ.

```

1  public class AdjacencyMatrixGraph<T> {
2      private int numVertices;
3      private int[][] matrix;
4      private T[] vertices;
5
6      public static final int INF = Integer.MAX_VALUE / 2; // To prevent
7          overflow
8
9      @SuppressWarnings("unchecked")
10     public AdjacencyMatrixGraph(int numVertices) {
11         this.numVertices = numVertices;
12         matrix = new int[numVertices][numVertices];
13         for (int i = 0; i < numVertices; i++) {
14             for (int j = 0; j < numVertices; j++) {
15                 if (i == j) matrix[i][j] = 0;
16                 else matrix[i][j] = INF;
17             }
18         }
19         vertices = (T[]) new Object[numVertices];
20     }
21
22     // ... (rest of the methods from the provided class)
23
24     public void addEdge(int source, int destination, int weight) {
25         matrix[source][destination] = weight;
26         matrix[destination][source] = weight; // For undirected graph
27     }
28
29     public int getEdge(int source, int destination) {
30         return matrix[source][destination];
31     }
32
33     public int[][] floydWarshall() {
34         int[][] dist = new int[numVertices][numVertices];
35         for (int i = 0; i < numVertices; i++) {
36             for (int j = 0; j < numVertices; j++) {
37                 dist[i][j] = matrix[i][j];
38             }
39         }

```

```

40     for (int k = 0; k < numVertices; k++) {
41         for (int i = 0; i < numVertices; i++) {
42             for (int j = 0; j < numVertices; j++) {
43                 if (dist[i][k] + dist[k][j] < dist[i][j]) {
44                     dist[i][j] = dist[i][k] + dist[k][j];
45                 }
46             }
47         }
48     }
49
50     return dist;
51 }
```

7.7 Проблеми во кои се применуваат алгоритми со графови

Графовите како структура се користат за репрезентација на големо множество од проблеми. Следствено на тоа, постојат и голем број на алгоритми кои ги решаваат овие проблеми. Во продолжение ќе дадеме осврт на неколку проблеми кои се решаваат со користење на алгоритми со графови. Дел од алгоритмите веќе ги видовме во претходниот текст, а дополнително ќе разработиме уште неколку алгоритми. Останатите проблеми само ќе ги спомнеме и опишеме. Сите проблеми имаат решенија кои понекогаш може да бидат релативно едноставни, но и многу комплексни. Секој од овие проблеми и алгоритми е неопходен дел од критичните компоненти на модерната компјутерска односно пресметковна инфраструктура во различните индустрии.

7.7.1 Наоѓање на најкраток пат

Овие алгоритми (од кои дел веќе видовме во претходното поглавје) се користат тогаш кога треба да го најдеме најкраткото растојание помеѓу две темиња во смисла на најмала сумарна тежина, најмал број на поминати работи и сл. Се користат во многу апликации како GPS системи за навигација, рутирачки протоколи и слично.

7.7.2 Поврзаност на граф

Алгоритмите за наоѓање на поврзаност во граф се користат за проверка дали две темиња во граffот се поврзани. На пример во насочен граф, силна поврзаност

е кога патувајќи од било кое теме може да стигнеме до било кое друго теме. Се користат за проверка на поврзаност во социјални мрежи за препорака на пријатели и наоѓање на заеднички пријатели.

7.7.3 Мрежен проток

Наоѓање на оптимална патека на пренос на материјали или податоци низ мрежа. На пример колку податоци можеме да пратиме од еден до друг компјутер без да го надминеме капацитетот на линкот? Овој тип на проблеми се користат во компании за логистика и транспорт. Се користи max-flow алгоритам за оптимизација на рутите за стоки и материјали во синџирите за снабдување.

7.7.4 Детекција на циклуси

Овој проблем дефинира одредување дали во графот има или нема циклуси (патека која почнува и завршува во истото теме). Решението на овој проблем има многу апликации кои вклучуваат детекција на циркуларни зависности во процесите.

7.7.5 Партиционирање и боење

Боењето на графовите е доделување на боја на темињата така што да нема две соседни темиња со иста боја. Партиционирањето е делење на графот според определени свойства. Решенијата на овој проблем се користат за решавање на проблеми како доделување на фреквенции на мрежните келии за мобилни телефони, каде што две соседни келии не треба да имаат иста фреквенција.

7.7.6 Споредување на графови

Споредувањето на графови е процес на наоѓање кореспонденција помеѓу темињата (а понекогаш и работите) на два графа кои задоволуваат одредени ограничувања. Оваа кореспонденција може да ги нагласи сличностите или разликите помеѓу двета графа. На пример ова може да се користи во системи за доделување на задачи каде што задачите се доделуваат на луѓе кои се најспособни да ги извршат.

7.7.7 Хамилтонови и Ојлерови патеки и циклуси

Како што претходно беа дефинирани, Хамилтоновите патеки ги посетуваат темињата точно по еднаш, додека Ојлеровите истото го прават за работите. Може

да се користат за наоѓање на оптимална патека на машина за печатење на електронски плочки.

7.7.8 Артикулациони точки

Идентификација на темињата кои, кога би биле отстранети, го зголемуваат бројот на поврзани компоненти за еден. Се користи во дизајн на мрежи за откривање на критичните точки, чиј што испад би можел да ја подели мрежата.

7.7.9 Проблем на покриеност на темињата

Избор на најмалиот број на темиња, така што секој раб добира барем едно од овие темиња. Се користи за дефинирање на безбедносни проблеми, како што е избор на минимален број на чувари потребни за да се набљудуваат сите коридори.

7.7.10 Детекција на планарни графови

Одредување дали графот може да биде нацртан без да се вкрстат работите (освен кај темињата). Се користи за дизајнирање на подлогата на електронските плочки каде што проводниците не треба да се допираат.

7.7.11 Изоморфизам на графови

Се користи за определување дали два графа може да се претопат еден во друг со промена на имињата на темињата. Примената ја има во биоинформатика каде што се користи за идентификација на слични молекуларни структури.

7.7.12 Проблем на патувачки трговец - Traveling Salesman Problem (TSP)

Ова е еден од најпознатите проблеми. Целта е да се најде најкраката можна рута која ќе ги посети сите градови и ќе го врати трговецот назад во почетниот град. Се користи за решавање на логистички проблеми за сервиси за достава.

7.7.13 Тополошко сортирање

Тополошкото сортирање е линеарно подредување на темињата во насочен граф, така што кај секој раб $U -> V$, U доаѓа пред V во подредувањето. Ова подредување е возможно само кај насочени ациклични графови (Directed Acyclic Graphs (DAGs)). Често се користи за решавање на зарачи на правење распоред

на производството каде што производството на една компонента зависи од производството на друга компонента. Еден пристап за решавање на проблемот е пристап базиран на DFS:

1. Направете празен стек и направете DFS изминување на графот
2. После рекурзивно повикување на DFS за соседните темиња, додадете го темето во стекот.
3. Стекот на крајот е тополошки сортиран.

Во продолжение е дадена пример имплементација која е компатибилна со репрезентацијата на граф со листа на соседи која ја дефинирајме претходно.

```

1 // DFS utility function used for topological sorting
2 private void topologicalSortUtil(T vertex, Set<T> visited, Stack<T>
3     stack) {
4     visited.add(vertex);
5     for (T neighbor : getNeighbors(vertex)) {
6         if (!visited.contains(neighbor)) {
7             topologicalSortUtil(neighbor, visited, stack);
8         }
9         stack.push(vertex);
10    }
11
12    public List<T> topologicalSort() {
13        Stack<T> stack = new Stack<>();
14        Set<T> visited = new HashSet<>();
15
16        for (T vertex : adjacencyList.keySet()) {
17            if (!visited.contains(vertex)) {
18                topologicalSortUtil(vertex, visited, stack);
19            }
20        }
21
22        List<T> order = new ArrayList<>();
23        while (!stack.isEmpty()) {
24            order.add(stack.pop());
25        }
26        return order;
27    }

```

7.7.14 Минимално распнувачко дрво (Minimum spanning tree - MST)

MST е подмножество од работите на поврзан, ненасочен граф во кој работите ги поврзуваат сите темиња, без циклуси и со минималната можна вкупна тежина на работите. Ако графот не е поврзан, нема распнувачко дрво, а камоли минимално распнувачко дрво. Како структура се користи за кластерирање, алгоритми за апроксимација, дизајн на телекомуникациски мрежи и многу други примени. За наоѓање на MST се користат алгоритмите на Прим и Крускал кои се описаны во понатамошниот текст.

Алгоритмите на Крускал и Прим се алчни алгоритми, што значи дека го прават локално оптималниот избор на секој чекор. Овој локален избор води до глобално оптимално решение за проблемот со наоѓање на MST. И двата од овие алгоритми гарантираат минимално опфатено дрво на крајот од нивното извршување. Изборот кој алгоритам да се користи може да зависи од специфичните детали и барања на дадениот проблем или од природата на влезниот граф.

Алгоритмот на Прим

Чекорите на алгоритмот на Прим се следните:

1. Иницијализирај го MST со едно теме.
2. Додека постојат темиња надвор од MST:
 - (а) Избери го работ со најмала тежина кој поврзува теме од MST со некое од надворешните темиња.
 - (б) Додај го темето во MST

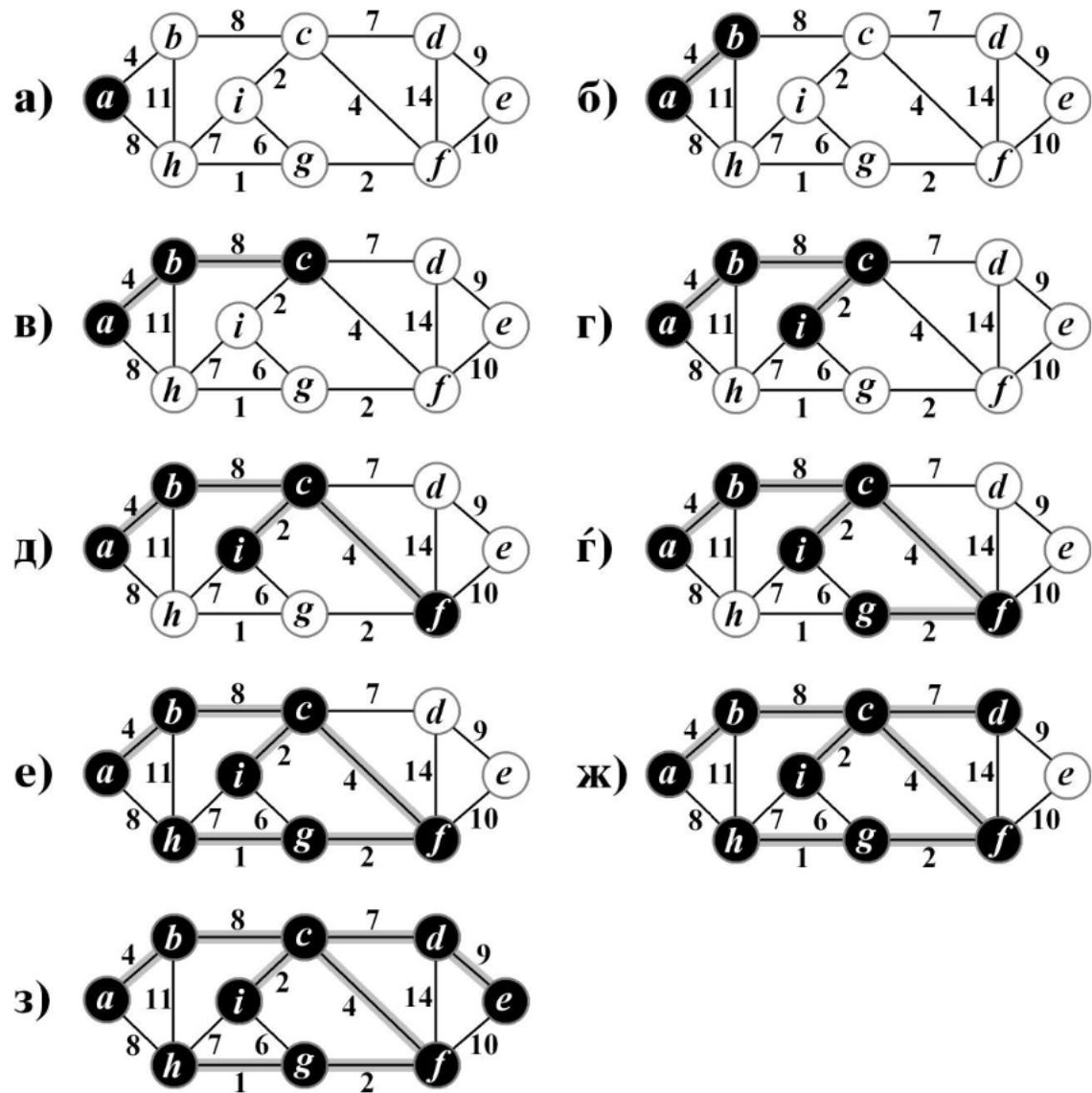
За поефикасна имплементација може да се користат приоритетни редици или Хип дрва.

Алгоритмот на ПРИМ е илустриран на сликата 7-11

Алгоритмот на Крускал

Чекорите на алгоритмот на Крускал се следните:

1. Започни со листа од сите работи
2. Сортирај ги работите според тежината
3. Иницијализирај празен граф
4. За секој раб од сортираната листа:
 - (а) Ако додавањето на работ не формира циклус, додај го во MST.



Слика 7-11: Илустрација на Прим

(б) Додај го темето во MST

Алгоритмот е илустриран на сликата 7-12

7.8 Едноставни проблеми со графови

Задача 1. Креирање на граф

Ваша задача е да креирате неориентиран нетежински граф со матрица на соседство, каде темињата како информација содржат буква. Графот го креирате според наредбите кои се добиваат. Ќе ви биде дадена низа од команди што можат да бидат од следните типови:

CREATE [број] - треба да креирате нов граф со дадениот број на темиња. Вредностите во темињата ќе бидат буквите од англиската азбука, според нивниот редослед. Така ако имате 3 темиња буквите ќе бидат: A, B и C. ADDEdge [број1] [број2] - треба да креирате ребро меѓу темињата со реден број број1 и реден број број2. DELETEEDGE [број1] [број2] - треба да го избришете реброто меѓу темињата со реден број број1 и реден број број2. ADJACENT [број1] [број2] - треба да испечатите 1 доколку темињата со реден број број1 и реден број број2 се соседни, во спротивност 0. PRINTMATRIX - Треба да ја испечатите матрицата на соседство PRINTNODE [број] - Треба да ја испечатите информацијата (т.е. буквата) за дадениот реден број на теме

Во првата линија на влезот е даден бројот на команди кои ќе следуваат.

Влез: Прво е даден бројот N на команди кои ќе следуваат после креирањето на графот. Потоа следува команда за иницијалното креирање на графот. На крај следуваат N линии коишто ги претставуваат командите што треба да се извршат на креираниот празен граф.

Излез: Се печати излезот од оние команди кои вклучуваат некакво печатење.

Пример:

Влез:

5

CREATE 4

ADDEdge 0 3

PRINTMATRIX

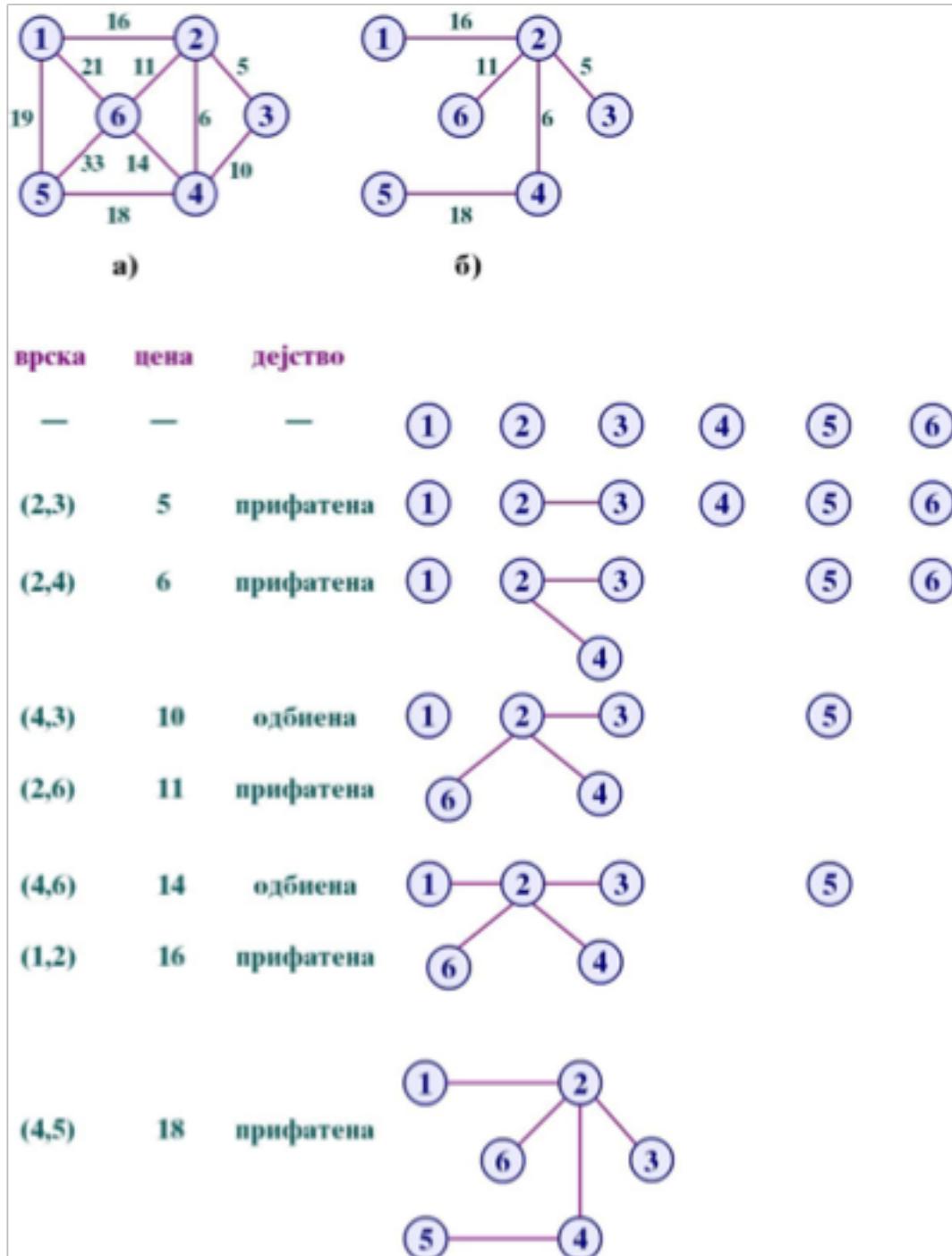
PRINTNODE 2

ADJACENT 0 2

DELETEEDGE 3 0

Излез:

0 0 0 1



Слика 7-12: Илустрација на Крускал

```

0 0 0 0
0 0 0 0
1 0 0 0
C
0

```

Решение

За ова ни е потребно најпрво да го прочитаме и зачуваме бројот на команди кој што ќе следува после креирањето на графот, во некоја променлива N . Потоа следно ја читаме командата за креирање за графот и според зададените параметри го иницијализираме истот, односно креираме граф со зададениот број на темиња. Гледаме од самата задача дека треба да се печати матрица на соседност, па според тоа најповолно ќе биде да се користи таа репрезентација на граф - `AdjacencyMatrixGraph`. За да може да се испечати матрицата, во оваа класа треба да додадеме метод што ќе го прави истото:

```

1 public void printMatrix() {
2     for(int i=0;i<numVertices;i++) {
3         for(int j=0;j<numVertices;j++) {
4             System.out.print(matrix[i][j] + " ");
5         }
6         System.out.println();
7     }
8 }

```

Следно во N итерации ги читаме командите што треба да се извршат, и ги парсирате истите. Во овој случај најпрво треба да се види клучниот збор за да знаеме каква операција треба да се прави. Потоа, ја повикуваме соодветната функција врз графот, со спроведените параметри (доколку има).

```

1 import java.util.Scanner;
2
3 //Вметни класа AdjacencyMatrixGraph<T>
4
5 public class GraphCreate {
6
7     public static void main(String[] args) {
8         Scanner sc = new Scanner(System.in);
9
10        int N = sc.nextInt();
11        sc.nextLine();

```

```

12
13     String pom[] = sc.nextLine().split(" ");
14     int brteminja = Integer.parseInt(pom[1]);
15     AdjacencyMatrixGraph<Character> g = new
16         AdjacencyMatrixGraph<Character>(brteminja);
17     for (int j = 0; j < brteminja; j++)
18         g.addVertex(j, (char) ((int) 'A' + j));
19
20     for (int i = 1; i < N; i++) {
21         pom = sc.nextLine().split(" ");
22         switch (pom[0]) {
23             case "ADDEdge":
24                 g.addEdge(Integer.parseInt(pom[1]),
25                           Integer.parseInt(pom[2]));
26                 break;
27             case "DELETEEdge":
28                 g.removeEdge(Integer.parseInt(pom[1]),
29                           Integer.parseInt(pom[2]));
30                 break;
31             case "ADJACENT":
32                 System.out.println(g.isEdge(Integer.parseInt(pom[1]),
33                                         Integer.parseInt(pom[2])) ? 1 : 0);
34                 break;
35             case "PRINTMATRIX":
36                 g.printMatrix();
37                 break;
38             case "PRINTNODE":
39                 System.out.println(g.getVertex(Integer.parseInt(pom[1])));
40                 break;
41             default:
42                 System.out.println("Nevalidna komanda: "+pom[0]);
43                 break;
44         }
45     }
46 }
47
48 }
```

Задача 2: Од матрица на соседност во листа на соседи

Да се напише функција којашто даден граф претставен со матрица на соседност ќе го даде истиот претставен со листа на соседи.

Решение

Најпрво треба да се инстанцира објект од тип AdjacencyListGraph. Следно, со итерирање на низата во која што се чуваат информациите за јазлите може да се додадат истите во ново креираниот граф. Потоа, со итерирање на секоја редица од матрицата, соодветно може да се додадат ребрата помеѓу јазлите. За оваа цел ќе треба да ги имаме вклучено и двете репрезентации (класи) на графот, и во класата AdjacencyMatrixGraph би се вметнала следната функција којашто ќе го прави потребното:

```

1 public AdjacencyListGraph<T> toAdjacencyList() {
2     AdjacencyListGraph<T> result = new AdjacencyListGraph<>();
3
4     for(int i=0;i<numVertices;i++) {
5         result.addVertex(vertices[i]);
6     }
7
8     for(int i=0;i<numVertices;i++) {
9         for(int j=0;j<numVertices;j++) {
10             if(matrix[i][j] > 0) {
11                 result.addEdge(vertices[i], vertices[j]);
12             }
13         }
14     }
15
16     return result;
17 }
```

Задача за вежбање: Да се напише функција која што ќе ја прави обратната конверзија (од листа на соседи во матрица на соседност).

Задача 3: Најкраток пат во нетежински граф

За даден јазел во нетежински граф, да се најдат најкратките патишта до сите други јазли.

Решение

За да ја постигнеме оваа цел можеме да го искористиме BFS алгоритамот за пребарување, бидејќи знаеме дека секогаш ги посетува прво најблиските јазли. Ќе треба само малку да се измени истиот со тоа што ќе се низа на растојанија. Почнуваме од тоа што знаеме дека растојанието од дадениот јазел до самиот себе е 0. За сите други јазли може да зачуваме некоја невозможна вредност како бесконечност, или -1, и според тоа ќе знаеме дали пат е најден до даден јазел.

```

1 public int shortestPathsFrom(int vertexIndex) {
2     boolean visited[] = new boolean[numVertices];
3     int distances[] = new int[]
4     for(int i=0;i<numVertices;i++) {
5         visited[i] = false;
6         distances[i] = -1;
7     }
8     visited[vertexIndex] = true;
9     distances[vertexIndex] = 0;
10    System.out.println(vertexIndex + ": " + vertices[vertexIndex]);
11
12    Queue<Integer> q = new LinkedQueue<>();
13    q.enqueue(vertexIndex);
14
15    int tmp;
16
17    while(!q.isEmpty()) {
18        tmp = q.dequeue();
19        for(int i=0;i<numVertices;i++) {
20            if(isEdge(tmp,i)) {
21                if(!visited[i]) {
22                    visited[i] = true;
23                    distances[i] = distances[tmp] + 1;
24                    System.out.println(i + ": " + vertices[i]);
25                    q.enqueue(i);
26                }
27            }
28        }
29    }
30 }
```

Задача 4: Изминување на лавиринт

Нека е даден еден совршен лавиринт, односно лавиринт кој има само еден пат од една точка на лавиринтот до било која друга. Нека лавиринтот биде даден во следната форма (како влез од карактери).

```
6,6
#####
# # ##
# # S#
# # ##
# E #
#####
```

Првите 2 бројки се димензиите на лавиринтот, а потоа следува самиот лавиринт. Во лавиринтот секое поле е означено со даден знак. Доколку знакот е '#', тоа значи дека на тоа поле од лавиринтот не смее да се стапнува, а на сите други полиња може да се оди. Полето означено со 'S' е startното поле, односно полето од каде што треба да се почне со изминување, додека полето означено со 'E' е крајното поле на изминување.

Треба да се најде единствениот пат од полето 'S' до полето 'E'.

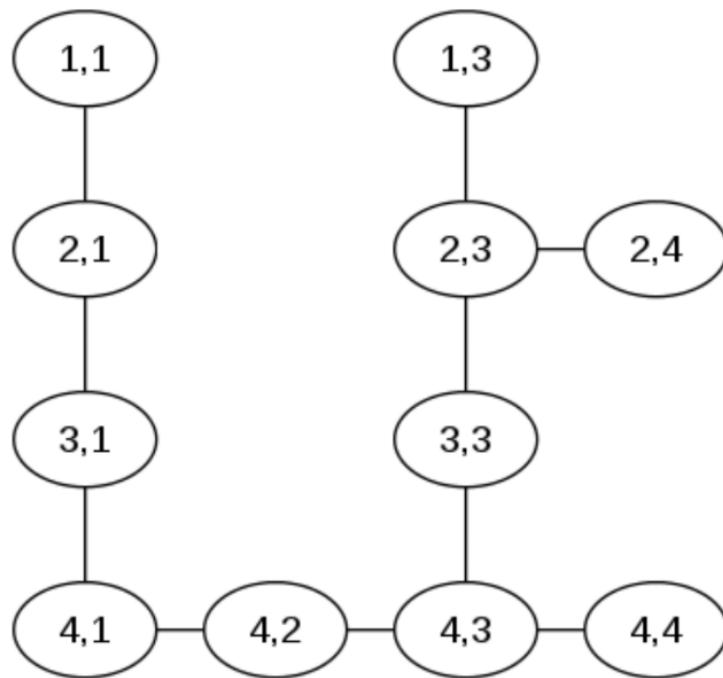
Решение

Дадениот лавиринт може да го претставиме преку граф со тоа што јазлите ќе ни бидат полињата во коишто може да се минува, а ребра ќе има помеѓу соседните такви полиња. Пример, лавиринтот даден погоре би изгледал како на следната слика:

Покрај креирањето на самиот граф, ќе треба соодветно да се запамтат кои се почетниот и крајниот јазел. Имајќи го сето тоа, може да се искористи нерекурзивната функција за пребарување по длабочина, почнувајќи од зададениот startен јазел, со тоа што ќе се направи мала промена истата да застане во тој момент кога ќе дојде до крајниот јазел, и дополнително јазлите ќе се вадат од стекот само кога ќе нема повеќе каде да се оди од тој јазел, односно истиот ќе нема непосетени соседи, за да може да се зачува целосната патека. Тој стек на крај само ќе може да се преврти во нов стек, за да се испечати соодветната патека од почеток до крај.

Ќе искористиме граф претставен со листа на соседи, па ќе треба оваа нова DFS-базирана функција да се додаде во класата `AdjacencyListGraph`.

```
1 public void findPath(T startVertex, T endVertex) {
2     Set<T> visited = new HashSet<>();
3     Stack<T> invertedPath = new Stack<>();
```



Слика 7-13: Репрезентација на лавиринтот со граф

```

4
5     visited.add(startVertex);
6     invertedPath.push(startVertex);
7
8     while(!invertedPath.isEmpty() && !invertedPath.peek().equals(endVertex))
9     {
10        T currentVertex = invertedPath.peek();
11        T tmp = currentVertex;
12
13        for(T vertex : getNeighbors(currentVertex)) {
14            tmp = vertex;
15            if(!visited.contains(vertex)) {
16                break;
17            }
18
19            if(!visited.contains(tmp)) {
20                visited.add(tmp);
21                invertedPath.push(tmp);
22            } else {

```

```

23         invertedPath.pop();
24     }
25 }
26
27 Stack<T> path = new Stack<>();
28
29 while(!invertedPath.isEmpty()) {
30     path.push(invertedPath.pop());
31 }
32
33 while(!path.isEmpty()) {
34     System.out.println(path.pop());
35 }
36 }
```

Следно само треба во главната програма да ја вклучиме оваа класа со новата функција, да го прочитаме влезот и соодветно да го креираме графот (со тоа што ќе треба најпрво да се иницијализира графот, па да се креираат јазлите со соодветните ознаки, и потоа соодветно да се додадат ребра помеѓу нив). На крај, треба само да се повика новата функција на така креираниот граф.

```

1 import java.util.Scanner;
2
3 //Вметни класа AdjacencyListGraph<T> вклучувајчи ја новта функција.
4
5 public class Maze {
6     public static void main(String[] args) {
7         Scanner sc = new Scanner(System.in);
8
9         String tmp = sc.nextLine();
10        String parts[] = tmp.split(",");
11
12        int m = Integer.parseInt(parts[0]);
13        int n = Integer.parseInt(parts[1]);
14
15        String lines[] = new String[m];
16
17        AdjacencyListGraph<String> mazeGraph = new AdjacencyList<>();
18
19        String startVertex, endVertex;
20
21        for(int i=0;i<m;i++) {
```

```

22     lines[i] = sc.nextInt();
23
24     for(int j=0;j<n;j++) {
25         if(lines[i].charAt(j) != '#') {
26             mazeGraph.addVertex(i + "," + j);
27
28             if(lines[i].charAt(j) == 'S') {
29                 startVertex = i + "," + j;
30             } else if(lines[i].charAt(j) == 'E') {
31                 endVertex = i + "," + j;
32             }
33             if(i>0 && lines[i-1].charAt(j) != '#') {
34                 mazeGraph.addEdge((i-1) + "," + j, i + "," + j);
35             }
36             if(j>0 && lines[i].charAt(j-1) != '#') {
37                 mazeGraph.addEdge(i + "," + (j-1), i + "," + j);
38             }
39         }
40     }
41 }
42
43     sc.close();
44
45     mazeGraph.findPath(startVertex, endVertex);
46 }
47 }
```

Задача 5: Тополошко сортирање

Да се имплементира тополошко сортирање на јазлите во даден граф, односно да се подредат јазлите во графот така што сите ребра да покажуваат од лево кон десно.

Решение

Од самиот опис на проблемот можеме да сфатиме дека на некој начин овде предност би имале јазлите коишто воопшто немаат или имаат најмал број на влезни ребра, па потоа ќе следуваат нивните соседи и така натаму. Треба да забележиме дека ова е возможно да се направи само кај ацикличен насочен граф. Знаејќи го ова, можеме да ги изминеме сите јазли и да пуштиме пребарување по длабочина од нив, доколку веќе не биле посетени. Со ваквото рекурзивно изминување мо-

жеме јазлите да ги ставаме во стек кога веќе нема да имаат непосетени соседи. На овој начин, на дното на стекот ќе се наоѓаат оние јазли коишто имаат влезни, но не излезни ребра, а на врвот, односно последно додадени, ќе бидат оние јазли кои што немаат влезни ребра. Баш тоа и ни треба.

За ова, како што е напоменато погоре, ќе треба да се искористи стек податочна структура. Може да се вклучи некоја од имплементациите или да се земе веќе готовата имплементација од `java.util` пакетот.

```
1 import java.util.Stack;
```

Следно, ќе ги додадеме овие методи во во класата `AdjacencyListGraph`:

```
1 public void topologicalSort() {
2     Set<T> visited = new HashSet<>();
3
4     Stack<T> s = new Stack<>();
5
6     for(vertex: adjacencyList.keySet()) {
7         dfsVisit(vertex, visited, s);
8     }
9
10    while(!s.isEmpty()) {
11        System.out.println(s.pop());
12    }
13 }
14
15 private void dfsVisit(T vertex, Set<T> visited, Stack<T> s) {
16     if(!visited.contains(vertex)) {
17         visited.add(vertex);
18         for(T v : getNeighbors(vertex)) {
19             dfsVisit(v, visited, s);
20         }
21         s.push(vertex);
22     }
23 }
```

Задача за вежбање: Да се имплементира тополошко сортирање за граф претставен со матрица на соседство.

7.9 Напредни проблеми со графови

Задача 1: Минимално распнувачко дрво

Да се напише функција за наоѓање на листата од ребра кои го формираат минималното распнувачко дрво на даден граф.

Решение

Постојат два добро познати алгоритми за решавање на овој проблем - алгоритмите на Крускал и Прим. Овде ќе ги погледнеме и двата, имплементирани за граф претставен со матрица на соседства.

Алгоритам на Крускал: Започнува да го гради минималното распнувачко дрво така што на почеток графот го гледа како шума од $|V|$ дрва (секој јазел е посебно дрво). Следно, треба да се итерираат сите ребра во растечки редослед. На тој начин, се разгледува следното ребро, и се додава во минималното распнувачко дрво доколку не ја уништува карактеристиката на дрво (не додава циклус). Ова можеме да го осигураме така што за секој јазел ќечуваме од кое дрво е дел. Како што беше напоменато погоре, на почеток секој јазел е посебно дрво, и како што ќе се додаваат ребра овие дрва ќе се спојуваат.

За ова најпрво ќе ни треба некоја класа која што ќе претставува ребро со дадена тежина.

```

1  public class Edge{
2      private int fromVertex, toVertex;
3      private float weight;
4      public Edge(int from, int to, float weight) {
5          this.fromVertex = from;
6          this.toVertex = to;
7          this.weight = weight;
8      }
9
10     public int getFrom() {
11         return this.fromVertex;
12     }
13     public int getTo() {
14         return this.toVertex;
15     }
16     public float getWeight() {
17         return this.weight;
18     }
19 }
```

Следно, во класата AdjacencyMatrixGraph треба да додадеме метод за листање на сите ребра, метод за правење на унија помеѓу 2 дрва, и потоа самиот алгоритам:

```

1  private List<Edge> getAllEdges() {
2      List<Edge> edges = new ArrayList<>();
3
4      for(int i=0;i<numVertices;i++) {
5          for(int j=0;j<numVertices;j++) {
6              if(isEdge(i,j)) {
7                  edges.add(new Edge(i, j, matrix[i][j]));
8              }
9          }
10     }
11
12     return edges;
13 }
14
15 private void union(int u, int v, int[] trees) {
16     int findWhat, replaceWith;
17     if(u<v) {
18         findWhat = trees[v];
19         replaceWith = trees[u];
20     } else {
21         findWhat = trees[u];
22         replaceWith = trees[v];
23     }
24
25     for(int i=0;i<trees.length;i++) {
26         if(trees[i] == findWhat) {
27             trees[i] = replaceWith;
28         }
29     }
30 }
31
32 public List<Edge> kruskal() {
33     List<Edge> mstEdges = new ArrayList<>();
34     List<Edge> allEdges = getAllEdges();
35
36     allEdges.sort((a,b) -> a.getWeight() - b.getWeight());
37

```

```

38     int trees[] = new int[numVertices];
39
40     for(int i=0;i<numVertices;i++)
41         trees[i] = i;
42
43     for(Edge e: allEdges) {
44         if(trees[e.getFrom()] != trees[e.getTo()]) {
45             mstEdges.add(e);
46
47             union(e.getFrom(), e.getTo(), trees);
48         }
49     }
50
51     return mstEdges;
52 }
```

Алгоритам на Прим: Минималното распнувачко дрво се гради почнувајќи од еден јазел и во секоја итерација се додаваат јазли на едно дрво кое постојано се доградува. Откако ќе се избере произволно почетно теме, кон него се додава атребра од множеството на ребра на темињата кои се споени со него. Во следните чекори се разгледуваат сите ребра кои не припаѓаат на веќе изграденото дрво, но излегуваат од темиња што припаѓаат на него. Реброто (односно темето) што се додава кон минималното распнувачко дрво во граffот е она реброшто го задоволува условот да не се спојува со теме што припаѓа на минималното дрво и има најмала тежина.

Ова може да се постигне со тоа како ќе се додава некој јазел, ребрата кои што излегуваат од истиот да се додаваат во приоритетна редица, каде што приоритетот би била помалата тежина. За таа цел ќе треба да се вклучи и класата за приоритетна редица:

```

1 import java.util.Queue;
2 import java.util.PriorityQueue;
```

Повторно, го додаваме методот во класата AdjacencyMatrixGraph:

```

1 public List<Edge> prim(int startVertexIndex) {
2     List<Edge> mstEdges = new ArrayList<>();
3     Queue<Edge> q = new PriorityQueue<>((a,b) -> a.getWeight() -
4                                         b.getWeight());
5
6     boolean included = new boolean[numVertices];
7
8     for(int i=0;i<numVertices;i++) {
```

```

8         included[i] = false;
9     }
10
11     included[startVertexIndex] = true;
12
13     for(int i=0;i<numVertices;i++) {
14         if(isEdge(startVertexIndex,i)) {
15             q.enqueue(new Edge(startVertexIndex, i,
16                     matrix[startVertexIndex][i]));
17         }
18     }
19
20     while(!q.isEmpty()) {
21         Edge e = q.dequeue();
22
23         if(!included[e.getTo()]) {
24             included[e.getTo()] = true;
25             mstEdges.add(e);
26             for(int i=0;i<numVertices;i++) {
27                 if(!included[i] && isEdge(e.getTo(),i)) {
28                     q.enqueue(new Edge(e.getTo(), i, matrix[e.getTo()][i]));
29                 }
30             }
31         }
32     }
33     return mstEdges;
34 }
```

Задача: Да се имплементираат алгоритмите на Крускал и Прим за граф претставен со листа на соседи.

Задача 2: Поплава

При обилни врнежи настапуваат поплави со кои се оштетуваат патиштата. Оштетувањата можат да бидат одрони, поплавени патишта или срушени мостови. Во таков случај патната мрежа во една држава станува дисконектирана и задача на соодветните служби е што е можно побрзо да обезбедат поврзаност на сите градови помеѓу себе. Оштетувањата на патиштата не се од иста категорија на сериозност, т.е. за секое од оштетувањата е потребно различно време за патот да се санира и да се направи прооден. Поради обемот на штетите и итноста на поправка, веќе е започнато воспоставување на поврзаноста и дел од патиштата

се санирани. Но, со цел оптимизација на процесот, на нас ни е доделена задачата да најдеме приоритетни патишта кои треба да се санираат со цел сите градови да бидат поврзани помеѓу себе со најмалку еден пат.

Влез:

Во првиот ред е даден бројот на градови, M.

Во вториот ред е даден бројот на патишта меѓу градовите, N.

Во третиот ред е даден бројот на веќе санирани патишта. P.

Во наредните M реда се дадени имињата на градовите.

Во следните N реда се дадени парови на имиња на градови, проследени со цел број што претставува време кое е проценето дека е потребно за да се расчисти-/санира делницата меѓу тие два града.

Во последните P реда се дадени парови од градови каде е завршена санацијата на патиштата (пред да ни го дадат проблемот на нас) и тие се веќе проодни.

Излез:

Во првиот ред се печатат два броја: бројот на патишта кои се останати да се санираат, и времето потребно за санација на тие патишта.

Потоа се печатат сите парови на градови помеѓу кои треба да бидат поправени патиштата, секој во посебен ред.

Пример:

Влез:

5

6

3

Skopje

Kumanovo

SvetiNikole

Veles

Shtip

Skopje Veles 5

Skopje Kumanovo 3

Skopje SvetiNikole 6

Kumanovo SvetiNikole 4

Shtip Veles 4

Shtip SvetiNikole 2

Skopje SvetiNikole

Shtip SvetiNikole

Kumanovo SvetiNikole

Излез:
1 4
Veles Shtip

Објаснување: Скопје, Свети Николе, Куманово и Штип се веќе поврзани со, со тоа што имасанирани патишта помеѓу нив. Велес е отсечен поради оштетувања и има две опции за поврзување: кон Скопје или кон Штип. Патот кон Штип побрзо би се санирал, па доволно е да се санира само тој пат со цел да немаме отсечени градови во државата.

Решение

Во самиот проблем е наведено дека крајната цел е да се има поврзана мрежа во која ќе има барем 1 пат од еден до друг град, со тоа што овде ни се дадени некои почетни патишта кои веќе се вклучени. Можеме да увидиме дека тоа што треба да се најде е минимално распнувачко дрво, со тоа некои ребра веќе се вклучени, значи треба да се најдат само ребрата коишто треба да се додадат за да се добие истото. Ова може да се постигне со користење на алгоритамот на Крускал, со измена во тоа што уште од почеток некои јазли веќе ќе бидат дел од исто дрво. Значи, низата во која се чува кој јазел од кое дрво е дел нема да биде иницијализирана внатре во функцијата, туку ќе биде дадена како аргумент, а во истата ќе се стави истата вредност за оние градови кои се на патиштата кои што веќе се санирани.

Најпрво ќе ја напишеме новата изменета функција за алгоритамот на Крускал, со веќе зададени вредности за тоа кој јазел од кое дрво (компонентата) е дел:

```

1 public List<Edge> adaptedKruskal(int trees[]) {
2     List<Edge> mstEdges = new ArrayList<>();
3     List<Edge> allEdges = getAllEdges();
4
5     allEdges.sort((a,b) -> a.getWeight() - b.getWeight());
6
7     for(Edge e: allEdges) {
8         if(trees[e.getFrom()] != trees[e.getTo()]) {
9             mstEdges.add(e);
10
11             union(e.getFrom(), e.getTo(), trees);
12         }
13     }
14
15     return mstEdges;

```

16 }

Следно, ќе ја напишите главната програма каде што ќе се чита влезот, соодветно ќе се креира графот и информациите за тоа кои јазли се дел од почетното поврзано дрво, и соодветно ќе се повика новата изменета функција:

```

1 import java.util.Scanner;
2 import java.util.HashMap;
3
4 public class Flood {
5     public static void main(String[] args) {
6         Scanner sc = new Scanner(System.in);
7
8         int M = sc.nextInt();
9         int N = sc.nextInt();
10        int P = sc.nextInt();
11        sc.nextLine();
12
13        HashMap<String, Integer> mapping = new HashMap<>();
14
15        AdjacencyMatrixGraph<String> cityNetwork = new
16            AdjacencyMatrixGraph<>();
17
18        int trees[] = new trees[M];
19
20        for(int i=0;i<M;i++) {
21            String city = sc.nextLine();
22            mapping.put(city, i);
23            cityNetwork.addVertex(i, city);
24            trees[i] = i+1;
25        }
26
27        for(int i=0;i<N;i++) {
28            String line = sc.nextLine();
29            String parts[] = line.split(" ");
30            cityNetwork.addEdge(mapping.get(parts[0]),
31                mapping.get(parts[1]), Integer.parseInt(parts[2]));
32        }
33
34        for(int i=0;i<P;i++) {
35            String line = sc.nextLine();
36            String parts[] = line.split(" ");

```

```

35         trees[mapping.get(parts[0])] = 0;
36         trees[mapping.get(parts[1])] = 0;
37     }
38
39     sc.close();
40
41     List<Edge> resultEdges = cityNetwork.adaptedKruskal(trees);
42
43     float suma = 0;
44
45     for(Edge e : resultEdges) {
46         suma+=e.getWeight();
47     }
48
49     System.out.println(resultEdges.size(), suma);
50
51     for(Edge e : resultEdges) {
52         System.out.println(cityNetwork.getVertex(e.getFrom()) + " " +
53                           cityNetwork.getVertex(e.getTo()));
54     }
55 }
```

Задача 3: Водовод

Дадена е водоводна мрежа во даден град. Во мрежата постои еден извор на вода и поголем број на крајни корисници. Постојат и неколку дистрибуциски јазли низ кои поминува водата и кои може да се поврзани преку повеќе патеки. Во самата мрежа можно е да има повеќе патеки низ кои водата стигнува до крајните корисници.

Мрежата е дадена преку граф од јазли (кои можат да бидат изворот, дистрибуциските јазли или крајните корисници). Јазлите се поврзани со цевки. Притоа кај секоја од цевките ја има дадено и нејзина карактеристика- пад на притисок кој се случува доколку водата помине низ неа.

За сите крајни корисници дадена е и надморската висина на која се наоѓаат. Корисниците добиваат вода доколку разликата од почетниот притисок и вкупната сума на падот на притисокот по патот до нив е позитивен број. Пример: почетен притисок е 100, водата поминува низ цевки со пад на притисок од 30, 50 и 10. Корисникот ќе добие вода само доколку се наоѓа на надморска висина помала од 10 (висина $<(100 - (30+50+10))$).

Во иницијалната конфигурација на мрежата, сите корисници добиваат вода. Но, во мрежата се случуваат неколку дефекти, кои го зголемуваат падот на притисокот кај дадени цевки за неколку пати.

За дадена иницијална мрежка, како и локација и сериозност на дефекти, ваша задача е да ги испечатите крајните корисници кои нема да добијат вода во таква конфигурација на системот.

Влез: На влез прво е даден бројот на јазли во мрежата N

Во наредните N линии се даени бројот на јазол, па неговата надморска висина. Изворот е секогаш прв со реден број 0. Бројот до изворот е притисокот во мрежата на ниво на изворот. Дистрибуциските јазли се на надморска висина 0

На крај следат крајните корисници, до кои после редниот број е дадена и нивната надморска висина

Потоа следи бројот M, кој го означува бројот на цевки во мрежата. Во наредните M редови се дадени три броеви: почетно теме на цевката, крајно теме на цевката и пад на притисок на таа цевка

На крај следи бројот P, кој го означува бројот на дефекти

Во наредните P редици се дадени цевките кај кои има дефекти, на следен начин: почетно теме на цевката, крајно теме на цевката и уште еден природен број, кој кажува колку пати сега е зголемен падот на притисокот од иницијалниот.

Излез:

На излез се печатат броевите на јазлите на крајните корисници кај кои нема да има вода

Пример влез:

5
0 100
1 0
2 0
3 10
4 20
5
0 1 5
0 2 10
1 2 30
1 3 10
2 4 20
1
2 4 5

Излез:

4

Објаснување:

Мрежата има 5 јазли: Извор со почетен притисок од 100. Два дистрибуциски јазли (1 и 2) и два крајни корисници- 3 на надморска висина од 10 и 4 на надморска висина од 20

Во мрежата има 5 цевки: од 0 до 1 со пад на притисок од 5
од 0 до 2 со пад на притисок од 10
од 1 до 2 со пад на притисок од 30
од 1 до 3 со пад на притисок од 10
од 2 до 4 со пад на притисок од 20

Во мрежата се случуваат следните дефекти (само еден): На цевката помеѓу јазлите 2 и 4 падот на притисок се зголемува 5 пати од првичниот. Што значи дека оваа цевка сега ќе има пад на притисок од $5*20=100$

Со тоа, минималната патека до корисникот 4 ќе го има следниот пад на притисок: $10+100=110$. Иницијалниот притисок бил 100, корисникот се наоѓа на надморска висина од 20, што значи дека до него нема да стигне вода.

Забелешка:

Притисоците кај дистрибуциските јазли не се сумираат, се смета дека притисокот оди по патот со најмал отпор. Дополнително, размислете за следните нешта:

Дали некој од добро познатите алгоритми за графови го решава проблемот?

Дали е важна насоченоста на цевките?

Решение

Најпрво треба да се прочита влезот и соодветно да се креира графот. Дополнително треба да се ажурираат тежините според влезот, за тие цевки на кои што има дефект. За ова треба да има и соодветна функција во граф класата која што ќе може да ја ажурира тежината на реброто од еден до друг јазел. Кажано ни е дека притисокот оди по патот со најмал отпор, што значи дека треба да се најдат таквите патишта од изворот до другите јазли, најважно до јазлите коишто претставуваат крајни корисници. Ова може да се направи со помош на алгоритамот на Дијикстра. Потоа за крајните корисници може да се провери дали разликата на почетниот притисок (притисокот на изворот) со минималниот пад на притисок од изворот до тој корисник, и на крај одземена и надморската висина на корисникот, ќе резултира во поголемо од 0. Доколку не е поголемо од 0, тој корисник нема да има вода.

¹ `import java.util.Scanner;`
² `import java.util.HashMap;`

³

```
4  public class Plumbing {
5      public static void main(String[] args) {
6          Scanner sc = new Scanner(System.in);
7
8          int n = sc.nextInt();
9
10         AdjacencyListGraph<Integer> waterNetwork = new
11             AdjacencyListGraph<>();
12         int values[] = new int[n];
13
14         for(int i=0;i<n;i++) {
15             int ind = sc.nextInt();
16             int val = sc.nextInt();
17
18             waterNetwork.addVertex(ind);
19             values[ind] = val;
20         }
21
22         int m = sc.nextInt();
23
24         for(int i=0;i<m;i++) {
25             waterNetwork.addEdge(sc.nextInt(), sc.nextInt(), sc.nextInt());
26         }
27
28         int p = sc.nextInt();
29
30         for(int i=0;i<p;i++) {
31             int startVertex = sc.nextInt();
32             int endVertex = sc.nextInt();
33             waterNetwork.adjustEdgeWeight(startVertex, endVertex,
34                 waterNetwork.getNeighbors(startVertex).get(endVertex));
35         }
36
37         sc.close();
38
39         Map<Integer, Integer> shortestPaths = waterNetwork.shortestPath(0);
40
41         for(int i=1;i<n;i++) {
42             if(values[i] != 0) {
43                 if(values[0] - (shortestPaths.get(i) + values[i]) <= 0) {
44                     System.out.print(i + " ");
45                 }
46             }
47         }
48     }
49 }
```

```

43     }
44   }
45 }
46   }
47 }
48 }
```

7.10 Задачи за вежбање

Задача 1: Вкупен број на патишта

Да се испечати вкупниот број на патишта со должина N кои почнуваат од некое фиксно теме V во неориентиран нетежински граф. **Влез:** Во првиот ред е даден бројот на јазли, во вториот ред е даден бројот на ребра, а потоа во следните редови се дадени ребрата во графот. Во претпоследниот ред се дадени темето V и во последниот ред, должината N на патиштата. **Излез:** Испечатете го вкупниот број на патишта со должина N . Јазлите во патот можат да се повторуваат.

Пример:

Влез:

4 5 0 1 1 2 2 3 0 2 1 3 3 2

Излез: 6

Во дадениот пример, постојат 6 патишта - 3 2 1, 3 2 3, 3 2 0, 3 1 0, 3 1 2 и 3 1 3, кои почнуваат од јазолот 3 и имаат должина 2.

Задача 2: Компоненти на сврзаност

За дадено теме да се прикажат сите темиња кои припаѓаат на иста компонента на сврзаност во даден граф. Темињата припаѓаат на иста компонента на сврзаност во графот доколку постои пат помеѓу нив. На влез прво се внесува бројот на темиња, потоа за секое теме се пишува бројот на темиња со кои е поврзано и индексите на темињата. На крај се внесува индексот на темето за кое ќе се бара решението.

Пример:

Влез:

10

1 5
3 2 4 5
3 1 3 4
3 2 4 5

4 1 2 3 5

4 0 1 3 4

2 7 8

2 6 8

2 6 7

0

4

Излез: 0 1 2 3 4 5

Задача 3: Дедо Мраз на распуст

Во земјата Лапонија живее Дедо Мраз. Во слободното време кога не е Нова Година, додека џуцињата си работат на играчките за следната година, Дедо Мраз има хоби. Тој сака да одгледува рибички. Но тој своите рибички ги одгледува во природни езера. Езерата се меѓусебно поврзани со рекички, и рекичките течат од едно езерце до друго. Рибите од едно езеро слободно можат преку рекичките да отидат во друго езеро. Секоја пролет дедо мраз сака да прави порибување на езерцата со нови рибички. Ваша задача е да му кажете на Дедо Мраз доколку тој пушти нови рибички во езерцето X, во колку други езерца ќе можат рибичките сами да стигнат, а со тоа да нема потреба тој самиот да ги порибува тие езерца.

Влез: Во првата линија од влезот е даден број $N < 15$ бројот на езерца. Во втората линија е даден број $U < 20$ бројот на реки меѓу езерцата. Во следните U линии се дадени парови од 2 броја R и Q , што значи постои рекичка која тече од R до Q , каде R и Q се броеви на езерцата. Во последната линија е даден број L , во кое езерце Дедо Мраз ќе ги пушти рибичките.

Излез: Се испишува бројот, колку езерца освен почетното ќе бидат порибени.

Пример:

Влез: 11 19 3 3 7 8 7 3 1 7 0 0 7 2 6 3 2 0 0 9 6 10 1 2 2 8 5 7 4 3 10 4 3 9 7 10
9 4 4 10 7

Излез: 7

Задача 4: Патарини

Бојан сака да го посети Берлин. Но, Бојан има ограничени средства, па истиот мора да избира по кој пат е најповољно да се движи. За таа цел, тој изнајмил автомобил и купил мапа на Европа. На неа има ‘ N ’ градови. Помеѓу секои два града може да има најмногу еден директен транспортен пат (патот е двонасочен). За секој од тие патишта, Бојан знае колкав број патарини треба да се поминат при изминување на соодветниот пат. Можете ли да му помогнете на Бојан, и

да ја откриете рутата од Скопје до Берлин, на кој треба да се поминат најмал можен број на патарини?

Влез: Во првата линија е запишан еден цел број ‘N‘ ($2 \leq N \leq 1000$), кој го означува бројот на градови. Секој од градовите е означен со единствен број id - кој се движи од 1 до N. Во втората линија се запишани два цели броја ‘A‘ и ‘B‘ ($1 \leq A, B \leq N$), кои ги означуваат id-ата на градовите Скопје (A) и Берлин (B). Во третата линија е запишан еден цел број ‘M‘ ($N \leq M \leq 1500$), кој го означува бројот на директни патишта. Во секој од следните M редови се запишани по три цели броја ‘X‘, ‘Y‘ ($1 \leq X, Y \leq N$) и ‘K‘ ($1 \leq K \leq 1000$), кои означуваат дека постои двонасочен пат од X до Y, и на истиот се наоѓаат K патарини.

Излез: Отпечатете го бараниот најмал можен број на патарини.

Забелешка: Дозволено е користење на готови класи од Java API.

Пример:

Влез:

```
10
7 8
13
1 2 387
2 3 831
4 1 820
5 4 204
5 6 304
4 7 381
5 8 238
9 5 214
9 10 126
8 6 709
4 3 3
6 7 732
3 1 488
```

Излез:

```
823
```

Литература

- [1] A. D. Watt and F. D. Brown, *Java Collections: An Introduction to Abstract Data Types, Data Structures and Algorithms.* Wiley, 2001.
- [2] B. J. Evans, J. Clark, and D. Flanagan, *Java in a Nutshell.* "O'Reilly Media, Inc. 2023.
- [3] R. Lafore, *Data structures and algorithms in Java.* Sams publishing, 2017.