## 3. Магацини и Редови

Во ова поглавје се опишани три податочни структури: магацин (stack), ред и ред со приоритет. На почетокот се опишуваат разликите на овие струкури во однос на полињата, потоа се опишува секоја од овие структури. Во последниот дел се разгледува операција во која магацинот има посебна улога: парсирање на аритметички изрази.

Постојат значителни разлики помеѓу структурите на податоците и алгоритмите кои беа разгледувани во претходните поглавја наспроти оние кои се разгледуваат овде. Предмет на дискусија ќе бидат три основни разлики, пред да се премине на детално разгледување на секоја од структурите.

Полињата – структури на податоци кои се разгледуваа досега – како и многу други структури на податоци кои се изучуваат понатаму во оваа книга (поврзани листи, дрва и т.н.) се соодветни за типовите на податоци кои може да се сретнат во апликациите за работа со бази на податоци. Типично се употребуваат за персонални податоци, инвентура, финансиски податоци и слично – податоци кои кореспондираат на објекти или активности од реалниот свет.Овие структури го олеснуваат пристапот до податоците: го прават едноставно внесувањето, бришењето и пребарувањето за одредени елементи.

Структурите и алгоритмите кои ќе бидат разгледани во ова поглавје, од друга страна пак, се почесто употребувани како програмерски алатки. Во основа тоа се повеќе концептуални помагала одошто комплетни уреди за чување на податоци. Нивниот животен век е типично пократок во споредба со структурите од типот на бази на податоци. Се креираат и користат за извршување на одредена задача во текот на работата на програмата, а кога задачата е извршена тие се отстрануваат.

Во поле може да се пристапи до секој елемент, било веднаш – доколку е познат неговиот индекс – или пак со пребарување на секвенца на ќелии се додека не биде пронајден. Во податочните структури во ова поглавје пак пристапот е ограничен: само еден елемент може да биде прочитан или одстранет во одреден моомент.

Интерфејсот на овие структури е дизајниран да го наметне ваквиот ограничен пристап. Пристапот кон други елементи (теоретски) не е дозволен.

Магацините, редовите и редовите со приоритети се поапстрактни ентитети од полињата и од многу други структури за чување на податоци. Примарно се дефинирани со нивниот интерфејс: дозволените операции кои можат да бидат спроведени над нив. Основниот механизам кој се користи за нивна имплементација вообичаено е невидлив за нивниот корисник.

Механизмот за реализација на магацинот, на пример, може да биде поле, како што е тоа покажано во ова поглавје, или пак може да биде и поврзана листа. Основниот механизам пак за редови на приоритети може да биде поле или пак посебен вид на дрво наречен хип (heap). За имплементирање на една структура на податоци со друга ќе се дискутира и во делот за Апстрактните Типови на Податоци (Abstract Data Types – ADTs) во поглавјето бр. 5 „Поврзани листи“.

### 3.1 Магацини

Магацинот дозволува пристап само до еден податочен елемент: елементот кој е последен внесен. Ако се избрише овој елемент, може да се пристапи до претпоследниот внесен елемент, и т.н. Оваа особина е корисна во многу програмски ситуации. Во овој дел се опишува употреба на магацин за проверка дали малите, средните и големите загради се балансирани во изворен код на програма. На крајот на поглавјето се опишува улогата на магацинот за парсирање (анализирање) на аритметички изрази како 3\*(4+5).

Магацинот е исто така помошно средство за алгоритми кои се применуваат кај некои комплексни податочни структури. Во поглавјето „Бинарни дрва“ се користи за изминување на јазлите на дрвото. Во поглавјето „Графови“ се применува за пребарување на темињата на граф (техника која може да се користи за наоѓање пат низ лавиринт).

Повеќето микропроцесори користат архитектура базирана на магацин. Кога се повикува метод неговата повратна адреса и аргументи се вметнуваат во магацин, а кога ќе заврши методот се вадат од магацин. Операциите со магацини се вградени во микропроцесорот.

Некои постари калкулатори користеле архитектура базирана на магацин. Наместо внесување на аритметички изрази со загради, меѓурезултатите се внесуваат во магацин. Повеќе за овој приод се зборува во делот за парсирање на аритметички изрази во последната секција од ова поглавје.

### 3.2 Java код за магацин

Во следниот код се имплементира магацин со класата StackX.

// stack.java

// demonstrates stacks

// to run this program: C>java StackApp

////////////////////////////////////////////////////////////////

class StackX

{

private int maxSize; // size of stack array

private long[] stackArray;

private int top; // top of stack

//--------------------------------------------------------------

public StackX(int s) // constructor

{

maxSize = s; // set array size

stackArray = new long[maxSize]; // create array

top = -1; // no items yet

}

//--------------------------------------------------------------

public void push(long j) // put item on top of stack

{

stackArray[++top] = j; // increment top, insert item

}

//--------------------------------------------------------------

public long pop() // take item from top of stack

{

return stackArray[top--]; // access item, decrement top

}

//--------------------------------------------------------------

public long peek() // peek at top of stack

{

return stackArray[top];

}

//--------------------------------------------------------------

public boolean isEmpty() // true if stack is empty

{

return (top == -1);

}

//--------------------------------------------------------------

public boolean isFull() // true if stack is full

{

return (top == maxSize-1);

}

//--------------------------------------------------------------

} // end class StackX

////////////////////////////////////////////////////////////////

class StackApp

{

public static void main(String[] args)

{

StackX theStack = new StackX(10); // make new stack

theStack.push(20); // push items onto stack

theStack.push(40);

theStack.push(60);

theStack.push(80);

while( !theStack.isEmpty() ) // until it’s empty,

{ // delete item from stack

long value = theStack.pop();

System.out.print(value); // display it

System.out.print(“ “);

} // end while

System.out.println(“”);

} // end main()

} // end class StackApp

////////////////////////////////////////////////////////////////

Методот main() во класата StackApp креира магацин за 10 елементи, внесува 4 елементи во магацинот и потоа ги печати, бришејќи еден по еден елемент се додека магацинот не се испразни.

Излезот од програмата е:

80 60 40 20

Се забележува дека редоследот на податоците е обратен. Бидејќи последниот елемент кој е внесен прв се испишува, 80 се јавува прв во излезната листа.

#### 3.2.1 Методи на класата StackX

Конструкторот креира нов магацин со големина специфицирана со аргументот. Податочни членови на класата за магацинот се променлива за максималната големина (големината на полето), самото поле, променлива top која го чува индексот на елементот на врвот на магацинот. (Потребно е да се специфицира големината на магацинот само затоа што се имплементира со поле. Ако се имплементира со поврзана листа не е потребна спецификација на големината.)

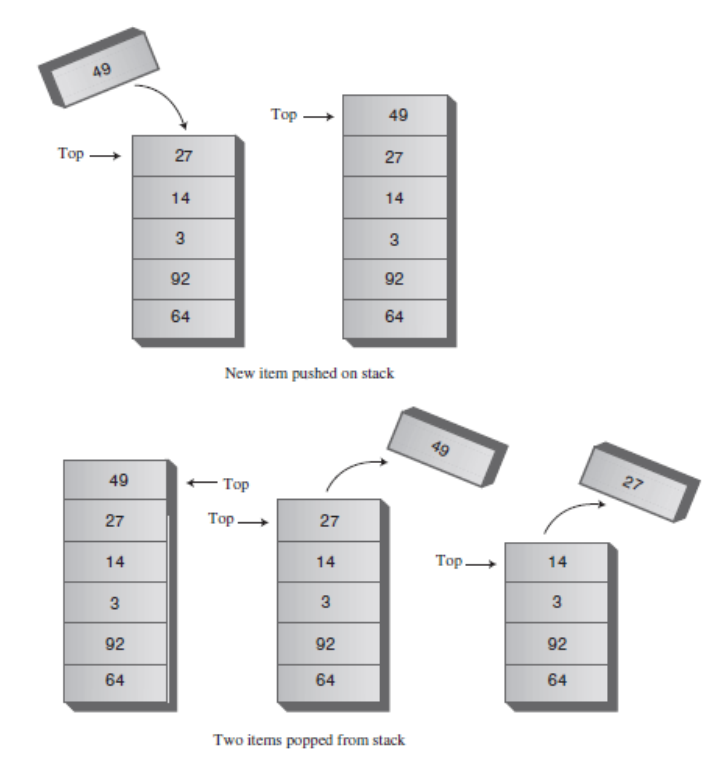
Методот push() го инкрементира top така да покажува на позицијата над претходниот top и таму го сместува елементот. Забележете дека top се инкрементира пред да се внесе елементот.

Методот pop() ја враќа вредноста на врвот и потоа го декрементира top. Ова всушност го брише елементот на врвот од магацинот, на таа вредност не може да и се пристапи иако останува во полето (се додека не се внесе внесе друг елемент во ќелијата).

Методот peek() ја враќа вредноста на врвот на магацинот, без да го менува магацинот.

Методите isEmpty() и isFull() враќаат true ако магацинот е празен или полн, соодветно. Променливата top се наоѓа на позиција -1 кога магацинот е празен и на позиција maxSize-1 кога магацинот е полн.

Сликата 3.1 покажува како работи магацинот.



Слика 3.1 Операции кои ги извршуваат методите на класата StackX

Во продолжение се опишува справување со грешките кога се внесува елемент во магацин кој е полн или кога се брише елемент од празен магацин. Во претходнта програма справувањето со овие гранични случаи е оставено на програмерот. Треба да се провери дали магацинот е полн пред внесување елемент:

if( !theStack.isFull() )

insert(item);

else

System.out.print(“Can’t insert, stack is full”);

Исто така со повикување на isEmpty() во main() пред да се повика pop() се проверува дали магацинот е празен пред да се изврши бришење на елемент.

#### 3.2.2 Пример за примена на магацин: Испишување на збор во обратен редослед

Кога програмот се извршува бара корисникот да внесе збор а по притискање на Enter се испишува зборот во обратен редослед.

Магацин се користи за превртување на буквите. Прво, карактерите се издвојуваат еден по еден од влезниот стринг и се внесуваат во магацинот. Потоа тие се бришат од магацинот и се печатат. Заради карактеристиката „последен внесен прв избришан“ (Last-In-First-Out) магацинот ги превртува карактерите. Во продолжение е даден кодот на програмата за превртување на зборови

// reverse.java

// stack used to reverse a string

// to run this program: C>java ReverseApp

import java.io.\*; // for I/O

////////////////////////////////////////////////////////////////

class StackX

{

private int maxSize;

private char[] stackArray;

private int top;

//--------------------------------------------------------------

public StackX(int max) // constructor

{

maxSize = max;

stackArray = new char[maxSize];

top = -1;

}

//--------------------------------------------------------------

public void push(char j) // put item on top of stack

{

stackArray[++top] = j;

}

//--------------------------------------------------------------

public char pop() // take item from top of stack

{

return stackArray[top--];

}

//--------------------------------------------------------------

public char peek() // peek at top of stack

{

return stackArray[top];

}

//--------------------------------------------------------------

public boolean isEmpty() // true if stack is empty

{

return (top == -1);

}

//--------------------------------------------------------------

} // end class StackX

////////////////////////////////////////////////////////////////

class Reverser

{

private String input; // input string

private String output; // output string

//--------------------------------------------------------------

public Reverser(String in) // constructor

{ input = in; }

//--------------------------------------------------------------

public String doRev() // reverse the string

{

int stackSize = input.length(); // get max stack size

StackX theStack = new StackX(stackSize); // make stack

for(int j=0; j<input.length(); j++)

{

char ch = input.charAt(j); // get a char from input

theStack.push(ch); // push it

}

output = “”;

while( !theStack.isEmpty() )

{

char ch = theStack.pop(); // pop a char,

output = output + ch; // append to output

}

return output;

} // end doRev()

//--------------------------------------------------------------

} // end class Reverser

////////////////////////////////////////////////////////////////

class ReverseApp

{

public static void main(String[] args) throws IOException

{

String input, output;

while(true)

{

System.out.print(“Enter a string: “);

System.out.flush();

input = getString(); // read a string from kbd

if( input.equals(“”) ) // quit if [Enter]

break;

// make a Reverser

Reverser theReverser = new Reverser(input);

output = theReverser.doRev(); // use it

System.out.println(“Reversed: “ + output);

} // end while

} // end main()

//--------------------------------------------------------------

public static String getString() throws IOException

{

InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader br = new BufferedReader(isr);

String s = br.readLine();

return s;

}

//--------------------------------------------------------------

} // end class ReverseApp

////////////////////////////////////////////////////////////////

Класата Reverser овозможува превртување на стринг. Најважна компонента на класата е методот doRev(), кој го изведува превртувањето, користејќи магацин. Магацинот се креира во методот doRev() и големината на магацинот се одредува според должината на влезниот стринг.

Во main() корисникот внесува стринг, креира објект Reverser со стрингот како аргумент на конструкторот, го повикува методот doRev() и ја печати вратената вредност, која го претставува превртениот стринг. Во продолжение е дадена интеракција со програмата:

Enter a string: part

Reversed: trap

Enter a string:

#### 3.2.3 Пример за примена на магацин: Спарување на загради

Магацините често се применуваат за парсирање на одредени видови на текстуални стригови. Стрингови обично се линии на код во некој програмски јазик, а програмите кои ги парсираат се преведувачи (compilers).

Во изразите напишани во програмски код се сретнуваат големи загради {} (braces), средни загради [] (brackets) и мали загради () (parentheses). Подоцна отворените загради треба да се затворат пред оние кои се јавуваат порано. На пример:

c[d] // correct

a{b[c]d}e // correct

a{b(c]d}e // not correct; ] doesn’t match (

a[b{c}d]e} // not correct; nothing matches final }

a{b(c) // not correct; nothing matches opening {

Програмата за спарување на загради работи на тој начин што чита карактери од стрингот и отворените загради ги внесува во магацин. Кога ќе прочита затворена заграда ја вади отворената заграда од врвот на магацинот и ја споредува со прочитаната заграда. Ако заградите не се од ист вид се јавува грешка. Исто така, ако нема отворени загради во магацинот кои се спаруваат со затворените или ако постои во магацинот заграда која останува неспарена, се јавува грешка.

Нека е даден коректен стринг

a{b(c[d]e)f}

Во табела 3.1 е прикажано како изгледа магацинот по секое читање на карактер од стрингот. Втората колона на табелата ја прикажува содржината на магацинот при читање од дното (најлево) до врвот (најдесно) на магацинот.

Прочитаните карактери кои не се загради не се внесуваат во магацинот туку се игнорираат.

Табела 3.1 Содржина на магацинот кај програмата за спарување на загради

|  |  |
| --- | --- |
| Прочитан карактер | Содржина на магацинот |
| a |  |
| { | { |
| b | { |
| ( | {( |
| c | {( |
| [ | {([ |
| d | {([ |
| ] | {( |
| e | {( |
| ) | { |
| f | { |
| } |  |

Овој приод работи бидејќи парот на загради кои отворени последни треба први да се затворат. Тоа е во согласност со особината на магацинот Last-In-First-Out.

Во продолжение е даден кодот на програмата за парсирање brackets.java. Методот кој го изведува парсирањето се нарекува check() и припаѓа на класата наречена BracketChecker.

// brackets.java

// stacks used to check matching brackets

// to run this program: C>java BracketsApp

////////////////////////////////////////////////////////////////

import java.io.\*; // for I/O

class StackX

{

private int maxSize;

private char[] stackArray;

private int top;

//--------------------------------------------------------------

public StackX(int s) // constructor

{

maxSize = s;

stackArray = new char[maxSize];

top = -1;

}

//--------------------------------------------------------------

public void push(char j) // put item on top of stack

{

stackArray[++top] = j;

}

//--------------------------------------------------------------

public char pop() // take item from top of stack

{

return stackArray[top--];

}

//--------------------------------------------------------------

public char peek() // peek at top of stack

{

return stackArray[top];

}

//--------------------------------------------------------------

public boolean isEmpty() // true if stack is empty

{

return (top == -1);

}

//--------------------------------------------------------------

} // end class StackX

////////////////////////////////////////////////////////////////

class BracketChecker

{

private String input; // input string

//--------------------------------------------------------------

public BracketChecker(String in) // constructor

{ input = in; }

//--------------------------------------------------------------

public void check()

{

int stackSize = input.length(); // get max stack size

StackX theStack = new StackX(stackSize); // make stack

for(int j=0; j<input.length(); j++) // get chars in turn

{

char ch = input.charAt(j); // get char

switch(ch)

{

case ‘{‘: // opening symbols

case ‘[‘:

case ‘(‘:

theStack.push(ch); // push them

break;

case ‘}’: // closing symbols

case ‘]’:

case ‘)’:

if( !theStack.isEmpty() ) // if stack not empty,

{

char chx = theStack.pop(); // pop and check

if( (ch==’}’ && chx!=’{‘) ||

(ch==’]’ && chx!=’[‘) ||

(ch==’)’ && chx!=’(‘) )

System.out.println(“Error: “+ch+” at “+j);

}

else // prematurely empty

System.out.println(“Error: “+ch+” at “+j);

break;

default: // no action on other characters

break;

} // end switch

} // end for

// at this point, all characters have been processed

if( !theStack.isEmpty() )

System.out.println(“Error: missing right delimiter”);

} // end check()

//--------------------------------------------------------------

} // end class BracketChecker

////////////////////////////////////////////////////////////////

class BracketsApp

{

public static void main(String[] args) throws IOException

{

String input;

while(true)

{

System.out.print(“Enter string containing delimiters: “);

System.out.flush();

input = getString(); // read a string from kbd

if( input.equals(“”) ) // quit if [Enter]

break;

// make a BracketChecker

BracketChecker theChecker = new BracketChecker(input);

theChecker.check(); // check brackets

} // end while

} // end main()

//--------------------------------------------------------------

public static String getString() throws IOException

{

InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader br = new BufferedReader(isr);

String s = br.readLine();

return s;

}

//--------------------------------------------------------------

} // end class BracketsApp

////////////////////////////////////////////////////////////////

Методот main() од класата BracketsApp чита линија текст внесен од корисникот, креира објект BracketChecker со овој текст како аргумент и потоа го повикува методот check() со објектот BracketChecker. Ако има грешки во изразот методот check() го испишува бројот на карактерот каде се јавува грешка. На пример за изразот

a{b(c]d}e

се испишува

Error: ] at 5

Во спротивно синтаксата е коректна.

#### 3.2.4 Ефикасност на магацините

Кај магацините може да се внесуваат елементи (операција push) или бришат елементи (операција pop) за константно време О(1). Времето на извршување не зависи од бројот на елементи во магацинот и според тоа операциите се брзи. Не се потребни споредби или преместувања на елементи.

### 3.2 Редови

Зборот ред (queue) на англиски означува ред на чекање. Во комјутерската наука ред е податочна структура која е слична на магацин но кај редот првиот внесен елемент прв го напушта редот (First-In-First-Out, FIFO). Кај магацинот последниот внесен елемент прв го напушта магацинот (LIFO). На слика 3.2 е претставен илустративен пример за ред



Слика 3.2 Ред од луѓе

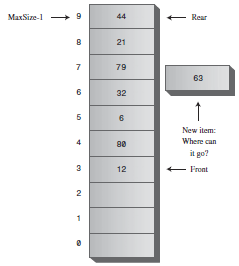
Редовита исто како магацините се користат како програмерска алатка. Во поглавјето XXX ќе биде претставен пример во кој ред ќе биде упторебен за пребарување на граф. Редовите исто така се користат и за моделирање на ситуации од реалниот живот како што се на пример луѓето кои чекаат на ред на шалтер, авиони кои чекаат на ред за полетување или пак пакети на податоци кои чекаат да бидат пренесени преку Интернет.

Постојат разновидни редови кои во позадина си ја вршат својата работа во компјутерскиот (или мрежен) оперативен систем. Редовите исто така ги помнат податоците за притиснатите тастери на вашата тастатура.. На овој начин на пример, доколку доколку користите текст процесор а вашиот компјутер е моментално зафатен правејќи нешто друго кога вие притискате тастер, отчуканиот знак нема да биде загубен; тој чека во редот се додека текст процесорот најде време да го прочита. Употребата на редовите гарантира дека отчуканите тастери ќе останат во ред се додека не бидат обработени.

### 3.3 Кружен ред

Кога ќе се внесе нов член во редот Предниот показател се поместува нанапред, кон повисоките броеви во полето. Кога пак ќе се острани еден член, Задниот показател исто така се поместува нанапред. Ваквиот аранжман е можеби спротивен на интуицијата, бидејќи сите луѓе во редот пред киното се поместуваат нанапред кон челото, како что еден човек го напушта редот. Секогаш кога ќе се избрише еден член во редот, би можело да се поместат сите членови во редот но тоа не би било многу ефикасно. Наместо тоа, сите членови се задржуваат на своите места, а се поместуваат само почетокот и крајот на редот.

Проблемот со ваквата организација е што набргу крајот на редот ќе се најде на крајот на полето (највисокиот индекс). Дури и ако има празни ќелии во почетокот на полето (затоа што се одстранети со REM, сепак не може да се внесе нов елемент бидејќи Задниот показател не може да оди понатаму. Или пак може? Ваквата ситуација е прикажана на сликата 4.8 XXX.

Слика 4.8 Задниот показател е на крајот на полето

#### 3.3.1 Обвиткување наоколу

За да се избегне проблемот со неможноста да се вметнат нови членови во редот дури и кога тој не е полн, Предниот и Задниот показател се „обвиткуваат наоколу“ , кон почетокот на полето. Резултат на ова е *Кружен Ред* (понекогаш се нарекуваат и *прстенести бафери*).

Може да се види дека обвиткувањето функционира: внесете доволно елементи за да Задниот показател биде доведен до врвот на полето (Индекс 9). Потоа, отсранете неколку елементи од почетокот на полето, а потоа внесете нов елемент. Ќе се види дека Задниот показател се обвиткува наоколи од Индекс 9 кон Индекс 0; новиот елемент ќе биде внесен таму. Ваквата ситуација е покажана на сликата 4.9.

Ако се внесат неколку нови елементи, Задниот показател се поместува нагоре, како што и се очекува. Забележително е дека Задниот показател се обвитал наоколу и сега е под Предниот показател, што е обратна ситуација во однос на оригиналната.Тоа би можело да се нарече и прекината секвенца (*broken sequence*). Елементите во редот се поставени во две различни секвенци во однос на полето.

Слика 4.9 Задниот показател е обвитан наоколу

Доколку се избришат доволно елементи и Предниот показател исто така се обвиткува наоколу, така што се добива оригиналната ситуација, при што Предниот е пред Задниот показател. Сите елементи се во единствен, континуиран редослед.

#### 3.3.2 JAVA код за Ред

Во рамките на queue.java програмата е содржана и класата на редот (Queue class) со примена на методите insert(), remove(), peek(),isFull(), isEmpty(), и size().

Програмата main() генерира ред од пет ќелии, внесува четири елементи, отстранува три и пак додава четири нови елементи. При шестото вметнување сеактивира особината „обвиткување наоколу“, по што сите елементи се најпрвин остранети а потоа и прикажани. Излезот изгледа вака:

40 50 60 70 80

Листингот 4.4 XXX го прикажува изгледот на queu.java програмот:

ЛИСТИНГ 4.4 The queue.java Program

// queue.java

// demonstrates queue

// to run this program: C>java QueueApp

////////////////////////////////////////////////////////////////

class Queue

{

private int maxSize;

private long[] queArray;

private int front;

private int rear;

private int nItems;

//--------------------------------------------------------------

public Queue(int s) // constructor

{

maxSize = s;

queArray = new long[maxSize];

front = 0;

rear = -1;

nItems = 0;

}

//--------------------------------------------------------------

public void insert(long j) // put item at rear of queue

{

If (rear == maxSize-1) // deal with wraparound

rear = -1;

queArray[++rear] = j; // increment rear and insert

nItems++; // one more item

}

//--------------------------------------------------------------

public long remove() // take item from front of queue

{

long temp = queArray[front++]; // get value and incr front

if(front == maxSize) // deal with wraparound

front = 0;

nItems--; // one less item

return temp;

}

//--------------------------------------------------------------

public long peekFront() // peek at front of queue

{

138 **CHAPTER 4** Stacks and Queues

return queArray[front];

}

//--------------------------------------------------------------

public boolean isEmpty() // true if queue is empty

{

return (nItems==0);

}

//--------------------------------------------------------------

public boolean isFull() // true if queue is full

{

return (nItems==maxSize);

}

//--------------------------------------------------------------

public int size() // number of items in queue

{

return nItems;

}

//--------------------------------------------------------------

} // end class Queue

////////////////////////////////////////////////////////////////

class QueueApp

{

public static void main(String[] args)

{

Queue theQueue = new Queue(5); // queue holds 5 items

theQueue.insert(10); // insert 4 items

theQueue.insert(20);

theQueue.insert(30);

theQueue.insert(40);

theQueue.remove(); // remove 3 items

theQueue.remove(); // (10, 20, 30)

theQueue.remove();

theQueue.insert(50); // insert 4 more items

theQueue.insert(60); // (wraps around)

theQueue.insert(70);

theQueue.insert(80);

while( !theQueue.isEmpty() ) // remove and display

{ // all items

long n = theQueue.remove(); // (40, 50, 60, 70, 80)

System.out.print(n);

System.out.print(“ “);

}

System.out.println(“”);

} // end main()

} // end class QueueApp

Избравме пристап со кој во полињата на класата Queu (ред) не само што ги вклучуваат предниот и Задниот показател, туку и бројот на елементи кои се моментално присутни во редот: nItems. Постојат некои редови кои не го користат ова поле, но на оваа алтернатива ќе се навратиме подоцна.

#### 3.3.2.1 insert() Mетода

Insert()методата претпоставува дека редот не е полн. Тоа не е прикажано во main(), но нормалноби било да се повика insert() веднаш по повикувањето на isFull() и добивањето на false() повратниот резултат. (Вообичаена практика е да се воспостави контрола на полноста во insert() рутината и да се предизвика појава на исклучок доколку е направен обидот да се уфрлат нови елементи во полн ред.)

Нормално е дека вметнувањето нов елемент подразбира и зголемување на Задниот показател и вметнувањето на ќелија кон која сега покажува Задниотт показател. Сепак, доколку Заднинскиот показател укажува на врвот на полето, при максимална вредност на maxSize-1, тогаш тој мора да се обвитка околу дното на полето пред да настане вметнувањето на ќелијата. Тоа се постигнува со поставување на Задниот показател (rear) на вредност -1, така што кога ќе настане инкрементот, Задниот показател ќе добие вредност 0 односно дното на полето. Конечно, настанува и инкрементацијата на nItems.

#### 3.3.2.2 remove() Mетода

Методата remove() подразбира дека редот не е празен. Потребно е претходно да се повика isEmpty() процедурата за да се обезбеди вистинитоста пред да биде повикана remove() процедурата, или пак оваа контрола на грешка да се вгради во remove().

Методата секогаш се спроведува така што најпрвин се обезбедува вредноста на Предниот показател и потоа се инкрементир неговата вредност. Сепак, доколку инкрементацијата на Предниот показател ја поставува неговата вредност надвор од полето, тогаш тој мор да биде обвиткан околу и поставен на 0. Повратната вредност привремено се меморира се додека не се испита оваа можност. Конечно, се пристапува на декрементација (намалување) на nItems.

#### 3.3.2.3 peek() Mетода

Peek() методата е многу едноставна: таа ја дава вредноста на Предниот показател. Некои имплементации дозволуваат увид и во вредноста наЗадниот показател; ваквите рутини вообичаено се среќаваат како peekFront ili peek Rear, или пак само како front() и rear().

#### 3.3.2.4 isEmpty(), isFull() и size() Mетоди

Овие методи се базираат на nItems полето, соодветно проверувајќи ги неговите нулти вредности, maxSize вредноста или пак враќајки ја неговата вредност.

#### 3.3.3 Имплементација без пребројување на елементите

Вклучувањето на полето nItems во класата Queu (Ред), предизвикува мало оптоварување во однос на insert() и remove() рутините, така што тие мораат да дадат вредности кои се соодветно зголемени односно намалени. Ова можеби не наликува на некаков посебен товар, но доколку се работи со огромни количини на внесувања и бришења, може сериозно да влијае врз ефикасноста.

Според тоа, некои имплементации на редовите функционираат без бројачот на елементи и се базираат на полињата за Предниот и Задниот показател, со цел да се отцени дали се редовите се празни или полни и/или колку елементи има во нив. Кога е тоа спроведено, isEmpty(), isFull() и size() рутините стануваат изненадувачки компликувани бидејќи секвенцата на елементи може да биде прекината или континуирана, како што межевме да се увериме.

При тоа се појавува и чуден проблем. Поентерите кон Предниот и Задниот показател подразбираат одредени позиции кога е редот полн, тно тие можат да ги завземат истите позиции и кога редот е празен. Во таква ситуација еден ред може да биде посматран, во исто време, како да е иполн и празен.

Ваквиот проблем може да се разреши така што полето ќе се оформи на начин да биде за една ќелија поголемо од максималниот број на елементи кои ќе бидат поставени во него. Листингот 4.5 ја покажува примената на Queue класата која го вклучува пристапот без броење на елементите. Ваквата класа го користи приодот без пребројување.

***ЛИСТИНГ 4.5*** The Queue Class Without nItems

class Queue

{

private int maxSize;

private long[] queArray;

private int front;

private int rear;

//--------------------------------------------------------------

public Queue(int s) // constructor

{

maxSize = s+1; // array is 1 cell larger

queArray = new long[maxSize]; // than requested

front = 0;

rear = -1;

}

//--------------------------------------------------------------

public void insert(long j) // put item at rear of queue

{

if(rear == maxSize-1)

rear = -1;

queArray[++rear] = j;

}

//--------------------------------------------------------------

public long remove() // take item from front of queue

{

Queues 141

long temp = queArray[front++];

if(front == maxSize)

front = 0;

return temp;

}

//--------------------------------------------------------------

public long peek() // peek at front of queue

{

return queArray[front];

}

//--------------------------------------------------------------

public boolean isEmpty() // true if queue is empty

{

return ( rear+1==front || (front+maxSize-1==rear) );

}

//--------------------------------------------------------------

public boolean isFull() // true if queue is full

{

return ( rear+2==front || (front+maxSize-2==rear) );

}

//--------------------------------------------------------------

public int size() // (assumes queue not empty)

{

if(rear >= front) // contiguous sequence

return rear-front+1;

else // broken sequence

return (maxSize-front) + (rear+1);

}

//--------------------------------------------------------------

} // end class Queue

Забележлива е сложеноста на isFull(), и isEmpty size() во овој пример. Ваквиот пристап, без броење на елементите ретко сесреќава во практиката, така што нема да биде детаљно анализиран.

***3.3.3.1 Ефикасност на редовите***

Како и во случајот со магацините, елементите можат да бидат внесувани и отстранувани од редовите во 0 (1) интервал.