ПОДАТОЧНИ СТРУКТУРИ И АНАЛИЗА НА АЛГОРИТМИ

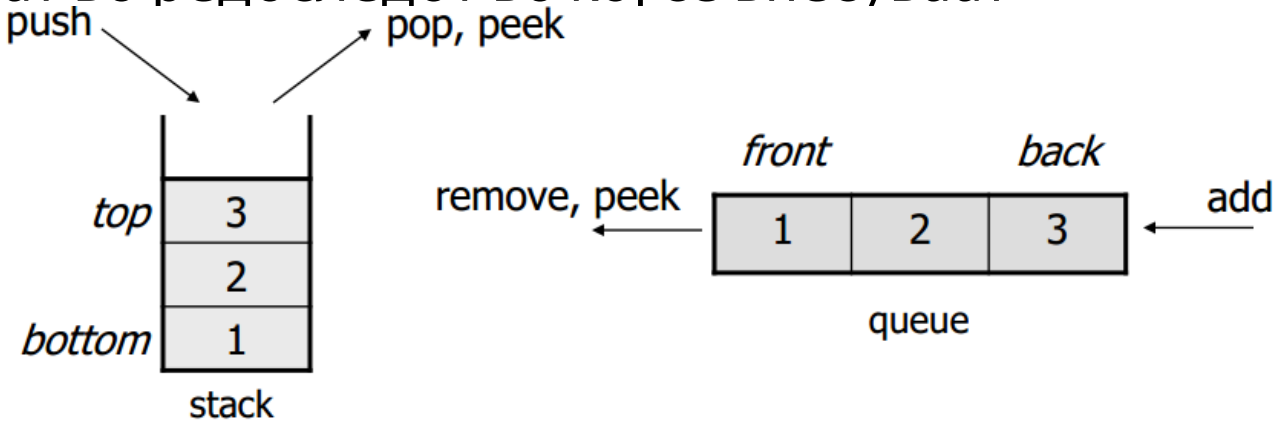
ЧАС 4: МАГАЦИНИ И РЕДОВИ

АУДИТОРИСКИ ВЕЖБИ

БОЈАНА ВЕЛИЧКОВСКА

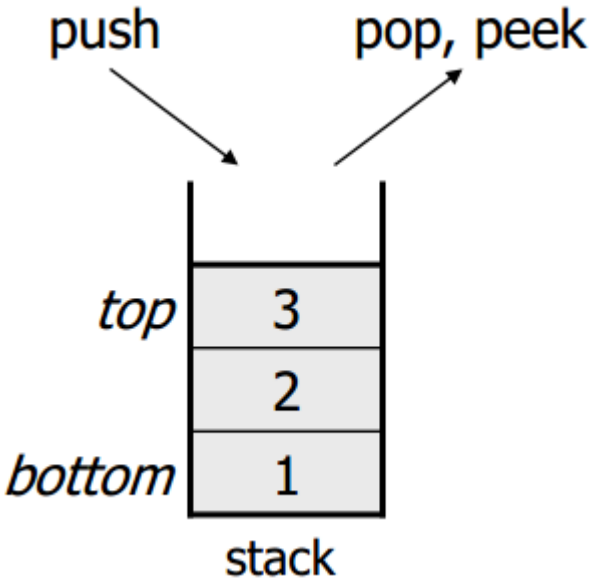
МАГАЦИНИ И РЕДОВИ

- Понекогаш добро е да користиме колекции од податоци кои се помоќни од низите и определени операции ги извршуваат побрзо
- Магациниte и редовите, како и листите се апстрактни типови на податоци (ADT) кои дозволуваат додавање и вадење на елементи само од едниот крај, не од внатрешноста
- Магацини – елементите се вадат обратно од редоследот во кој се внесуваат
- Редови - елементите се вадат во редоследот во кој се внесуваат



МАГАЦИН (STACK)

- Last In First Out (LIFO)
- Може да се испитува последно додадениот елемент, односно врвот (top)
- Основни операции:
  - push() – додава елемент на врв
  - pop() – вади и враќа елемент од врв
  - peek() – враќа елемент од врв
  - isEmpty() – проверува дали магацинот е празен



КЛАСАТА STACK <E>

- Се наоѓа во пакетот java.util
- Класата Stack користи низа за имплементирање на магацин

`Stack<E>()`	constructs a new stack with elements of type **E**
`push(value)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

МАГАЦИНИ

- Магацинот не се изминува на досега познатиот начин

```
Stack<Integer> s = new Stack<Integer>();
...
for (int i = 0; i < s.size(); i++) {
    do something with s.get(i);
}
```

- Мора да извадиме елемент од магацинот за да може да го прикажеме

```
// process (and destroy) an entire stack
while (!s.isEmpty()) {
    do something with s.pop();
}
```

МАГАЦИНИ

```
Stack<Integer> s = new Stack<Integer>();
for(int i = 0; i < 5; i++)
    s.push( i );
int limit = s.size();
for(int i = 0; i < limit; i++)
    System.out.print( s.pop() + " ");
//или
// while( !s.isEmpty() )
// System.out.println( s.pop() );
```

МАГАЦИНИ

- Нека имаме метод кој прима магацин од цели броеви и го враќа најголемиот елемент

```
// Precondition: !s.isEmpty()
public static void max(Stack<Integer> s) {
    int maxValue = s.pop();
    while (!s.isEmpty()) {
        int next = s.pop();
        maxValue = Math.max(maxValue, next);
    }
    return maxValue;
}
```

- Што е погрешно???

МАГАЦИНИ

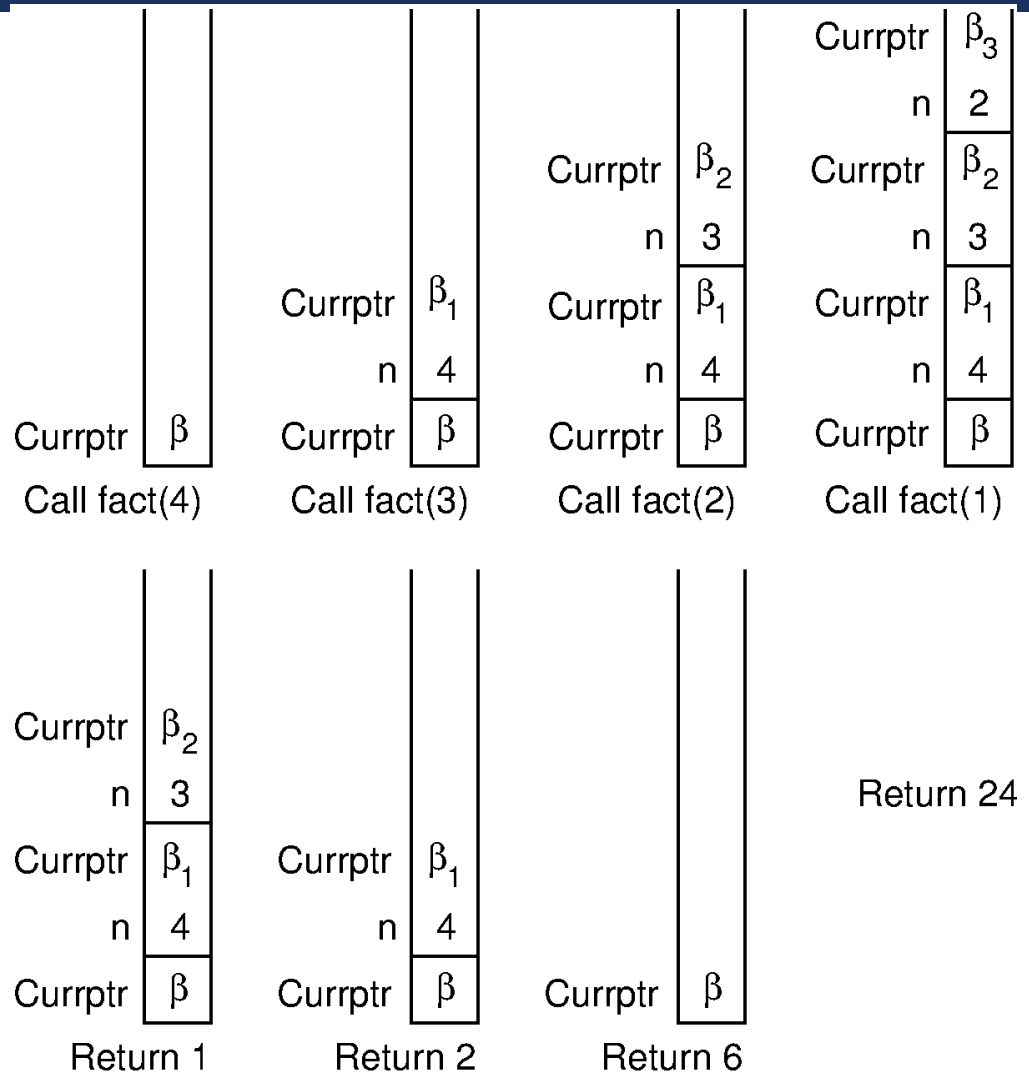
- Кодот го уништи магацинот

```
public static void max(Stack<Integer> s) {
    Stack<Integer> backup = new Stack<Integer>();
    int maxValue = s.pop();
    backup.push(maxValue);
    while (!s.isEmpty()) {
        int next = s.pop();
        backup.push(next);
        maxValue = Math.max(maxValue, next);
    }
    while (!backup.isEmpty()) {    // restore
        s.push(backup.pop());
    }
    return maxValue;
}
```

УПОТРЕБА НА МАГАЦИНИ

- Програмски јазици и компајлери:
  - Рекурзија
  - Повикот на методи (функции) се поставува на магацин (call = push, return = pop)
  - Компајлерите користат магацин за евалуирање на изрази
- Совпаѓање на парови од нешта:
  - Дали два стрингови се палиндроми
  - Совпаѓање на загради во израз
  - Претворање и пресметка на infix -> postfix/prefix изрази
- Напредни алгоритми:
  - Пребарување со враќање наназад
  - Undo за враќање на претходните операции

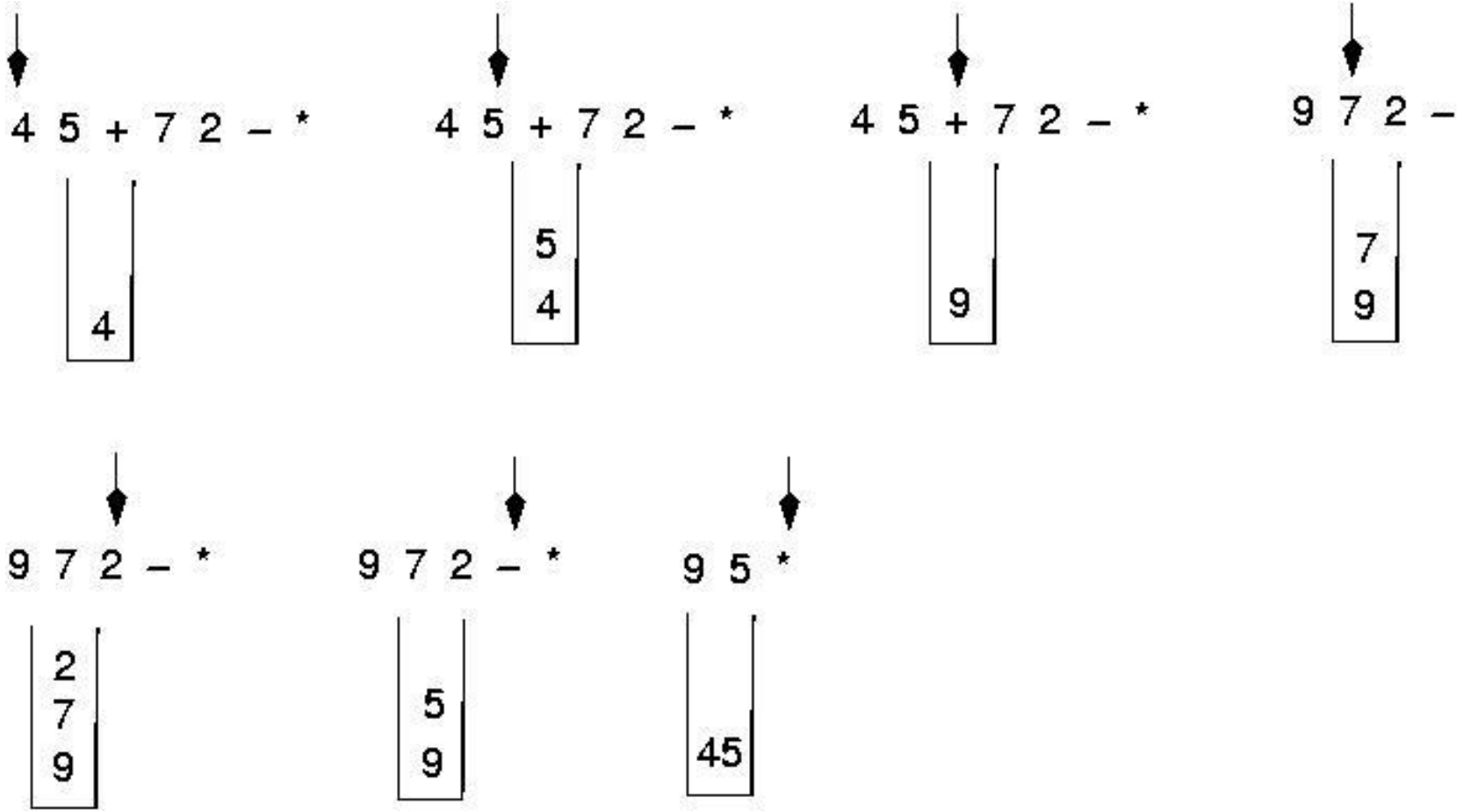
УПОТРЕБА НА МАГАЦИНИ - РЕКУРЗИЈА



ЗАДАЧА 5: ПРЕСМЕТУВАЊЕ НА POSTFIX ИЗРАЗ

- Изразот 4 * 5 + 2 + 6 * 7 (infix нотација), претворен во postfix израз изгледа вака:
  - 4 5 * 2 + 6 7 * +
- Користиме магацин за да го пресметаме postfix изразот
  - Ако наидеме на операнд, го поставуваме на магацин
  - Ако наидеме на оператор, ја извршуваме соодветната операција на два броја кои ги вадиме од магацин. Резултатот го поставуваме на магацин

ЗАДАЧА 5: ПРЕСМЕТУВАЊЕ НА POSTFIX ИЗРАЗ



ЗАДАЧА 5: ПРЕСМЕТУВАЊЕ НА POSTFIX ИЗРАЗ

```
public static float calculatePostfix(String postfix)
   {
      int result=0;
      Stack <Integer> s = new Stack(); // декларација на магацин со податоци од тип integer

      for(int i=0; i<postfix.length(); i++) // се разгледува должината на внесениот postfix израз
      {
         char znak = postfix.charAt(i);  // се зема карактерот на позиција i
         if(!isOperator(znak)) // проверуваме дали станува збор за број (а не за знак)
         {
            s.push(znak – '0');  //48 е позицијата на '0' во UTF     // се сместува во магацин со push ASCII табела (0-48:
9=57)
         }
```

ЗАДАЧА 5: ПРЕСМЕТУВАЊЕ НА POSTFIX ИЗРАЗ

```
else
        {    result=0;
            switch(znak) // ако станува збор за знак пристапува до switch
            {
                case '+':      result+=s.pop()+s.pop(); // резултат е збир од двата последно додадени членови
                    s.push(result);             break; // резултатот се запишува во магацинот
                case '-':       result+=-(s.pop())+s.pop();
                    s.push(result);             break;
                case '*':       result+=s.pop()*s.pop();
                    s.push(result);              break;
                case '/':         result+=s.pop()/s.pop();
                    s.push(result);
                    break;
                default:
                    break;
            }           }       }
    return s.pop();  //на врв на магацин е сместен резултатот
  }
```

ЗАДАЧА 5: ПРЕСМЕТУВАЊЕ НА POSTFIX ИЗРАЗ

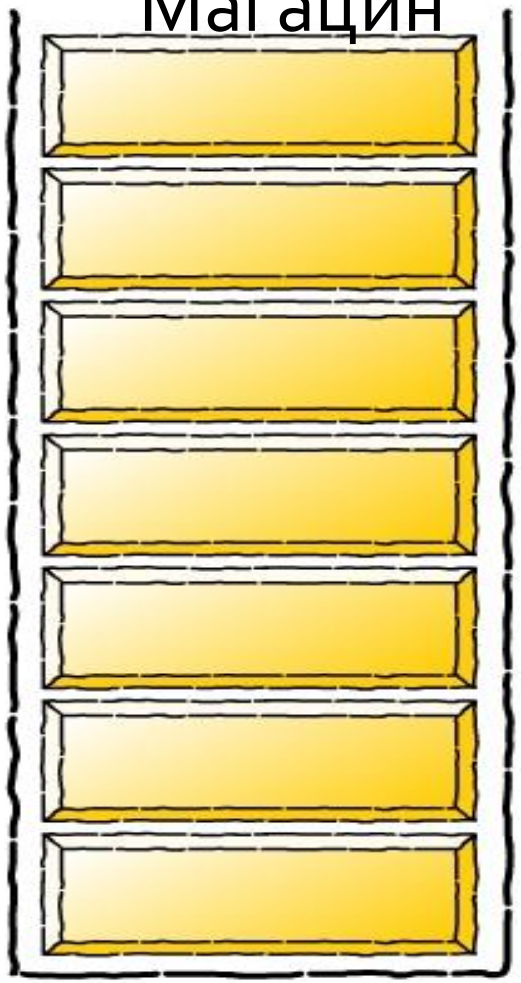
```
public static void main(String[] args) {


        Scanner in = new Scanner(System.in);
        System.out.println("Vnesete aritmetichki izraz vo postfix
notacija");
        String s = in.nextLine();
        System.out.println(s+" = "+ calculatePostfix(s));


    }
```

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ

Магацин

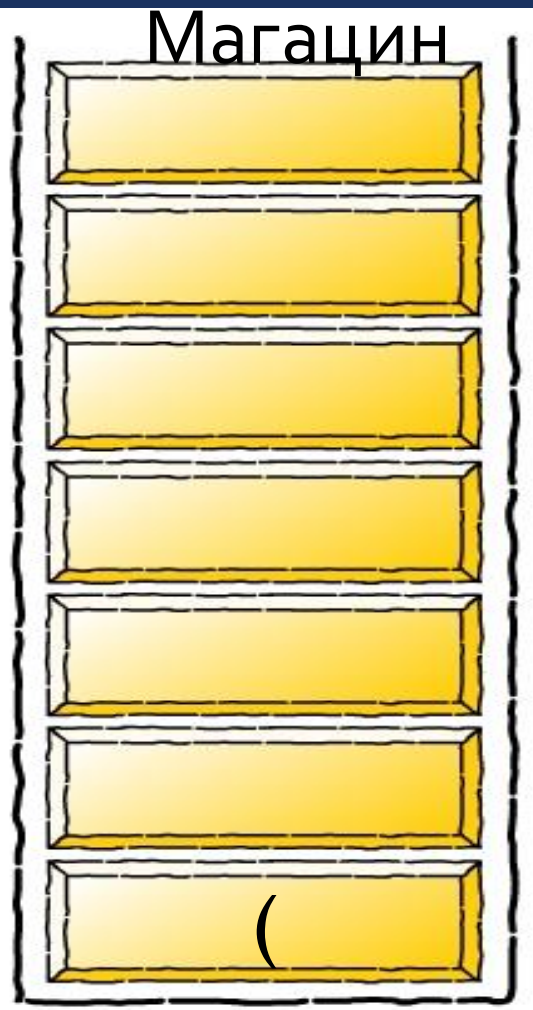


Infix израз

(a + b - c) * d – (e + f)

Postfix израз

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ

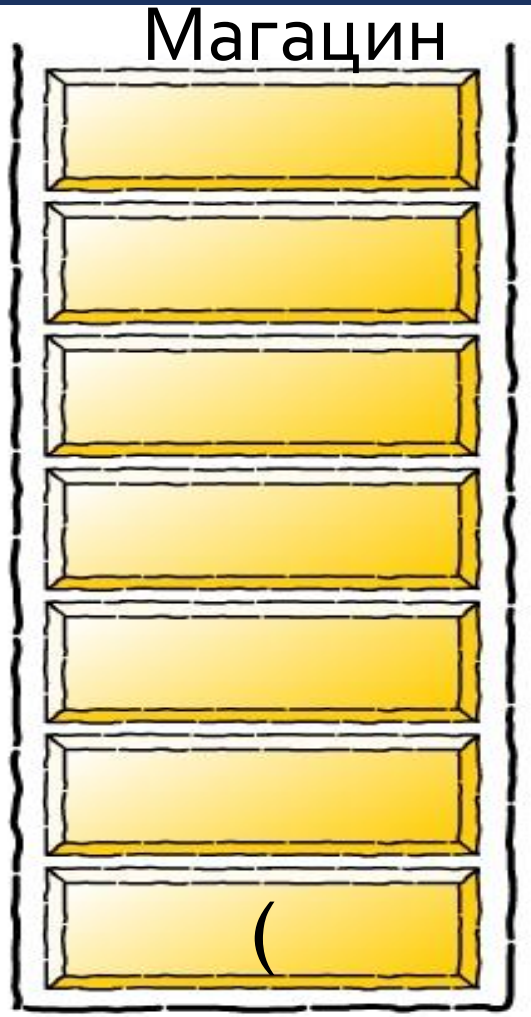


Infix израз

a + b - c) * d – (e + f)

Postfix израз

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



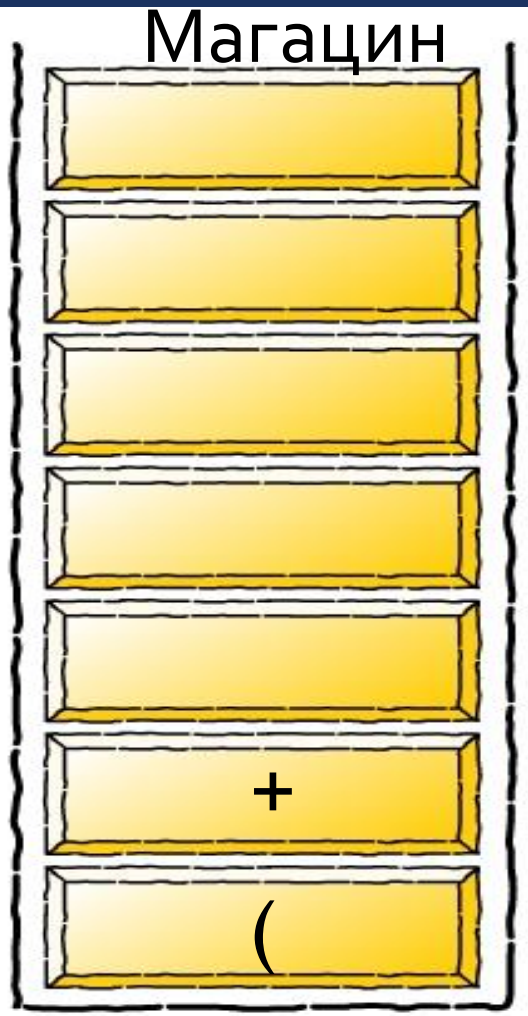
Infix израз

+ b - c) * d – (e + f)

Postfix израз

a

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



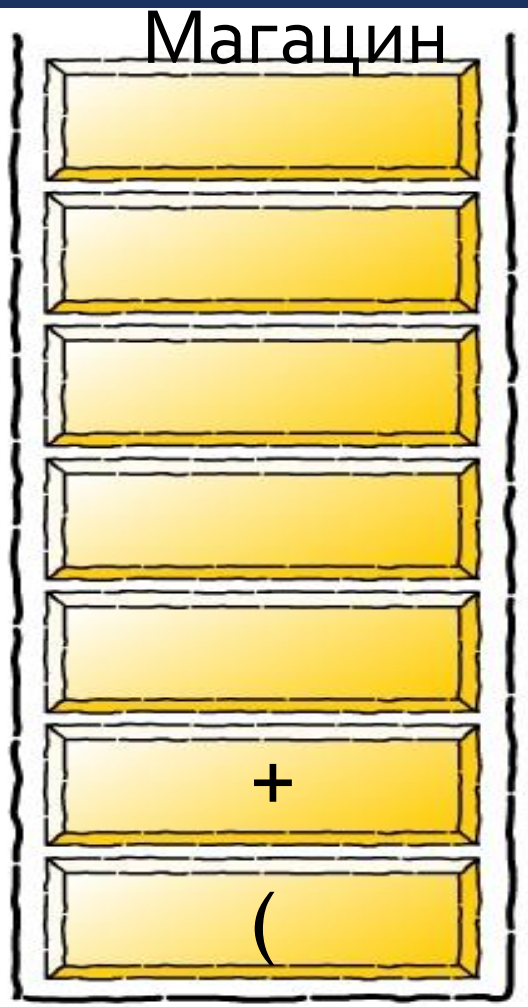
Infix израз

b - c) * d – (e + f)

Postfix израз

a

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



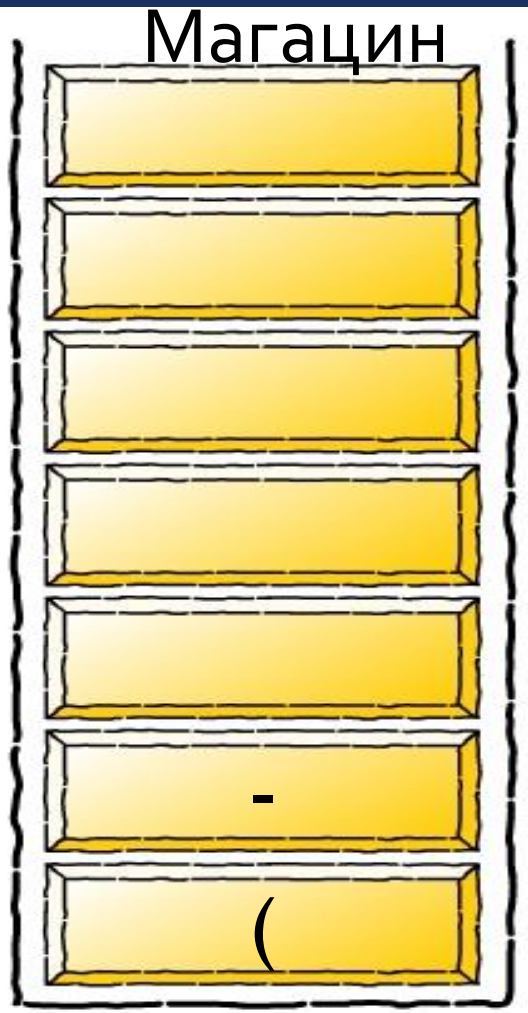
Infix израз

- c) * d – (e + f)

Postfix израз

a b

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



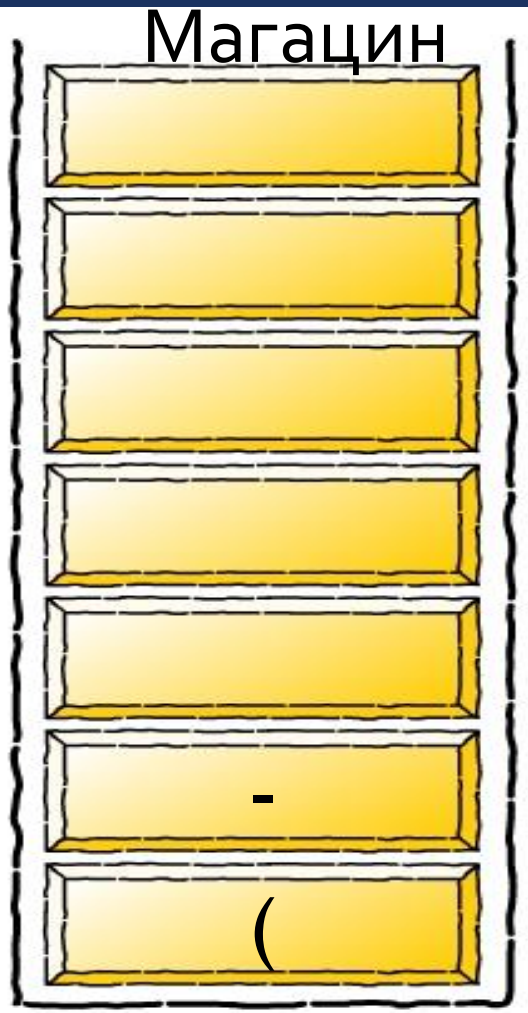
Infix израз

c) * d – (e + f)

Postfix израз

a b +

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



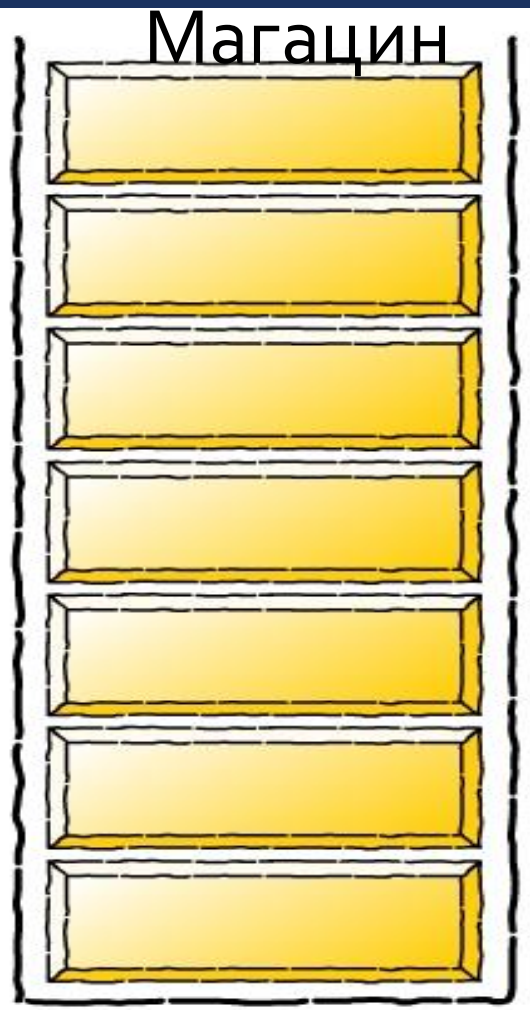
Infix израз

) * d – (e + f)

Postfix израз

a b + c

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



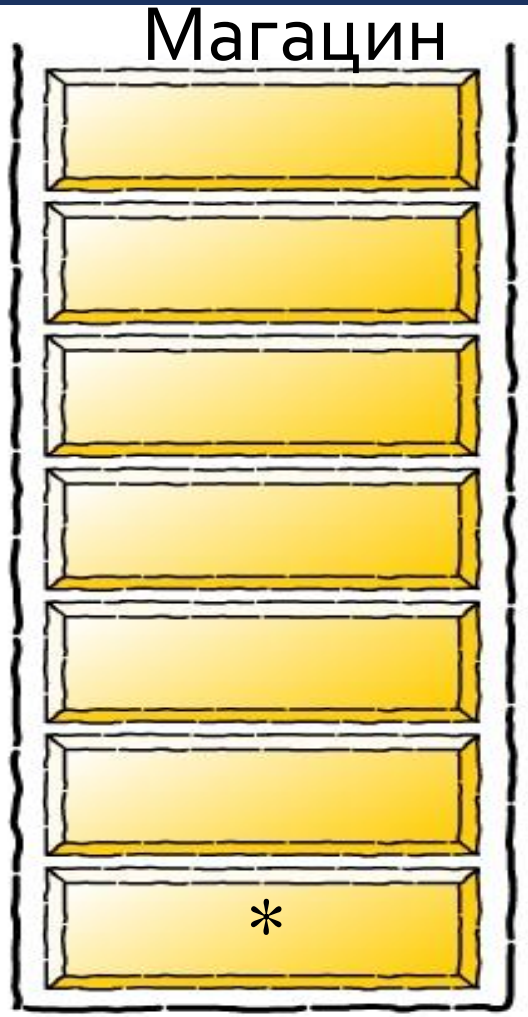
Infix израз

* d – (e + f)

Postfix израз

a b + c -

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



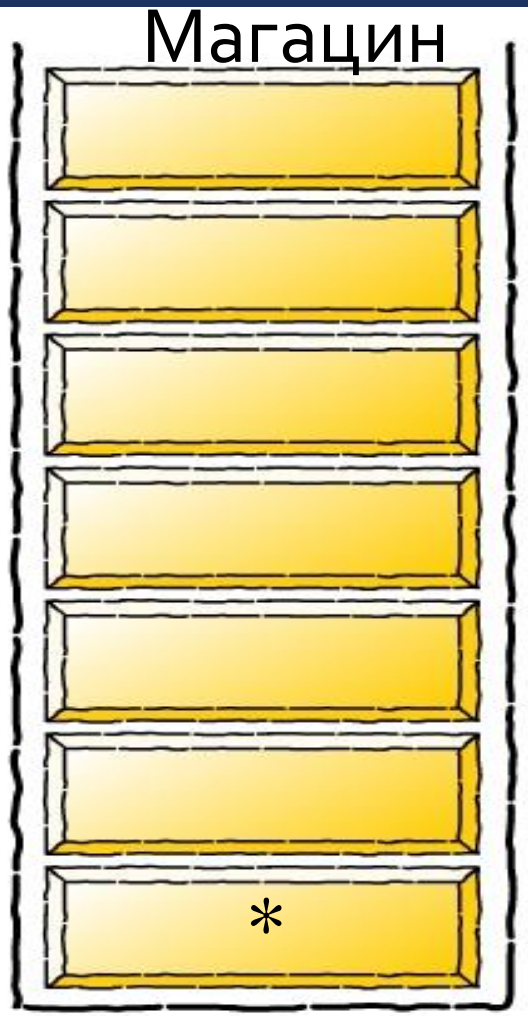
Infix израз

d – (e + f)

Postfix израз

a b + c -

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



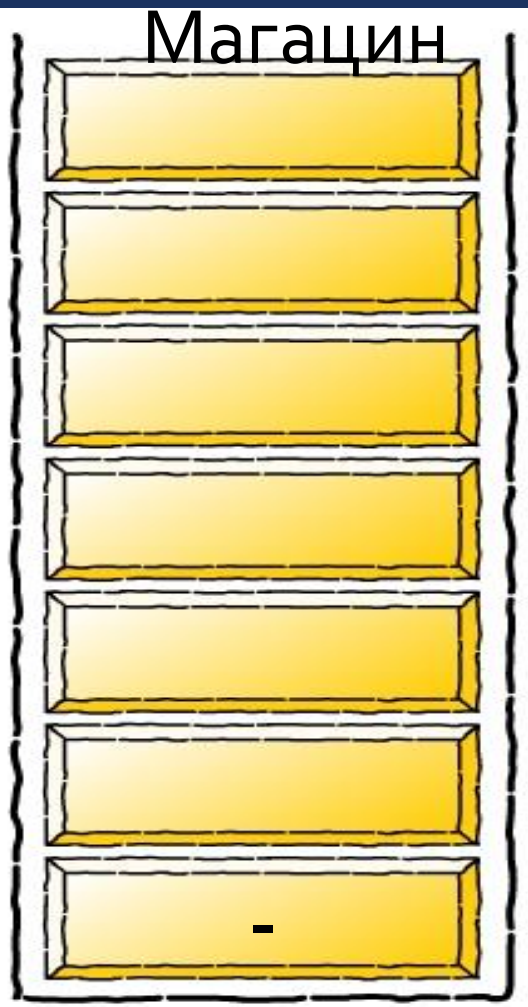
Infix израз

– (e + f)

Postfix израз

a b + c - d

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



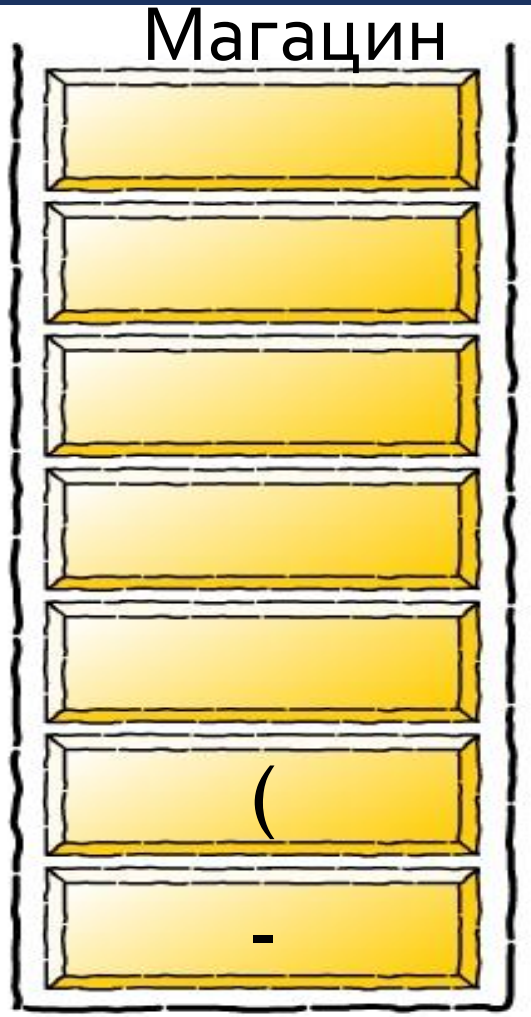
Infix израз

(e + f)

Postfix израз

a b + c – d *

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



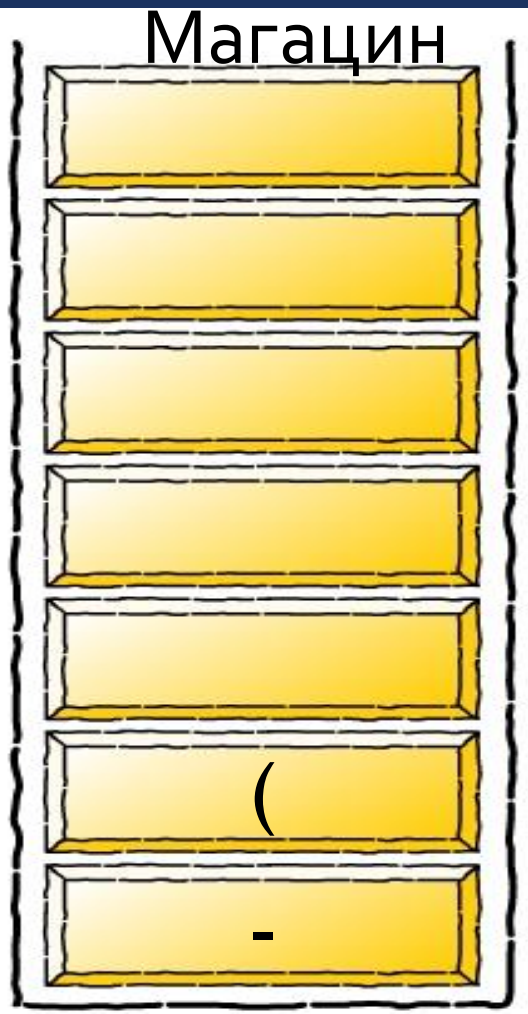
Infix израз

e + f)

Postfix израз

a b + c – d *

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



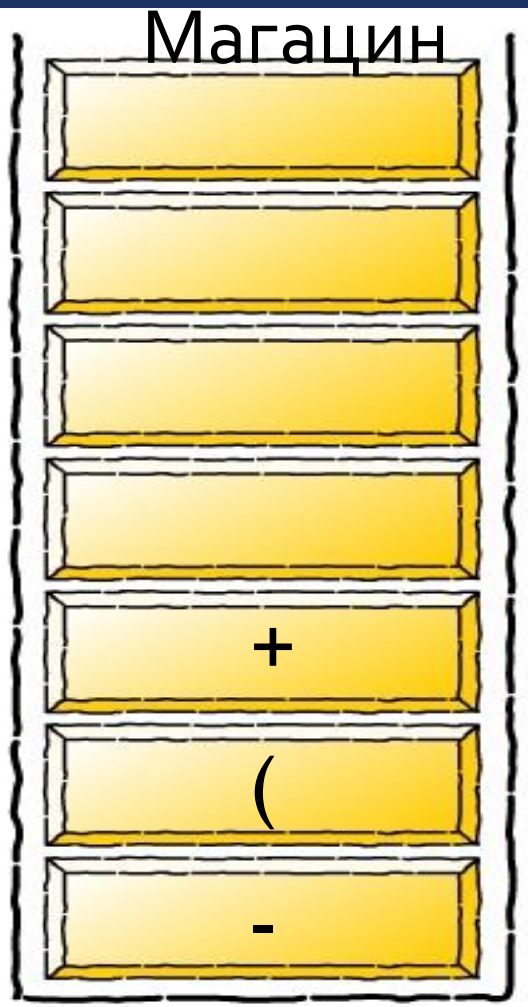
Infix израз

+ f)

Postfix израз

a b + c – d * e

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



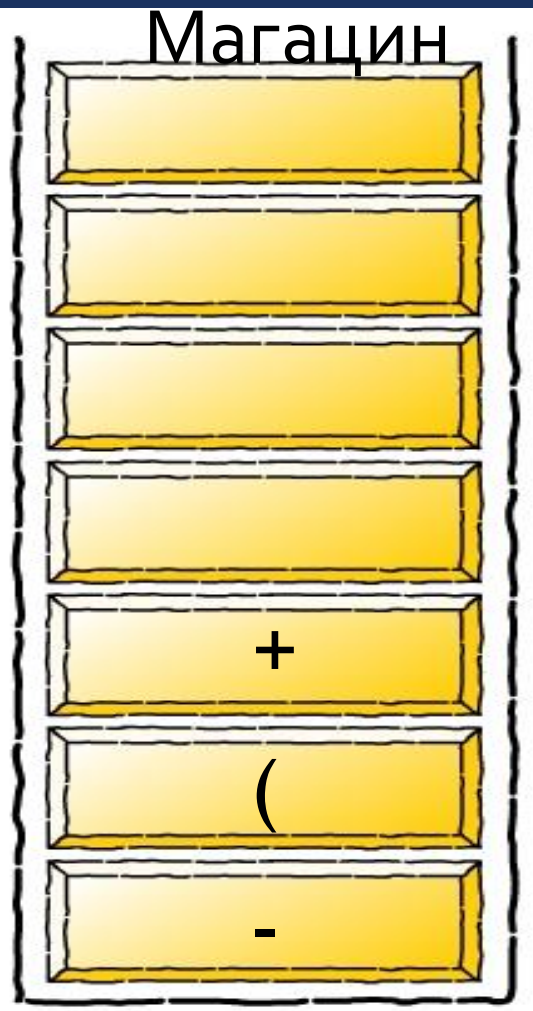
Infix израз

f)

Postfix израз

a b + c – d * e

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



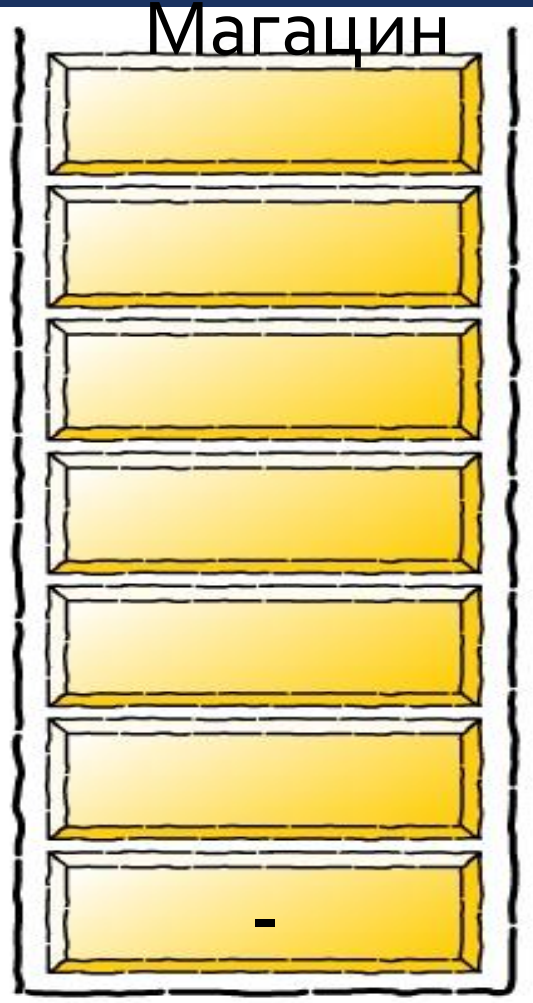
Infix израз

)

Postfix израз

a b + c – d * e f

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ

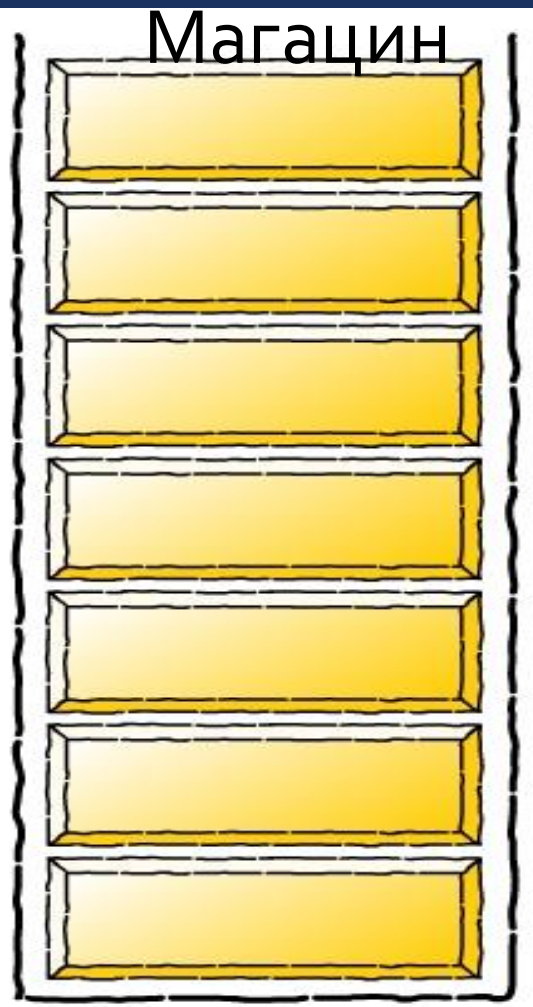


Infix израз

Postfix израз

a b + c – d * e f +

ПРЕТВОРАЊЕ НА INFIX ВО POSTFIX ИЗРАЗ



Infix израз

Postfix израз

a b + c – d * e f + -

РЕДОВИ

- Редот може да се визуелизира како редица од клиенти кои чекаат да бидат опслужени
- Новодојдените клиенти се сместуваат на крај на редот
- Клиентот на почеток на редот ќе биде опслужен прв
- First Come First Served
- Оперативните системи користат редици за обезбедување на услуги (сервиси)

ИНТЕРФЕЈСОТ QUEUE<E>

```
import java.util.*;
public interface Queue
{
 public boolean isEmpty();
 public void add( Object o );
 public Object remove() throws NoSuchElementException;
 public void clear();
}
```

ИМПЛЕМЕТАЦИЈА НА РЕД

- Во Јава редот се имплеметира преку класата LinkedList
  - LinkedList обезбедува методи за додавање и бришење на елементи од двете страни на листата, што е доволно за имплементирање на ред
  - Пример
    - Queue<String> name = new LinkedList<String>();

ИМПЛЕМЕТАЦИЈА НА РЕД

```
import java.util.LinkedList;
import java.util.Queue;
public class QueueImplementation {
    public static void main(String[] args) {
        Queue <Integer> q = new LinkedList();
        for(int i=0;i<5;i++)
        {
            q.add(i);
        }
      while(!q.isEmpty())
      {
          System.out.print(q.peek()+" ");
          q.remove();
      }  }   }
```

ИСКЛУЧОЦИ

- Механизам за справување со грешки за време на извршување
- Во Java исклучоците се објекти
- Исклучоците се исфрлаат кога кодот или JRE ќе наидат на неочекуван услов или состојба
- Исфрлениот исклучок се фаќа од код кој се справува со него

ИСФРЛАЊЕ НА ИСКЛУЧОЦИ

```
if (insertIndex >= A.length) {
throw new BoundaryViolationException("No element at index" +
insertIndex); // се создава и фрла објект
}
```

- Името на исклучокот треба да биде дескриптивно
- JRE има исклучок ArrayIndexOutOfBoundsException кој автоматски ќе се генерира и го прави истото (слично) како и кориснички дефинираниот исклучок BoundaryViolationException

ИСФРЛАЊЕ НА ИСКЛУЧОЦИ

- За BoundaryViolationException мора да се напише кориснички дефинирана класа

```
class BoundaryViolationException extends Exception
{
    protected String s;
    public BoundaryViolationException (String s)
    {
        this.s=s;
    }
}
```

ИСФРЛАЊЕ НА ИСКЛУЧОЦИ

- Кога дефинираме метод, добра практика е да наведеме (дефинираме) кои исклучоци може да ги исфрли

```
public void goShopping() throws
hoppingListTooSmallException, OutOfMoneyException
{
// method body . . .
}
```

СПРАВУВАЊЕ СО ИСКЛУЧОЦИ

- Кога исклучокот се исфрла, истиот мора да биде фатен. Поинаку програмата ќе терминира
- **try – catch** блок

```
try    main_block_of_statements
catch (exception_Class1   variable1)
    block_of_statements1
```

- Некоја од наредбите во **try** блокот може да исфрли исклучок

СПРАВУВАЊЕ СО ИСКЛУЧОЦИ

- **try – catch** блок

```
try    main_block_of_statements
catch (exception_Class1  variable1)
    block_of_statements1
catch (exception_Class2  variable2)
    block_of_statements2
```

- catch блокот го фаќа исклучокот и се справува со грешката

СПРАВУВАЊЕ СО ИСКЛУЧОЦИ

```
try {   int result = 99/0;    //this statement throws an exception
//other statements may appear here;      }
catch  (ArithmetichException e){
System.out.println("ArithmetichException caught");  }
```

- Ако некој услов во **try** блокот исфрли исклучок, останатите наредби во истиот блок се прескокнуваат и контролата се префрла на **catch** блокот

СПРАВУВАЊЕ СО ИСКЛУЧОЦИ

```
public void myMethod() throws  MyException{
//some code here.
if  (some condition) throw new MyException("MyException: reason");
}
public void  yourMethod(){
try {
myMethod();
}
catch (MyException e){
//code to handle the exception
}   }
```

СПРАВУВАЊЕ СО ИСКЛУЧОЦИ

```
Class MyException extends Exception {
public MyException(String s) {
super (s);
}
}
```

СПРАВУВАЊЕ СО ИСКЛУЧОЦИ

```
try     main_block_of_statements
catch (exception_Class1   variable1)
    block_of_statements1
catch (exception_Class2   variable2)
    block_of_statements2
finally //опционален дел
    block_of_statementsn //овие наредби секогаш се извршуваат (ако постои finally)
```