KRMILNI STAVKI

POGOJNI STAVEK

```
if (pogoj) {
    koda
}
```

~ sintaksa: pravilo zapisovanja

= nastavi vrednost

== primerja vrednosti

```
if (pogoj) {
    koda
} else {
    koda
}
```

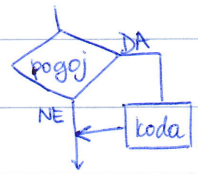
Console.log izpiše podatke

```
if (pogoj 1) { koda }
else if (pogoj 2) { koda }
⋮
else { koda }
```

DIAGRAM POTEKA = flowchart

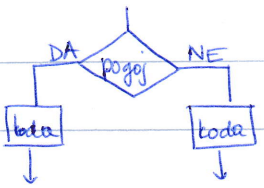
vstopni blok

if:



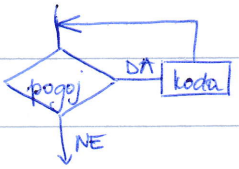
odločitveni blok

if, else:



PONAVLJALNI STAVEK = loop, zanka

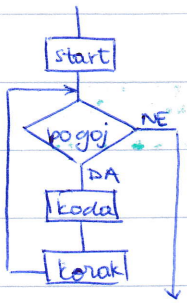
```
while (pogoj) {
    koda
}
```



% modul = ostanek celoštevilskega deljenja

~ ustavljivost algoritma: če se za vse veljavne vhodne podatke algoritem prej ali slej ustavi

```
for ( start ; pogoj ; korak ) {
    koda
}
```



<= je ≤

⇒ podoben while, uporabi se pri kakršnem koli štetju

TABELA = array

~ vektor: enorazsežnostna tabela

~ matrika: dvorazsežna tabela

⇒ tabele imajo skalarne elemente

štetje el. v tabelah: od 0 do n-1

ENORAZSEŽNA TABELA

```
ime_tabele = [ element_1 , element_2 , ... , element_n-1 ] ;
```

čuvaj = sentinel

&& je enako AND

⇒ je oblika znotrajpasovne informacije o koncu niza podatkov

↳ zunajpasovna bi bila podana dolžina niza

```
t = [ e1 , e2 , ... , en-1 , 0 ] ;
for ( i = 0; t[i] != 0 ; i ++ ) {
    // koda za i-ti element
}
```

DEMORGANOVA ZAKONA
$\neg(a \wedge b) \Leftrightarrow (\neg a) \vee (\neg b)$
$\neg(a \vee b) \Leftrightarrow (\neg a) \wedge (\neg b)$

DVORAZSEŽNA TABELA

```
ime_tabele =[[ el_0_0 , el_0_1 , ... , el_0_j-1 ],
             [ el_1_0 , el_1_1 , ... , el_1_j-1 ],
                    :
             [ el_i-1_0 , el_i-1_1 , ... , el_i-1_j-1 ]]
ime_tabele [i][j];
```

i: število vrstic
j: število stolpcev

PODPROGRAMI = funkcije, procedure, rutine

~ en del programa, je poimenovan in ima natančno določen način vnosa vhodnih podatkov

⇒ podprogram definiramo in kličemo

```
function ime_funkcije (argumenti) {
    koda (+ return vrednost);
}
ime_funkcije (vhodni_podatki);
```

↳ formalni parametri (kopija dejanskih)

↳ dejanski parametri

~ lokalne spremenljivke: dodatne spremenljivke, omejene na podprogram

~ globalne spremenljivke: dostopne vseposvod, razen, če ima podprogram lokalno spremenljivko z istim imenom

ISKANJE NAPAK

~ hrošč (bug): napaka v programski kodi

~ razhroščevanje (debugging): iskanje napak

načini razhroščevanja:

- sled programa (program trace)
- razhroščevalnik (debugger)

SLED PROGRAMA = program trace

⇒ zapisovanje pomembnih dogodkov med izvajanjem programa, za kasnejšo analizo

⇒ z uporabo konzole

console.log("a","b")
↳ izpiše več stvari, ločenih z vejico

RAZHROŠČEVALNIK

⇒ podobno kot sled, le da program lahko izvajamo korak za korakom

~ breakpoint, watch, step over, step into, step out

• za spremenljivke ↷ ↓ ↑

PODAJANJE PARAMETROV PO SKLICU = referenca

⇒ skalarji se obnašajo drugače kot tabele, ker hranijo samo lokacijo podatkov, ne pa tudi podatkov

```
tab1 = [1,2,3]        lokacija za [1,2,3] → tab1
tab2 = tab1           lokacija za [1,2,3] → tab2
```

↳ tab1 in tab2 kličeta ISTE podatke

⇒ če podamo ime tabele kot parameter, se spreminja „originalna" tabela

⇒ če želimo narediti kopijo tabele, jo moramo kopirati element po element.

NAČRTOVANJE ALGORITMOV IN PODATKOV

⇒ načrtovanje z vrha navzdol (= top-down design)

postopno načrtovanje (= stepwise design)

razgradnja problema (= problem decomposition)

- sistem postopoma razčlenjujemo: najprej določimo, kako bo sistem izgledal in deloval, nato ga razdelimo na podsisteme, ki imajo vsak svojo funkcijo & način povezovanja z ostalimi deli, po potrebi podsisteme še naprej razdelimo na enostavnejše dele

⇒ strukturirano programiranje: vsak program je mogoče sestaviti iz treh elementov:

- zaporedno izvajanje
- odločanje
- ponavljanje = iteracija

⇒ načrtovanje z dna navzgor (= bottom-up design)

- začnemo s podrobnostmi, končamo s končno organizacijo
- testiranje & preizkušanje je možno že prej

⇒ načrtovanje komponent (= piece part design)

~ brute force: preverjanje vseh obstoječih možnosti

~ reverse polish notation (RPN): postfix zapis

OBJEKT

~ abstraktni podatkovni tip: matematični model, ki poleg podatkov definira tudi operacije nad temi podatki

⇒ programska tvorba, ki pod istim imenom združuje podatke & operacije

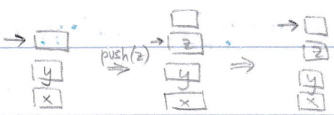
- podatki ali lastnosti
- operacije ali metode
- selektor (.) – izberemo želeno lastnost / postopek

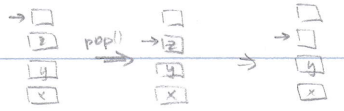
```
ime_objekta = Object();
ime_objekta.lastnost = num;
ime_objekta.metoda = function(parametri) {
  // koda
};
ime_objekta.metoda(ime_objekta.lastnost);
```

SKLAD = stack

⇒ podoben tabeli, le da lahko dostopamo le do zadnjega elementa (LIFO)

```
sklad = Object();
sklad.podatki = [];
sklad.vrh = 0;        ← vrh vedno kaže na mesto, kamor pride naslednji element
sklad.push = function(element) {
    sklad.podatki[sklad.vrh] = element;
    sklad.vrh += 1; };
sklad.pop = function() {
    sklad.vrh -= 1;
    return sklad.podatki[sklad.vrh]; };
sklad.size = function() {
    return sklad.vrh; };
sklad.clear = function() {
    sklad.vrh = 0; };
```





* še vedno je notri z, ampak bo overwritten

videz kode

⇒ zamik v desno:

- da je branje enostavnejše
- koda, ki se izvaja zaporedno, je zapisana ena pod drugo
- koda, ki je del ponavljalnega ali odločitvenega stavka, je zamaknjena - nakazovanje odvisnosti

⇒ imena spremenljivk in podprogramov

- imena naj čim bolj predstavljajo pomen spremenljivke / funkcije
- namesto presledka lahko uporabljamo podčrtaj (lažja berljivost),
- camelCase - vsaka beseda se začne z veliko začetnico

dokumentiranje kode

⇒ opombe

- neposredno v programski kodi
- kratke razlage funkcionalnosti podprogramov, pogoji parametrov ...

komentarji v JS:
// enovrstični
/** večvrstični **/

⇒ obsežnejše dokumentiranje programske kode

- posebni dokumenti
- za uporabnike in vzdrževalce
- piše s pomočjo softwara (documentation tools)

omejevanje območja spremenljivk

⇒ za dober nadzor nad kodo

- čim več lokalnih spremenljivk
- v podprogramih uporabljamo spremenljivke preko parametrov & return
- spremenljivke, ki pašejo skupaj, dodamo v en objekt

APT = abstraktni podatkovni tip
PS = podatkovna struktura

PODATKOVNE STRUKTURE & ABSTRAKTNI PODATKOVNI TIPI

⇒ prednost abstraktnih podatkovnih tipov je ta, da uporabnika ne zanima način hrambe podatkov, ampak le kako delujejo metode

~ podatkovna struktura: kot je abstraktni podatkovni tip prijazen uporabniku, je podatkovna struktura namenjena razvijalcu; predstavlja način fizične izvedbe abstraktnega podatkovnega tipa

SLOVAR -APT = dictionary / asociativna tabela = associative array

⇒ zbirka parov ključev & gesel

⇒ metode slovarja:

- insert dodajanje para
- delete brisanje para
- modify spreminjanje para
- lookup iskanje gesla s ključem

! vsak ključ se lahko pojavi samo enkrat

! deklariranje slovarja UČ str. 96

- slovar je objekt, slovar.podatki so dvodimenzionalna tabela
- slovar.konec nam pove, na katero mesto shranimo naslednji par
- lookup primerja podan parameter z vsemi ključi v podatkih, če ga ni, -1

VRSTA -APT = queue

⇒ podobno skladu, le da deluje po principu FIFO

⇒ uporablja se kot medpomnilnik (buffer), obstaja tudi krožni medpomnilnik

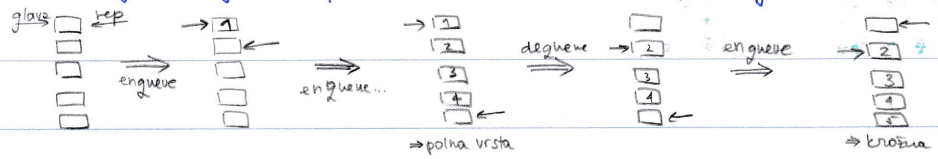
⇒ metode:

- enqueue pisanje v vrsto
- dequeue branje z vrste

deklaracija vrste UČ str. 100

* in enaka

- vrsta je objekt, podatki so seznam, rep & glava sta 0, ko je vrsta prazna

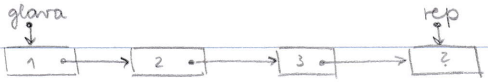


POVEZAN SEZNAM – PS = linked list

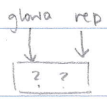
⇒ uporablja se namesto tabele

⇒ sestavljen iz posameznih blokov podatkov, ki vsebujejo informacijo o naslednjem bloku

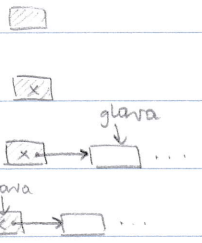
~ element ali vozlišče (= node): posamezen podatkovni blok




```
seznam = Object();
seznam.glava = Object();        ⇒ čuvaj
seznam.glava.podatek = "?";
seznam.glava.naslednji = "?";
seznam.rep = seznam.glava;
```

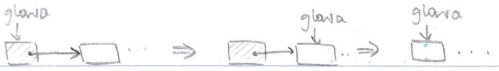



```
seznam.insertBeginning = function (vrednost) {
    var nov = Object();
    nov.podatek = vrednost;
    nov.naslednji = seznam.glava;
    seznam.glava = nov;
};
```

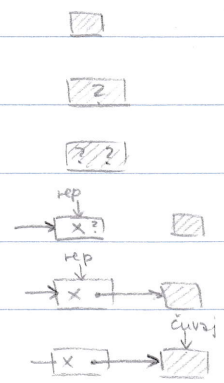


seznam.pop = seznam.removeBeginning
↳ deluje tol. seznam.pop()

deklaracija seznam.removeBeginning UČ str. 104



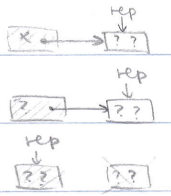

```
seznam.insertEnd = function (vrednost) {
    var cuvaj = Object();
    cuvaj.podatek = "?";
    cuvaj.naslednji = "?";
    seznam.rep.podatek = vrednost;
    seznam.rep.naslednji = cuvaj;
    seznam.rep = cuvaj; };
```



~ dvojno povezan seznam (= doubley linked list): podobno povezanemu seznamu, le da vsako vozlišče vsebuje tudi sklic na svojega predhodnika

```
seznam.delete = function (vrednost) {
    var ta;
    ta = seznam.glava;
    while (ta != seznam.rep) {
        if (ta.podatek == vrednost) {
            if (ta.naslednji == seznam.rep) {
                seznam.rep = ta;
            }
            ta.podatek = ta.naslednji.podatek;
            ta.naslednji = ta.naslednji.naslednji;
            return;
        }
        ta = ta.naslednji;
    }};
```

če je naslednjik vozlišča, ki ga bomo izbrisali, hkrati tudi rep, bo nov rep postal trenutno vozlišče



REKURZIJA = recursion

~ Drostejev učinek: vizualna reprezentacija rekurzije

~ rekurzija v računalništvu: proces izvajanja določenega algoritma, pri čemer je eden od korakov algoritma tudi izvajanje algoritma samega

⇒ potrebno je poskrbeti, da se rekurziven algoritem ne bi ponavljal v neskončnost, zato upoštevamo, da mora tak algoritem imeti:

- enostaven osnovni postopek, ki ne vsebuje rekurzije (= base case)
- eno ali več rekurzivnih pravil, ki vse primere poenostavljajo proti osnovnemu primeru

REKURZIJA & ITERACIJA

~ iteracija (= iteration) : algoritem, pri katerem postopke / izračune ponavljamo s ponavljalnim stavkom for ali while

⇒ rekurzija je ponavadi počasnejša, ampak se jo uporablja pri kompleksnejših programih, kjer je iterativen zapis nepregleden in zapleten npr. Fibbonacci

ČASOVNA ZAHTEVNOST PROGRAMA

~ časovna zahtevnost (= time complexity) : količina časa, ki ga algoritem porabi za svoje izvajanje; ocenimo tako, da preštejemo, koliko osnovnih operacij mora algoritem izvesti

⇒ običajno nas zanima najslabši možen scenarij

⇒ pri majhnih vhodnih podatkih nas dejanski čas izvajanja ne zanima, zato opazujemo oceno zahtevnosti v odvisnosti od naraščanja vhodnih podattov ~ asimptotična ocena zahtevnosti

⇒ uporabljamo zapis z velikim O (opisuje obnašanje funkcije pri velikem / neskončnem argumentu)

$T(n) = \frac{1}{2}n^2 + \frac{1}{2}n + 1$ ⇒ opuščamo linearen & konstanten člen, ker nimata velikega vpliva

$T(n) = O(n^2)$

- konstantna $O(1)$
- linearna $O(n)$
- kvadratna $O(n^2)$
- logaritemska $O(\log n)$ npr binarno iskanje
- eksponentna $O(2^n)$

reurziven Fibonacci:

C_n – št. klicanja funkcije

$C_0 = C_1 = 1$; $C_n = C_{n-1} + C_{n-2} + 1$

$\lim_{n\to\infty} \frac{C_n}{C_{n-1}} = \frac{1+\sqrt{5}}{2} = 1{,}618$ (zlati rez)

↳ vsak +n je poreča za 1,618-krat

$T(n) = O(1{,}618^n)$