**Stog realiziran statickim poljem**

*Struktura podataka* koja opisuje stog realiziran poljem:

#define MAXSTOG 100

typedef struct{

int vrh;

tip polje[MAXSTOG];

}Stog;

*1) Dodavanje elementa na stog*

int dodaj(tip element, Stog \*stog){

if (stog->vrh >= MAXSTOG-1) return 0;

stog->vrh++;

stog->polje[stog->vrh] = element;

return 1;

}

*2)Skidanje elementa sa stoga*

int skini(tip \*element, Stog \*stog){

if (stog->vrh < 0) return 0;

\*element = stog->polje[stog->vrh];

stog->vrh--;

return 1;

}

*3)Inicijalizacija praznog stoga*

void init\_stog(Stog \*stog){

stog->vrh = -1;

}

**Stog realiziran listom**

U prethodnom poglavlju stog smo realizirali statičkim poljem. Ta realizacija je imala veliki nedostatak

jer je postojala mogućnost prepunjenja polja. Sada ćemo stog implementirati listom što će nam

omogućiti da stavimo na stog gotovo neograničen broj elemenata. Jedino ograničenje predstavljat će

veličina radnog spremnika u računalu. Podsjetimo se koje operacije moramo ostvariti da bi stog

ispravno radio.

*Struktura podataka* koja opisuje stog realiziran listom:

struct at{

tip element;

struct at \*sljed;

};

typedef struct at atom;

typedef struct{

atom \*vrh;

}Stog;

*1) Dodavanje elementa na stog*

int dodaj(tip element, Stog \*stog){

atom \*novi;

if((novi=(atom\*)malloc(sizeof(atom)))!=NULL){

novi->element=element;

novi->sljed=stog->vrh;

stog->vrh=novi;

return 1;

}

else return 0;

}

*2)Skidanje elementa sa stoga*

int skini(zapis \*element, Stog \*stog){

atom \*pom;

if (stog->vrh==NULL) return 0;

\*element=stog->vrh->element;

pom=stog->vrh->sljed; //adresa novog vrha

free(stog->vrh); // obriši stari vrh

stog->vrh = pom; // postavi novi vrh

return 1;

}

*3)Inicijalizacija praznog stoga*

void init\_stog(Stog \*stog){

stog->vrh=NULL;

}

Red realiziran cirkularnim poljem

Osim statičkim poljem red se može ostvariti i povezanom listom. Veliki nedostatak implementacije

reda statičkim poljem je mogućnost prepunjenja jer imamo polje ograničene veličine (*MAXRED*).

Da bi omogućili razlikovanje punog i praznog reda, jedan element cirkularnog polja uvijek mora biti

prazan. Red je prazan ako vrijedi ulaz==izlaz, a pun je ako vrijedi (ulaz+1)%MAXRED==izlaz.

*Struktura podataka* koja opisuje red realiziran cirkularnim poljem:

#define MAXRED 100

typedef struct{

int ulaz, izlaz;

tip polje[MAXRED];

}Red;

*1) Dodavanje elementa u red*

int dodaj(tip element, Red \*red){

if ((red->ulaz +1) %n == red->izlaz) return 0;

red->ulaz++;

red->poljel[red->ulaz] = element;

return 1;

}

*2)Skidanje elementa iz reda*

int skini(tip \*element, Red \*red){

if (red->ulaz == red->izlaz) return 0;

red->izlaz++;

red->izlaz %=n;

\*element =red->polje[red->izlaz];

return 1;

}

*3)Inicijalizacija praznog reda*

void init\_stog(Stog \*stog){

stog->vrh = -1;

}

Red realiziran listom

U prethodnom poglavlju red smo realizirali statičkim poljem. Ta realizacija je imala veliki nedostatak

jer je postojala mogućnost prepunjenja polja. Sada ćemo red implementirati listom što će nam

omogućiti da stavimo u red gotovo neograničen broj elemenata. Jedino ograničenje predstavljat će

veličina radnog spremnika u računalu. Podsjetimo se koje operacije moramo ostvariti da bi red

ispravno radio

*Struktura podataka* koja opisuje red realiziran listom:

struct at{

tip element;

struct at \*sljed;

};

typedef struct at atom;

typedef struct {

atom \*ulaz, \*izlaz;

}Red;

*1) Dodavanje elementa u red*

int DodajURed(tip element, Red \*red){

atom \*novi;

if(novi=malloc(sizeof(atom))) {

novi->element=element;

novi->sljed=NULL;

if (red->izlaz==NULL) red->izlaz=novi; //ako je bio prazan

else red->ulaz->sljed=novi; //inace, stavi na kraj

red->ulaz=novi; //zapamti zadnjeg

return 1;

}

return 0;

}

*2)Skidanje elementa iz reda*

int SkiniIzReda(tip \*element, Red \*red) {

atom \*stari;

if (red->izlaz) { //ako red nije prazan

\*element = red->izlaz->element; //element koji se skida

stari = red->izlaz; //zapamti trenutni izlaz

red->izlaz = red->izlaz->sljed; // novi izlaz

free (stari); //oslobodi memoriju skinutog

if (red->izlaz == NULL) red->ulaz = NULL; //prazan red

return 1;

}

return 0;

}

*3)Inicijalizacija praznog reda*

void init\_red(Red \*red){

red->ulaz = NULL;

red->izlaz = NULL;

}

Linearna jednostruko povezana lista

Linearna lista je struktura podataka koja se sastoji od uređenog niza elemenata odabranih iz nekog

skupa podataka. Pojedini elemenat liste zvat ćemo atom, a smatrat ćemo da je lista prazna ukoliko je

njen broj elemenata 0. Osim statičkim poljem listu možemo realizirati i dinamičkom podatkovnom

strukturom koja se sastoji od pokazivača na prvi element liste i od proizvoljnog broja atoma.

Atom jednostruko povezane liste sastoji se od podatkovnog dijela i od pokazivača na sljedeći element liste:

*1) Dodavanje elementa u listu*

int dodaj(atom \*\*glavap, tip element){

atom \*novi, \*p;

if((novi=(atom \*)malloc(sizeof(atom)))==NULL) return 0;

novi->element=element;

if(\*glavap==NULL || (\*glavap)->element >= element){

novi->sljed=\*glavap;

\*glavap=novi;

}

else{

for(p=\*glavap; p->sljed && (p->sljed)->element < element; p=p->sljed);

novi->sljed=p->sljed;

p->sljed=novi;

}

return 1;

}

*2) Brisanje elementa iz liste*

int brisi(atom \*\*glavap, tip element){

atom \*p;

for(; \*glavap && (\*glavap)->element!=element; glavap=&((\*glavap)->sljed));

if(\*glavap){

p=\*glavap;

\*glavap=(\*glavap)->sljed;

free(p);

return 1;

}

else return 0;

}

*3) Traženje elementa u listi*

atom \*trazi(atom \*glava, tip element){

atom \*p;

for(p=glava; p!=NULL; p=p->sljed){

if(p->element==element) return p;

}

return NULL;

}

Binarno stablo

Stablo je konačan skup čvorova pri čemu je jedan čvor *korijen*, a ostali čvorovi podijeljeni su u

disjunktne podskupove od kojih je svaki stablo ‐ *podstabla*.

*‐ roditelj(i)=* i/2 za i 1; kada je i=1, čvor i je korijen, nema roditelja

*‐ lijevo\_dijete(i)=* 2\*i ako je 2\*i n; kad je 2\*i>n čvor i nema lijevog djeteta

*‐ desno\_dijete(i)=* 2\*i+1 ako je 2\*i+1n; kad je 2\*i+1>n čvor i nema desnog djeteta

* + *inorder:*  lijevo podstablo → korijen → desno podstablo
  + *preorder:* korijen → lijevo podstablo → desno podstablo
  + *postorder:* lijevo podstablo → desno podstablo → korijen

typedef struct cv{

tip element;

struct cv \*lijevo\_dijete;

struct cv \*desno\_dijete;

}cvor;

*1) Dodavanje čvora u stablo*

cvor \*upis(cvor \*korijen, tip element){

if(korijen==NULL){

korijen=(cvor\*)malloc(sizeof(cvor));

if(korijen){

korijen->element=element;

korijen->lijevo=korijen->desno=NULL;

}else{

printf("Ne mogu zauzeti memoriju za novi element!\n", element);

}

}else if(element < korijen->element){

korijen->lijevo=upis(korijen->lijevo, element);

}else if(element > korijen->element){

korijen->desno=upis(korijen->desno, element);

}else{

printf("Podatak %d vec postoji!\n", element);

}

return korijen;

}

*3) Traženje čvora u stablu*

cvor \*trazi(cvor \*korijen, tip element){

while(korijen && korijen->element != element){

if(element < korijen->element) korijen=korijen->lijevo;

else korijen=korijen->desno;

}

return korijen;

}