ASP završni

Povezane liste

Atom:

```
typedef struct atom {
  int element;
  struct atom *sljed;
} atom;
```

Dodavanje na početak:

```
int dodaj (atom **glavap, int element) {
  atom *novi;
  if ((novi = (atom *) malloc(sizeof(atom))) == NULL)
        return 0;
  novi->element = element;
  if (*glavap == NULL || (*glavap)->element >= element) {
    novi->sljed = *glavap;
    *glavap = novi;
  }
}
```

Dodavanje u sredinu:

```
int dodaj (atom **glavap, int element) {
  atom *novi, *p;
  if ((novi = (atom *) malloc(sizeof(atom))) == NULL)
        return 0;
  novi->element = element;
  for (p = *glavap; p->sljed &&(p->sljed)->element < elemen
t; p = p->sljed);
        novi->sljed = p->sljed;
        p->sljed = novi;
}
```

Brisanje s početka:

```
int brisi (atom **glavap, int elem) {
  atom *p;
  for (; *glavap && (*glavap)->elem != elem; glavap = &((*g
lavap)->sljed));
  if (*glavap) {
    p = *glavap;
    *glavap = (*glavap)->sljed;
    free (p);
        return 1;
  } else return 0;
}
```

Brisanje iz sredine:

```
int brisi (atom **glavap, int elem) {
  atom *p;
  for (; *glavap && (*glavap)->elem != elem; glavap = &((*g
lavap)->sljed));
  if (*glavap) {
    p = *glavap;
    *glavap = (*glavap)->sljed;
    free (p);
        return 1;
  } else return 0;
}
```

Ispis:

```
void ispis(atom *glava) {
    while (glava) {
        printf("%s\n", glava->element);
        glava = glava->sljed;
    }
}
```

Stog

Pojedina operacija dodaj ili brisi zahtijeva jednako vremena bez obzira na broj pohranjenih podataka.

Statičkim poljem

Definicija:

```
#define MAXSTOG 5
typedef struct {
    int vrh, polje[MAXSTOG];
} Stog;

void init_stog(Stog *stog){
    stog->vrh = -1;
}
```

Dodaj:

```
int dodaj (int element, Stog *stog) {
  if (stog->vrh >= MAXSTOG-1) return 0;
  stog->vrh++;
  stog->polje[stog->vrh] = element;
  return 1;
}
```

Skini:

```
int skini (int *element, Stog *stog) {
    if (stog->vrh < 0) return 0;
```

```
    *element = stog->polje[stog->vrh];
    stog->vrh--;
    return 1;
}
```

Listom

Definicija:

```
typedef struct atom {
  tip element;
  struct atom *sljed;
} atom;
typedef struct {
    atom *vrh;
} Stog;

void init_stog(Stog *stog){
    stog->vrh = NULL;
}
```

Dodaj:

```
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;
}
```

Skini:

```
int skini (tip *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;
}
```

Red

Red (queue) je linearna lista kod koje se umetanje u listu izvodi na jednom, a brisanje iz liste na njenom drugom kraju (načelo FIFO). Kraj reda na kojem se obavlja umetanje naziva se ulaz (stražnji kraj), drugi kraj je izlaz (prednji kraj).

Statičkim polje

Jedan element niza je prazan i time se omogućuje razlikovanje praznog i punog reda.

- prazan red: `ulaz == izlaz`
- puni red: `(ulaz + 1) % maxelemenata == izlaz`

Definicija:

```
typedef struct {
    tip polje[MAXRED];
    int ulaz, izlaz;
} Red;
```

Dodaj:

```
int dodaj (tip element, Red *red) {
 if ((red->ulaz+1) % n == red->izlaz) return 0;
 red->ulaz++;
 red->ulaz %= n;
 red->polje[red->ulaz] = element;
 return 1;
}
```

Skini:

```
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;
}
```

Listom

Definicija:

```
typedef struct atom {
    int element;
    struct atom *sljed;
} atom;
typedef struct {
    atom *ulaz, *izlaz;
} Red;

void init_red(Red *red){
    red->ulaz = NULL;
    red->izlaz = NULL;
}
```

Dodaj:

```
int dodaj (int 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 red bio prazan
            else (red->ulaz)->sljed = novi; // inace, stavi
 na kraj
        red->ulaz = novi;               // zapamti zadnjeg
        return 1;
    }
    return 0;
}
```

Skini:

```
int skini (int *element, Red *red) {
  atom *stari;
  if (red->izlaz) {                 // ako red nije prazan
    *element = red->izlaz->element;  // element koji se ski
da
    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;
}
```

Raspršeno adresiranje (hashing)

Za ključ k uvedemo funkciju koja mapira vrijednost ključa u indeks pod kojim se taj ključ nalazi u hash tablici.

Ako se više ključeva mapira na isto mjesto u hash tablici dolazi do problema kolizije. Kolizija se rješava: **ulančavanjem**, **linearnim ispitivanjem**, **kvadratnim ispitivanjem**, **dvostruko raspršenim adresiranjem**.

Ulančavanje

Ako je mjesto popunjeno, samo nižemo na *taj element* u povezanu listu *ostale elemente*.

Definicija:

```
typedef struct cvor {
    int broj;
    struct cvor *sljed;
} cvor;

// U glavnom
```

```
cvor **hash = (cvor **) malloc(sizeof(cvor *) * M);
// kraj u glavnom

void init_hash(cvor **hash) {
    int i;
    for (i = 0; i < M; i++) {
        hash[i] = NULL;
    }
}
```

Dodaj:

```
int dodaj(int broj, cvor **hash) {
    int adr = adresa(broj);
    cvor *novi = (cvor *) malloc(sizeof(cvor)), *p;
    if (novi == NULL) return 0;
    novi->broj = broj;
    novi->sljed = NULL;
    if (hash[adr] == NULL) {
        hash[adr] = novi;
    } else {
        p = hash[adr];
        while(p->sljed) p = p->sljed;
        p->sljed = novi;
    }
    return 1;
}
```

Linearno ispitivanje

Izračuna se adresa i onda se u for petlji dodaje `i` dok ne nađemo prazno mjesto. Uspis:

```
void Upis(struct zapis* hash, struct zapis element) {
  int indeks;
  int adresa = Adresa(element.sifra);
  for (int i = 0; i < M; i++){
    indeks = (adresa + i) % M;
    if (hash[indeks].sifra == 0) {
        hash[indeks] = element;
        break;
    }
  }
}
```

Kvadratno ispitivanje

Izračuna se adresa i onda se u for petlji dodaje `c1 * i + c2 * i * i` (c1, c2 proizvoljno biramo) dok ne nađemo prazno mjesto.

```
void Upis(struct zapis* hash, struct zapis element) {
  int indeks;
  unsigned long int adresa = Adresa(element.sifra);
  for (int i = 0; i < M; i++) {
    // fmod je modulo za float-ove
```

```
    indeks = fmod((adresa + c1*i + c2*i*i), M);
    if (hash[indeks].sifra == 0) {
       hash[indeks] = element;
       break;
    }
  }
}
```

Dvostruko raspršeno adresiranje

Imamo dvije neovisne funkcije za dobivanje adrese.

```
void Upis(struct zapis* hash, struct zapis element) {
  int indeks;
  int h1 = Adresa1(element.sifra);
  int h2 = Adresa2(element.sifra);
  for (int i = 0; i < M; i++) {
    indeks = (h1 +i*h2) % M;
    if (hash[indeks].sifra == 0) {
      hash[indeks] = element;
      break;
    }
  }
}
```

Grafovi

Ima vrhove i spojnice/bridove. Nisu bili u zadnje 3 godine na završnima.

Stabla

Skup čvorova. Postoji korijen (također čvor) i na njega se nižu “djeca”.

Stupanj je broj podstabala nekog čvora. Skup krajnjih čvorova su listovi. Stupanj stabla je maksimalni stupanj od svih čvorova nekog stabla.

Razina (level) nekog čvora određuje se iz definicije da je korijen razine 1, a da su razine djece nekog čvora razine k jednaki k+1. *U biti gleda se u kojem su čvorovi horizontalnom redu po redu.*

Dubina (depth) stabla je jednaka maksimalnoj razini nekog čvora u stablu.

Binarno stablo

Binarno stablo je stablo koje se sastoji od nijednog, jednog ili više čvorova **drugog** stupnja. (Svaki čvor ima maksimalno dvoje djece, minimalno niti jedno.) Kod binarnog stabla razlikujemo lijevo i desno podstablo svakog čvora.

Statičko polje

- `roditelj(i)= i/2` za `i != 1`
    - kada je `i == 1`, čvor i je korijen pa 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

Dinamičkom strukturom

Postoji 3 standardna načina obilaska stabla:

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

Definicija:

```
struct cvor {
  tip podatak;
  struct cvor *lijevo_dijete;
  struct cvor *desno_dijete;
};
```

Dodaj:

```
struct cvor* NoviCvor(int elem) {
    struct cvor* novi = (cvor *) malloc(sizeof(struct cvor)
);
    novi->podatak = elem;
    novi->lijevo = NULL;
    novi->desno = NULL;
```

```
    return(novi);
}
struct cvor* dodaj(struct cvor* cvor, tip elem) {
    if (cvor == NULL) {
        return(NoviCvor(elem));
    }
    else {
        if (elem <= cvor->podatak)
            cvor->lijevo = dodaj(cvor->lijevo, elem);
        else
            cvor->desno = dodaj(cvor->desno, elem);
        return(cvor);
    }
}
```

Traži:

```
int trazi (struct cvor* cvor, int trazeno) {
    if (cvor == NULL) {
        return 0;
    } else {
        if (trazeno == cvor->podatak) {
            return 1;
        } else {
            if (trazeno < cvor->podatak) {
                return(trazi(cvor->lijevo, trazeno));
            } else {
```

```
                return(trazi(cvor->desno, trazeno));
            }
        }
    }
}
```

Gomila

Prioritetni red je struktura podataka u kojem se skida prvo podatak koji ima najveću vrijednost (najveći prioritet). Najbolje se prikazuje gomilom.

Gomila je potpuno binarno stablo gdje se čvorovi mogu uspoređivati nekom uređajnom relacijom (npr. <=) i gdje je bilo koji čvor u smislu te relacije veći ili jednak od svoje djece. *Npr. kod maxheap-a roditelj je uvijek veći od svoje djece ili minheap-a gdje je roditelj uvijek manji od svoje djece.*

Ubacivanje elementa (ubaci) O(n * log(n)):

```
// element polja moze biti bilo sta
// zato se koristi typedef
typedef int tip;
// ubacuje vrijednost iz A[k] na gomilu pohranjenu u A[1:k-1]
void ubaci (tip A[], int j) {
  int i, k;
```

```
  tip novi;
  k = j;
  i = j/2;
  novi = A[j];
  while ((i > 0) && (A[i] < novi)) {
    A[k] = A[i]; // spusti roditelja na vecu razinu
    k = i;
    i /= 2;    // roditelj od A[i] je na A[i/2]
  }
  A[k] = novi;
}
```

Ubacivanje elementa (stvori, podesi) O(n):

```
// element polja moze biti bilo sta
// zato se koristi typedef
typedef int tip;
// potpuna binarna stabla s korijenima A[2*i]
// i A[2*i+1]kombiniraju se s A[i] formirajuci
// jedinstvenu gomilu
// 1 <= i <= n
void podesi (tip A[], int i, int n) {
  int j;
  tip stavka;
  j = 2*i;
  stavka = A[i];
  while (j <= n ) {
```

```
    // Usporedi lijevo i desno dijete (ako ga ima)
    if ((j < n) && (A[j] < A[j+1])) j++;
    // j pokazuje na vece dijete
    if (stavka >= A[j]) break;  // stavka je na dobrom mjes
tu
    A[j/2] = A[j];              // vece dijete podigni za r
azinu
    j *=2;
  }
  A[j/2] = stavka; // pohrani stavku
}
// premjesti elemente A[1:n] da tvore gomilu
void StvoriGomilu (tip A[], int n) {
  int i;
  for (i = n/2; i >= 1; i--)
        podesi (A, i, n);
}
```

Heapsort:

```
// Heap sort - podesavanje gomile
void Podesi (tip A[], int i, int n) {
  int j;
  tip stavka;
  j = 2*i;
  stavka = A[i];
  while (j <= n ) {
```

```
    if ((j < n) && (A[j] < A[j+1])) j++;
    if (stavka >= A[j]) break;
    A[j/2] = A[j];
    j *=2;
  }
  A[j/2] = stavka;
}

// Heap sort - inicijalno stvaranje gomile
void StvoriGomilu (tip A[], int n) {
  int i;
  for (i = n/2; i >= 1; i--)
    Podesi (A, i, n);
}

// Heap sort
void HeapSort (tip A[], int n) {
  // A[1:n] sadrzi podatke koje treba sortirati
  int i;
  StvoriGomilu (A, n);
  for (i = n; i >= 2; i--) {
        // Zamijeni korijen i zadnji list, skrati polje za
1 i podesi gomilu
        Zamijeni (&A[1], &A[i]);
    Podesi (A, 1, i-1);
  }
}
```