Strukture podataka

Hash

tablice

Definicija

Hash tablica je struktura podataka koja omogućuje direktan pristup podacima izračunavanjem njihovog položaja u polju na osnovu vrednosti ključa.

Hash funkcije

$|K|$ - kardinalnost skupa vrednosti ključeva (tj. broj elemenata),

N - broj elemenata koje treba smestiti u tablicu,

M – dimenzija polja

$\Rightarrow |K| >> N \wedge M \geq N$.

$|K| > M$, $\Rightarrow$ *hash* funkcija nema jednoznačno preslikavanje, tj. važi $(\exists (i, j) \in K)(i \neq j \Rightarrow h(i) = h(j))$.

Pojava da se različiti ključevi preslikavaju u istu adresu naziva se **kolizija**, a ključevi koji izazivaju koliziju nazivaju se **sinonimi**.

Karakteristike *hash* funkcija

Osnovni zahtevi koje treba da ispuni *hash* funkcija su:

- da adresira celu tablicu,
- ne generiše adrese van tablice.

Osobine **dobre** *hesh* funkcije su:

- da izbegava kolizije,
- da rasipa vrednosti ravnomerno po čitavoj tablici i
- da se jednostavno (tj. brzo) izračunava.

Specijalne vrste *hash* funkcija

- **perfektna hash funkcija** - različite ključeve mapira u različite adrese $(\forall (i, j) \in K)(h(i) = h(j) \Rightarrow i = j)$
- **minimalna perfektna hash funkcija** - perfektna hash funkcija kod koje je broj mogućih adresa u koje se preslikavaju ključevi jednak broju ključeva
  $f : k \rightarrow [0, |K| - 1]$
- **minimalna perfektna hash funkcija sa održavanjem redosleda** - $(\forall (i, j) \in K)(i > j \Rightarrow h(i) > h(j))$

Najčešće metode za izračunavanje *hash* funkcija

- metod **deljenja**,
- metod **sredine kvadrata**,
- metod **množenja**,
- **Fibonačijev** metod,
- metod **presavijanja**,
- metod **ekstrakcije** (izvlačenja) i
- metod **transformacije osnove**.

Metod deljenja

$$h(k) = k \bmod M$$

Za M se obično bira neparan broj ili stepen broja 2 ($M = 2^p$) ili prost broj. Najbrža za izračunavanje je funkcija oblika:

$$\mathbf{h(k) = k \bmod 2^p}\ ,$$

uzima poslednjih **p** bitova broja **k**, odnosno maskira ostale bitove. Ova funkcija nema dobru karakteristiku zato što ne zavisi od svih bitova broja **k**. Zato se M bira tako da bude prost broj.

Metod sredine kvadrata

Zasniva se na konačnoj preciznosti celobrojne aritmetike. Ako sa **w** označimo veličinu reči (odnosno broj bitskih pozicija za predstavljanje broja), tada je maksimalni broj koji se može predstaviti veličine **W = 2^w**, i sva izračunavanja se obavljaju po modulu tog broja.

$$h(k) = \left\lfloor \frac{M}{W}(k^2 \bmod W) \right\rfloor$$

$$\frac{W}{M} = 2^{w-p}$$

$$h(k) = (k * k) >> (w - p)$$

Metod množenja

Metod množenja je varijanta metoda sredine kvadrata, kod koga se ključ ne množi samim sobom, već nekom, pažljivo izabranom, konstantom **a**.

$$h(k) = \left\lfloor \frac{M}{W}(ak \bmod W) \right\rfloor$$

$$(a \cdot a') \bmod W = 1$$

Broj **a'** naziva se inverzni broj broja **a** po modulu W. Postojanje inverznog broja je vrlo važno jer omogućuje rekonstrukciju vrednosti dobijene proizvodom **ak**, odnosno množenjem adrese sa **a'** ponovo se dobija vrednost ključa, pa se ona ne mora pamtiti u tablici. Jedna od mogućih vrednosti koja se koristi za 32-bitnu aritmetiku je

a = 2 654 435 769, odnosno njegova inverzna vrednost **a'** = 340 573 321.

Fibonačijev metod

Ovaj metod zasniva se na "zlatnoj proporciji", koja se definiše kao odnos dva pozitivna cela broja **x** i **y** za koje važi:

$$\frac{x}{y} = \frac{x+y}{x} \Rightarrow x^2 - xy - y^2 = 0 \quad \Longrightarrow \quad \varphi^2 - \varphi - 1 = 0 \Rightarrow \varphi = \frac{1 \pm \sqrt{5}}{2}$$

φ = x/y

Recipročna vrednost pozitivnog korena

$$\varphi^{-1} = \frac{2}{1+\sqrt{5}} = \left(\frac{2}{1+\sqrt{5}}\right)\left(\frac{\sqrt{5}-1}{\sqrt{5}-1}\right) = \frac{\sqrt{5}-1}{2} \approx 0.618033887$$

Fibonačijev metod je metod množenja kod koga se konstanta **a** bira kao ceo broj, uzajamno prost sa **W**, najbliži vrednosti **φ^{-1}W**. Na primer, ako je W = 2^{16} za **a** se dobija 40503. Dobra osobina ove funkcije je da pravilno rasipa uzastopne vrednosti ključeva.

Metod presavijanja

Metod presavijanja sastoji se u podeli celobrojnog ključa na više delova nad kojima se zatim sprovodi neka aritmetička operacija. Na primer:

- $h_1(257346542) = (257+346+542) \text{ mod } M = 1145 \text{ mod } M$,
- $h_2(257346542) = (2+57+34+65+42) \text{ mod } M = 200 \text{ mod } M$

Metod h_1 biće pogodniji za veće (npr. M = 1000), a h_2 za manje tablice (npr. M = 100). Na osnovu veličine tablice treba izabrati broj cifara. Postoje varijante ovog metoda u kojima se okreće redosled cifara u podgrupama, na primer:

- $h_3(257346542) = (257+643+542) \text{ mod } M = 1442 \text{ mod } M$

Metod ekstrakcije

Metod ekstrakcije koristi izdvajanje dela ključa za izračunavanje adrese, na primer:

- $h_4(257346542) = 542$ - poslednje 3 cifre,
- $h_5(257346542) = 2573$ - prve 4 cifre,
- $h_6(257346542) = 25742$ - prve 3 + poslednje 2 cifre.

Maskiranje može da se obavi i na binarnoj reprezentaciji broja, čime ovo postaje jedna od najbržih funkcija za izračunavanje. Metod deljenja je specijalan slučaj ekstrakcije, ukoliko se koristi tablica veličine 2^p, obzirom da predstavlja izdvajanje **p** najnižih bitova ključa.

Metod transformacije osnove

Metod transformacije osnove koristi prevođenje ključa iz jedne brojne osnove u drugu. Na primer:

- $h_7(123_{10}) = 173_8 \bmod M$,
- $h_8(345_{10}) = 423_9 \bmod M$.

Tipovi ključeva

- celobrojni pozitivni br. - h: k$\rightarrow$\{0, 1, ..., M-1\}

- realni br.- npr. ako deo mantise broja predstavlja ključ. Funkcija **frexp** vraća mantisu i eksponent realnog broja *x*.

  double frexp(double *x*, int ****expptr* **);**

- karakter - k = (c_0, c_1, c_2, c_3, ..., c_{n-1})

$$h(k) = c_0 \bmod M \qquad h(k) = \sum_{i=0}^{n-1} c_i \qquad h(k) = \left(\sum_{i=0}^{n-1} 2^{a\cdot(n-i-1)} \cdot c_i \right) \bmod W$$

Primer izračunavanja hash funkcije

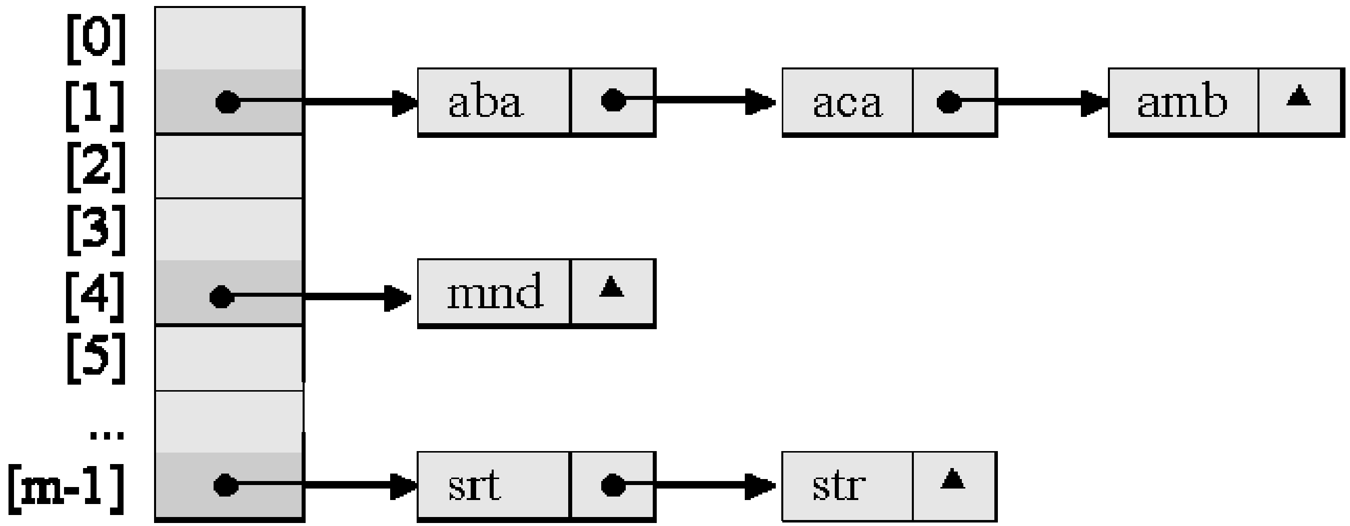
```
unsigned int hashCode (double d)
{
    if (d == 0)
            return 0U;
    else{
            int exponent;
            double mantissa = frexp (d, &exponent);
            return (unsigned int)(2 * fabs(mantissa) - 1) * ~0U;
    }
}
```

```
unsigned int hashCode (char* s)
{
    unsigned int res = 0;
    unsigned int a = 7;
    for (int i = 0; s[i] != 0 ; i++)
            res = res << a ^ s[i];
    return res;
}
```

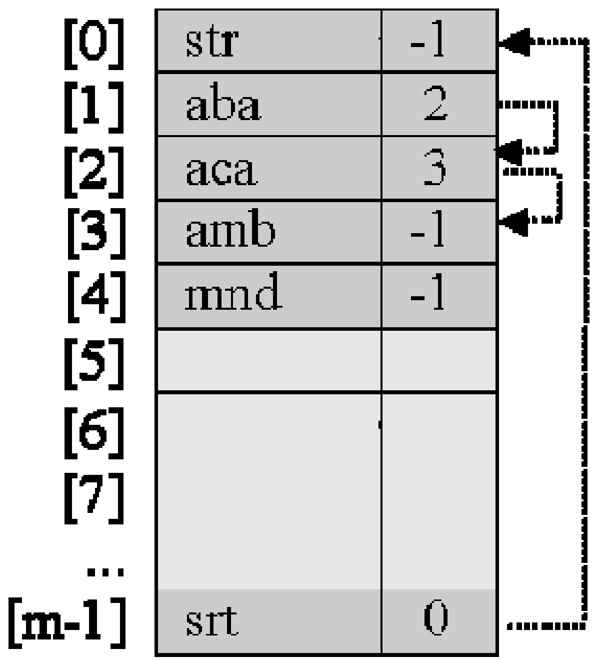
Rešavanje kolizije

- spoljašnje ulančavanje sinonima,
- unutrašnje ulančavanje sinonima i
- otvoreno adresiranje

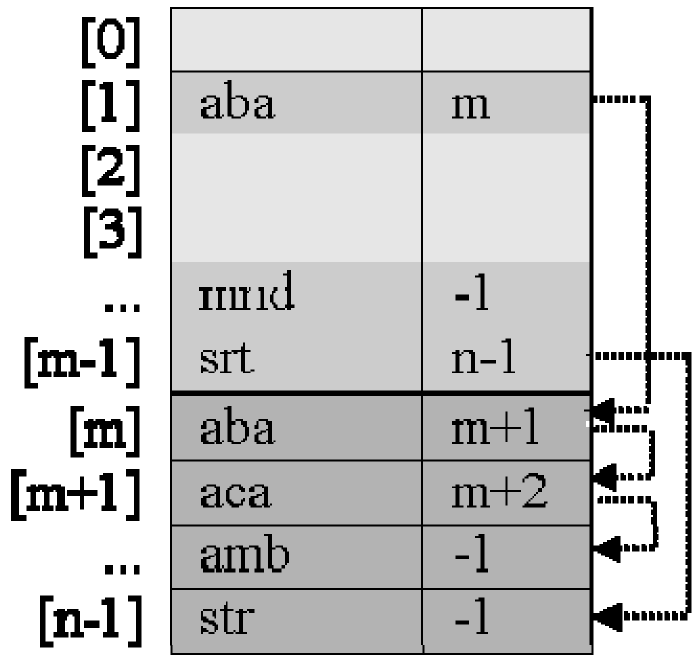
Spoljašnje ulančavanje sinonima



Unutrašnje ulančavanje sinonima



Jedinstveni adresni prostor



Adresni prostor podeljen u dve particije

Otvoreno adresiranje

Ukoliko *hash* funkcija ne vrati adresu sa slobodnom lokacijom, kod tablica sa otvorenim adresiranjem, nastavlja se sa traženjem slobodne lokacije korišćenjem **sekundarne transformacije**.

$$h_i(k) = (h(k) + c(i)) \bmod M, \quad i = 0, 1, \ldots, M - 1$$

Funkcija c(i) naziva se sekundarna transfomacija i treba da ima sledeća dva svojstva:

• c(0) = 0, čime se obezbeđuje da se prvi pokušaj poklopi sa primarnom (hash) transformacijom i

• skup { c(0) mod M, c(1) mod M, ... c(M-1) mod M } mora sadržati sve cele brojeve iz intervala [0, M-1], kako bi se obezbedilo adresiranje čitavog adresnog prostora tablice.

Najčešće korišćene sekundarne transformacije

- **linearno traženje** - $c(i) = \alpha \cdot i$, gde je α uzajamno prost broj sa M. U slučaju kada je $\alpha \neq 1$ ova transformacija se naziva modifikovano linearno traženje.
- **kvadratno traženje** - $c(i) = \alpha \cdot i^2$
- **sekundarna hash funkcija** - $c(i) = i \cdot h'(k)$, gde je $h'(k)$ *hash* funkcija različita od primarne transformacije.

Srednji broj pristupa

Ako **faktor popunjenosti** (engl. *load factor*) označimo sa FP i definišemo kao:

FP = broj_elemenata_u_tablici / veličina_tablice

Tabela 5.1	**Srednji broj pristupa tablici pri uspešnom i neuspešnom traženju**		
	linearno traženje	**kvadratno traženje**	**sekundarna hash funkcija**
uspešno traženje	$\frac{1}{2}\left(1+\frac{1}{1-\mathrm{FP}}\right)$	$1-\ln(1-\mathrm{FP})-\frac{\mathrm{FP}}{2}$	$\frac{1}{\mathrm{FP}}\ln\frac{1}{1-\mathrm{FP}}$
neuspešno traženje	$\frac{1}{2}\left(1+\frac{1}{(1-\mathrm{FP})^2}\right)$	$\frac{1}{1-\mathrm{FP}}-\mathrm{FP}-\ln(1-\mathrm{FP})$	$\frac{1}{1-\mathrm{FP}}$

Rasute tablice

Hash tablice koje koriste unutrašnje ulančavanje ili otvoreno adresiranje za rešavanje problema sinonima, odnosno tablice koje se u potpunosti mogu implementirati preko strukture tipa polja, nazivaju se **rasute tablice** (engl. *scatter table*).

HashObject – element *hash* tablice

```
template <class T, class R>
class HashObject
{
protected:
        T key;
        R* record;
public:
        HashObject() { key = (T)0; record = NULL; }
        HashObject(T k) { key = k; record = NULL; }
        HashObject(T k, R* object) { key = k; record = object; }
        ~HashObject() { deleteRecord(); }
        HashObject& operator = (HashObject const& obj){
                key = obj.key;
                record = obj.record;
                return *this;
        }
        bool operator == (HashObject const& obj){
                return record == obj.record;
        }
        void deleteRecord() { if(record) {delete record; record = NULL;} }
        T getKey() {return key;}
        R* getRecord() {return record;}
        bool isEqualKey(T k) {return key==k;}
        void print(){ cout << key << "|" << record;}
};
```

HashTable

klasa apstraktne *hash* tablice

```
template <class T, class R>
class HashTable
{
protected:
            unsigned int length; // velicina tablice
            unsigned int count;  //broj elemenata u tablici
protected:
            unsigned int h(HashObject<T,R> obj){
                        return (f(obj.getKey())%length);
            }

// primarna transformacija
virtual unsigned int f(int i){ return abs(i); }
virtual unsigned int f(double d) {
            if (d == 0) return 0;
            else
            {
                        int exponent;
                        double mantissa = frexp (d, &exponent);
                        return (unsigned int) ((2 * fabs(mantissa) - 1) * ~0U);
            }
    }
```

HashTable

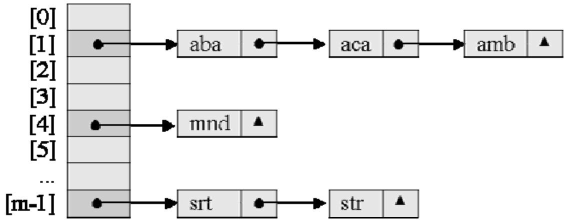
klasa apstraktne *hash* tablice

```
            virtual unsigned int f(char* s)
            {
                        unsigned int res = 0;
                        unsigned int a = 7;
                        for (int i = 0; s[i] != 0 ; i++)
                                    res = res << a ^ s[i];
                        return res;
            }
            // sekundarna transformacija
            virtual unsigned int g(unsigned int i)
            {
                        return (i + 1) % length;
            }

public:
            unsigned int getLength() { return length;}

            virtual double getLoadFactor (){
                        return (double)count / (double)length;
            }
};
```

ChainedHashTable

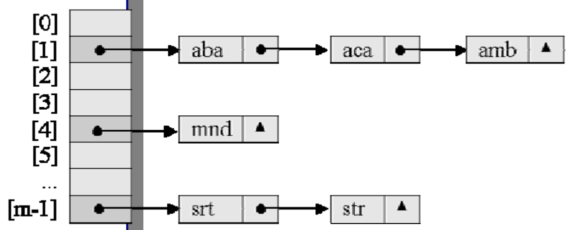



```
template <class T, class R>
class ChainedHashTable : public HashTable<T,R>
{
protected:              // polje lančanih listi
        SLList<HashObject<T,R> >* array;

public:
        ChainedHashTable (unsigned int len){
                length = len;
                count = 0;
                array = new SLList<HashObject<T,R> >[len];
}
        ~ChainedHashTable(){
                HashObject<T,R> obj;
                for(unsigned int i=0; i<length; i++){
                        try{
                                obj = array[i].getHeadEl();
                                while(true){
                                        obj.deleteRecord();
                                        obj = array[i].getNextEl(obj);
                                }
                        }
                        catch(SBPException* e){}
                }
                delete [] array;
        }
```

ChainedHashTable

```
void insert(HashObject<T,R> obj){
            array [h(obj)].addToHead(obj);
            count++;
}
void withdraw(HashObject<T,R> obj){
            array [h(obj)].deleteEl(obj);
            count--;
}
void withdraw(T key){
            HashObject<T,R> obj = find(key);
            withdraw(obj);
}
HashObject<T,R> find (T key){
            HashObject<T,R> obj;
            unsigned int i = f(key)%length;
            obj = array[i].getHeadEl();
            while(!(obj.isEqualKey(key)))
                        obj = array[i].getNextEl(obj);
            return obj;
  }
};
```



ScatterObject – element rasute tablice

```
 template <class T, class R>
class ScatterObject : public HashObject<T,R>
{
public:
		int status; // 0-slobodan, 1-obrisan, 2-zauzet
public:
		ScatterObject() : HashObject<T,R>(){ status = 0; }
		ScatterObject(T k) : HashObject<T,R>(k){ status = 0; }
		ScatterObject(T k, R* object) : HashObject<T,R>(k){ status = 0; }
};
```

ChainedScatterObject - element rasute tablice sa ulančavanjem sinonima

```
 template <class T, class R>
class ChainedScatterObject : public ScatterObject<T,R>
{
public:
		long next;
public:
		ChainedScatterObject() : ScatterObject<T,R>() {next=-1;}
		ChainedScatterObject(T k) : ScatterObject<T,R>(k) {next=-1;}
		ChainedScatterObject(T k, R* object)
		: ScatterObject<T,R>(k,object) {next=-1;}
		ChainedScatterObject(T k, R* object, unsigned int n)
				: ScatterObject<T,R>(k,object) {next=n;}
};
```

ChainedScatterTable

```
 template <class T, class R>
class ChainedScatterTable : public HashTable<T,R>
{
protected:
        ChainedScatterObject<T,R>* array;

public:
        ChainedScatterTable (unsigned int len){
                length = len;
                count = 0;
                array = new ChainedScatterObject<T,R>[len];

        }

        ~ChainedScatterTable(){ delete [] array; }
```

Umetanje elementa u rasutu tablicu sa ulančavanjem sinonima

```
void insert(ChainedScatterObject<T,R> obj)
{
        if(count == getLength ())
                throw new SBPException("The table is full!");
        long probe = h(obj);
        if(!array[probe].free)
        {
                while (array [probe].next != -1)
                {
                        probe = array [probe].next;
                }
                long tail = probe;
                probe = g(probe);
                while (!array [probe].free && probe!=tail)
                        probe = g(probe);
                if(probe==tail)
                    throw new SBPException("Poor secondary transformation!");
                array [tail].next = probe;
        }
        array[probe] = obj;
        array[probe].status = 2; // zauzet
        array[probe].next = -1;
        count++;
}
```

Traženje elementa u rasutu tablicu sa ulančavanjem sinonima

```
ChainedScatterObject<T,R> find(T key)
{
        long probe = f(key)%length;
        while(probe != -1)
        {
                if(!array[probe].isEqualKey(key))
                        probe = array[probe].next;
                else
                        return array[probe];
        }
        throw new SBPException("Element not found!");
}
```

Brisanje

```
void withdraw (T key)
{
        if (count == 0) throw new SBPException("Table is empty");
        long probe = f(key)%length;
        long prev = -1;
        while (probe != -1 && !array[probe].isEqualKey(key)){
                prev = probe;
                probe = array [probe].next;
        }
        if (probe == -1)
                throw new SBPException("Element not found!");
        if( prev != -1) { // brise se sinonim
                array[prev].next = array[probe].next;
                array[probe].deleteRecord();
                array[probe].status = 1; // obrisan
        }
```

Brisanje - nastavak

```
        else{
                if(array[probe].next == -1){
                        array[probe].deleteRecord();
                        array[probe].status = 1; // obrisan
                }
                else{
                        long nextEl = array[probe].next;
                        array[probe].deleteRecord();
                        array[probe] = array[nextEl];
                        array[probe].next = array[nextEl].next;
                        array[nextEl] = ChainedScatterObject<T,R>();
                        array[nextEl].status = 1; // obrisan
                }
        }
        count--;
}
```

OpenScatterTable – klasa rasute tablice sa otvorenim adresiranjem

```
 template <class T, class R>
class OpenScatterTable : public HashTable<T,R>
{
protected:
		ScatterObject<T,R>* array;
public:
		OpenScatterTable (unsigned int len){
				length = len;
				count = 0;
				array = new ScatterObject<T,R>[len];

		}
		~OpenScatterTable()
		{
				delete [] array;
		}
...
```

OpenScatterTable

```
unsigned int findUnoccupied (ScatterObject<T,R> obj)
{
        unsigned int hash = h(obj);
        unsigned int probe = hash;
        if(array[probe].status < 2)  return probe;
        do{
                probe = g(probe);
                if (array[probe].status < 2)  return probe;
        }while(probe!=hash);
        throw new SBPException("The table is full");
}

long findMatch (T key)
{
        unsigned int probe = f(key)%length;
        for (unsigned int i = 0; i < length; i++)
        {
                if (array[probe].status == 0) return -1;
                if (array[probe].isEqualKey(key)) return probe;
                probe = g(probe);
        }
        return -1;
}
```

OpenScatterTable

```
void insert(ScatterObject<T,R> obj){
        if(count == getLength ())
                throw new SBPException("The table is full!");
        unsigned int offset = findUnoccupied (obj);
        array[offset] = obj;
        array[offset].status = 2; // zauzet
        count++;
}

ScatterObject<T,R> find(T key) {
        long offset = findMatch(key);
        if (offset >= 0)
                return array[offset];
        throw new SBPException("Element not found!");
}

void withdraw (T key){
        if (count == 0) throw new SBPException("Table is empty");
        long offset = findMatch(key);
        if (offset < 0) throw new SBPException ("Object not found!");
        array[offset].status = 1; // obrisan
        array[offset].deleteRecord();
        count--;
}
```