Slovenská technická univerzita

Dátové štruktúry a algoritmy

Vypracované otázky do roku 2009

Peter Macko a kolektív fiitkara

Otvorené otázky

Aký je najhorší prípad časovej zložitosti sprístupnenia prvku v tabuľke s n rozptýlenými prvkami (hash tabuľke) so zreťazením?

O(n)

Ako zobrazí rozptylová funkcia prvky?

Všetky prvky budú v tomto prípade umiestnené v jednom prvku tabuľky. A teda ak dáme vyhľadať posledne vložený prvok je nutné prejsť celý spájaný zoznam až do konca.

Čo označuje premenná W a D v nasledujúcom algoritme?

```
FLOYD-WARSHALL'(W)
1  n ← rows[W]
2  D ← W
3  for k ← 1 to n
4       do for i ← 1 to n
5            do for j ← 1 to n
6                 do d_ij ← min(d_ij, d_ik + d_kj)
7  return D
```

W označuje: je to matica, v ktorej sú uložené dĺžky jednotlivých hrán grafu.

D označuje: je to matica, v ktorej sa nachádzajú najkratšie cesty medzi bodmi

Initial-single-source a relax

V jednej algoritmickej formulácii Bellmanovho-Fordovho algoritmu pre hľadanie nejkratších ciest z daného východiska v orientovanom grafe sa používajú funkcie initialize-single-source a relax (pozri nižšie). Napíšte ich.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to |V[G]| − 1
3       do for each edge (u, v) ∈ E[G]
4            do RELAX(u, v, w)
5  for each edge (u, v) ∈ E[G]
6       do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

4 initialize-single-source
2 relax
3 v akom prípade vracia algoritmus true a v akom false?

```
Initialize-single-source(G,s){
```

```
        for each vertex v ∈ V[G] do {
                d[v] = infinity;
                π[v] = nil;
        }
        d[s] = 0;
}
```

```
Relax(u,v,ω) {
        if (d[v] > d[u] + ω(u,v)) then {
                d[v] = d[u] + ω(u,v);
                π[v] = u;
        }
}
```

Algoritmus vracia flase, ak sa v grafe nachádza cyklus negatívnej cesty

V opačnom prípade vracia true.

Napíšte algoritmus distributívneho usporiadania (bucket sort)

```
Bucket-sort(A){
        n = length(A);
        for i=0 to n-1 do {
                insert A[i] into B[[nA[i]]]
        }
        n = lenght(B);
        for i=0 to n-1 do {
                usporiadaj B[i] insert sort-om
        }
        Spoj zoznamy B[0],B[1]…B[n-1] dohromady v správnom poradí
}
```

(4b) Definujte AVL strom

V AVL strome platí že rozdiel medzi výškou jeho podstromov je v interval **<-1, 1>**

Takisto platí pre každý koreň, že hodnota jeho ľavého dieťaťa je **nižšia,** ako hodnota koreňa a hodnota pravého nasledovníka je **vyššia**, ako hodnota koreňa.

(2b) Aký je odhad časovej zložitosti sprístupnenia ľubovoľného prvku v najhoršom prípade?

$O(\log_2 n)$

(5b) Napíšte algoritmus usporadúvania vkladaním (insert sort)

```
Insert-sort(A,n){
        for j=2 to n do {
                key=A[j];
                i=j-1;
                while i>0 and A[i]>key do {
                        A[i+1]=A[i];
                        i--;
```

```
                }
                A[i+1]=key;
        }
}
```

(2b) Aký je odhad jeho časovej zložitosti?

O(n^2)

(2b) Aký je odhad jeho pamäťovej zložitosti

O(1) – nevytvárajú sa žiadne pomocné polia

(4b) Uvažujte známy algoritmus vyhľadávania v binárnom vyhľadávacom strome. Napíšte jeho iteratívnu verziu.

```
Iterative-tree-search(T,k){
        While T<>nil and k<>DATA(T) do
                If(k<DATA(T))
                        then T = LCHILD(T);
                        else T = RCHILD(T);
        return T;
}
```

Uvažujte algoritmus znárodneného rýchleho usporiadanie

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2     then q ← RANDOMIZED-PARTITION(A, p, r)
3          RANDOMIZED-QUICKSORT(A, p, q − 1)
4          RANDOMIZED-QUICKSORT(A, q + 1, r)
```

(4b)Napíšte funkciu randomized-partition

```
Randomuzed-partition(A,p,r){
        i = random(p,r);
        Exchange(A[r],A[i]);
        return Partition(A,p,r);
}
```

```
Partition(A,p,r) {
        x = A[r];
        i = p;
        for (j = p to r-1) do {
                if (A[j] <= x) then {
                        Exchange(A[i],A[j]);
                        i++;
                }
        }
        Exchange(A[i],A[r]);
        return i;
}
```

(7b) Vysvetlite a algoritmom implementujte operáciu HEAPIFY (bolo to popísané nejakou omáčkou, ale v zásade išlo o toto)

```
HEAPIFY(heap, i) {
	l = left(i);
	r = right(i);

	if (l <= heap-size(heap) and heap[l] > heap[i])
		then largest = l;
		else largest = i;

	if( r<= heap-size(heap) and heap[r] > heap[largest])
		then largest = r;

	if (largest != i) then {
		Exchange(heap[i],heap[largest]);
		HEAPIFY (heap, largest);
	}
}
```

```
int left(i){ // v pripade ze zaciatok pola ma index 0
	return 2*i+1; // v pripade ze zaciatok pola je 1: 2*i
}
```

```
int right(i) { // v pripade ze zaciatok pola ma index 0
	return 2*i+2; // v pripade ze zaciatok pola je 1: 2*i+1
}
```

Funkcia heapify zabezpečuje to, aby bola daná množina haldou. To zabezpečí tak, že usporiada ľavý aj pravý podstrom podľa pravidiel binárnej hlady.

Funkcia heapify dostáva ako parameter pole heap – haldu a premenu *i* identifikujúcu prvok, ktorý treba skontrolovať. Na začiatku sa zistí, či ľavý potomok prvku *i* existuje, a či je väčší ako *i*. Väčší z nich sa potom priradí do premennej *largest*. Ďalej sa testuje, či existuje pravý potomok prvku i, a či je väčší ako doteraz nájdene maximum, ak hej priradí sa ako hodnota premennej *largest*.

V poslednom kroku funkcie sa zisťuje či najväčší prvok nie je *i*, lebo ak by tomu tak bolo ďalšia kontrola nie je potrebná a prvok *i* je správne umiestnený. V opačnom prípade sa vymení prvok *i* s najväčším potomkom a ide sa kontrolovať, či bol nový prvok dobre umiestnený (rekurzívne tou istou funkciou).

Vyber max. prvku z haldy

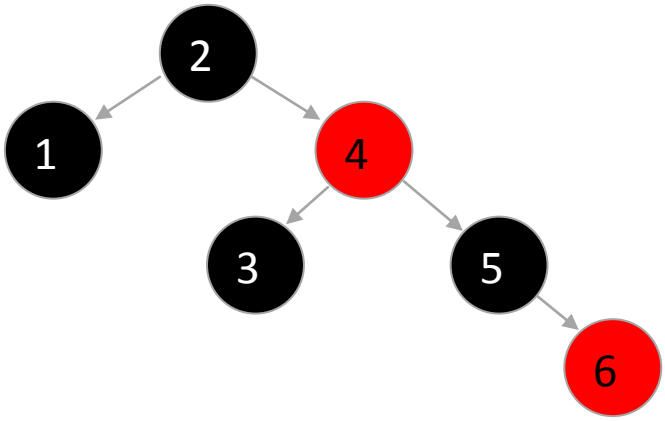
```
Extract-max(heap){
	if (heap-size(heap) < 1)
	then error;

	Max = heap[0];
	Heap[0] = heap[heap-size(heap)-1];
```

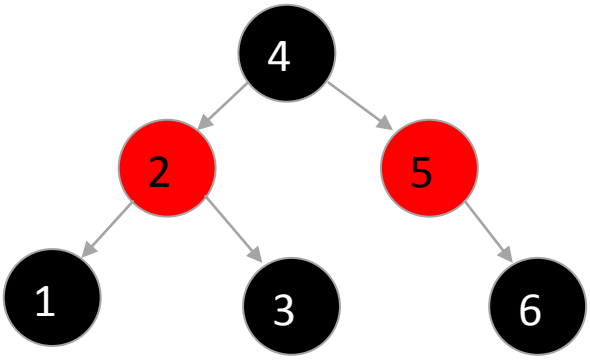
```
        Set-heap-size(heap-size(heap)-1);
        HEAPIFY(heap,0);
        return max;
}
```

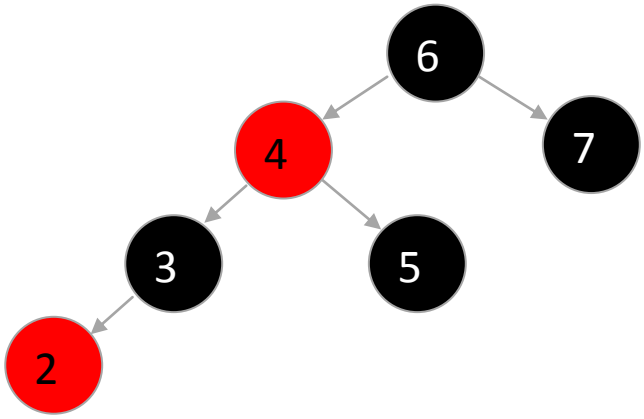
(4b) Definujte čierno-červený strom

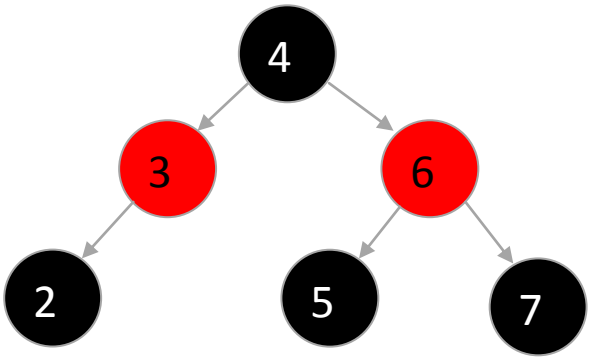
Červeno čierne stromy sú odvodené z binárnych vyvažovacích stromov. Na rozdiel od nich však obsahujú navyše príznak farby. Ten môže byť buď červený alebo čierny. Ďalej platia pravidlá, že koreň je vždy čierny, deťmi červeného prvku môžu byť iba čierne prvky. Od koreňa ku každému listu musí byť rovnaký počet čiernych vrcholov.

(3b) Na príklade predveďte pravú a ľavú rotáciu v červeno-čiernom strome.









(2b) Napíšte asymptotický odhad výšky 2-3 stromu o n prvkov

$O(\log_2 n)$

MERGE SORT

(5b) implementujte algoritmus

```
Merge-sort(A,p,r){
        if (p < r){
                q = ⌊(p + r)/2⌋;
                Merge-sort(A, p, q);
                Merge-sort(A,q+1,r);
                Merge(A,p,q,r);
```

```
        }
}
```

```
Merge(A,p,q,r){
        s = q - p + 1;
        t = r - q;
        L = A[p..q];
        R = A[q+1..r];
        L[s+1] = infinity;
        R[t+1] = infinity;
        A[p..r]= MergeArray(L,R);
}
```

```
MergeArray(L,R){
        i = 1;
        j = 1;
        for k=1 to s+t do
                if (L[i] <= R[j]) then {
                        A[k] = L[i];
                        i++;
                } else {
                        A[k] = R[j];
                        j++;
                }
        }
}
```

(2b) napíšte asymptotický odhad rastu časovej zložitosti

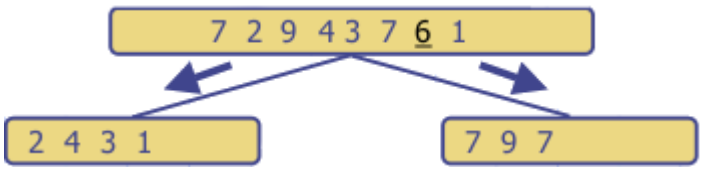
O(n log n) - vo všetkých prípadoch

(2b) napíšte asymptotický odhad rastu pamäťovej zložitosti

O(n)

(8b) QUICK SORT Nepamätám si presné zadanie, každopádne boli 4 podúlohy po 2 body a chceli všetky tieto veci:

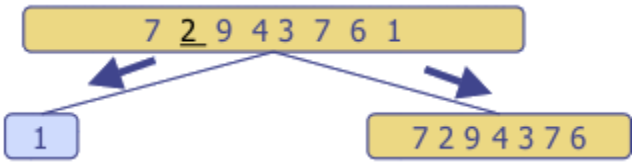
a) **aká je asymptotická zložitosť v najlepšom prípade, uveďte príklad naň príklad**
v najlepšom prípade je pivot umiestnený presne v strede poľa, a preto sa interval rozdelí na dve rovnaké polovice. V tomto prípade je zložitosť O(n log n)



b) **aký pivot je najvhodnejší**
najvhodnejší pivot je taký, ktorý rozdelí postupnosť na dve rovnako veľke (+-1) postupnosti

c) aká je asymptotická zložitosť v najhoršom prípade, uveďte príklad

Najhorší prípad je ak je pivot stanovený na hraničný bod postupnosti, teda buď na maximum alebo minimum. V tomto prípade je zložitosť $O(n^2)$



d) aká je asymptotická zložitosť v priemernom prípade

O(n log n)

formálna špecifikácia binárneho stromu

binárny strom je graf, ktorý neobsahuje cykly pričom každý prvok obsahuje maximálne dvoch potomkov. Tí sa označujú ako pravý a ľavý. Tieto pravidlá musia rekurzívne platiť pre všetky prvky stromu.

implementácia Dijkstrov alg. na výpočet najkratších ciest z vrcholu do ostatných vrcholov v grafe

```
Dijstras(G, ω, s){
	Initialize-single-source(G,s);
	S = 0;
	Q = V[G];
	while Q != 0 do {
		u = Extract-min(Q);
		S = S U {u};
		for each vertex v Є Adj[u] do
			Relax(u,v,ω);
	}
}
```

algoritmus na zistenie najväčšieho prvku v BVS

```
get-max-bvs(graf){
	while(graf.right != NULL) do
		graf = graf.right;

	return graf.value;
}
```

triedenie prirodzeným zlučovaním

```
# Original data is on the input tape; the other tapes are blank
 function merge_sort(input_tape, output_tape, scratch_tape_C, scratch_tape_D)
     while any records remain on the input_tape
         while any records remain on the input_tape
             merge( input_tape, output_tape, scratch_tape_C)
             merge( input_tape, output_tape, scratch_tape_D)
```

```
        while any records remain on C or D
            merge( scratch_tape_C, scratch_tape_D, output_tape)
            merge( scratch_tape_C, scratch_tape_D, input_tape)

# take the next sorted chunk from the input tapes, and merge into the single
# given output_tape.
# tapes are scanned linearly.
# tape[next] gives the record currently under the read head of that tape.
# tape[current] gives the record previously under the read head of that tape.
# (Generally both tape[current] and tape[previous] are buffered in RAM ...)
function merge(left[], right[], output_tape[])
    do
       if left[current] ≤ right[current]
           append left[current] to output_tape
           read next record from left tape
       else
           append right[current] to output_tape
           read next record from right tape
   while left[current] < left[next] and right[current] < right[next]
   if left[current] < left[next]
       append current_left_record to output_tape
   if right[current] < right[next]
       append current_right_record to output_tape
   return;
```

Shellovo triedenie

```
Shell-sort(pole, n){
      inc = round(n/2);
      while inc > 0 do {
              for i = inc to n do {
                      temp = pole[i];
                      j = i;
                      while j >= inc and pole[j - inc] > temp do {
                              pole[j] = pole[j - inc];
                              j = j - inc;
                      }
                      pole[j] = temp;
              }
              inc = round(inc/2.2);
              // moze byť aj inc = inc/2; za predpokladu ze inc je
              // integer a zaokruhli sa smerom nadol.
      }
}
```

(5b) Abstraktný typ údajov množina treba špecifikovať

súčasťou každého prvku množiny je údaj (kľúč), ktorý ho jednoznačne určuje

najdôležitejšie operácie nad množinou sú okrem vytvorenia vkladanie prvku, zrušenie prvku, nájdenie hodnoty prvku, zistenie, či prvok patrí do množiny a operácia, či je nejaký údaj množinou.

Uveďte formálnu špecifikáciu. Ako sa takýto druh množiny zvykne nazývať?

```
Mnozina              Create();
Mnozina              Insert(mnozina, prvok);
mnozina              Delete(mnozina, prvok);
elementValue  Value(mnozina, prvok);
boolean              IsIn(mnozina, prvok);
boolean              IsSet(prvok);
```

Takýto druh množiny sa nazýva **SET**.

(5b) Napíšte algoritmus na výpočet hĺbky binárneho stromu.

```
get-depth(tree){
      if (tree == NULL) then return 0;
      else {
            l = get-depth(tree->lchild);
            r = get-depth(tree->rchild);
      }
      if( l > r)
            then return l + 1;
            else return r + 1;
}
```

(6b) Napíšte algoritmus na pridanie prvku do binárnej haldy.

```
Heap-insert(heap, node){
      Set-heap-size(heap-size(heap)+1,heap);
      i = heap-size(heap)-1;

      while (i > 1 and node > heap[parent(i)]) do {
            heap[i] = heap[parent(i)];
            i = parent(i);
      }
      heap[i] = node;
}
```

(6b) Operácia PUSH abstraktného typu údajov zásobník je implementovaná funkciou v jazyku C s nasledujúcou deklaráciou:

void push(STACK *s, int i);

Vysvetlite, nakoľko takáto implementácia operácie PUSH zodpovedá jej formálnej špecifikácii a ak v niečom nezopovedá, v čom?

Zodpovedá čiastočne v tom, že sa do nej vkladá prvok do premennej STACK

Nezodpovedá v tom, že

- funkcia implementuje operáciu PUSH špecializovanú na prvky typu int,

- funkcia nevracia ako hodnotu zásobník v stave po vykonaní operácie

(vracia void a nie údaj typu STACK)

(6b) Abstraktný typ údajov pole treba špecifikovať tak, aby nebolo prípustné sprístupňovanie prvku poľa s indexom, ktorý je mimo deklarovaných hraníc. Napíšte axiómu alebo axiómy, ktoré túto vlastnosť zakotvia.

A Є array, i Є index, x Є value

WRITE(A,x,i) = if (i > LOW(A) and i < HIGH(A)) then WriteArray(A,x,i); else error;

(6b) Pri jednom spôsobe dokazovania čiastočnej správnosti programov odvodzovaním príslušného tzv. induktívneho výrazu sa používajú axiómy a odvodzovacie pravidlá. Napíšte odvodzovacie pravidlo pre cyklus.

{p and t} B {p}

_____

{p} WHILE t do B {p and not t}

(5b) Napíšte stav priebehu usporadúvania postupnosti položiek [852, 362, 902, 156, 752, 264, 365, 196, 062, 756, 095, 891, 666] podľa algoritmu prirodzeného zlučovania na 4 páskach po druhom kroku (čo je zapísané na každej z pások).

1.	2.	3.	4.
362	852	362	156
852	156	902	264
902	752	264	365
062	196	365	752
196	095	062	095
756	891	756	666
666	891		

Úplná správna odpoveď je aj ak napíše len prvý a štvrtý stĺpec.

Ak to niekto usporadúval zostupne, riešenie je iné, ale treba ho uznať.

(10b) Napíšte algoritmus hľadania tranzitívneho uzáveru v orientovanom grafe.

```
Transitive-closure(G){
      n = |V[G]|;
      for i = 0 to n do
            for j = 0 to n do
                  if (i = j or (i,j) Є E[G])
                  then t_ij^(0) = 1;
                  else t_ij^(0) = 0;
      for k = 0 to n do
            for i = 0 to n do
                  for j = 0 to n do
                        t_ij^(k) = t_ij^(k-1) V (t_ik^(k-1) Λ t_kj^(k-1));
      return T^(n);
```

```
}
```

(11 b) Je daná nasledujúca postupnosť prvkov v poli *A = (5, 13, 2, 22, 0, 17, -4, 8,4).* Vysvetlite princíp činnosti algoritmu usporiadania haldou HEAPSORT tak, že popíšete jednotlivé fázy algoritmu následne pre postupnosť A znázornite detailne postupnosť krokov jeho výpočtu vo všetkých jeho fázach. Vyznačte, ktorá časť reprezentuje haldu a ktorá predstavuje čiastkové výsledky usporiadania.

```
heap-sort(){
	heap-build(heap);
	for i = length(A)-1 downto 1 do {
		Exchange(a[0],a[i]);
		Set-heap-size(heap-size(heap)-1,heap);
		Heapify(heap,0);
	}
	Return heap;
}
```

Algoritmus heapsort začína tým, že si vytvorí haldu tak, že zbehne známy algoritmus heapify na prvky od jeho polovice po koniec. To zabezpečí, že všetky prvky budú zrovnané tak, ako to vyžaduje halda (potomkovia musia mať vždy nižšiu hodnotu ako rodič).

Potom nasleduje cyklus ktorý najskôr zamení najväčší prvok haldy s prvkom na konci haldy (teda jedným z najmenších). Následne zmení veľkosť haldy o jednu a spustí algoritmus hepify. Keďže zmenil veľkosť haldy, najväčší prvok je umiestnený na konci a nie je ho možné v tomto algoritme preniesť. Vďaka tomu sa na vrch dostane ďalší najväčší prvok zo zostávajúcej postupnosti.

Tento algoritmus pokračuje až po prvok s indexom 2 keďže potom ostáva iba jeden prvok je jasné že je najmenší, a preto ostáva na svojom prvom mieste.

```
Heap-build(heap){
	Set-heap-size(length(heap),heap);
	n = ⌊heap − size(heap)/2⌋;

	for i = n downto 0 do {
		heapify(heap,i);
	}
}
```

```
Heapify(heap, i){
	l= lchild(i);
	r = rchild(i);

	if( l <= heap-size(heap) and heap[l]> heap[i])
		then largest = l;
		else largest = i;

	if (r<= heap-size(heap) and heap[r]>heap[largest])
		then largest = r;
```

```
        if (largest != i){
                exchange(Heap[largest],heap[i]);
                heapify(heap, largest);
        }
}
```

5, 13, 2, 22, 0, 17, -4, 8,4 → (prechod na haldu) 22, 17, 8, 13, 5, 2, 4, -4, 0 → 0, 17, 8, 13, 5, 2, 4, -4,|22 → 17, 8, 13, 5, 2, 4, 0, -4 |22 → -4, 8, 13, 5, 2, 4, 0 |17, 22 → 13, 8, 5, 2,-4, 4, 0 |17, 22 → ..... → -4 | 0, 2, 4, 5, 8, 13, 17, 22 (v tejto fáze je už pole zoradené)

Testové otázky

(2b) Operácia rotácie zachováva poradie prvkov v strome podľa

a) preorder

b) inorder

c) postorder

(2b) V tabuľke s rozptýlenými prvkami (hash tabuľke) sa rozptylová funkcia navrhuje ako

a) jednoznačná, ale nemusí byť jedno-jednoznačná

b) jedno-jednoznačná

c) nemusí byť jednoznačná, lebo sa súčasne navrhuje mechanizmus rozriešenia kolízií

(2b) Kruskalov algoritmus hľadania minimálnej kostry grafu z výhodou využíva reprezentáciu množiny uzlov pomocou

a) Slovníka

b) Binárneho vyhľadávacieho stromu

c) Fakturovanej množiny

d) Haldy

e) 2-3 stromu

(2b) Shellovo usporadúvanie je založené na

a) výmenách susedných prvkov
b) dvojcestnom zlučovaní
c) výmenách dvoch prvkov so zmenšovaním vzdialenosti medzi prvkami, ktoré porovnávajú
d) rozdelením postupnosti na menšie a hľadaní maximálneho a minimálneho prvku v nich
e) rozdelením postupnosti na menšie a rekurzívnej aplikácii algoritmu usporadúvania

(3b) V tabuľke s rozptýlenými prvkami (hash tabuľke) pri otvorenom rozptýlení prvkov:

a) prvky sú rovnomerne rozložené
b) prvky môžu byť aj mimo tabuľky
c) kolízie sa riešia modifikáciou rozptylovej funkcie
d) kolízie nevznikajú
e) kolízie sa riešia reťazením prvkov

Vo svetovej literatúre sa niekedy jednotlivé spôsoby riešenia kolízií označujú rôzne, dokonca rôzne spôsoby tým istým názvom, čo našlo odraz aj našich textoch, písaných v minulosti. Na prednáškach som uviedol členenie na spôsoby:

- *otvoreným rozptýlením,*
- *vnútorným reťazením,*
- *vonkajším reťazením*

a v tomto zmysle je správna odpoveď jednoznačná a len jedna správna.

Ako ocenenie tých, ktorí študovali aj iné zdroje než prednášky, uznám aj odpoveď (e).

(3b) V binárnom vyhľadávacom strome dva prvky:

a) musia vždy mať rôzne kľúče
b) môžu mať rovnaké kľúče ak sú spojené hranou
c) môžu mať rovnaké kľúče jedine ak je jeden v ľavom, a druhý v pravom podstrome
d) môžu mať rovnaké kľúče len ak je jeden pravým nasledovníkom druhého
e) môžu mať rovnaké kľúče len ak je jeden ľavým nasledovníkom druhého

(3b) V binárnom vyhľadávacom strome sa najväčší prvok nachádza

a) vždy v koreni
b) môže, ale nemusí byť v koreni
c) nikdy v koreni

Prečo?
V binárnom vyhľadávacom strome platí, že ľavé dieťa je vždy menšie, ako koreň a pravé dieťa je vždy väčšie ako koreň. To znamená, že v prípade, keď máme iba jeden prvok v strome je určite koreňom a je zároveň najväčším prvkom. Za druhé, ak vložíme do stromu najprv maximum postupnosti a potom vkladáme ostatné prvky, bez vyvažovania, bude takisto koreň maximom stromu, lebo všetky prvky sa umiestnia do jeho ľavého podstromu.

(3b) Implementácia množiny pomocou binárneho vyhľadávacieho stromu v porovnaní s implementáciou pomocou vyváženého stromu je z hľadiska najhoršieho prípadu výpočtovej zložitosti

a) v podstate rovnaká
b) lepšia
c) horšia

Prečo?
V binárnom vyhľadávačom strome nie je zaručené, že strom je vyvážený. To znamená, že niektorý z podstromov môže na svojej ľavej strane obsahovať oveľa viac prvkov, ako na svojej pravej alebo naopak. Toto takisto zhoršuje výpočtovú zložitosť vkladania, vyberania a mazania prvku pričom tá môže byť v najhoršom prípade O(n). V prípade vyváženého stromu to tak nie je keďže po vložení prvku a nastatí nevyváženosti na jednej, či druhej strane sa vyvolá rotácia. To zabezpečuje, že výška stromu nebude nikdy rovná počtu prvkov a takisto, že vyhľadávanie, mazanie a pridávanie prvkov bude mať zložitosť O(log n).

(2b) Vlastnosť AVL stromu:

a) každý AVL strom je binárny vyhľadávací strom

b) niektorý AVL strom je binárny vyhľadávací strom
c) žiadny AVL strom nie je binárny vyhľadávací strom

(2b) V binárnej halde platí

a) ľavý podstrom obsahuje vždy prvky s nižšími hodnotami kľúčov ako pravý
b) ľavý podstrom môže, ale nemusí obsahovať prvky s nižšími hodnotami kľúčov ako pravý
c) ľavý podstrom nikdy neobsahuje prvky s nižšími hodnotami kľúčov ako pravý