Riadny termin 2009/2010

```
1. Hash funkcia
a, linear probing
b, kvadratic probing
(definicie!)
```

a. Linear probing - Linear probing - pri kolízii sa skúša miesto o n prvkov ďalej, ak je obsadené, tak miesto o ďalších n prvkov ďalej, atď. (n je väčšinou 1, ale nemusí byť). Veľkosť tabuľky musí byť nesúdeliteľná s n. (by haro)
b. Quadratic probing - pri kolízii sa skúša miesto, ktoré je od pôvodného ďalej o hodnotu nejakého kvadratického polynómu, napr. x^2 + x, tak ak je obsadené miesto ktoré dá hashovacia funkcia, tak sa skúša miesto o 1^2+1 = 2 miesta ďalej od pôvodného, ak je to obsadené tak miesto o 2^2 + 2 = 6 miest ďalej od pôvodného (nie predchádzajúceho), atď. Aby fungovalo, tabuľka môže byť naplnená max. z polovice a jej veľkosť musí byť prvočíslo, viď http://www.brpreiss.com/books/opus5/html/page241.html . Použitý polynóm je väčšinou x^2. (by haro)

```
c. 2. kedy je sortovaci algoritmus stabilny
```

- Usporadúvací algoritmus je stabilný, ak vždy zachová originálne poradie elementov s rovnakými kľúčmi
- Ak elementy s rovnakými kľúčmi sú neodlíšiteľné, tak nie je potrebné sa zaoberať stabilitou algoritmu
- (napr. ak kľúčom je samotný element)
- Zachovať originálne poradie elementov je dôležité napr. pri viacnásobnom usporiadaní – najprv podľa priezviska a potom podľa mena.
- Každý nestabilný algoritmus sa dá implementovať ako stabilný tým, že sa zapamätá originálne poradie elementov a pri zhodných kľúčoch sa berie do úvahy toto poradie
- Viacnásobné usporiadanie je možné obísť vytvorením jedného kľúča, ktorý je zložený z primárneho, sekundárneho, atď. kľúča usporiadania
    - Takéto úpravy nestabilných algoritmov majú negatívny vplyv na výpočtovú zložitosť.

(Navrat's prednaska)

3. Dijkstra
a, single source
b, relax
(obidve doplnit do programu)

```
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;
      }
}
```

(Peter Macko’s pdf)

4. Merge sort
a, Merge
(napisat)
b, analyza zlozitosti

```
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++;
            }
      }
}
```

(Peter Macko’s pdf)

5. AVL, CC strom
(pravidla a vyvazovanie)

Toto hadam kazdy vie ale aj tak sem dam nejake veci:

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.

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.
odhad časovej zložitosti sprístupnenia ľubovoľného prvku v najhoršom prípade:
O(log2n)

(Peter Macko's pdf)

6. Rabinov-Karpov algoritmus

Namiesto priameho porovnávania reťazca so vzorom sa porovnavajú výstupy hešovacích funkcií. Porovnáva sa heš vzoru s hešom vybraného podreťazca (vyberá sa podreťazec taký dlhý ako je dĺžka vzoru). Pokiaľ sa heše zhodujú, uskutočnuje sa porovnávanie jednotlivých znakov.

algorithm RabinKarp:

Input pole znakov T, dĺžka n
pole znakov P, dĺžka m

```
hP := hash(P[1..m])
hT := hash(T[1..m])
for i from 1 to n-m+1
        if hT = hP
                if T[i..i+m-1] = P
                        return i
        hT := hash(T[i+1..i+m])
return nenašiel sa výskyt
```

Hešovacia funkcia by mala mať tieto vlastnosti:
– jednoducho vypočítateľná
– s malou pravdepodobnosťou kolízií
– Heš posunutého podreťazca by mal byť jednoducho odvoditeľný z predchádzajúceho hešu (táto vlastnosť výrazne uľahčí výpočet a algoritmus sa tým stáva omnoho efektívnejší ako naivný)
(Navrat's prednaska – porovnavanie retazcov str. 15)

7. Quicksort
Randomized Partition
(napisat)

```
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;
}
```

(Peter Macko's pdf)

8. Heapsort
(asi stacilo napisat ako sa jednotlive casti nasledujú)

```
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;
}
```

Pre istotu aj heapify a heap build:

```
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);
      }
}
```

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.