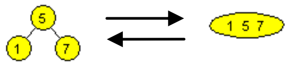
Stack:
CREATE | PUSH | POP | TOP | ISEMPTY
create, push a pop vracajú pointer na stack, top vracia pointer na element (ELM) a isempty vracia bool.

Queue:
CREATE | INSERT | DELETE | FRONT | ISEMPTY
create, insert a delete vracajú pointer na queue, front ELM a isempty bool

2-3 Strom: zložitosť každej operácie = O(logn)
Každý vnútorný vrchol má 2 alebo 3 potomkov.
Všetky listy sú rovnako vzdialené od koreňa.
Dáta sú uložené (až) v listoch.

2-3-4 Strom : je 2-3 strom, kde 3 dvojuzly sú spojené do jedného štvoruzla



Č-Č Strom: zložitosť každej operácie = O(logn)
Každý vrchol je čierny alebo červený, koreň je vždy čierny.
Každý list je čierny (posledný vrchol môže byť červený, no má potomkov „NULL“, ktorí sú čierny)
Každý červený vrchol má oboch potomkov čiernych.
Každá cesta z ľubovoľného pevne zvoleného vrcholu do listov obsahuje rovnaký počet čiernych vrcholov
Najdlhšia cesta je najviac dvakrát tak dlhá ako najkratšia cesta –strom je „vyvážený“

AVL: zložitosť každej operácie = O(logn)
je BVS strom práve vtedy, ak pre každý vrchol x v strome platí:
|(výška ľavého podstromu vrcholu x) -(výška pravého podstromu vrcholu x)| ≤ 1
faktor vyváženia bf: bf(x) = výška(ľavý podstrom(x)) –výška(pravý podstrom(x))

hashovanie:
Možnosti riešenia problému kolízie:
– zreťazenie: Navrhnutie takej štruktúry, ktorá bude schopná uchovávať viacero prvkov s rovnakou rozptylovou hodnotou
– otvorené adresovanie: Umiestnenie jedného z kolidujúcich prvkov na iné miesto v tabuľke
Základná hashovacia funkcia: h‘(k): U -> {0,1....n-1}
Linear probing - pri kolízií sa skúša miesto o c_1 prvkov ďalej, ak je obsadené, tak miesto o ďalších c_1 prvkov ďalej, atď...
Problém sú strapce (clusters) keď sa prvky naukladajú vedľa seba a musíme skúšať veľa miest.
Predpis: $h(k) = (h'(k) + \mathbf{c_1 i}) \bmod n$, pre i=0,1,...n-1 ; väčšinou sa používa $c_1 = 1$
Quadratic probing - pri kolízií sa skúša miesto, ktoré je od pôvodného ďalej o hodnotu nejakého kvadratického polynómu.
Aby fungovalo, tabuľka môže byť naplnená max. z polovice a jej veľkosť musí byť prvočíslo. Použitý polynóm je väčšinou x^2.
Predpis: $h(k) = (h'(k) + \mathbf{c_1 i} + \mathbf{c_2 i^2}) \bmod n$, pre i=0,1,...n-1 ; väčšinou sa používa $c_1=0$ $c_2=1$ tj i^2
Double hashing – pri kolízii sa hashuje znova, druhou funkciou $h_2(k)$: U -> {0,1..n-1} a tak sa hľadá nové miesto
Predpis: $h(k) = (h'(k) + \mathbf{i * h_2(k)}) \bmod n$, pre i=0,1,...n-1

Heap:
```
heap_extract_max(heap){
   if heap-size(heap) < 1                   // ak je tam menej ako 1 cislo
      then error
   max = heap[0]                            // indexujeme od 0, berieme prvy prvok
   heap[0] = heap[heap-size(heap)-1]        // na zaciatok hodime posledny prvok
   heap-size(heap)--                        // zmensime velkost haldy
   HEAPIFY(heap, 0)                         // poprehadzujeme prvky novej haldy
   return max
}
HEAPIFY(heap, i){
   l = left(i)                                        // ak je max na heap[0] left(i) vracia 2*i + 1
   r = right(i)                                       // right(i) vracia 2*i + 2
   if l <= heap-size(heap) and heap[l] > heap[i]      // hladame najvacsi prvok spomedzi l, r a i
      then largest = l
   else largest = i
   if r <= heap-size(heap) and heap[r] > heap[largest]   // kontrlujeme aj ci taky index je este v halde
      then largest = r
   if largest ≠ i
      then Exchange( heap[i], heap[largest] )         // ak najvacsi nie je i, tak ich prehodime
             HEAPIFY(heap, largest)                   // a pokracujeme tam, kde bol povodne najvacsi
}
```

```
BUILD-HEAP(array){                                  // robime z pola haldu -> array[0...n-1]
   heap-size(heap) = length(array)                  // velkost haldy bude velkost pola
   for i = (int)(length(array) / 2 – 1) downto 1    // ideme od spodu hore, zaciname dolnou cel. castou z (velkost / 2 – 1)
      do HEAPIFY(heap, i)                           // prehadzujeme prvky od indexu i
}
HEAPSORT (heap){
   BUILD-HEAP(heap);                       // z nasho pola heap spravi haldu
   for i = length(heap) -1 downto 1 do     // od posledneho prvku az k druhemu
      Exchange(heap[0],heap[i]);           // vymeni prvy s poslednym (na konci vymeni prve dva)
      heap-size(heap)--                    // zmensi velkost haldy
      HEAPIFY (heap,0);                    // a uprace novu haldu
   return heap;                            // nakoniec vrati vzostupne usporiadane pole ( od najmen. po najv. )
}

Heap-INSERT(heap, key){                    // original posted by Gondy -> prerobene na indexovanie od 0 by LihO
   heap-size(heap) = heap-size(heap) + 1            // zvacsime si pole
   i = heap-size(heap) – 1                          // zistime index posledneho prvku
   while i > 0 and heap[PARENT(i)] < key do         // kym je to za prvym prvkom a parent cisla, ktore chceme vlozit je mensi
      heap[i] = heap[PARENT(i)]                     // tak na to miesto kam sme chceli dat key bachni parenta
      i = PARENT(i)                                 // a ako novy index i oznac index povodneho parenta
   heap[i] = key                                    // ak je parent vacsi ako kluc, alebo sme uz na indexe 0, tak tam bachni key
}
```

Dijkstra:

```
inicializuj-jedno-vychodisko (G, s){
   for každý vrchol v z množiny vrcholov V[G]
      do d[v] = ∞                                   // d[v] je odhad najkratsej cesty z vychodiska do vrcholu v
         π[v] = NIL                                 // π[v] je predchadzajuci vrchol na ceste z vychodiska do v
   d[s] = 0                                         // d[s] je 0, pretoze s je vychodisko
}

Relax(u,v,w){
   if d[v] > d[u] + w(u,v) then            // ak je nova odhadovana najkratsia cesta kratsia ako ta co tam uz je
      d[v] = d[u] + w(u,v)                 // tak prepis tu staru nasou novou
      π [v] = u                            // a uloz vrchol, z ktoreho sme tam isli
}

Dijkstra(G, w, s){
   inicializuj-jedno-vychodisko (G, s)
   S = prazdna mnozina                     // S je mnozina vrcholov, do ktorych vieme najkratsie cesty
   Q = mnozina vrcholov V[G]               // Q je mnozina vrcholov, do ktorych cesty este nevieme
   while Q nie je prazdna mnozina
      do u = Extrakt-Min(Q)                // vyber vrchol z mnoziny Q s najmensim odhadom najkratsej cesty
         S = S zjednotenie {u}                      // prida tento vrchol u do mnoziny S
         for každý vrchol v z mnoziny adj[u]        // pre kazdy vrchol v, ktory je spojeny hranou s u
            do Relax(u, v, w)
}
```

Bellman-Ford:

```
Bellman-Ford(G, w, s){
   Inicializuj-jedno-vychodisko(G, s)
   for i = 1 to |V[G]| -1 do                        // pocet vrcholov – 1 krat
      for každú hranu (u, v) z množiny hrán H[G] do
         Relax(u, v, w)                             // zrelaxuje kazdu hranu
   for každú hranu (u, v) z množiny hrán H[G] do
      if d[v] > d[u]+w(u, v) then                   // test na zaporny cyklus
         return FALSE
   return TRUE                                      // true sa vrati ak tam nie je zaporny cyklus
}
```

Floyd-Warshall:

```
Floyd-Warshall(W){                          // vstup. argument je matica, kde su dlzky jednotlivych hran
   n = rows[W]                              // n = pocet riadkov matice
   D^(0) = W
   for k = 1 to n do
      for i = 1 to n do
         for j = 1 to n do
            d_ij^(k) = min(d_ij^(k-1) , d_ik^(k-1) + d_kj^(k-1))   // ak je priama cesta i->j dlhsia ako „obchadzka“ i->k->j, tak ju prepis
   return D                                 // vrati maticu D, kde su najkratsie cesty medzi bodmi
}
```

Kruskal:

```
MST- Kruskal( G, w ){            // MST = minimum spanning tree = minimalna kostra grafu
   A=Ø                           // mnozina A, ktora bude obsahovat hrany MST
   for každý vrchol v z V[G] do
      Make-Set(v)                                        // vytvor pomocny strom z vrcholov (bez hran)
      usporiadaj hrany v H v neklesajúcom poradí podľa váhy w
      for každú hranu (u, v) z H v poradí podľa neklesajúcej váhy do
         if Find-Set(u) ≠ Find-Set(v) then               // ak v strome este neexistuje cesta medzi u a v
            pridaj hranu (u, v) do mnoziny A             // pridaj hranu do mnoziny A (medzi hrany MST)
            Union(u,v)                                   // „spoj“ u a v aj v nasom pomocnom strome
   return A                                              // vrati mnozinu hran MST
}
```

Jarníkov (-Primov –Dijkstrov):

```
MST-PRIM(G, w, r)
for each u ∈ V[G]                           // pre kazdu hranu
     do key[u] ← ∞                          // prirad hrane kluc ( nekonecno )
        π[u] ← NIL                          // vrchol, z ktoreho sme do u isli...
key[r] ← 0                                  // r je vychodisko ( ako pri Dijkstrovi )
Q ← V[G]                                    // mnozina vsetkych vrcholov
while Q ≠ Ø
     do u ← EXTRACT-MIN(Q)                  // vyberieme najmensiu hranu, u = jeden vrchol z tej hrany
        for each v ∈ Adj[u]                 // pre kazdy vrchol v, do ktoreho sa da dostat z u
            do if v ∈ Q and w(u, v) < key[v]   // ak sme v tom vrchole este neboli (t.j. < nekonecno )
                  then π[v] ← u
                       key[v] ← w(u, v)     // oznac ho za navstiveny (t.j. prirad mu novy kluc + vrchol)
```

1. Vybrať ľubovoľný vrchol a označiť ho ako spracovaný
2. Z rezu vybrať minimálnu hranu e a vložiť ju do MST.
3. Nespracovaný vrcholhrany e označiťako spracovaný.

… pri výbere hrany sa vyberá z množiny hrán, ktorých jeden vrchol sme už navštívili,
celý algoritmus skladá akoby nový (po celý čas súvislý) graf...

InsertionSort: -> best case: $O(n)$ worst = average = $O(n^2)$

```
INSERTION-SORT(A){
   for j = 2 to length[A] do      // zacneme druhym prvkom v poli A[1..n]
      key = A[j]                  // zalohujeme si jeho hodnotu do premennej key
      i = j-1                     // i je index kam ho chceme vlozit ( zaciname s i = 1 )
      while i >0 and A[i]>key do  // ak je key mensi ako hodnota ktora je uz v usporiadanej casti
         A[i+1] = A[i]            // tak hodnotu posun do prava
         i = i-1                  // prechadzame usporiadanu cast z prava do lava
      A[i+1] = key                // ked narazime na prvok, ktory je mensi ako nas kluc, tak dalej nejdeme, ulozime key
}
```

QuickSort: -> best case = average = O(nlogn) worst = O(n^2)

```
PARTITION(A, l, r){                 // Array, left, right
   pivot = A[r]                     // vezmeme si za pivota posledny prvok
   i = l                            // i je index kam sa ulozi najblizsi prvok mensi ako pivot
   for j = l to r – 1 {             // postupne ideme cez prvky od l po r-1
      if A[j] <= pivot {            // ak je ten prvok mensi ako pivot (pri i = j sa musi vymenit sam so sebou aby zbehlo i++)
         exchange( A[i], A[j] )     // hodime ten prvok na index, kam sa ma ulozit
         i++                        // index si posunieme
      }
   }                       // teraz su na indexoch l az i-1 prvky mensie ako pivot, na i az r-1 vacsie ako pivot a na r je nas pivot
   exchange( A[i], A[r] )  // hodime teda pivota z konca pola na index i (prehodime ho na spravnu poziciu)
   return i                // vratime index, na ktory sme ulozili pivota
}

QUICKSORT(A, l, r){
   If l < r {                               // zabezpeci, ze pole ma minimalne 2 prvky
      indexPivota = PARITITION(A, l, r)     // rozdel na dve polovice podla pivota
      QUICKSORT(A, l, indexPivota - 1)      // usporiadaj lavu polovicu
      QUICKSORT(A, indexPivota + 1, r)      // pravu polovicu
   }
}

RANDOMIZED-PARTITION(A, l, r){
   i = RANDOM(l, r)                // nahodny index z vnutra pola
   exchange(A[i], A[r])            // hodime ten nahodne vybraty prvok na koniec (spravime tym z neho pivota)
   return PARTITION(A, l, r)       // potom ako predtym (podla posledneho prvku…)
}
```

MergeSort: -> vždy O(nlogn)

```
MERGE(A, l, m, r){                 // Array (pole), left, middle, right (indexy)
   sizeL = m – l + 1
   sizeR = r – m
   L = A[l..m]                     // lava polovica pola
   R = A[m+1..r]                   // prava polovica pola
   L[sizeL + 1] = ∞                // za koniec pola dame nekonecno
   R[sizeR + 1] = ∞
   A[l..r] = MERGEARRAY(L, R)      // zlucime polia L a R
}

MERGEARRAY(L, R){
   i = j = 0
   B = Make-Array( size(L) + size(R) )      // vytvori pomocne pole B o velkosti tych dvoch
   for k = 0 to size(L) + size(R) – 1 do    // skladame nove pole
      if L[i] <= R[j]                       // polia L a R su uz usporiadane
         then B[k] = L[i]
              i++
      else
         then B[k] = R[j]
              j++
   return B                                 // vrati pomocne pole, v ktorom budu usporiadane prvky z L a R
}

MERGESORT(A, l, r){
   If l < r {                  // zaisti nam, ze pole bude mat min. 2 prvky
      m = (int) (l + r)/2      // dolna cela cast (pri neparnych velkostiach pola bude lava „polovica“ vacsia)
      MERGESORT(A, l, m)       // usporiadaj lavu polovicu
      MERGESORT(A, m+1, r)     // pravu
      MERGE(A, l, m, r)        // zluc usporiadane polovice
   }
}
```

KMP:

```
PREDPONOVA FUNKCIA(P){          // π[k] = kolko prvych znakov retazca P_{k-x} tvori koncovku retazca P_k ( x= 1,2,3… )
   m← length(P)                 // π[k] bude vlastne tabulka π[1..m]
   π[1] ← 0                     // π[1] je vzdy 0 ( system je rovanky jak pri π v AZE )
   k ← 0
   for q← 2 to m do                      // zaciname s π[2]
      while k > 0 and P[k + 1] ≠ P[q] do  // zaciname s k=0,  prvy krat sa cyklus preskoci vzdy...
         k ← π[k]                        // k si prepiseme hodnotou π[k]
      if P[k + 1] = P[q] then             // ak sa k+prvy znak zhoduje so znakom na pozicii q (*poznamka)
         k++
      π[q] ← k                           // zapise zistenu hodnotu do tabulky ( π[q] )
   return π
}
```

*poznamka – zistujeme, predponovu funkciu retazca dlzky 6 (abcabc)
-> q = 2, k = 0 -> P[1] = P[2] ? nie π[2] = 0
-> q = 3, k = 0 -> P[1] = P[3] ? nie π[3] = 0
-> q = 4, k = 0 -> P[1] = P[4] ? ano π[4] = 1
-> q = 5, k = 1 -> tu neprejde druha podmienka a porovnavame P[2] = P[5], teda π[5] = 2 (ab je koncovka abcab)
… to ako sa meni k sa dost zle predstavuje

```
KMPPOROVNANIE(T, P){                     // hladame retazec P v texte T
   n← length(T)
   m← length(P)                          // logicky musi platit, ze n>=m
   π← PREDPONOVA FUNKCIA(P)              // vyplni sa cela tabulka π
   q← 0
   for i← 1 to n do                      // cely tento cyklus je vlastne o tom, ze hladame cislo q, t.j. kolko znakov
      while q> 0 and P[q + 1] ≠ T[i] do  // retazca P tvori koncovku retazca T_{1,i} (prvych i znakov textu )
         q ← π[q]
      if P[q + 1] = T[i] then
         q++
      if q = m then                      // ak pocet znakov koncovky = dlzka hladaneho retazca
         print "Vzor sa v retazci vyskytuje s posunom" i-m        // tak mame index, kde zacina vyskyt (i-m)
         q ← π[q]
}
```

Rabin-Karp:

1 2 3 4 5 6 7 8 9
a b b a b a b a b   = T   n = 9
a b a b   = P   m = 4
-> ľahko sa dá na podobnom príklade odvodiť všetko, čo nám bude treba
-> posledné porovnanie chceme spraviť T[6..9] s P[1..4], teda 6 = 9 – 4 + 1 => cyklus od 1 po n - m + 1
-> v cykle sa dopredu vypočítava hash => bolo by dobré, aby pri poslednom porovnaní hash nepočítalo (ACCESS VIOLATION)
čiže nič sa nestane keď pred počítanie hashu hodíme podmienku aby posledný reťazec končil na pozícii 9 (5 + 4 = i + m)

```
Rabin-Karp(T,P){
   hP = hash(P[1..m])                    // zahashovany retazec, ktory hladame
   hT = hash(T[1..m])                    // zahashovanych prvych m pismen textu
   for i from 1 to n-m+1
      if hT= hP and T[i..i+m-1] = P then  // ak sa rovnaju hashe, tak este skotroluje ci sa rovnaju aj retazce
         return i                        // vrati index vyskytu ( na hentom priklade hore by to vratilo 4 )
      if i+m <= n then                   // tento check v prednáške nie je
         hT = hash(T[i+1..i+m])          // vypocita sa hash pre dalsie porovnanie
   return nenašiel sa výskyt
}
```