Vypoctovy problem je urceny:  
 - mnozinou vstupnych hodnot  
 - mnozinou vystupnych hodnot  
 - priradenim, ktore kazdej vstupnej hodnote priradi vystupnu

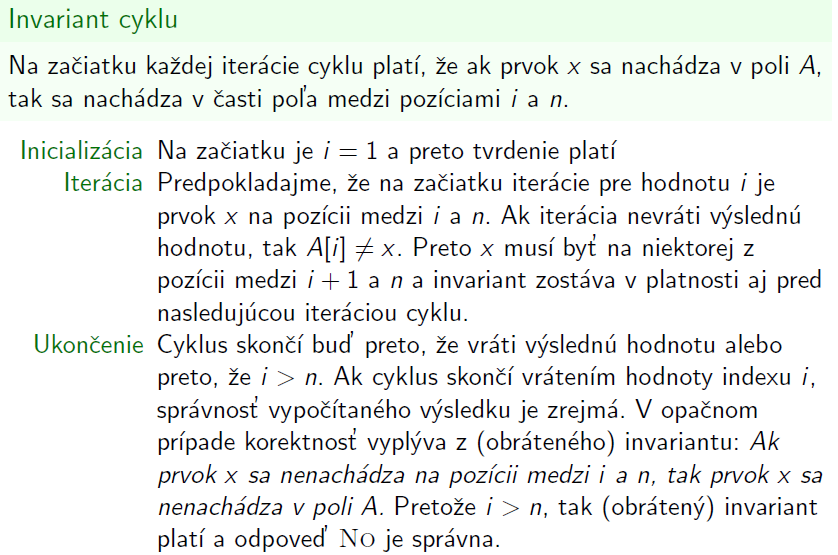
Algoritmus:   
 - nastroj na riesenie vypoctovych problemov  
 - vypoctova procedura, dostane vstupne hodnoty a vypocita zodpovedajuce vystupne  
 - popisuje postup transformacie vstupnych hodnot na vystupne

Datova struktura = sposob organizacie dat, ktory umoznuje pristup a modifikaciu dat

Ciastocna (parcialna) korektnost algoritmu = ak vypocet skonci, poskytne korektny vystup  
Uplnost (konvergencia) algoritmu = vypocet pre kazdy vstup skonci

Korektnost iterativneho algoritmu indukcnym dokazovanim:  
 - analyzujeme vsetky cykly, u vnorenych zaciname od cyklu najhlbsej urovne  
 - pre kazdy cyklus urcime jeho invariant, ktory plati pocas celeho vypoctu cyklu  
 - dokazeme ze invariant cyklu je pravdivy  
 - pomocou invariantu: 1. dokazeme konecnost algoritmu  
 2. dokazeme spravnost vypocitaneho vysledku

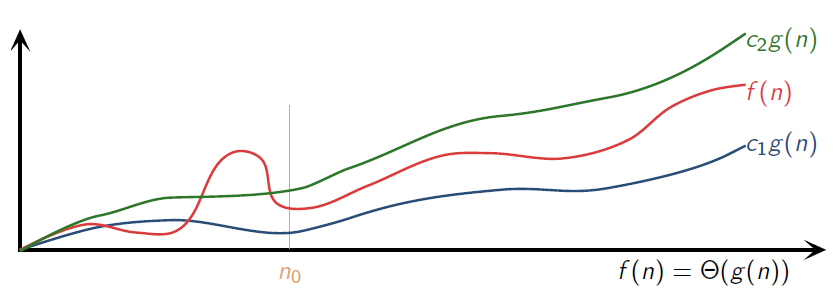
Invariant cyklu  
 1. Inicializacia = invariant je platny pred zaciatkom cyklu  
 2. Iteracia = ak invariant plati pred iteraciou cyklu, plati aj pred nasledujucou iteraciou  
 3. Ukoncenie = cyklus skonci, po ukonceni plati invariant a garantuje pozadovany efekt cyklu



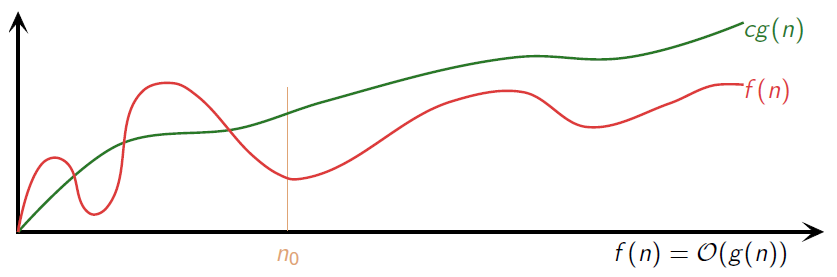
*Invariant cyklu pre linearny algoritmus hladania prvku x v poli A*

Asymptoticka notacia  
- vyuzivame pri urcovanie casovej zlozitosti algoritmov  
- umoznuje abstrahovat od detailov skutocnej zlozitosti  
- vyuziva sa aj pri inych zlozitostnych kriteriach, napr: pamatova zlozitost  
- je nutne specifikovat ci sa jedna o zlozitost v najlepsom, najhorsom, ocakavanom pripade...

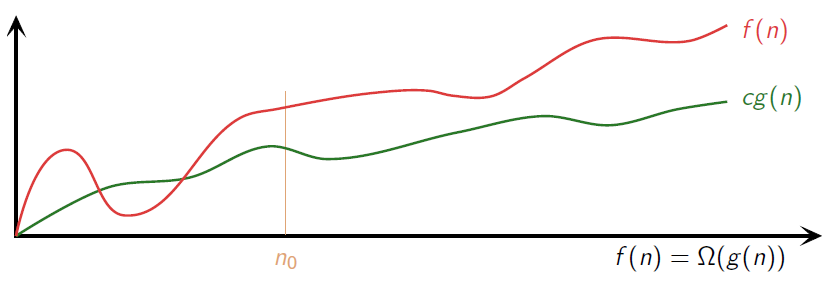
Θ notacia  
- pre danu funkciu g(n) oznacuje symbol Θ(g(n)) mnozinu funkcii:  
Θ(g(n)) = { f(n) | existuju kladne konstanty c1, c2, n0 take, ze 0 <= c1g(n) <= f(n) <= c2g(n) pre vsetky  
 n >= n0 }  
- ak f(n) = Θ(g(n)), potom funkcia f(n) rastie asymptoticky rovnako rychlo ako funkcia g(n)  
- napr: f(n) = 1/2n2 – 3n = Θ(n2), ale f(n) = 6n3 ≠ Θ(n2)  
- plati: f(n) = Θ(g(n)) len vtedy, ak plati f(n) = O(g(n)) a zaroven f(n) = Ω(g(n))



O notacia  
- pre danu funkciu g(n) oznacuje symbol O(g(n)) mnozinu funkcii:  
O(g(n)) = { f(n) | existuju kladne konstanty c, n0 take, ze 0 <= f(n) <= cg(n), pre vsetky n >= n0 }  
- ak f(n) = O(g(n)), potom funkcia f(n) rastie asymptoticky pomalsie ako funkcia g(n)  
- napr: f(n) = n = O(n2), ale f(n) = n3 ≠ O(n2)  
- plati: Θ(g(n)) ⊆ O(g(n))



Ω notacia  
- pre danu funkciu g(n) oznacuje symbol Ω(g(n)) mnozinu funkcii:  
Ω(g(n)) = { f(n) | existuju kladne konstanty c, n0 take, ze 0 <= cg(n) <= f(n), pre vsetky n >= n0 }  
- ak f(n) = Ω(g(n)), potom funkcia f(n) rastie asymptoticky rychlejsie ako funckia g(n)



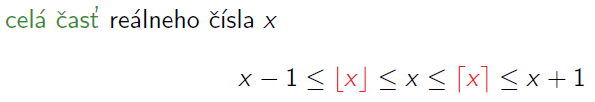
Vlastnosti asymptotickej notacie

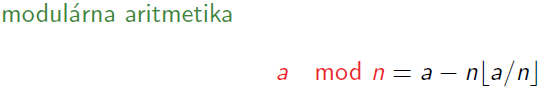
Tranzitivita  
f(n) = Θ(g(n)) a g(n) = Θ(h(n)), potom f(n) = Θ(h(n))  
f(n) = O(g(n)) a g(n) = O(h(n)), potom f(n) = O(h(n))  
f(n) = Ω(g(n)) a g(n) = Ω(h(n)), potom f(n) = Ω(h(n))

Reflexivita  
f(n) = Θ(f(n))   
f(n) = O(f(n))  
f(n) = Ω(f(n))

Symetria  
f(n) = Θ(g(n)) len vtedy, ked g(n) = Θ(f(n))

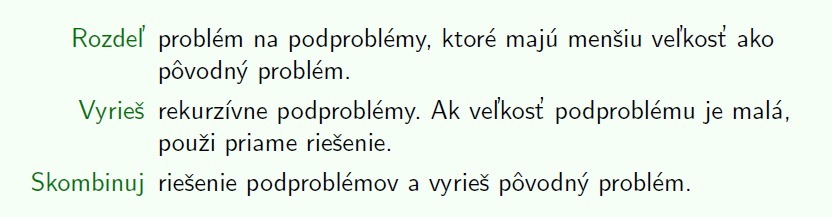
Transpozicia  
f(n) = O(g(n)) len vtedy, ked g(n) = Ω(f(n))

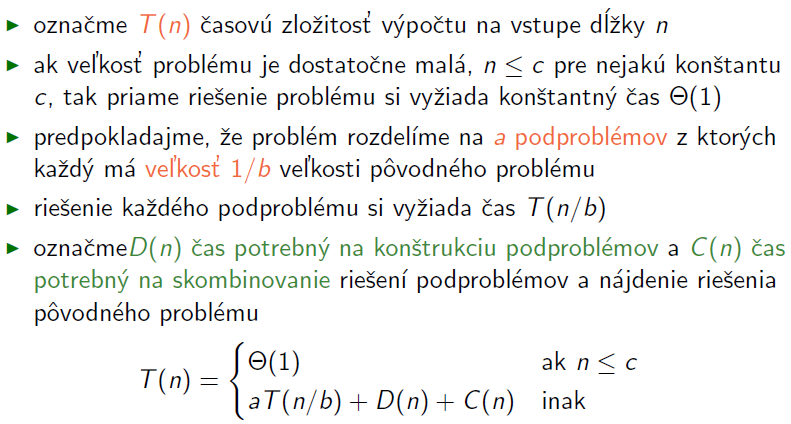




Zlozitost rekurzivneho algoritmu  
Ak program obsahuje rekurzivne volanie seba sameho, jeho zlozitost obvykle vyjadrujeme rekurentnou  
rovnicou, ktora vyjadruje zlozitost vypoctu na vstupe velkosti n pomocou zlozitosti vypoctu na mensich  
vstupoch.

Rozdel a panuj - princip

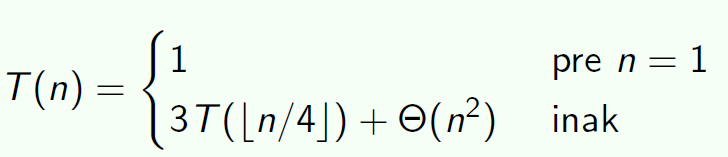
  
Rozdel a panuj – zlozitost

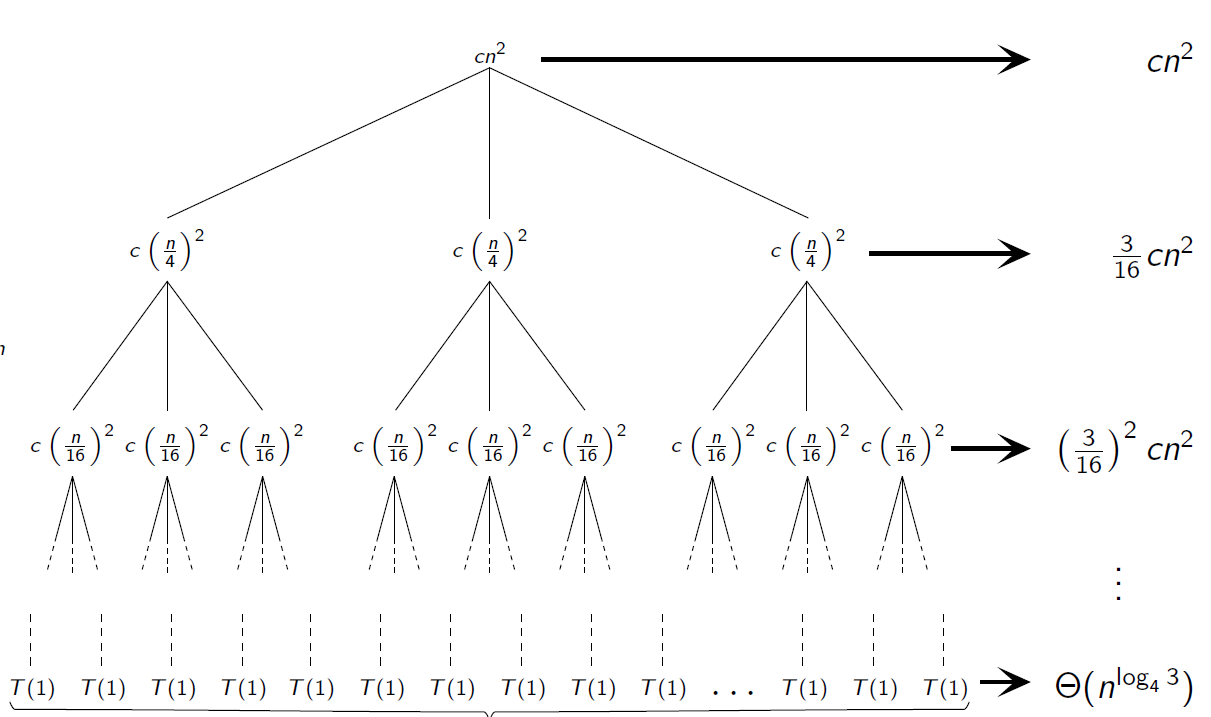


Riesenie rekurentnych rovnic

1. Substitucna metoda = “uhadneme” riesenie a dokazeme indukciou  
2. Metoda rekurzivneho stromu = skonstruujeme strom, ktoreho vrcholy vyjadruju zlozitost jednotlivych  
 rekurzivnych volani, vyslednu zlozitost dostaneme sumaciou  
3. Hlavna metoda (master method) = “vzorec” pre riesenie rovnice tvaru T(n) = aT(n/b) + f(n)

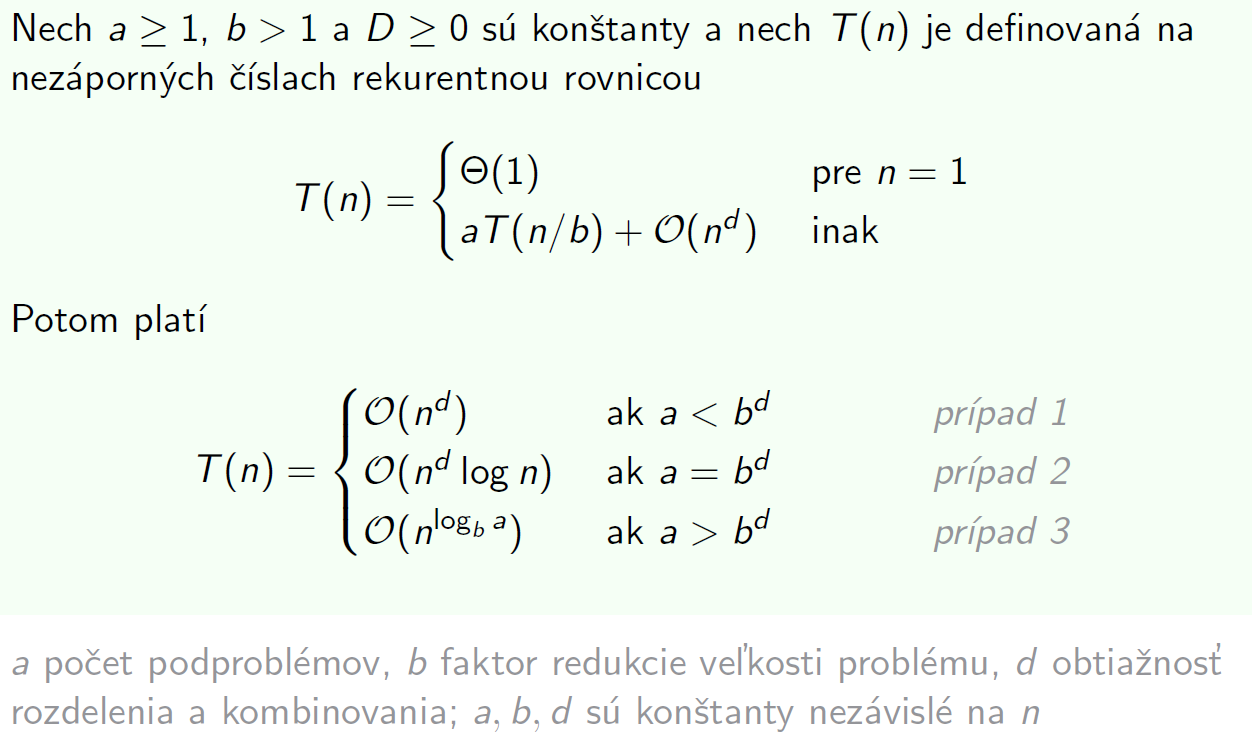
Metoda rekurzivneho stromu





Hlbka stromu: po dosadeni akeho vyrazu dostaneme n = 1 na kazdej urovni stromu ?  
V tomto pripade: n = 1 prave, ak n/4i = 1 => i = log4n

Hlavna metoda (Master method)



Stabilny algoritmus triedenia = zachovava relativne poradie prvkov s rovnakym klucom

Priestorova zlozitost = definovana analogicky ako casova, ale mierou zlozitosti nieje pocet krokov  
vypoctu ale mnozstvo obsadenej pamate v priebehu vypoctu

Extrasekvencna priestorova zlozitost = nezapocitava sa do nej pamat obsadena triedenou postupnostou

In situ (in place) = algoritmy, ktore maju extrasekvencnu priestorovu zlozitost konstantnu

Algoritmy zalozene na porovnavani

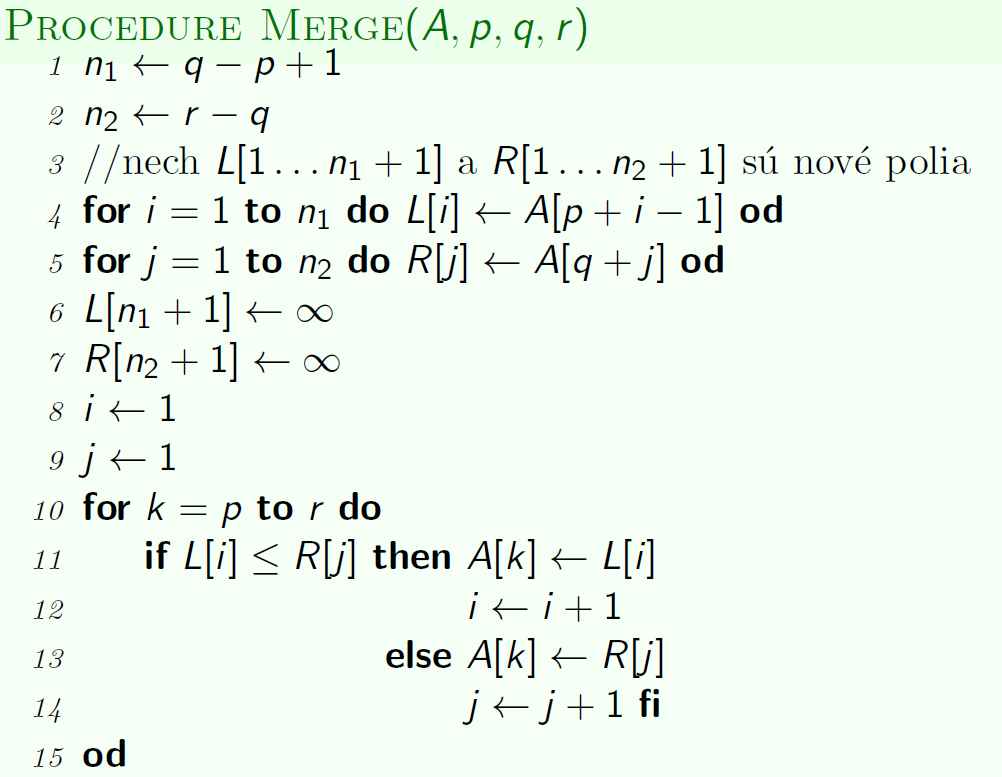
* Vkladanim (Insertion sort) = in situ, stabilny
* Vyberom (Selection sort) = in situ, nie je stabilny
* Spajanim (Merge sort) = nie je in situ, je asymptoticky casovo optimalny (n log n), stabilny
* Haldou (Heapsort) = asymptoticky casovo optimalny, in situ, nie je stabilny
* Rozdelovanim (Quicksort) = nie je casovo optimalny, extrasekvencna zlozitost a stabilita  
  zalezia od implementacie

Algoritmy, ktore ziskavaju informacie o vstupnej postupnosti inak ako porovnavanim prvkov

* Pocitanim (Counting Sort) = vstupne prvky su z mnoziny {0,...,k}
* Cislicove triedenie (Radix Sort) = rozsirene triedenie pocitanim
* Priehradkove triedenie (Bucket Sort) = vyzaduje znalost o pravdepodobnom rozdeleni cisel  
  na vstupe

Spajanie utriedenych postupnosti – MERGE

- ma 4 parametre: A = vstupne pole, p,q,r take, ze p <= q <= r  
- predpokladame, ze A[p ... q] a A[q+1 ... r] su utriedene  
- vysledkom mergu je jedna postupnost A[p ... r], ktora je cela utriedena

Riadky: 1,2,6,7,8,9 maju  
konstantnu zlozitost

for cykly na riadkoch 4 a 5 maju  
spolu zlozitost  
 Θ(n1 + n2) = Θ(q-p+1 + r–q)=Θ(n)

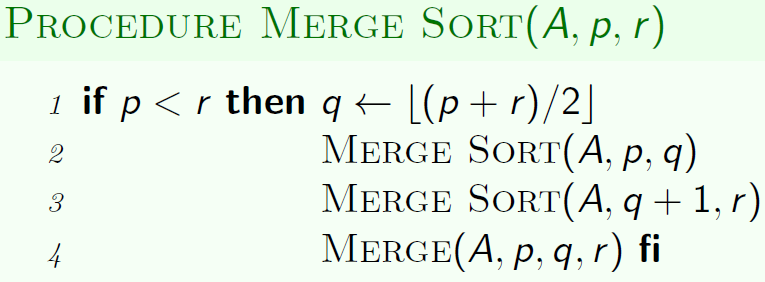
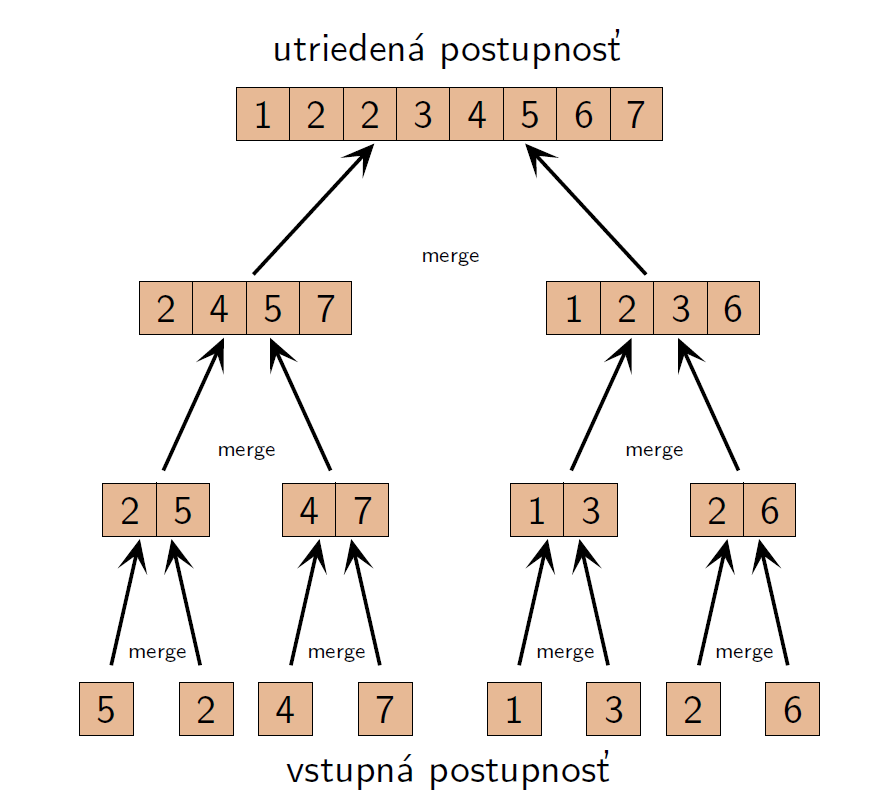
for cyklus na riadkoch 10-15 ma  
zlozitost Θ(r – p + 1), riadky 11-14  
maju konstantu zlozitost

=> zlozitost procedury Merge = Θ(n)

Invariant: Na zaciatku kazdej iteracie plati ze v poli A[p ... k-1] je k – p najmensich prvkov z  
postupnosti L[1 ... n1+1] a R[1 ... n2+1] a to v poradi podla velkosti. Tiez plati, ze L[i] a R[j] su  
najmensie prvky zo svojich postupnosti, ktore este neboli zaradene do vyslednej postupnosti.

Triedenie spajanim – MERGE SORT

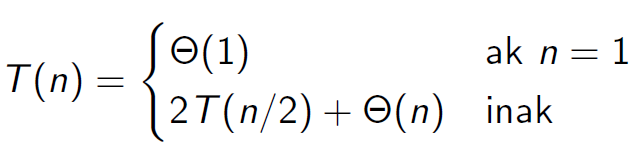
- algoritmus rozdeluj a panuj  
- vyuziva proceduru merge



Rozdelenie spociva vo vypocte indexu  
=> D(n) = Θ(1)

Rekurzivne triedenie dvoch postupnosti  
velkosti n/2 => casova zlozitost = 2T(n/2)

Zlozitost procedury Merge je Θ(n)  
=> C(n) = Θ



Casova zlozitost Merge Sortu je T(n) = Θ(n log n).

Problem inverzii

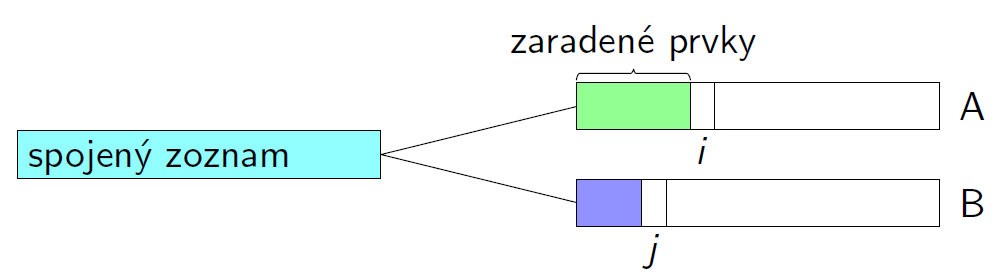
- je dana postupnost vzajomne roznych cisel a1, a2, ..., an  
- inverziou v postupnosti je dvojica indexov i, j takych, ze **i < j** a zaroven **ai > aj**  
- ulohou je najst vsetky inverzie v postupnosti

Naivne riesenie: otestuje vsetky dvojice indexov => zlozitost O(n2)

Rozdel a panuj riesenie:  
1. Postupnost rozdelime na dve postupnosti a1,...,an/2 a an/2 +1, ..., an  
2. V kazdej z podpostupnosti spocitame inverzie  
3. Spocitame inverzie medzi roznymi podpostupnostami

Pri 2. ulohe zaroven s pocitanim inverzii v podpostupnosti, podpostupnost utriedime  
Pri 3. ulohe spajame 2 utriedene podpostupnosti do jednej utriedenej a zaroven pocitame inverzie  
medzi prvkami tychto 2 podpostupnosti

=> Vzdy mame dve podpostupnosti A = a1, ..., ak a B = b1, ..., bl  
- obe tieto postupnosti su utriedene  
- vsetky prvky v podpostupnosti A maju nizsi index ako prvky v podpostupnosti B  
- porovnavame prvky ai a bj  
 -mensi z porovnavanych zaradime do vyslednej postupnosti  
 - ak ai < bj => prvok ai nie je v inverzii so ziadnym prvkom z postupnosti B  
 - ak bj < ai => prvok bj je v inverzii s kazdym prvkom ai,...,ak => pocet inverzii = k – i + 1

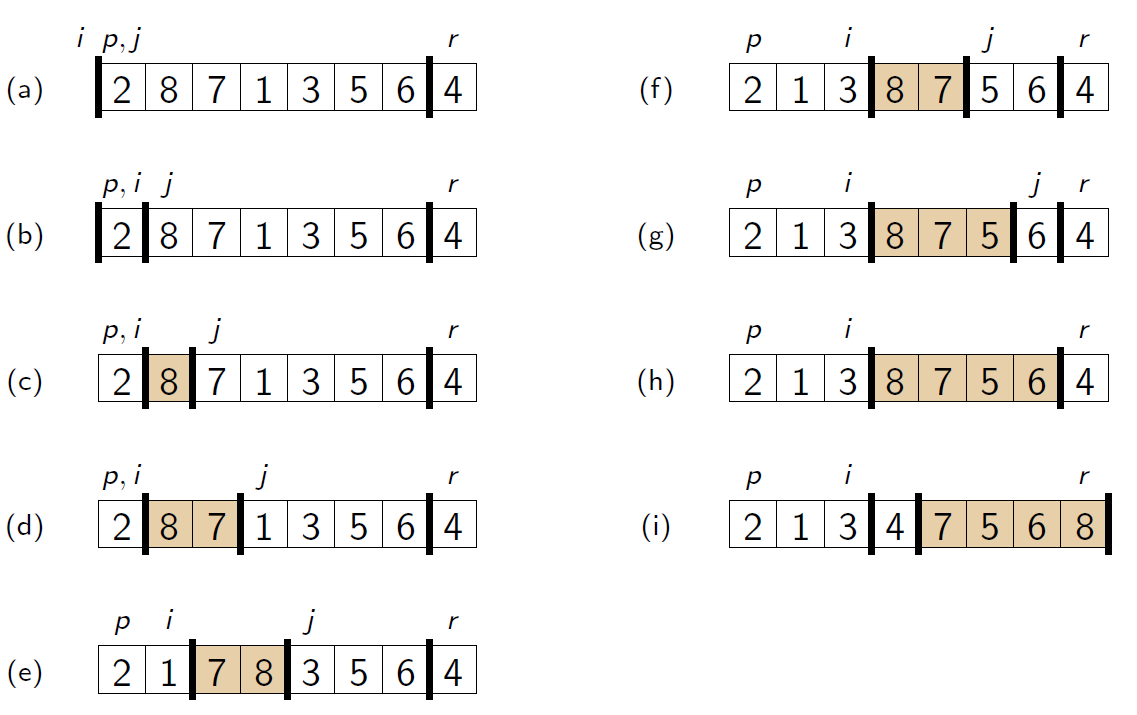


Zlozitost takehoto algoritmu je T(n) = O(n log n).

Triedenie rozdelovanim – QUICKSORT

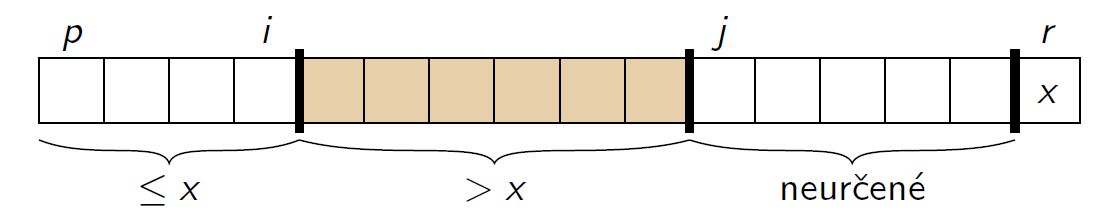
- rozdelenie vstupnej postupnosti A[p ... r] na dve postupnosti A[p ... q-1] a A[q ... r] tak, aby vsetky  
prvky postupnosti A[p ... q-1] boli mensie nanajvys rovne prvkom v postupnosti A[q ... r], na toto  
rozdelenie pouzivame pivota  
- obidve postupnosti utriedime  
- skombinovanie: obidve postupnosti su utriedene, a prvky v lavej casti su mensie rovne prvkom v pravej  
casti => nie je potrebny ziadny dodatocny vypocet a postupnost A[p ... r] je utriedena

Quicksort bez dodatocnej pamate:



*Rozdelenie pola na 2 pomocou pivota, bez dodatocnej pamate*

Drzime si 4 indexy v poli:  
p = prvy prvok pola  
i = posledny prvok prvej podpostupnosti, tj. posledny prvok mensi – rovny pivotovi  
j = prvok za druhou podpostupnostou, tj. prvok za poslednym prvkom vacsim ako pivot



Najhorsi pripad: T(n) = T(n-1) + T(0) + Θ(n) = T(n-1) + Θ(n) => T(n) = **Θ(n2)**Najlepsi pripad: T(n) = T(n/2) + T(n/2) + Θ(n) = 2T(n/2) + Θ(n) => T(n) = Θ(n log n)  
Ocakavana zlozitost (priemerna) je T(n) = Θ(n log n)

Triedenie porovnavanim  
- vyuzivaju len vzajomne porovnavanie prvkov (comparison based algorithms)  
- ziadne predpoklady o vlastnostiach vstupnej postupnosti  
- kazdy algoritmus zalozeny na porovnavani ma zlozitost **Ω(n log n)**

**Binarny strom**- strom, ktoreho kazdy vrchol, ma nanajvys 2 potomkov

**Uplny binarny strom  
-** strom, ktoreho vrcholy na vsetkych urovniach s vynimkou predposlednej maju prave dvoch naslednikov

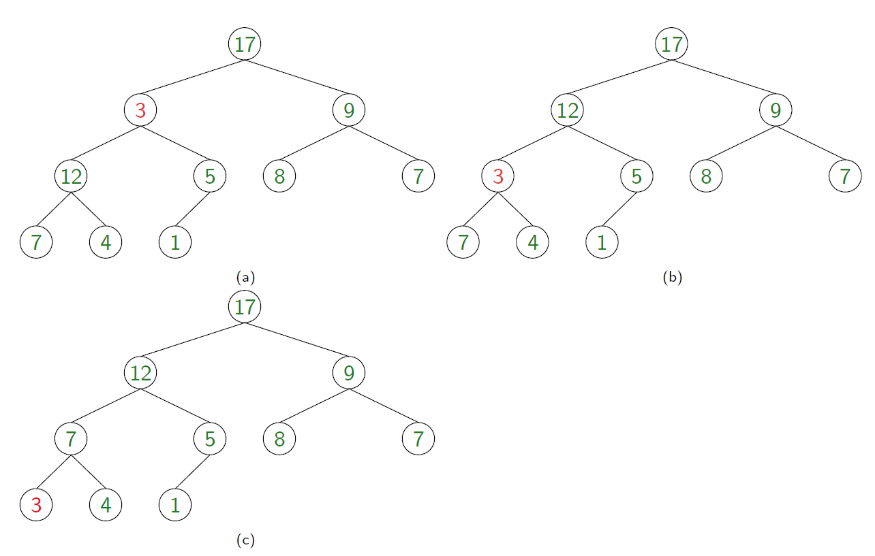
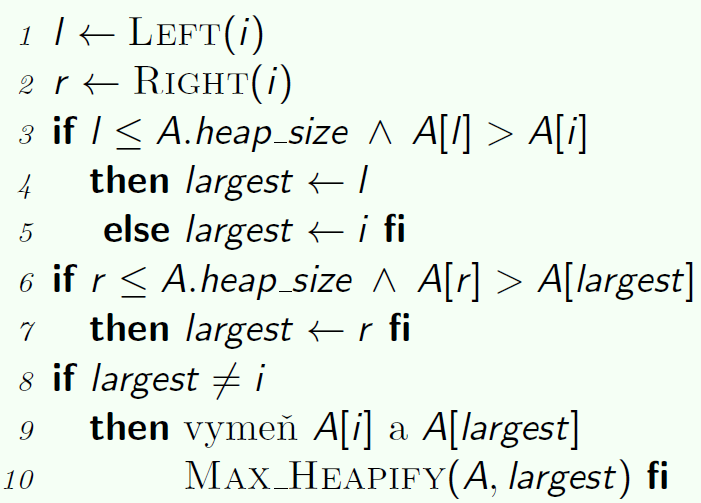
Halda (binarna)  
- datova struktura  
- realizovana polom  
- prvky pola zodpovedaju vrcholom binarneho stromu  
- uplny a zlava zarovnany(najnizsia uroven je zaplnena zlava doprava) binarny strom

Rodic vrcholu A[i] = cela dolna cast (i/2)  
Lavy potomok vrcholu A[i] = 2i  
Pravy potomok vrcholu A[i] = 2i + 1

Maximova halda  
- hodnota ulozena vo vrchole nie je vacsia ako hodnota ulozena v jeho rodicovi  
- najvacsi prvok haldy je ulozeny v koreni

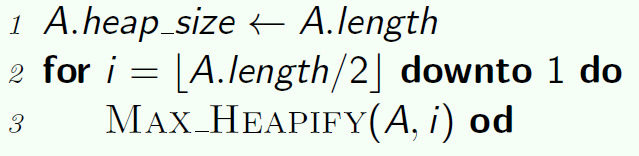
Minimova halda  
- hodnota ulozena vo vrchole nie je mensia ako hodnota v jeho rodicovi  
- najmensi prvok haldy je ulozeny v koreni

MAX\_HEAPIFY  
- garantuje platnost vlastnosti haldy  
- moze sa zacat od korena alebo od predposlednej urovne (tj. urovne nad listami) z prava  
- efektivnejsie ked sa zacne od konca (predposlednej urovne) tj. od indexu n/2  
- ak pole nieje korektna halda, rekurzivne ju upravuje na korektnu

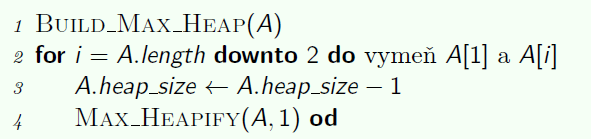


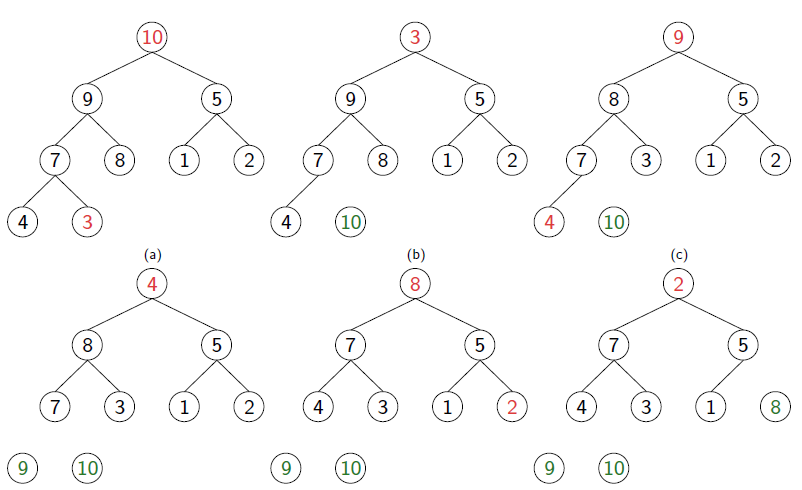
*Overenie haldy pre 1 podstrom*

Zlozitost vypoctu largest je konstanta + zlozitost rekurzivneho volania  
Rekurzivne volanie pre podstrom, ktory ma maximalne 2n/3 vrcholov je T(n) <= T(2n/3) + Θ(1)  
Riesenim rekurentnej rovnice dostavame T(n) = O(log n)  
Zlozitost sa da tiez vyjadrit ako O(h) kde h je vyska stromu

BUILD\_MAX\_HEAP  
- aplikacia procedury MAX\_HEAPIFY na prvky pola odspodu smerom nahor  


Z lubovolneho pola dlzky n vieme spravit korektnu binarnu haldu v O(n).

HEAPSORT  
- algoritmus triedenia haldov  
- najprv pomocou BUILD\_MAX\_HEAP vytvori korektnu binarnu haldu v case O(n)  
- vymeni prvy prvok s poslednym, zmensi velkost pola o 1 => posledny index odpadne  
- korektnost haldy obnovi zavolanim procedury MAX\_HEAPIFY na koren stromu  


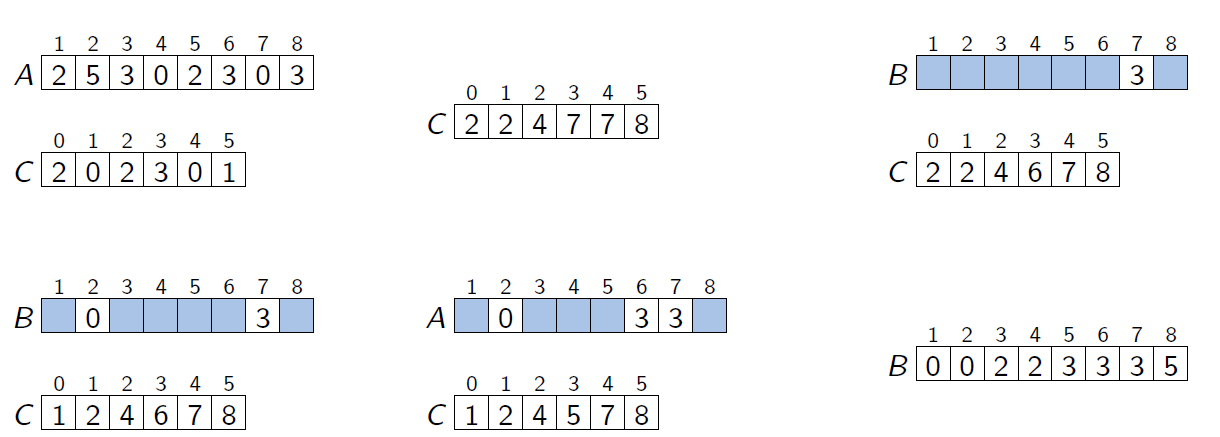
  
Zlozitost BUILD\_MAX\_HEAP =  
O(n)

Kazde z n-1 volani  
MAX\_HEAPIFY = O(log n)

=> HEAPSORT = O(n log n)

Prioritne fronty  
1. HEAP\_MAXIMUM = vrati najvacsi prvok mnoziny => staci nam vratit prvok v koreni maximovej haldy  
2. EXTRACT\_MAX = odstrani z mnoziny najvacsi prvok  
 - prvok z korena stromu odobereme  
 - do korena vlozime posledny prvok  
 - zmensime velkost pola o 1  
 - zavolame MAX\_HEAPIFY na koren stromu  
 - zlozitost O(log n)  
3. HEAP\_INCREASE\_KEY = zameni prvok x prvkom k, kde k >= x  
 - zmenime hodnotu vybraneho vrcholu   
 - obnovime vlastnost haldy, pricom vlastnost haldy mohla byt porusena len voci rodicovi  
 zlozitost O(log n)  
4. MAX\_HEAP\_INSERT = vlozi prvok do haldy  
 - na koniec pola vlozime prvok, ktory je mensi nez vsetky ostatne (- nekonecno)  
 - zvysime hodnotu vlozeneho prvku na pozadovanu, pomocou metody HEAP\_INCREASE\_KEY

Counting sort  
- predpoklada, ze vstupna postupnost obsahuje cele cisla z intervalu 0 ... k pre prirodzene k  
- nie je in situ  
- vstupne pole A[1 ... n]  
- vystupne pole, ktore bude obsahovat utriedenu postupnost B[1 ... n]  
- pole C[0 ... k], ktore sa vyuziva v priebehu vypoctu  
- linearne prechadza cele vstupne pole, ak narazi na prvok p, prvok C[p] zvysi o 1  
- po prejdeni celeho pola, upravuje pole C takto: C[p] = C[p-1] + C[p], tym dostane informaciu pre kazdy  
 prvok, kolko prvkov sa nachadza v utriedenej postupnosti pred nim  
- potom prechadza povodne pole od konca, narazi na prvok p, zisti hodnotu C[p] a do vysledneho pola  
 utriedenej postupnosti ho zaradi => B[p] = p a znizi hodnotu prvku C[p] o jedna  
- extrasekvencna zlozitost je Θ(n+k)  
- stabilny algoritmus  
- ak k = O(n), tak zlozitost triedenia pocitanim je Θ(n)



Radix sort  
- stabilny riadiaci algoritmus  
- casto pouzivany pre radenie retazcov rovnakej dlzky  
- zacina porovnavanie od poslednych znakov retazcov, pomocou volania stabilneho radiaceho algoritmu  
 (casto je to counting sort)  
- asymptoticka zlozitost je Θ(d(n+k)) za predpokladu, ze stabilne triedenie, ktore vyuziva, ma zlozitost  
 Θ(n+k), pre postupnost n cisel s d cislicami v k – arnej cislicovej sustave

Bucket sort  
- stabilny triediaci algoritmus  
- predpoklada vstupnu postupnost z intervalu [0 ... 1)  
- interval [0 ... 1) rozdelime na rovnako velke podintervaly – kose  
- vstupne prvky porozdeluje podla hodnot do kosov danych intervalov  
- na kazdy kos zavola iny stabilny triediaci algoritmus alebo sam seba rekurzivne  
- vyuziva sa pri radeni obrovskych dat, kde nieje mozne nacitat vsetky data naraz, alebo  
 ked by mohol byt kazdy kos radeny v inom vlakne alebo na inom stroji  
- zlozitost je O(m C(n/m)), kde m je pocet kosov, n je pocet prvkov, C(x) je zlozitost vyuzivaneho  
 triediaceho algoritmu  
- ocakavana zlozitost pre vstup s uniformne rozdelenymi cislami je Θ(n)

Datovy typ  
- mnozina hodnot, ktore moze premenna daneho datoveho typu nadobudnut  
- mnozina operacii, ktore su pre dany datovy typ povolene/definovane  
- nezavisi na konkretnej implementacii

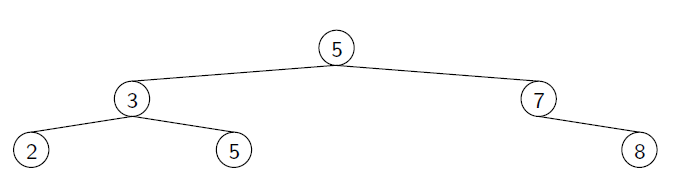
Jednoduchy (skalarny) datovy typ  
- data zaberaju vzdy konstantne mnozstvo pamate, zpristupnenie hodnoty trva konstantny cas  
- napr: ciselne typy, typ pravdivostnych hodnot...

Zlozeny datovy typ  
- implementacia zlozeneho datoveho typu je datova struktura  
- staticky = pevna velkost, casova zlozitost zpristupnenia je konstantna, napr: pole, k-tica  
- dynamicky = neohranicena velkost, casova zlozitost zpristupnenia prvku je funkciou zavislou na velkosti  
 napr: zoznam, zasobnik, fronta, strom, graf

Dynamicke datove struktury  
- mnozina objektov, v priebehu vypoctu mozme prvky pridavat, odoberat, inak modifikovat  
- k atributom objektu mozme pristupovat ak mame ukazatel na objekt  
- jeden z atributov objektu je jeho identifikator = kluc key  
- ak vsetky prvky maju rozne kluce, hovorime o mnozine obsahujucej kluce

Binarny vyhladavaci strom (BVS)  
- dynamicka datova struktura  
- kazdy vrchol stromu je jeden objekt  
- atributy objektu: kluc(key), ukazatele left(lavy potomok), right(pravy potomok), parent(rodic)  
- kluce su ulozene vzdy tak, ze plati:  
 Ak x je vrchol BVS:  
 -y je lavy syn vrcholu x, potom y.key <= x.key  
 -y je pravy syn vrcholu x, potom y.key >= x.key

Prvky ulozene v BVS mozme vypisat v 3 poradiach:  
1. inorder = hodnotu kluca v koreni vypiseme **medzi** vypisanim klucov v jeho lavom a pravom  
podstrome  
2. preorder = hodnotu kluca v koreni vypiseme **pred** vypisanim klucov v jeho lavom a pravom  
podstrome  
3. postorder = hodnotu kluca v koreni vypiseme **po** vypisani klucov v jeho lavom a pravom podstrome

inorder = 2,3,5,5,7,8

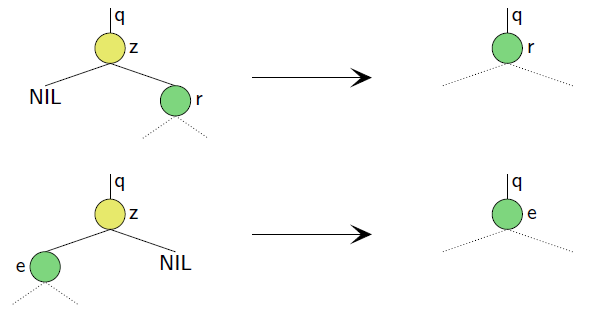
preorder = 5,3,2,5,7,8

postorder = 2,5,3,8,7,5

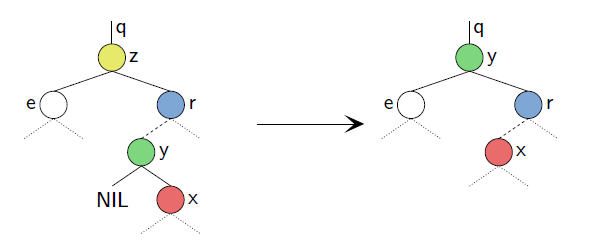
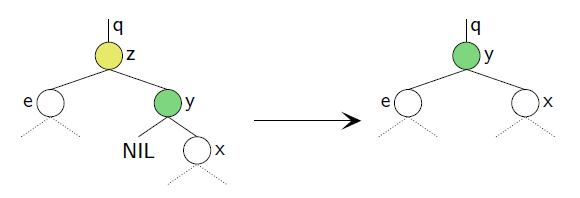
Operacie search, min, max, naslednik, predchodca, delete, insert su ohranicene zhora vyskou BVS <= n.

TREE\_INSERT = vlozenie prvku do BVS  
 - prechadzame strom (ako pri hladani prvku)  
 - hladame vrchol, ktoreho prislusny podstrom je prazdny (lavy podstrom ak vkladany prvok je  
 mensi ako kluc prislusneho vrchola, inak pravy podstrom)  
 - vkladany prvok sa stane lavym resp. pravym synom vrchola

TREE\_DELETE  
1. pripad = vrchol, ktory chceme zmazat je listom (tj. nema ziadneho syna)  
 - takyto vrchol mozme jednoducho odstranit

2. pripad = vrchol, ktory chceme zmazat ma jedneho syna  
 - syn uzla, ktory mazeme sa stane synom rodica uzla, ktory mazeme

3. pripad = vrchol, ktory chceme zmazat ma 2 synov  
 - najdeme vrchol y, ktory nahradi zmazany vrchol z  
 - A: y je v pravom podstrome vrchola z a zaroven nema laveho potomka  
 - B: y je v pravom podstrome vrchola z a ma laveho potomka => y najdeme pomocou  
 TREE\_MINIMUM pocinajuc vrcholom z (vrchol, ktory chceme zmazat)



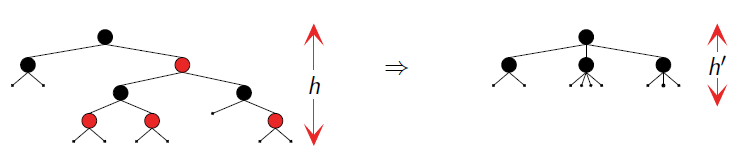
*Podpripad A Podpripad B*

Cerveno cierne stromy

- binarny vyhladavaci strom, ktoreho kazdy uzol je zafarbeny cervenou alebo ciernou farbou a splna:  
 1. koren stromu a vsetky listy su cierne  
 2. ak je vrchol cerveny, tak jeho otec je cierny  
 3. kazda cesta z vrcholu x do listu obsahuje rovnaky pocet ciernych vrcholov  
=> ak je vrchol cerveny, obaja jeho synovia su cierny  
- cierna hlbka stromu s korenom x je pocet ciernych vrcholov na lubovolnej ceste z korena do listu

Absorbovanim cervenych vrcholov do ciernych dostaneme strom, v ktorom ma kazdy vrchol 2,3 alebo 4  
potomkov a vsetky cesty z korena do listu maju rovnaku dlzku h‘.

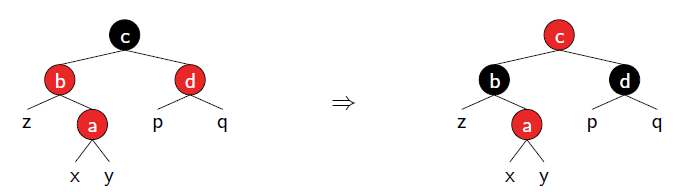
h‘ >= h/2 lebo na kazdej ceste moze byt nanajvys h/2 cervenych vrcholov  
h <= 2 log(n + 1)



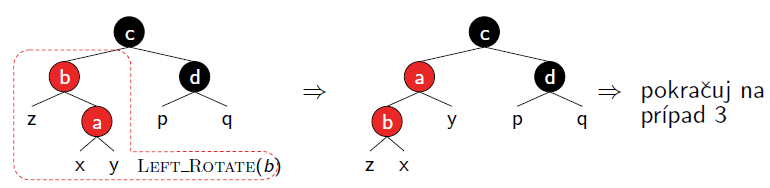
Operacie search, min, max, naslednik, predchodca sa implementuju rovnako ako pri BVS ich zlozitost je  
O(log n).  
Pri operaciach delete sa meni zafarbenie vrcholov, vyuziva sa pri tychto operaciach rotacia, zlozitost je  
O(log n).

Vlozenie prvku  
- vlozi prvok x do stromu rovnako ako pri BVS  
- zafarbi vrchol x na cerveno  
- mozu nastat 3 pripady:

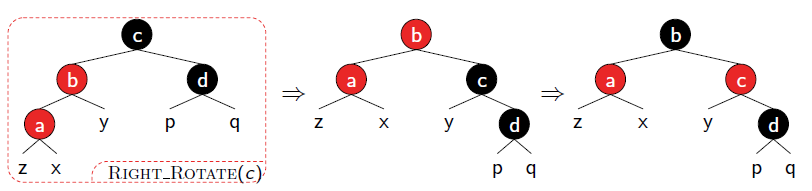
1. pripad = vrchol x a jeho stryko su cervene  
 - zafarbi otca a stryka ciernou farbou  
 - zafarbi praotca vrcholu x cervenou farbou



2. pripad = stryko vrcholu x je cierny a x je pravym synom  
 - preved lavu rotaciu okolo otca vrcholu x  
 - pokracuj na pripad 3



3. pripad = stryko vrcholu x je cierny, x je lavym synom  
 - preved pravu rotaciu okolo praotca vrchollu x  
 - vymen zafarbenie medzi otcom vrcholu x a novym bratom vrcholu x

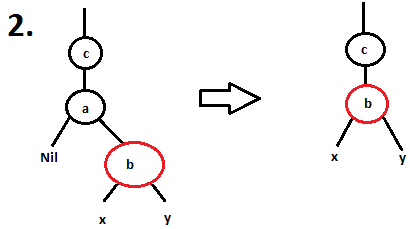
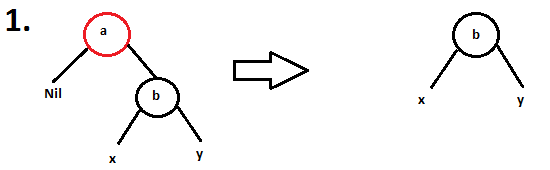


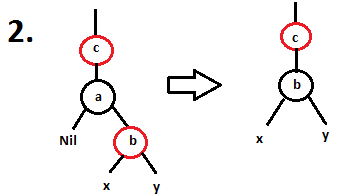
Zlozitost vkladania prvku je O(log n).

Odstranenie prvku

Odstranovany vrchol ma prave jedneho syna:  
1. Odstranovany vrchol je zafarbeny na cerveno = odstranim prvok sposobom ako pri BVS, odstranenim  
cerveneho vrcholu sa nemohla porusit ziadna vlastnost cerveno-cierneho stromu

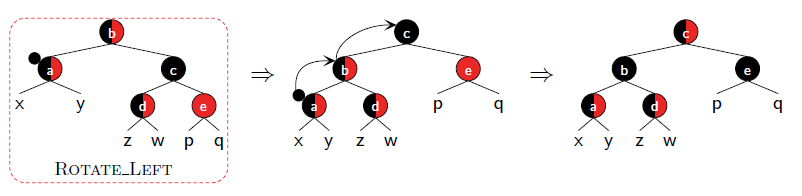
2. Odstranovany vrchol je zafarbeny na cierno = odstranim vrchol ako pri BVS, ak vrchol, ktory sme   
dali na poziciu odstraneneho je cerveny a jeho otec je tiez cerveny, novy vrchol zafarbim na cierno



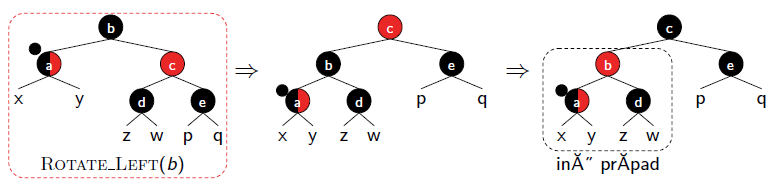


Odstranovany vrchol ma dvoch synov  
1. Naslednik odstranovaneho vrcholu je cerveny => odstranim vrchol sposobom ako pri BVS, tj  
najdem jeho naslednika a dam ho na jeho miesto, naslednik ziska farbu odstranovaneho vrcholu,   
na miesto naslednika ide jeho syn (pravy, kedze hladame naslednika a naslednik nema laveho syna)

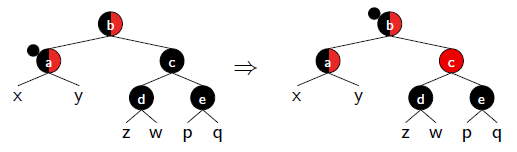
2. Naslednik odstranovaneho vrcholu je cierny => odstranim vrchol sposobom ako pri BVS, tj  
najdem jeho naslednika a dam ho na jeho miesto, naslednik ziska farbu odstranovaneho vrcholu,  
na miesto naslednika ide jeho pravy syn a tento syn zdedi jeho ciernu farbu, v tomto pripade  
ma syn naslednika dve farby (cervenu a ciernu alebo ciernu a ciernu) => jednu svoju a ciernu zdedenu,  
to sa dalej riesi korekciou farieb

Odstranovany vrchol ma dvoch synov – korekcia dvoch farieb  
1. Brat dvojfarebneho vrcholu je cierny, bratov pravy syn je cerveny:  
 - prevediem lavu rotaciu okolo otca dvojfarebneho vrcholu  
 - zafarbim praotca dvojfarebneho vrcholu farbou otca dvojfarebneho vrcholu  
 - presuniem zdedenu ciernu farbu z dvojfarebneho vrcholu na jeho otca => otec sa stane cierny  
 - brat dvojfarebneho vrcholu v povodnom strome sa stane cierny

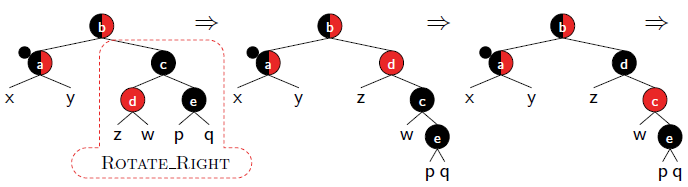
2. Brat dvojfarebneho vrcholu je cerveny, bratovi synovia su cierny  
 - prevediem lavu rotaciu okolo otca dvojfarebneho vrcholu  
 - vymenim farby medzi otcom a praotcom dvojfarebneho vrcholu  
 - pokracujem niektorym z dalsich moznych pripadov



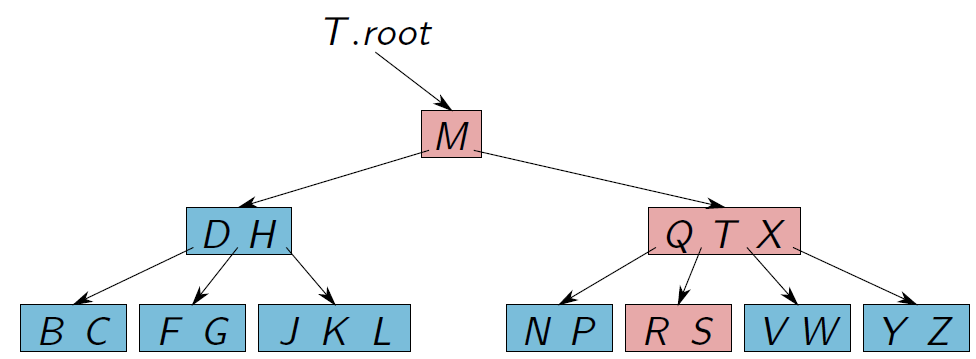
3. Brat dvojfarebneho vrcholu je cierny a jeho synovia su tiez cierny  
 - presuniem ciernu farbu z dvojfarebneho vrcholu a jeho brata na ich otca  
 - vrchol s dvomi farbami sa presunul blizsie ku korenu, pokracuj analogicky



4. Brat dvojfarebneho vrcholu je cierny, jeho pravy syn je cierny a jeho lavy syn je cerveny  
 - prevediem rotaciu okolo brata dvojfarebneho vrcholu  
 - vymenim farby medzi povodnym a novym bratom dvojfarebneho vrcholu  
 - pokracujem pripadom 1. (brat dvojfarebneho vrcholu cierny a jeho pravy syn cerveny)



B – stromy  
- vrcholy mozu mat vela naslednikov  
- vyska B-stromu moze byt vyrazne mensia ako log n vdaka vacsiemu poctu naslednikov  
- ak vnutorny vrchol stromu obsahuje n klucov, potom ma n+1 naslednikov  
- n klucov vo vnutornom vrchole vymedzuje n+1 intervalov, do ktorych patria kluce reprezentovane  
 kazdym z n+1 naslednikov  
- kazdy vrchol x ma atributy:  
 - x.n = pocet klucov ulozenych vo vrchole x  
 - x.n klucov v neklesajucom poradi => x.key1 <= x.key2 <= ... <= x.keyx.n  
 - x.leaf = booleovska premenna, ma hodnotu true ak vrchol x je listom  
- vsetky listy maju rovnaku hlbku

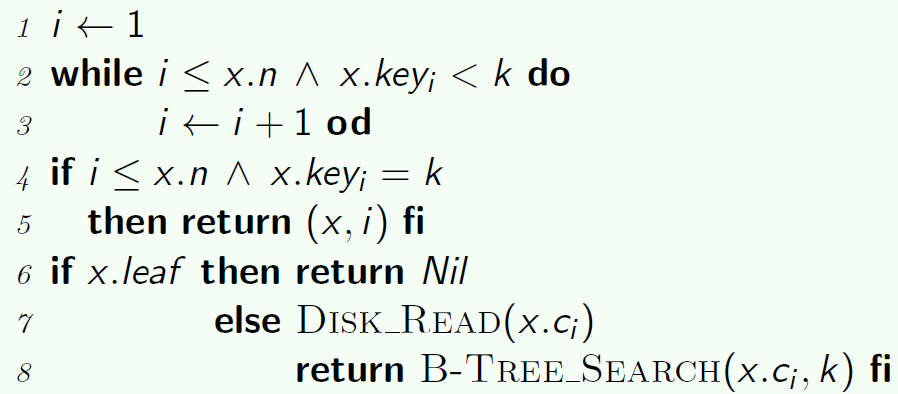


- vrcholy maju dolnu a hornu hranicu na pocet klucov, ktore obsahuju  
- hranica je t >= 2, tato hranica sa nazyva minimalny stupen B – stromu  
- kazdy vrchol okrem korena musi obsahovat aspon t – 1 klucov => kazdy vnutorny vrchol okrem  
 korena musi mat aspon t naslednikov  
- ak je strom neprazdny, koren musi obsahovat aspon 1 kluc  
- kazdy vrchol moze obsahovat nanajvys 2t – 1 klucov => moze mat nanajvys 2t naslednikov  
- vrchol, ktory ma presne 2t naslednikov nazyvame plny

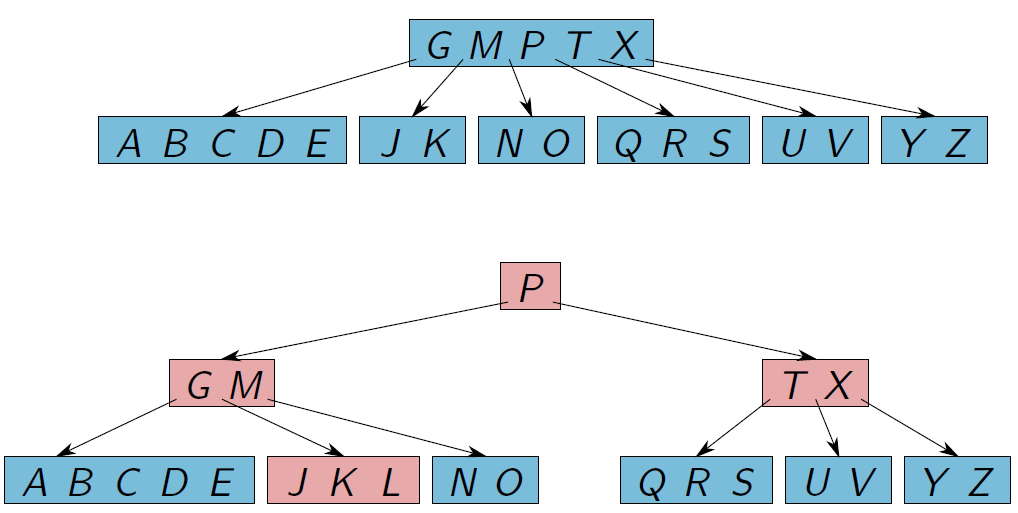
Ak B – strom T ma n >= 1 klucov, hlbku h a minimalny stupen t>= 2, potom h <= log t (n+1/2) + 1

Operacie nad B stromom: vytvorenie stromu, vyhladavanie, vkladanie a odstranovanie kluca

Vyhladavanie  
- pocet DISK\_READ operacii je ohraniceny vyskou stromu h  
- pocet opakovani cyklu 2 – 3 je nanajvys 2t  
- celkova zlozitost je O(th) = O (t log t n)



Vlozenie kluca  
- hladame list, do ktoreho ma byt novy kluc vlozeny  
- vytvorenie noveho listu ako v BVS nieje mozne, bola by porusena vlastnost minimalneho poctu klucov  
- kluc sa vklada do existujuceho listu   
- ak by vlozenim doslo k poruseniu vlastnosti maximalneho poctu klucov vo vrchole, tak list rozdelime na  
 dva  
- rozdelenim sa zvysi pocet naslednikov predchodcu povodneho listu, ak bol predtym plny, teraz je   
 porusena vlastnost maximalneho poctu naslednikov => je potrebne ho tiez rozdelit  
- v najhorsom pripade sa rozdelovanie zastavi az v koreni stromu  
- realizuje sa to v jednom priechode od korena smerom k listu  
- vzdy ked najdeme plny vrchol (vnutorny alebo list) rozdelime ho na dva nove, tym sa zaruci, ze  
 predchodca rozdelovaneho vrcholu nieje plny  
- pocet operacii DISK\_WRITE a DISK\_READ je O(h)  
- celkova zlozitost je O(th) = O( t log t n)

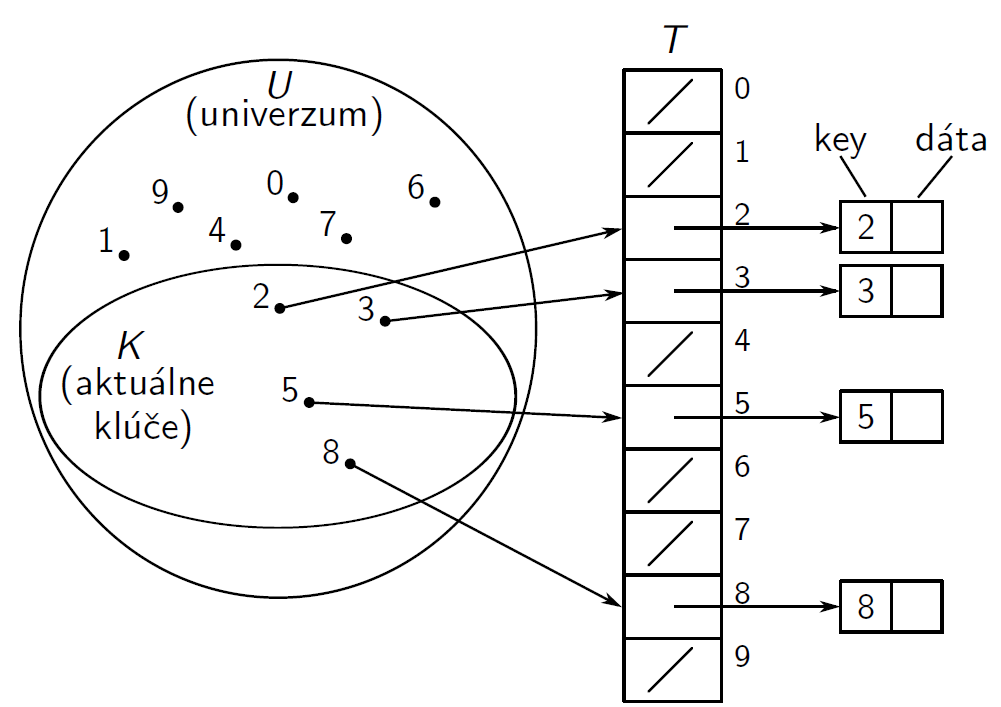
t = 3

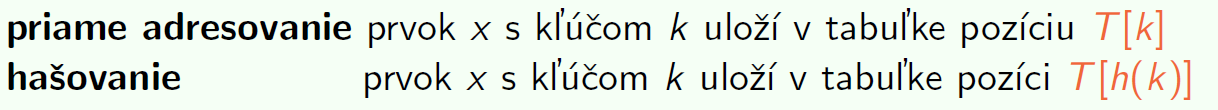
Odstranenie kluca  
- po odstraneni kluca moze klesnut pocet klucov vo vrchole pod minimalnu hranicu  
- ako pri vkladani kluca sa to realizuje v jednom priechode od korena k listu  
- ak odstranovany kluc sa nachadza v liste, procedura prechadza od korena k listu bez nutnosti navratu  
- ak sa kluc nachadza vo vnutornom vrchole, tak procedura postupuje od korena k listu s moznym  
 navratom do vrcholu, z ktoreho bol kluc odstraneny a nahradeny svojim predchodcom alebo nasledn.  
- celkova zlozitost je O(th) = O(t log t n)

B+ stromy  
- zaznamy s udajmi su len v listoch  
- listy su zretazene pri zachovani poradia podla klucov  
- vnutorne vrcholy indexuju listy  
- kluc sa najde az po dosiahnuti listu  
- jednoduchsie vkladanie a odstranovanie ako pri B stromoch  
- implementacia B+ stromu je jednoduchsia ako B stromu

Hasovanie  
- reprezentacia dynamickej mnoziny prvkov  
- podporovane operacie: INSERT, DELETE, SEARCH

Priame adresovanie  
- kazdy prvok ma priradeny kluc z univerza U = {0,1, ..., m-1}  
- ziadne dva prvky nemaju priradeny ten isty kluc  
- pouziva sa pole T[0 ... m-1]  
 - kazda pozicia v poli T zodpoveda jednemu klucu z univerza U  
 - ak mnozina obsahuje prvok x s klucom k, potom T[k] obsahuje ukazatel na prvok x, inak  
 je T[k] prazdne (NIL)  
- zlozitost operacii je konstanta

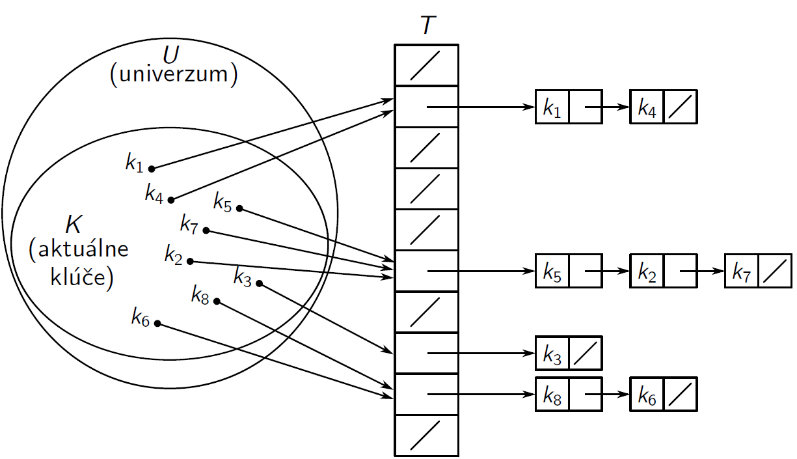
Vyhody  
- konstanta zlozitost operacii  
- jednoducha implementacia  
  
Nevyhody  
- ulozenie pola o velkosti univerza je neprakticke  
- ak mnozina prvkov je mala v porovnani  
s univerzom => obrovske plytvanie pamatou  
  
  
  
Hasovacia tabulka  
- ak je mnozina prvkov K mensia ako univerzum U => hasovacia tabulka vyuziva vyrazne menej priestoru  
 ako tabulka s priamym pristupom  
- potrebny priestor sa da zredukovat na Θ(|K|)  
- zlozitost operacii ostava O(1) ale v **ocakavanom pripade**, nie v najhorsom  
- vyuziva hasovaciu funkciu h: U -> {0,1, ..., m-1}  
- moze nastat kolizia, ked dva alebo viac klucov funkcia zahasuje na rovnaku poziciu: x ≠ y ale h(x) = h(y)



Riesenie kolizii:   
 - zretazene zoznamy (efektivnejsie)  
 - otvorene adresovanie

Vyber hasovacej funkcie:  
 - chceme minimalizovat pocet kolizii  
 - efektivny vypocet funkcie

Riesenie kolizii - zretazene zoznamy  
- slot j obsahuje ukazatel na zaciatok zoznamu, ktory obsahuje vsetky prvky s klucami zahasovanymi  
 na poziciu j => h(k) = j  
- zoznam je prazdny ak ziadny prvok nebol zahasovany na prislusnu poziciu  
- vlozenie prvku x do hasovacej tabulky T sa realizuje ako vlozenie prvku na zaciatok zoznamu T[h(x.key)]

INSERT: zlozitost konstantna aj v najhorsom  
pripade (za predp. Ze prvok x este nieje v tabulke)  
  
SEARCH: zlozitost v najhorsom pripade je umerna dlzke zoznamu => Θ(n), kde n je   
pocet prvkov ulozenych v tabulke

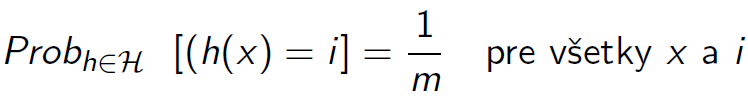
DELETE: asymptoticky rovnaka ako SEARCH (za predpokladu obojstranneho zoznamu)  
  
- zlozitost v ocakavanom pripade zalezi od vyberu hasovacej funkcie

Riesenie kolizii - otvorene adresovanie  
- vsetky kluce ukladame priamo do tabulky  
- kazdy slot tabulky obsahuje bud kluc, alebo hodnotu NIL  
- pri hladani kluca k  
 1. vypocitame hodnotu h(k) a testujeme obsah slotu h(k)  
 - ak slot h(k) obsahuje kluc k, vyhladanie je uspesne a ukoncene  
 - ak slot h(k) obsahuje NIL, vyhladavanie je neuspesne a ukoncene  
 - ak slot h(k) obsahuje kluc rozny od k, tak vypocitame novu poziciu v tabulke ako funkciu   
 k a poradoveho cisla testu (zacina sa od 0)  
 - testovanie konci ak sme nasli hodnotu NIL alebo kluc k  
- hasovacia funkcia je typu h: U x {0,1, ..., m-1} -> {0,1, ..., m-1}

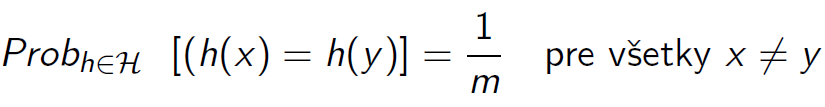
Otvorene adresovanie – odstranenie kluca  
- z pozicie j chcem odstranit kluc k  
- ak som po vlozeni kluca k do tabulky vlozil kluc k2 a pri jeho vkladani som testoval pozicu j =>  
 ak by som na poziciu j vlozil NIL, tak pri naslednom hladani kluca k2 by som dostal nespravny vysledok  
- namiesto hodnoty NIL sa preto pri mazani pouziva specialna hodnota DELETED  
- operacia INSERT berie DELETED slot ako prazdny, operacia SEARCH za neprazdny, ale obsahujuci inu  
 hodnotu, ako hladany kluc

Hasovacia funkcia  
- vyberame nahodne  
- mame fixnu mnozinu H funkcii z U do {0,1, ..., m-1} a na zaciatku kazdeho behu vypoctu nahodne jednu  
 vybereme

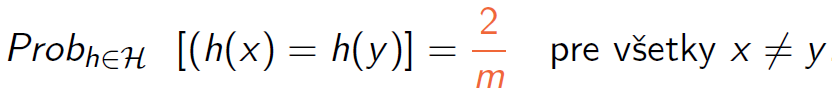
Uniformna mnozina hasovacich funkcii  
- ak pre nahodne vybranu hasovaciu funkciu h z H je pravdepodobnost vyberu kazdej pozicie rovnaka =>



Univerzalna mnozina hasovacich funkcii  
- ak pre kazde dva prvky univerza je pravdepodobnost kolizie najmensia mozna =>



Near – universal mnozina hasovacich funkcii  
- ak pre kazde dva prvky univerza je pravdepodobnost kolizie blizka najmensej moznej  
- v praxi castejsie vyuzivana



Riesenie kolizii - zretazene zoznamy

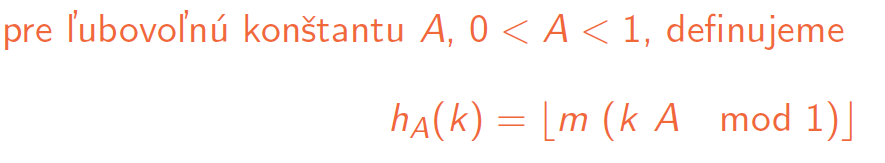
- nech d(x) je dlzka zoznamu na pozicii T[h(x)]  
- zlozitost vyhladania(vkladania, odstranovania) prvku x v hasovacej tabulke je v najhorsom pripade  
 O(1) (cas potrebny na vypocet hodnoty h(x)) plus d(x) => O (1 + d(x))  
- nech h je funkcia vybrata z univerzalnej mnoziny funkcii => ocakvana dlzka zoznamu je **n/m**

Koeficient zatazenia: **α = n/m**, m je pocet slotov tabulky, n pocet prvkov ulozenych v tabulke  
=> koeficient zatazenia vyjadruje priemerny pocet prkov zahasovanych na jednu poziciu  
- ocakavana zlozitost vyhladavania prvku je Θ(1 + α)  
- keby sa miesto zoznamu pouzi vyvazeny BVS, ocakavana zlozitost by bola O(1 + log(d(x)) =>  
 za predpokladu univerzalnej funkcie na O(1 + log α)  
- miesto zoznamu sa tiez moze pouzit dalsia hasovacia tabulka a postupovat rekurzivne, vznikne strom  
 hasovacich tabuliek

Hasovacia funkcia – metoda delenia  
- h(k) = k mod m  
- vyhody: rychlost  
- nevyhody: nevhodne spravanie pre niektore m  
- vhodne zvolene m je prvocislo

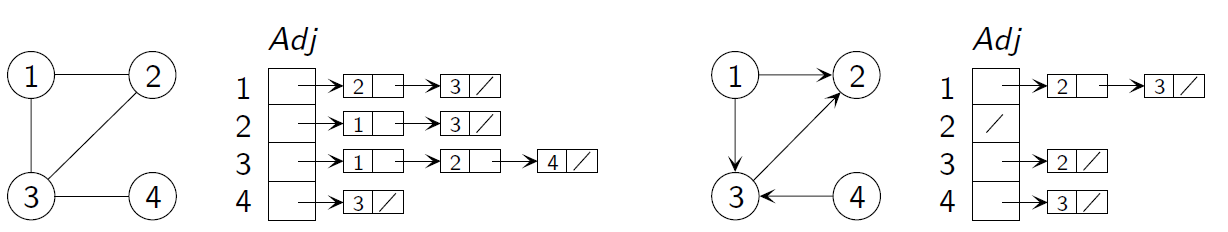
Hasovacia funkcia – metoda nasobenia  
- nech p je prvocislo, p > |U|  
- pre kazde cislo A ∊ {1, ..., p-1} definujeme funkciu hA: U -> {0, ..., p-1} predpisom  
- **hA(x) = (A x mod p) mod m**- mnozina funkcii M = {hA | A ∊ {1, ..., p} je univerzalna

Hasovacia funkcia – metoda binarneho nasobenia  
- jednoduchsia varianta metody nasobenia  
- nevyzaduje velke prvocisla  
- jednoduchsi vypocet  
- mnozina funkcii je skoro univerzalna



GRAFY  
- graf: G = (V,E) => mnozina vrcholov a hran  
- orientovany alebo neorientovany  
- ohodnoteny alebo neohodnoteny  
- sposoby reprezentacie grafu: 1. zoznam naslednikov, 2. incidencna matica  
- zlozitost grafovych algoritmov je vyjadrena ako funkcia poctu vrcholov |V| a hran |E|

Zoznam naslednikov  
- potrebujeme pole velkosti |V|  
- priestorova zlozitost Θ(V + E)  
- casova zlozitost vypisu vsetkych susedov vrcholu je O(degree(u))  
- casova zlozitost urcenia, ci (u,v) ∊ E je O(degree(u))



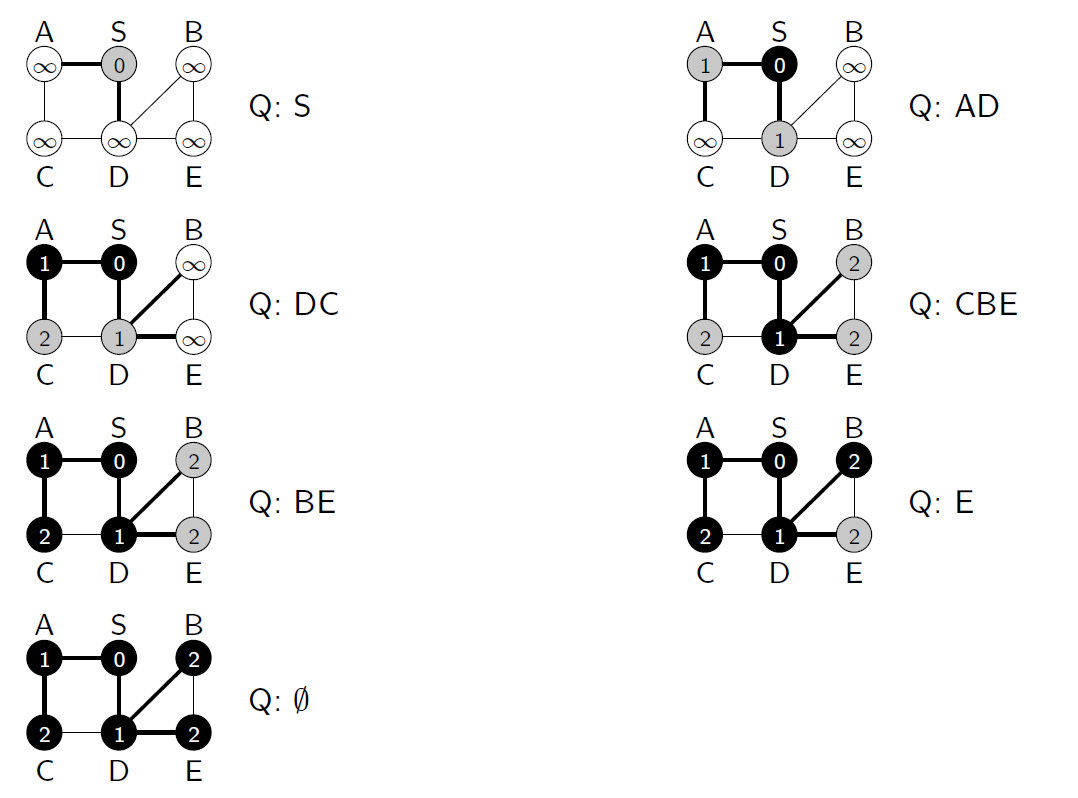
Incidencna matica  
- priestorova zlozitost je Θ(V2)  
- casova zlozitost vypisu vsetkych naslednikov vrcholu u: Θ(V)  
- casova zlozitost urcenia, ci (u,v) ∊ E je Θ(1)

Prehladavanie grafu do sirky  
- hrany prechadza po vrstvach  
- najde najkratsie cesty  
- najde silne suvisle komponenty neorientovaneho grafu  
- zlozitost je O(V+E), datova struktura FIFO

Vstup: graf G = (V,E) a vrchol s ∊ V

Vystup: pre kazdy vrchol v ∊ V  
 - v.d = vzdialenost z s do v ( = najmensi pocet hran na ceste z s do v)  
 - v.π = predchodca v na najkratsej ceste z s do v  
 - v.color = vrchol ma ciernu farbu prave ak je dosiahnutelny z vrcholu s, inak ma bielu farbu

- postupuje sa po vlnach  
- najprv preskumame vsetky vrcholy dosiahnutelne po 1 hrane, potom vsetky vrcholy dosiahnutelne  
 po 2 hranach, atd...  
- pre manipulaciu s vrcholmi sa pouziva prioritna fronta Q  
- v ∊ Q (fronty), prave ak ho dosiahla aktualna vlna, ale este sa z neho neposunula



- operacie s frontou maju konstantnu zlozitost, kazdy vrchol je vo fronte maximalne 1 krat => O(V)  
- zoznam naslednikov kazdeho vrcholu sa prechadza maximalne 1 krat, manipulacia s hranou ma  
 konstantnu zlozitost, celkovo **O(E)**  
- inicializacia ma zlozitost **Θ(V)**  
- celkova zlozitost BFS je O(V + E)  
- algoritmus BFS definuje atributmi π graf predchodcov  
- graf predchodcov je BFS strom

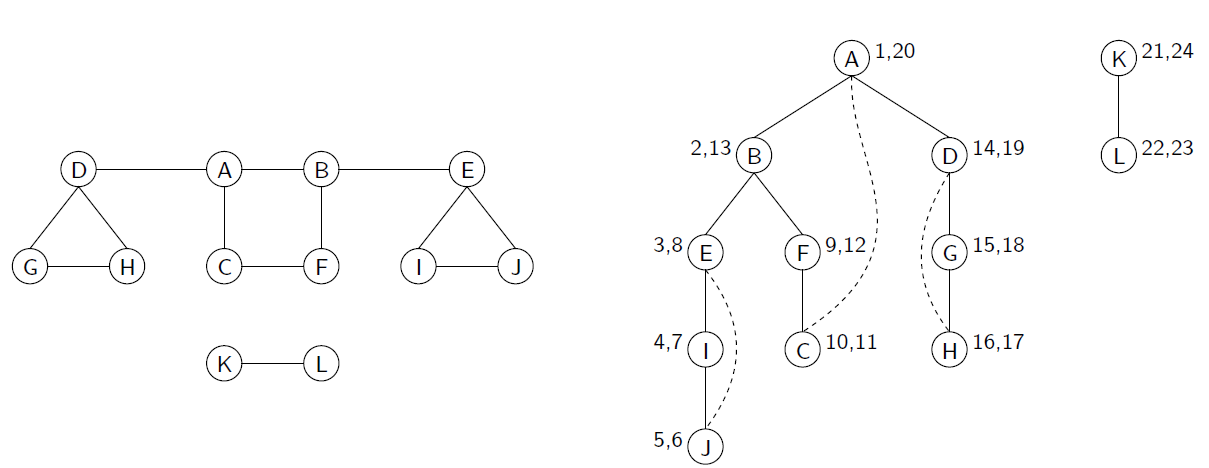
Prehladavanie grafu do hlbky  
- vzdy ide „dalej“, vrati sa len ak nema inu moznost  
- najde topologicke usporiadanie orientovaneho acyklickeho grafu  
- najde silne suvisle komponenty orientovaneho grafu  
- zlozitost je O(V+E), datova struktura LIFO

- nie je dany inicialny vrchol  
- na zaciatku a vzdy po dokonceni prieskumu vyberieme jeden z doposial nepreskumanych vrcholov  
 a zvolime ho za novy inicialny vrchol

Postup:  
- vyberieme nepreskumanu hranu, ktora vychadza z naposledy objaveneho vrcholu v takeho,   
 ze neboli preskumane vsetky z neho vychadzajuce hrany  
- ak vsetky hrany vychadzajuce z vrcholu v boli preskumane, tak sa vraciame naspat (backtrack)  
 do vrcholu, z ktoreho bol objaveny  
- prieskum konci ked su preskumane vsetky vrcholy dosiahnutelne z inicialneho vrcholu  
- definicia grafu predchodcov sa odlisuje od BFS  
- graf predchodcov tvori DFS les, ktory pozostava z niekolkych DFS stromov  
- DFS priraduje vrcholom farby a casove znamky

Farby:  
white – oznacuje nepreskumany vrchol  
grey – oznacuje, ze prieskum vrcholu uz zacal, ale este nebol dokonceny  
black – oznacuje, ze prieskum vrcholu bol ukonceny

Casove znamky:  
v.d – cas prvej navstevy vrcholu (discovery time)  
v.f – cas ukoncenia prieskumu vrcholu (finishing time)



- casova zlozitost DFS je Θ(V + E)

Vlastnosti casovych znamok  
- pre kazdy vrchol u plati: u.d < u.f  
- pre kazde dva vrcholy u, v plati prave jedna z podmienok:  
1. intervaly [u.d, u.f] a [v.d, v.f] su disjunktne a ani u ani v nie je naslednikom druheho z vrcholov v DFS  
strome  
2. interval [u.d, u.f] je cely obsiahnuty v intervale [v.d, v.f] a u je naslednikom v v DFS strome  
3. interval [v.d, v.f] je cely obsiahnuty v intervale [u.d, u.f] a v je naslednikom u v DFS strome

Vrchol v je dosiahnutelny z vrcholu u v DFS strome grafu G vtedy a len vtedy, ak u.d < v.d < v.f < u.f

Vlastnost bielej cesty  
v DFS strome grafu G je vrchol v naslednikom vrcholu u vtedy a len vtedy, ak v case u.d existuje  
cesta z u do v obsahujuca len biele vrcholy

Klasifikacia hran

Stromova hrana (tree edge) je hrana (u,v) obsiahnuta v DFS lese, pri prieskume hrany bol vrchol v biely  
u.d < v.d < v.f < u.f

Spatna hrana (back edge) je hrana (u,v) spajajuca vrchol u s jeho predchodcom v v DFS strome  
pri prieskume hrany bol vrchol v sedivy  
v.d < u.d < u.f < v.f

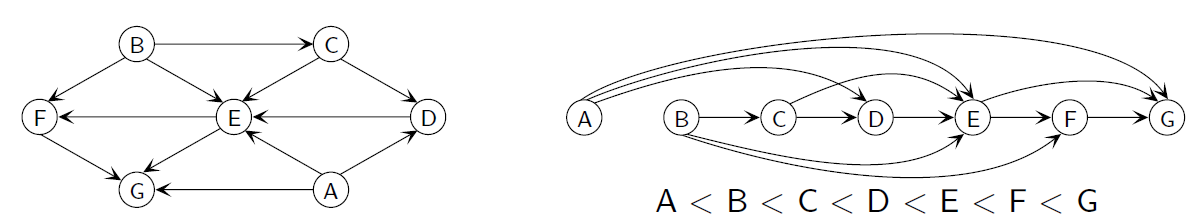
Dopredna hrana (forward edge) je hrana (u,v), ktora nepatri do DFS stromu a spaja vrchol u s jeho  
naslednikom v DFS strome  
pri prieskume hrany bol vrchol v cierny  
u.d < v.d < v.f < u.f

Priecna hrana (cross edge) vsetky ostatne hrany  
pri prieskume hrany bol vrchol v cierny  
v.d < v.f < u.d < u.f

Bipartitne grafy  
- jeho mnozinu vrcholov je mozne rozdelit na dve disjunktne mnoziny tak, ze ziadne dva vrcholy  
 z jednej mnoziny nie su spojene hranou  
- ak je graf bipartitny, neobsahuje cyklus neparnej dlzky  
- vrstvu Li tvoria vrcholy, ktorych vzdialenost od s je i (tj. v.d = i)

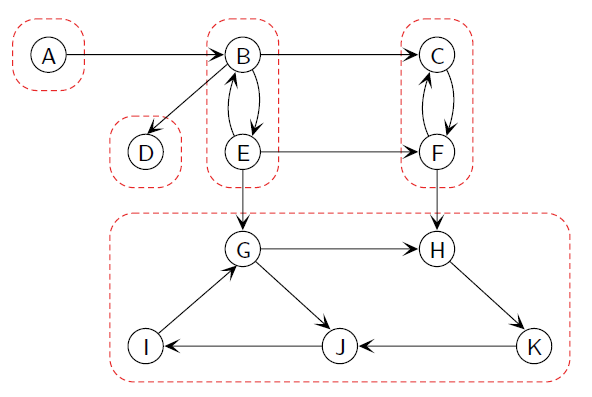
Nech G je suvisly graf a nech L0, L1, ... su vrstvy urcene BFS prieskumom grafu. Potom plati:  
1. V grafe neexistuje hrana, spajajuca dva vrcholy z tej istej vrstvy. Potom je graf bipartitny a   
zafarbenie vrcholov je urcene BFS vrstvami (parne vrstvy jedna farba, neparne druha farba)  
2. V grafe existuje hrana, ktora spaja dva vrcholy z jednej vrstvy. Potom graf obsahuje cyklus  
neparnej dlzky a nieje bipartitny.

Topologicke usporiadanie vrcholov grafu  
- linearne usporiadanie vsetkych vrcholov grafu take, ze ak graf obsahuje (u,v), tak v usporiadani vrchol  
 u predchadza vrcholu v  
- ak graf obsahuje cyklus, topologicke usporiadanie neexistuje  
- pre urcenie topologickeho usporiadania je zakladom DFS prieskum grafu

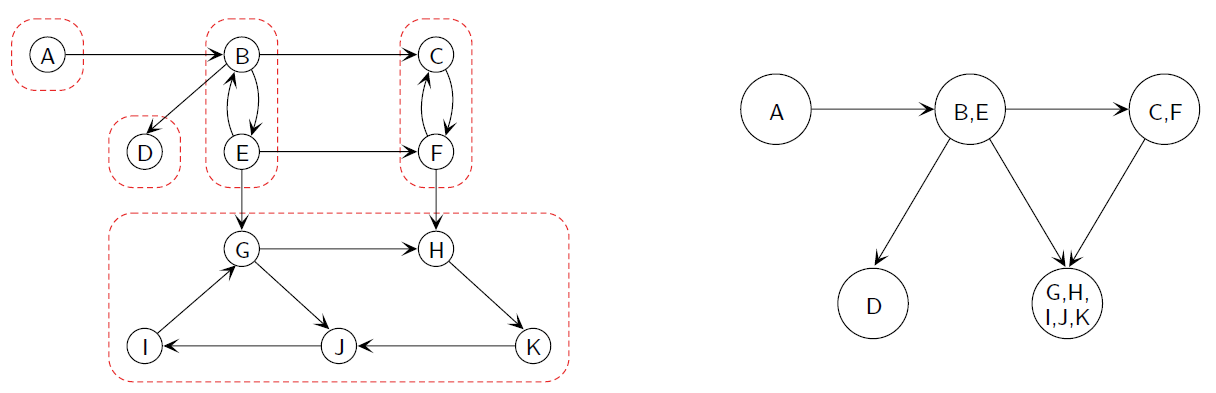


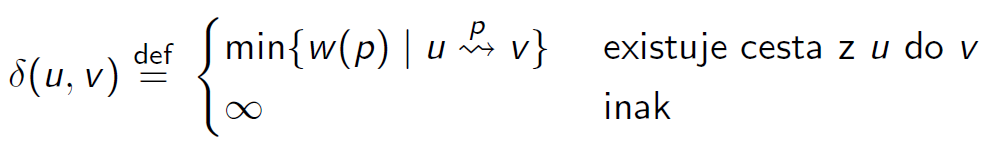
Orientovany graf je acyklicky, prave vtedy, ak pri DFS prieskume grafu ziadna hrana nie je klasifikovana  
ako spatna.

Silne suvisle komponenty (SCC)  
- maximalna mnozina vrcholov C ⊆ V taka, ze pre kazde u, v ∊ C plati, u 🡪 v a zaroven v 🡪 u

u 🡪 v plati prave ak v (orientovanom) grafe  
existuje (orientovana) cesta z vrcholu u   
do vrcholu v

Komponentovy graf  
- obsahuje jeden vrchol pre kazdu silne suvislu komponentu grafu  
- hranu obsahuje prave ak existuje hrana medzi vrcholmi prislusnych silne suvislych komponent

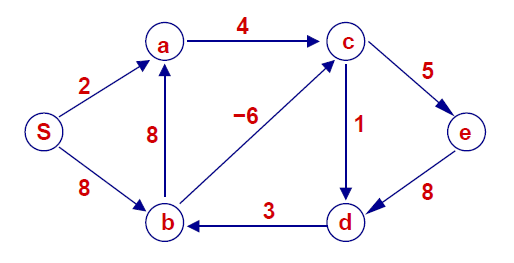


Ohodnotene hrany  
- vahova funkcia w: E -> R  
- dlzka cesty p je sucet dlzok (ohodnoteni) jej hran  
- dlzka najkratsej cesty z vrcholu u do vrcholu v je definovana:  


- najkratsia cesta z vrcholu u do vrcholu v je lubovolna cesta p (z u do v), pre ktoru plati w(p) = δ(u,v)

Najkratsie cesty  
1. najkratsie cesty z daneho vrcholu do vsetkych vrcholov (single source shortest path, SSSP)  
2. najkratsia cesta medzi danou dvojicou vrcholov (specialny pripad SSSP, pre tento problem nieje  
znamy ziadny asymptoticky rychlejsi algoritmus ako pre SSSP)  
3. najkratsie cesty medzi vsetkymi dvojicami vrcholov – da sa riesit opakovanou aplikaciou algoritmu  
pre SSSP ale existuju aj efektivnejsie algoritmy

Hrany so zapornou dlzkou

cyklus <b,c,d,b> ma dlzku -2

ziadna cesta z vrcholu s do vrcholu na   
cykle <b,c,d> nemoze byt najkratsou cestou

rovnako pre vrcholy dosiahnutelne z cyklu  
<b,c,d> (vrcholy a,e)

ak nejaka cesta z vrcholu u do vrcholu v  
obsahuje cyklus zapornej dlzky => δ(u,v) = -nek

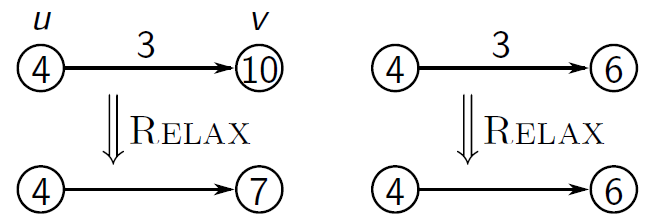
Ak graf obsahuje hrany so zapornou dlzkou:  
1. rozhodni, ci graf obsahuje cyklus zapornej dlzky  
2. ak graf neobsahuje cyklus zapornej dlzky, najdi najkratsie cesty

Najkratsia cesta nemoze obsahovat cyklus zapornej dlzky.  
Najkratsia cesta nemoze obsahovat cyklus kladnej dlzky.  
Najkratsia cesta moze obsahovat cyklus dlzky 0.

Kazda podcesta najkratsej cesty je najkratsou cestou.

hodnoty .π indikuju strom najkratsich ciest

Relaxacia  
- technika vyuzivana algoritmami pre hladanie najkratsich ciest  
- relaxacia hrany (u,v) je test, ci je mozne skonstruovat kratsiu cestu do v tak, ze prejdeme cez vrchol u,   
ak ano => aktualizujeme hodnoty v.d a v.π



Trojuholnikova nerovnost  
- pre kazdu hranu (u,v) ∊ E plati δ(s,v) <= δ(s,u) + w(u,v)

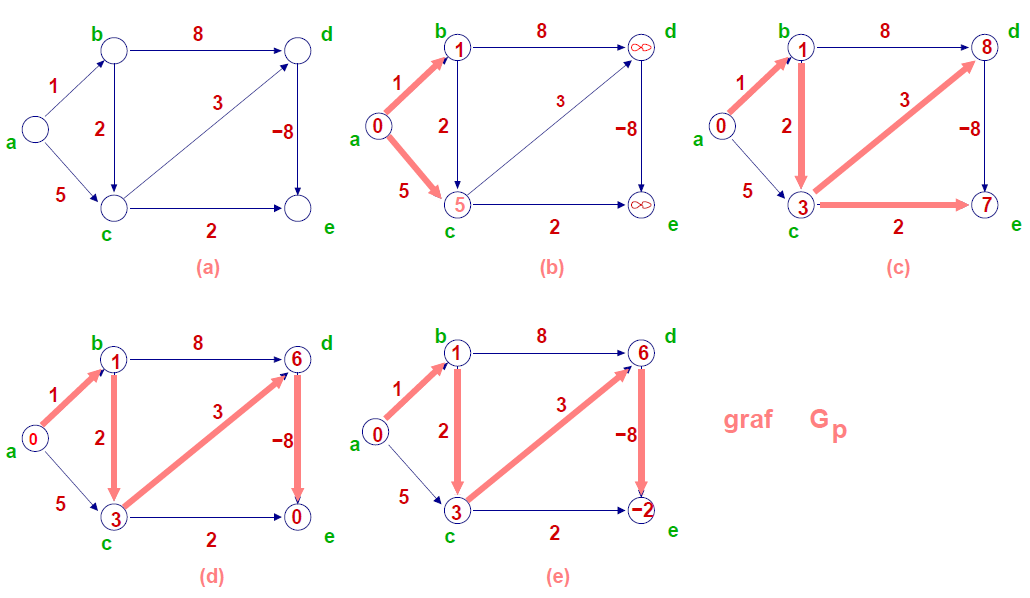
Horny odhad  
- pre kazdy vrchol v ∊ V plati v.d >= δ(s,v), ak hodnota v.d klesne na δ(s,v), tak sa uz nezmeni

Neexistencia cesty  
- ak neexistuje cesta z s do v, tak pocas celeho vypoctu plati: v.d = δ(s,v) = nekonecno

Konvergencia  
- ak s 🡪 u -> v je najkratsou cestou a ak u.d = δ(s,u) pred relaxaciou hrany (u,v), tak po relaxacii  
 plati: v.d = δ(s,v)

Ak v.d = δ(s,v) pre vsetky v ∊ V, potom graf predchodcov je strom najkratsich ciest

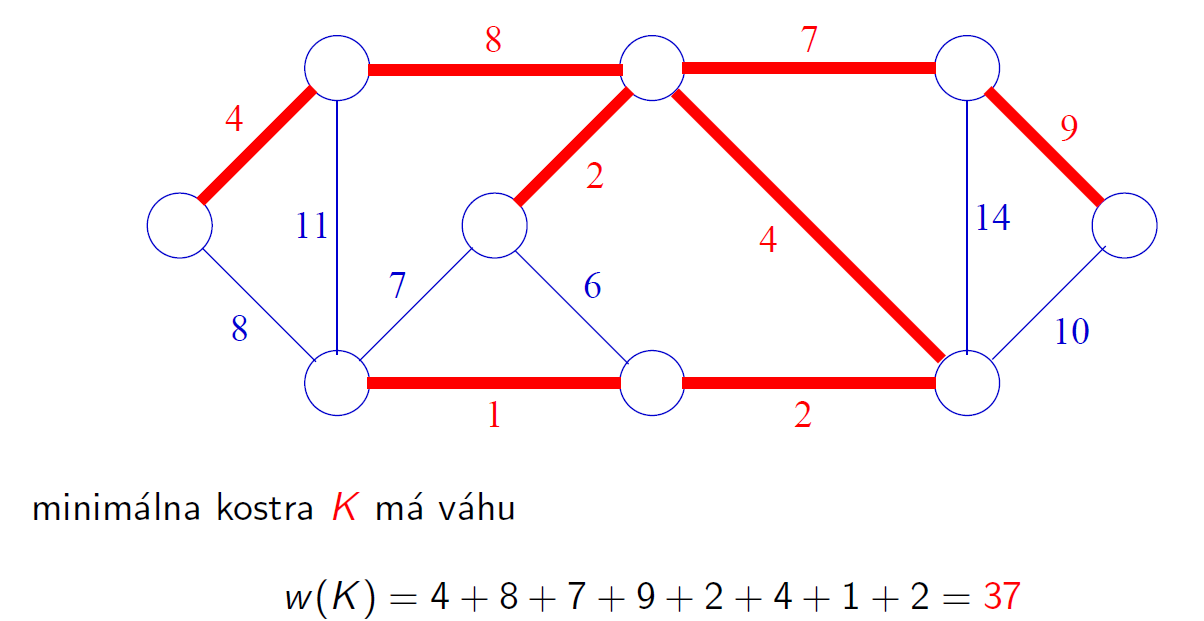
Bellman – Ford algoritmus  
- graf moze obsahovat hrany zapornej dlzky  
- ak graf neobsahuje cyklus zapornej dlzky dosiahnutelny z korena s => vypocita najkratsie cesty  
- ak graf obsahuje cyklus zapornej dlzky vrati false  
- je zalozeny na postupnom zlepsovani hodnot u.d  
- ak vrcholu u zlepsime hodnotu u.d, tak relaxujeme vsetky hrany (u,v) ∊ V  
- casova zlozitost je Θ(VE)

vypocet bellman-fordovho algoritmu pre graf s korenom a

Najkratsie cesty v orientovanom acyklickom grafe  
- optimalne poradie relaxacie hran je vypocitane pomocou topologickeho usporiadania vrcholov grafu  
- casova zlozitost je Θ(V+E)  
- z topologickeho usporiadania vyplyva, ze hrany kazdej cesty su relaxovane v poradi, v akom sa  
 vyskytuju na ceste

Djikstrov algoritmus  
- graf nesmie obsahovat hrany so zapornym ohodnotenim  
- zalozeny na BFS, pouziva sa prioritna fronta, klucmi su hodnoty v.d  
- udrzuje dve mnoziny:  
 1. S = vrcholy, pre ktore sa uz vypocitala dlzka najkratsej cesty  
 2. Q = prioritna fronta, Q = V \ S  
- vybera vrchol u ∊ Q s najmensou hodnotou u.d a relaxuje hrany vychadzajuce z u  
- zlozitost zavisi od sposobu reprezentacie prioritnej fronty Q

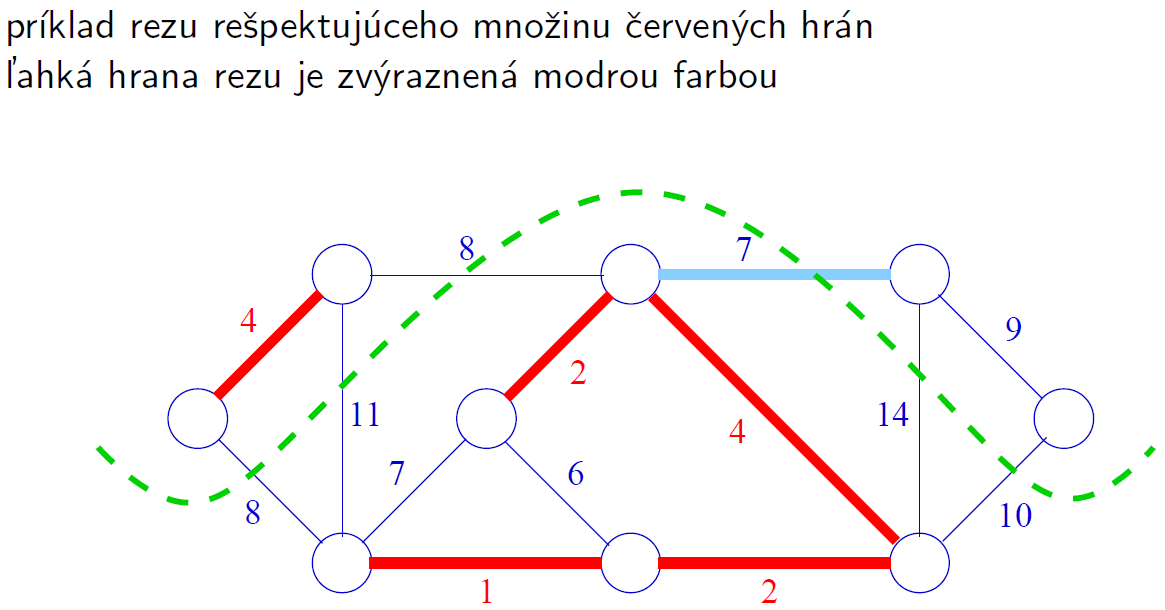
Minimalna kostra grafu  
- ak je dany suvisly neorientovany graf G = (V,H) spolu s ohodnotenim hran, tak kostra grafu je   
K ⊆ H prave ak graf G‘ = (V,K) je suvisly a acyklicky  
- vaha kostry je suma dlzok hran mnoziny K  
- minimalna kostra je kostra s minimalnou vahou



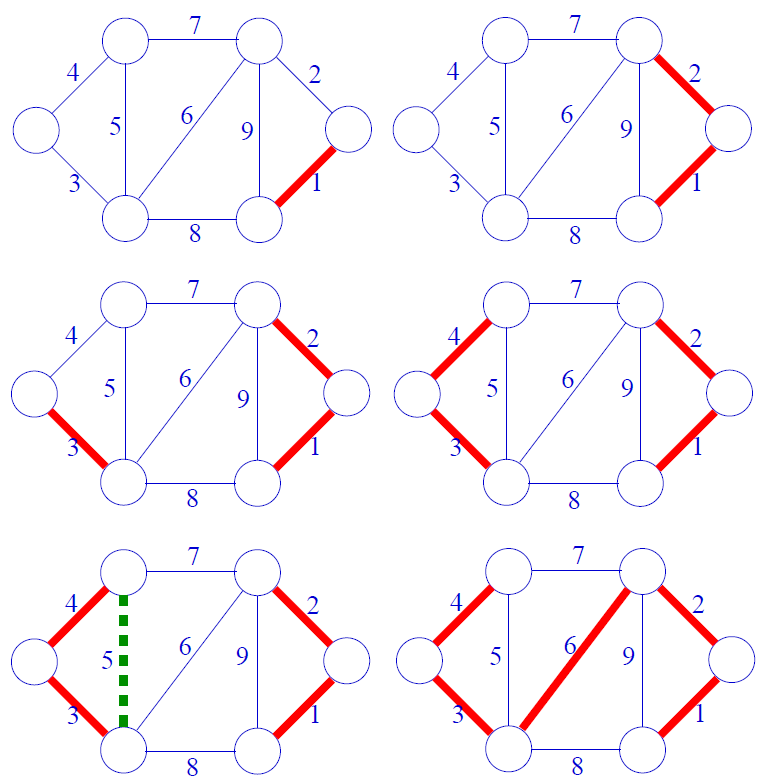
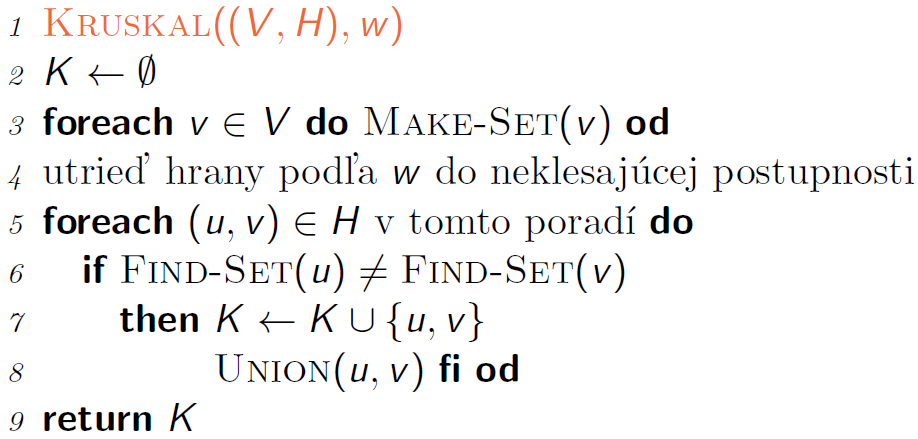
Za predpokladu jedinecneho  
ohodnotenia hran v grafe ma   
graf jednu minimalnu kostru

Ak hraju nemaju rozdielne  
ohodnotenie, v grafe moze  
existovat viac navzajom roznych  
minimalnych kostier

Hladanie bezpecnych hran  
- rez (S, V – S) v grafe je lubovolne rozdelenie mnoziny vrcholov do dvoch podmnozin  
- hrana h pretina rez prave ak jeden z jej koncovych vrcholov patri do S a druhy do V – S  
- rez (S, V – S) respektuje mnozinu hran K prave ak ziadna hrana z mnoziny K nepretina rez (S, V – S)  
- hrana h je lahkou hranou vzhladom k rezu (S, V – S) prave ak tento rez pretina a ma zo vsetkych  
 takych hran minimalnu vahu



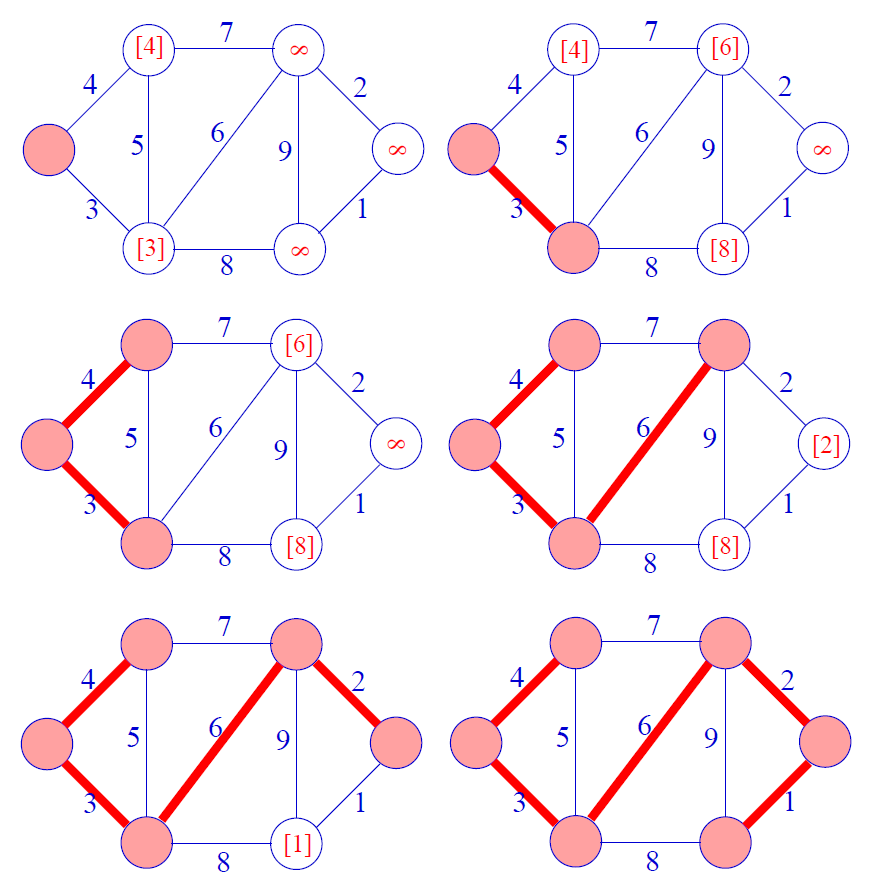
Kruskalov algoritmus  
- mnozina K tvori les, inicialne tvori kazdy vrchol grafu jeden strom lesa  
- v kazdej iteracii vyberame hranu h s minimalnou vahou zo vsetkych hran spajajucich rozne  
 komponenty z K



Inicializacia ma zlozitost Θ(|V|) + Θ(|H|log|H|)  
cyklus 5-8 prebehne |H| krat

Celkova zlozitost je O(|H|log|H|)

Primov algoritmus  
- mnozinu K tvori vzdy jeden strom  
- na zaciatku zvolime lubovolny vrchol r za koren a postupne budeme pridavat dalsie vrcholy, resp hrany  
- Vk je mnozina tych vrcholov, z ktorych vedie nejaka hrana h ∊ K  
- v kazdom kroku vybereme nejaku hranu h, ktora je lahka vzhladom k rezu (Vk, V – Vk) a pridame do K



Inicializacia ma zlozitost Θ(|V|)

V cykle while sa prevedie |V| operacii

Celkova zlozitost je O(|V|log|V|+|H|)