Obsah

Prednáška 01 – algoritmy, dátové štruktúry a zložitosť

Algoritmus je jednoznačný konečný opis (postupu riešenia) problému.

- Postup na vyriešenie úloh určitého typu
- vstup → výstup
- Jednoznačnosť krokov
- Všeobecnosť použitia
- Vlastnosti
    - Konečnosť
    - Efektívnosť

Algoritmus v informatike je jednoznačná, presná a konečná postupnosť operácií, ktoré sú aplikovateľné na množinu objektov alebo symbolov (čísiel, šachových figúrok, ingrediencií na bábovku).

Počiatočný stav týchto objektov je vstupom, ich **koncový stav** je výstupom. Počet operácií, vstupy a výstupy sú konečné (aj keď bežne počítame napr. s iracionálnym číslom π, vždy jeho číselnú reprezentáciu obmedzíme pri numerických výpočtoch na konečnú presnosť, napr π=3.14).

Jeden problém môže byť riešení viacerými algoritmami.

Vlastnosti algoritmov

- **Jednoznačnosť** – znamená, že každý krok algoritmu musí byť presne definovaný. Nesmie dovoľovať viac výkladov a jednoznačne je určený krok za ním nasledujúci.
- **Univerzálnosť** algoritmu – znamená, že je použiteľný pre riešenie veľkej skupiny úloh toho istého typu, líšiacich sa vstupnými údajmi (tak napr algoritmus pre hľadanie koreňov kvadratickej rovnice je použiteľný pre KAŽDÚ kvadratickú rovnicu).
- **Rezultatívnosť** – znamená, že algoritmus vždy musí po konečnom počte krokov dôjsť k nejakému riešeniu.
- **Správnosť** (korektnosť) algoritmu – znamená, že odpoveď pre každý vstup je správna a korektná – teda vyhovuje špecifikácii problému, ktorý algoritmus o sebe tvrdí, že rieši.
- **Efektívnosť** algoritmus – zahŕňa zdroje potrebné na vykonanie výpočtu – výpočtový čas, kapacita pamäte, počet správ pri komunikácií, a pod.

Dátová štruktúra

Dátová štruktúra je spôsob organizácie a uchovania dát, ktorý umožňuje efektívne využitie /spracovanie dát.

- Operácie – čo umožňuje s dátami robiť
- Konzistentnosť
- Vlastnosti
    - Súbežnosť (concurrency)
    - Perzistentnosť (persistency)
    - Dynamickosť (online/offline)
    - Efektívnosť

Operácie, ktoré s ňou môžeme realizovať:

- Vlož prvok do množiny
- Odstráň prvok z množiny
- Zisti, či prvok je v množine

Asymptotická zložitosť

Je to presné určenie počtu vykonaných operácií:

- veľmi náročné v prípade zložitých algoritmov
- skoro zbytočné pre jednoduché algoritmy

Jednoduchšie je analyzovať horné a dolné ohraničenia počtu vykonaných krokov.

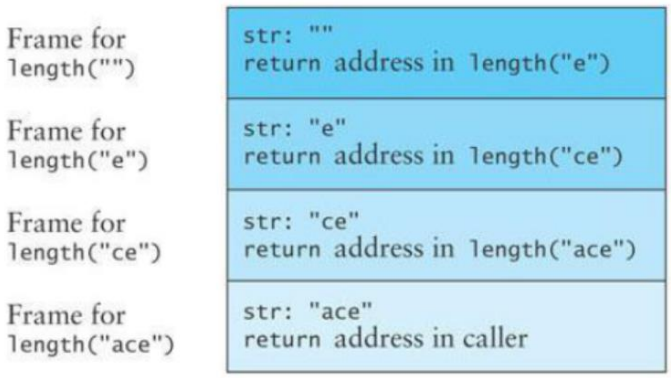
Asymptotická zložitosť je vyjadrenie pre takú (veľkú) veľkosť problému, aby sa prejavil rád rastu funkcie zložitosti v závislosti na veľkosti vstupu. Asymptoticky lepší algoritmus bude lepší pre všetky vstupy okrem konečného počtu malých vstupov.

Prednáška 02 – správa pamäte pri vykonávaní programu

Aktivačný rámec

Je to stavová informácia pre volanie funkcií. Pre vykonanie volania funkcie je potrebné uchovať nasledovné informácie:

- argumenty funkcie (hodnoty parametrov)
- adresa, kam sa má vrátiť vykonávanie programu po ukončení
- volania funkcie (návratová adresa pre return)
- lokálne premenné (hodnoty)



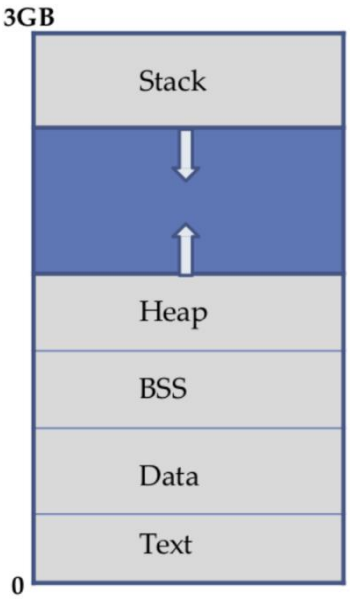
Pre každé volanie funkcie sa vytvorí **aktivačný rámec** (stack frame) a vloží sa do zásobníka volaní (call stack). (Úmyselné) **pretečenie zásobníka** volaní predstavuje bezpečnostné riziko: stack buffer overflow.

Program vs. proces

Program sa po preložení (compile), spojení s externými podprogramami (link) a načítaní do pamäti počítača (load) vykonáva (execute) – **proces.** Program je statický kód a statické dáta, proces je dynamická inštancia kódu, dát a ďalšieho.

Riadiaci blok procesu (process control block) spravuje:

- Stav procesu – new, ready running, waiting, ...
- Registre – %eip, %eax, ...
- Pamäť – všetko čo proces môže adresovať: kód, dáta, zásobník
- (stack), heap (halda)
- I/O – stav otvorenia-čítania súborov
- ...



Adresný priestor procesu (Process address space)

- **Text**: obsahuje program v strojovom jazyku, ktorý sa vykonáva, reťazce, konštanty, a ďalšie údaje na čítanie
- **Data**: inicializované globálne a statické premenné
- **BSS**: (Block Started by Symbol) neinicializované globálne a statické premenné
- **Stack** (zásobník): lokálne premenné bežiaceho procesu
- **Heap** (halda voľnej pamäti): dynamická pamäť procesu (môže sa zväčšovať aj zmenšovať) toto je pamäť, ktorú prideľuje malloc()

Typy prideľovania pamäte

1) **Statická veľkosť, statické prideľovanie**
   a) globálne premenné
   b) spojovač (linker) pridelí definitívne virtuálne adresy
   c) vykonateľný strojový program odkazuje na tieto pridelené adresy
2) **Statická veľkosť, dynamické prideľovanie**
   a) lokálne premenné
   b) prekladač predpíše prideľovanie v zásobníku
   c) posunutia voči ukazovateľu na vrch zásobníka (čo sú vlastne adresy premenných) sú priamo vo vykonateľnom strojovom programe
3) **Dynamická veľkosť, dynamické prideľovanie**
   a) ovláda programátor
   b) prideľuje sa v dynamickej voľnej pamäti (heap – halda)

Prednáška 03 – abstraktné dátové typy, vyhľadávanie, vyhľadávacie stromy

Dátový typ

Každá hodnota v programe má dátový typ. Množina použiteľných dátových typov je určená použitým programovacím jazykom. Dátový typ premennej určuje:

- Množinu hodnôt, ktoré možno dátovým typom reprezentovať
- Vnútornú reprezentáciu v počítači (využitie-kódovanie v pamäti)
- Prípustné operácie, ktoré možno nad hodnotami daného typu vykonávať

Abstraktné dátové typy (ADT)

Je to všeobecný model pre dátový typ (dátovú štruktúru) vyjadrený pomocou abstrakcie:

- Určíme operácie s dátovým typom a ich vlastnosti
- Abstrahujeme od konkrétnej implementácie

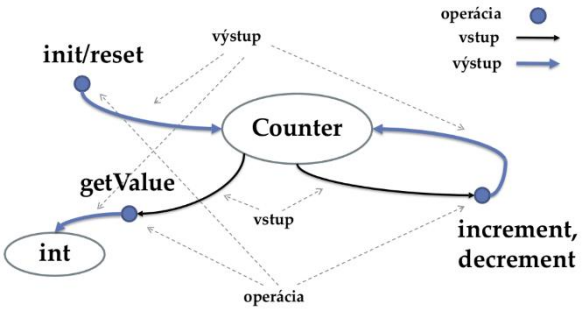
ADT môžeme implementovať rôznymi spôsobmi bez toho, aby to ovplyvnilo správnosť behu programu algoritmu, ktorý ADT používa.

Abstraktný dátový typ vs. dátová štruktúra

1) **Abstraktný dátový typ**
   - Množina typov údajov a operácií, ktoré sú špecifikované nezávisle od konkrétnej implementácie
   - Reprezentuje model zložitejšieho dátového typu
   - Abstraktný model
2) **Dátová štruktúra**
   - Implementácia ADT v programovacom jazyku
   - Reprezentácia typov údajov v ADT
   - Voľba algoritmov pre implementáciu operácií ADT

Definícia ADT

1) **Formálne** – signatúra a axiómy
   - **signatúra**



   - **axiómy** - opisujú vlastnosti – význam (sémantiku) operácií prostredníctvom ekvivalencie výrazov:

     Pre všetky C ∈ Counter platí:

     getValue (init) = 0

     getValue (increment(C)) = getValue(C) + 1

     getValue (decrement(C)) = getValue(C) – 1

2) **Programátorsky** – definícia rozhranie s operáciami
   - definícia rozhrania s operáciami

     int getValue();
   - možná implementácia
     - int getValue() { return value; }

Ďalšie ADT (s obmedzeným prístupom)

- **Zásobník** (Stack)
  c) Push (vlož na vrch zásobníka)
  d) Pop (vyber z vrchu zásobníka)
  e) LIFO – Last In, First Out
- **Rad-Front** (Queue)
  f) Enqueue (pridaj na koniec do radu-fronty)
  g) Dequeue (odober zo začiatku radu-fronty)
  h) FIFO – First In, First Out

Vyhľadávanie

Lineárne vyhľadávanie

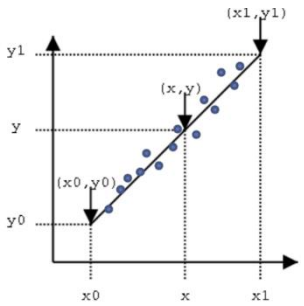
- O vstupnej postupnosti kľúčov nemám žiadne znalosti
- Algoritmus: prehľadám postupne všetky prvky
- Počet vykonaných operácií: najlepší prípad: O(1), najhorší prípad O(n), priemerne (n)

N	log N
10	4
1000	10
1 000 000	20
2 000 000 000	32

Binárne vyhľadávanie (metóda pólenia intervalu)

- Vždy skontrolujem prvok v strede intervalu, v ktorom hľadaný kľúč ešte môže byť
- Ak som našiel hľadaný kľúč, končím, inak pokračujem v zostávajúcej polovici intervalu
- Koľko bude porovnaní: najlepší prípad: O(1), najhorší prípad keď sa hľadaný kľúč v postupnosti nenachádza: O(log n) (koľko razy môžem skrátiť interval na polovicu, až kým nedostanem posledný prvok)

Interpolačné vyhľadávanie



- Predpokladajme rovnomernú distribúciu hodnôt kľúčov, na intervale <x0,x1> nadobúdajú hodnoty <y0,y1>
- Hľadám kľúč y ako určím čo najpravdepodobnejší výskyt – index x, taký, že k(ax) je blízko y?

$$\frac{y-y_0}{y_1-y_0} = \frac{x-x_0}{x_1-x_0}$$

$$x = x_0 + \frac{(y-y_0)*(x_1-x_0)}{y_1-y_0}$$

- Výpočtová zložitosť pre postupnosť s n prvkami je O(log log n) krokov

Dynamická množina poľom

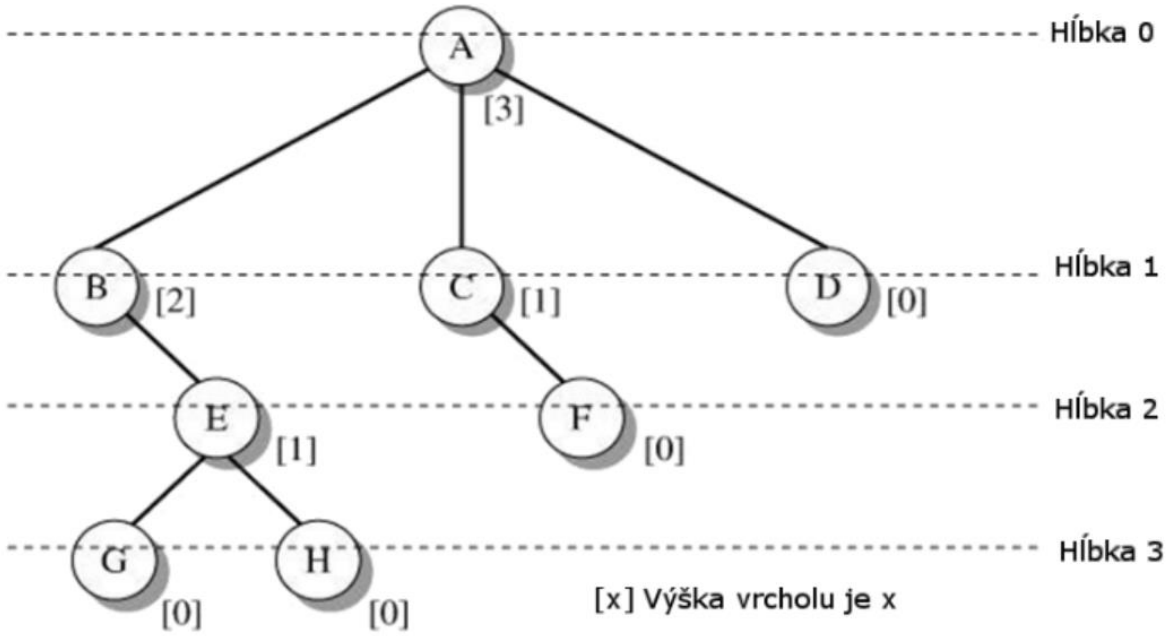
Vieme spraviť dátovú štruktúru, podporujúcu operácie:

- **Pridanie prvku** X – vyžaduje O(N) operácií, kde N je počet prvkov v postupnosti
- **Vyhľadanie prvku** X – vyžaduje O(log N) operácií, využijeme binárne vyhľadávanie
- **Odstránenie prvku** X – vyžaduje O(N) operácií, kde N je počet prvkov v postupnosti

Prednáška 04 – binárne vyhľadávacie stromy

Strom je súvislý neorientovaný graf bez cyklov. Strom, v ktorom je význačný vrchol – koreň (root) je **zakorenený**.

- Hĺbka vrcholu – počet hrán od koreňa stromu k danému vrcholu
- Výška vrcholu – dĺžka najdlhšej cesty z daného vrcholu k listu (koncovému vrcholu)
- Výška stromu – výška jeho koreňa



Binárny strom – operácie

- **CREATE**: vytvorenie prázdneho binárneho stromu
- **MAKE**: vytvorenie binárneho stromu z dvoch už existujúcich binárnych stromov a hodnoty
- **LCHILD**: vrátenie ľavého podstromu
- **DATA**: vrátenie hodnoty koreňa v danom binárnom strome
- **RCHILD**: vrátenie pravého podstromu
- **ISEMPTY**: test na prázdnosť

Binárny strom – Formálna špecifikácia

CREATE() → bintree
LCHILD(bintree) → bintree
RCHILD(bintree) → bintree

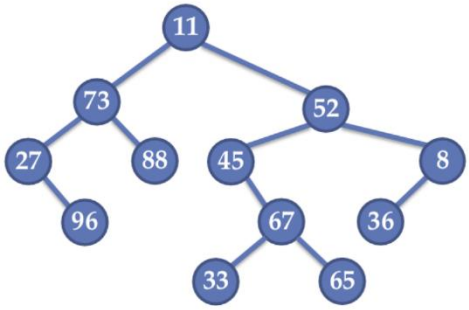
MAKE(bintree, item, bintree) → bintree
DATA(bintree) → item
ISEMPTY(bintree) → boolean

- Pre všetky p,r ∈ bintree, i ∈ item platí:

ISEMPTY(CREATE) = true
LCHILD(MAKE(p,i,r)) = p
DATA(MAKE(p,i,r) = i
RCHILD(MAKE(p,i,r)) = r

ISEMPTY(MAKE(p,i,r)) = false
LCHILD(CREATE) = error
DATA(CREATE) = error
RCHILD(CREATE) = error

Prehľadávanie binárnych stromov

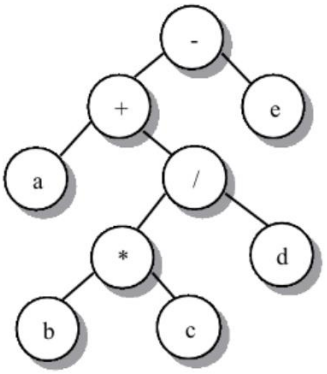


Existujú tri základné algoritmy:

- **pre-order** poradie prehľadávania:
  - koreň → ľavý podstrom → pravý podstrom
  - 11,73,27,96,88,52,45,67,33,65,8,36
- **in-order** poradie prehľadávania:
  - ľavý podstrom → koreň → pravý podstrom
  - 27,96,73,88,11,45,33,67,65,52,36,8
- **post-order** poradie prehľadávania:
  - ľavý podstrom → pravý podstrom → koreň
  - 96,27,88,73,33,65,67,45,36,8,52,11

Prehľadávanie stromu aritmetického výrazu

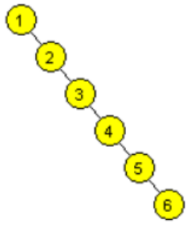
- **pre-order** prechádzanie stromu poskytne prefixový zápis výrazu
  - preorder (Prefix): - + a / * b c d e
- **in-order** prechádzanie stromu poskytne infixový zápis výrazu
  - inorder (Infix): a + b * c / d – e
- **post-order** prechádzanie stromu poskytne postfixový zápis výrazu
  - postorder (Postfix): a b c * d / + e -



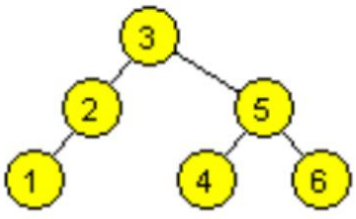
a + b*c/d -e

Analýza zložitosti – search

- Závisí od hĺbky h – O(h)
  a) **najhorší prípad** O(n) nájdenie uzla 6
  b) **priemerný a zároveň najlepší prípad** O(log n) nájdenie uzla 6



a)



b)

Analýza zložitosti – insert

Musíme nájsť miesto, kde môžeme prvok vložiť – časová zložitosť závisí od hĺbky stromu – O(h).

Najhorší prípad O(n): zoradená postupnosť – vytvárame nevyvážený strom – rýchle zväčšovanie hĺbky stromu. Napr. a) 1, 2, 3, 4, 5, 6

Priemerný prípad O(log n): náhodná postupnosť – vytvárame väčšinou „dobre“ vyvážený strom – pomalé zväčšovanie hĺbky stromu. Napr. b) 3, 5, 6, 2, 4, 1

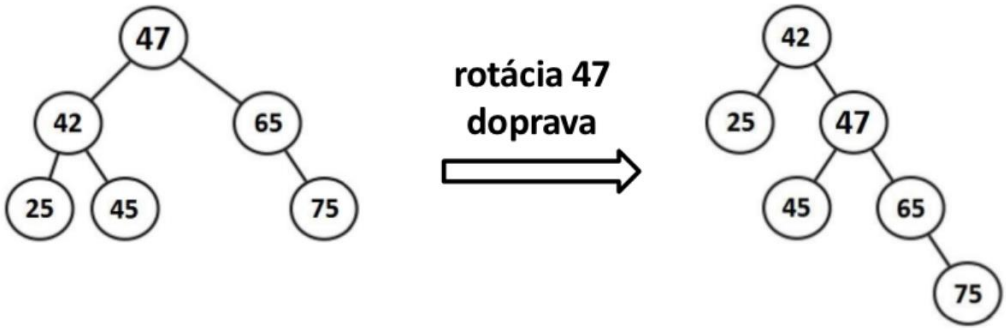
Vyvážené vyhľadávacie stromy – Prehľad

- **Základné vyvažovanie – podrobne**
  - AVL strom
  - Splay strom
- **Ostatné: definícia, insert, vlastnosti**
  - B stromy
  - (a,b) stromy: 2,3 a 2,3,4 stromy
  - Červeno-Čierne (Red-Black) stormy
  - Váhovo vyvážené
- **Optimálne binárne vyhľadávacie stromy**
- **A ďalšie:**
  - Trie – dynamická množina reťazcov
  - Radixový strom, lano, ...

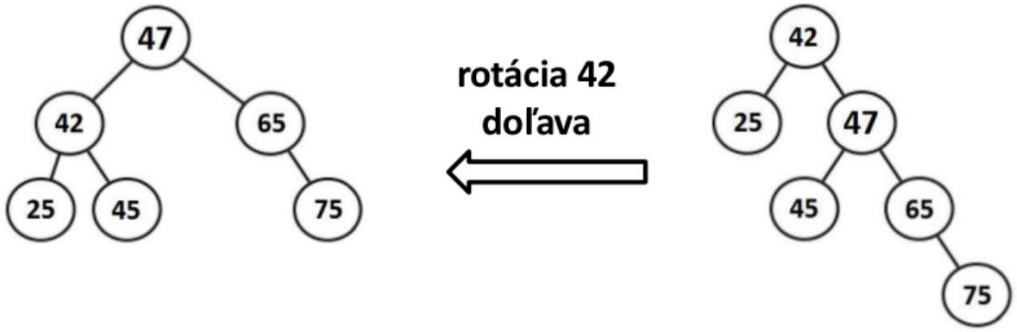
Rotácie stromu

Je to operácia, ktorá zmení štruktúru ale zachová usporiadanie. Zmena tvaru stromu – zmena výšky stromu. In-order poradie (oba stromy): 25, 42, 45, 47, 65, 75

Na získanie optimálnej zložitosti O(log n) musíme zabezpečiť, aby strom po vykonaní každej operácie zostal vyvážený



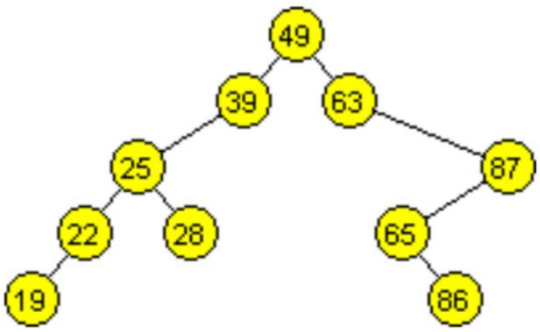
Zmena hĺbky 25(-1), 42(-1), 47(+1), 65(+1), 75(+1)



Zmena hĺbky 25(+1), 42(+1), 47(-1), 65(-1), 75(-1)

Prednáška 05 – pokročilé vyhľadávacie algoritmy, vyhľadávacie stromy a pod.

Nová operácia: nájdi k-ty prvok v strome



Prvé riešenie:

- Využiť in-order usporiadanie, zobrať k-ty prvok
- Zložitosť **O(k)**
- Napr. k=5
- In-order: 19, 22, 25, 28, **39**, 49, 63, 65, 86, 87

Lepšie riešenie:

- Využiť princíp QuickSelect algoritmu pri porovnaní vo vrchole pokračovať len v podstrome, v ktorom sa k-ty prvok nachádza
- **Potrebujeme pre každý vrchol x poznať:**
  - počet prvkov v podstrome strome s koreňom x
- Implementácia ako rozšírenie štandardnej dátovej štruktúry BVS, **rozšírime údaje pre vrchol**:
  - ľavý, pravý, rodič,
  - počet (prvkov v podstrome) tzv váha
- Rekurzívna definícia váhy
  - **váha**(v) = váha(ľavýPodstrom(v)) + váha(pravýPodstrom(v)) + 1
- Hodnoty váha vo vrcholoch upravujeme pri každej operácií ktorá mení štruktúru stromu: zložitosť **O(h),** kde h je výška stromu

Order statistic tree

Rozšírenie BVS stromu, pre každý vrchol BVS si navyše pamätáme počet prvkov v podstrome vrcholu tzv **váhu**. Hodnoty váhy vo vrcholoch upravujeme pri každej operácií ktorá mení štruktúru stromu (insert, delete).

Rozšírený strom podporuje navyše operácie:

- **select**(k) – nájdi k-ty najmenší prvok v množine
- **rank**(x) – nájdi poradie prvku x v usporiadanej postupnosti prvkov stromu

Zložitosť operácií je O(h), kde h je výška stromu.

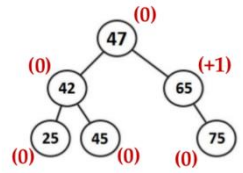
Výškovo vyvážené stromy (AVL stromy)

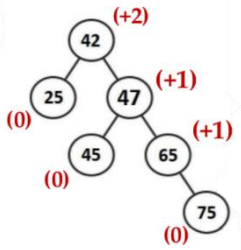
Označme **faktor vyváženia** (balance factor) vo vrchole:

- bf(v) = výška(pravýPodstrom(v)) – výška(ľavýPodstrom(v))

Výškovo vyvážený strom (AVL strom):

- |bf(v)| **≤ 1**, pre každý vrchol v

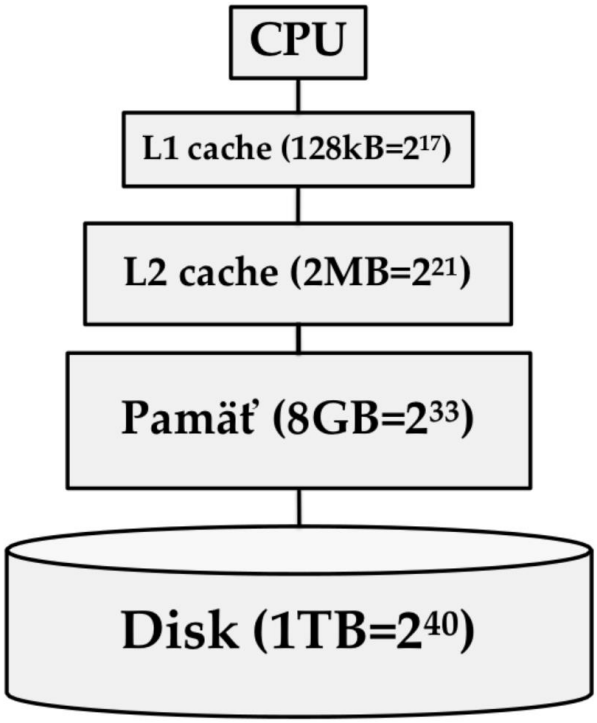




Operácie nad AVL stromom

- Ak operácia **nemení štruktúru stromu** (napr min, max, select, succ, pred) vykonávame rovnako ako pri BVS, ale navyše máme garantovanú zložitosť **O(log N),** kde N je počet vrcholov v AVL strome.
- Ak operácia **mení štruktúru stromu** (napr insert, delete), vykonáme rovnako ako pri BVS a následne dodatočne vyvážime strom rotáciami. Časová zložitosť jednej rotácie O(1). Vykonáme najviac h rotácií, kde h = O(log N), preto celková zložitosť insert aj delete je **O(log N).**

Aké rýchle sú operácie v počítači?



- 4GHz procesor ≈ 2^{32} B/s
- Načítať dáta do L1 cache trvá 2 takty CPU: ≈ 2^{31} B/s
- Načítať dáta do L2 cache trvá 30 taktov: ≈ 2^{27} B/s
- Načítať dáta do pamäte trvá 250 taktov: ≈ 2^{24} B/s
- Načítať dáta z nového miesta na disku trvá asi 8M inštrukcií: ≈ 2^{9} B/s

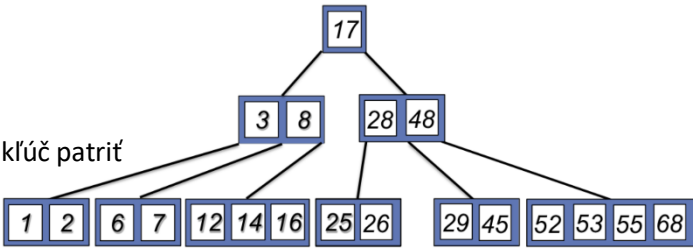
Porovnanie zložitostí algoritmov

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

B-strom (B-tree)

Myšlienka: vo vrcholoch sa budeme viac vetviť. Vyšší stupeň vetvenia vo vrcholoch => menšia výška stromu. B-strom rádu m je strom, v ktorom každý vrchol môže mať najviac m detí, pričom:

- Počet kľúčov vo vrchole je o 1 menší než počet jeho detí (kľúče rozdeľujú intervaly kľúčov v podstromoch)
- Všetky listy sú v rovnakej hĺbke
- Každý vnútorný vrchol okrem koreňa má $\lceil \frac{m}{2} \rceil$ aspoň detí
- Koreň je buď list alebo má 2 až m potomkov
- List obsahuje najviac m – 1 kľúčov
- Rád B-stromu m je nepárne číslo



Insert do B-stromu

- Vyhľadám list, do ktorého má nový kľúč patriť
- Vložíme nový kľúč do tohto listu
- Opravím chyby:
  - Ak by list už obsahoval príliš veľa kľúčov, rozdelíme ho na dva vrcholy, a prostredný kľúč posunieme vyššie (do rodiča)
  - Rekurzívne pokračuj až do koreňa: ak by rodič už obsahoval príliš veľa kľúčov, rozdeľ ho a stredný kľúč posun vyššie
  - V prípade koreňa (ak je to potrebné): koreň rozdelíme na dva vrcholy, a prostredný kľúč vytvorí nový koreň, čím sa zvýši celková výška B-stromu

(a,b) stromy

Ak umožníme ešte väčší stupeň vetvenia (b ≥ 2a), tak vyvažovanie pracuje efektívnejšie – (a,b) stromy:

- Pri insert/delete postačuje upraviť O(1) blokov
- V praxi preferovaný typ stromu v porovnaní s klasickým B-stromom

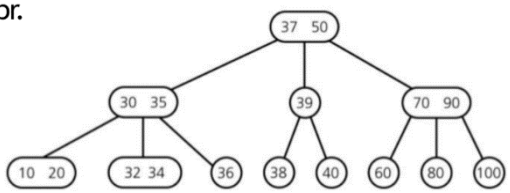
2-3 stromy (2-3 trees)

Špeciálny prípad B-stromu pre m=3.

- Každý vnútorný vrchol má dve alebo tri deti.
- Všetky listy sú v rovnakej hĺbke a každý obsahuje najviac 2 hodnoty.

B-stromy sú vždy vyvážené, takže 2-3 strom je dobrý príklad vyváženého vyhľadávacieho stromu

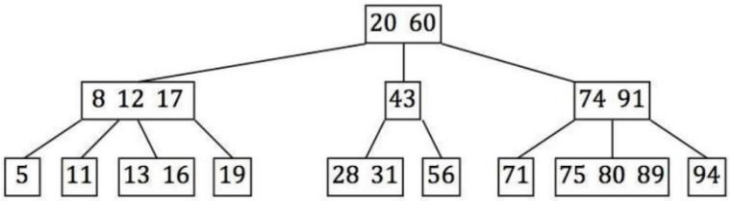
Napr.



2-3-4 stromy (2-3-4 trees)

Špeciálny prípad B-stromu, resp. (a,b) stromu pre (2,4)

- Každý vnútorný vrchol má dve, tri alebo štyri deti
- Všetky listy sú v rovnakej hĺbke a každý obsahuje najviac 3 hodnoty



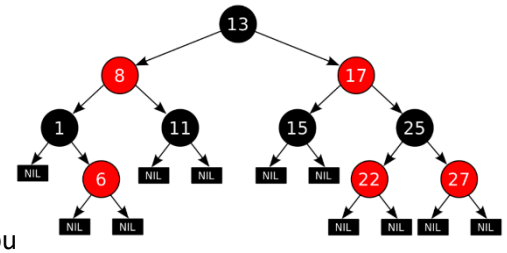
Insert do 2-3-4 stromu

- Nájdi list, do ktorého sa bude hodnota vkladať.
- Počas hľadania, keď narazíš na 4-vrchol, tak ho rozbaľ.
- Ak je list, do ktorého vkladáme 2-vrchol alebo 3-vrchol, tak vlož do listu.
- Ak je list (po vložení) 4-vrchol, tak ho rozbaľ tak, že prostrednú hodnotu vlož do rodiča a vkladanú hodnotu vlož do príslušného listu.
    - Miesto v rodičovi sa určite nájde, keďže sme pri ceste dole rozbalili všetky 4-vrcholy. Preto nemusíme rekurzívne postupovať hore ako v prípade 2-3 stromov.

Červeno-čierny strom (red-black tree)

Je to binárny vyhľadávací strom s vrcholmi ofarbenými na červeno alebo čierno, taký že:

- Cesty z koreňa do listov majú rovnaký počet čiernych vrcholov a tento počet označujeme čierna výška stromu
- Koreň je čierny
- Listy neobsahujú dáta a sú čierne
- Ak je vrchol červený, tak jeho deti sú čierne



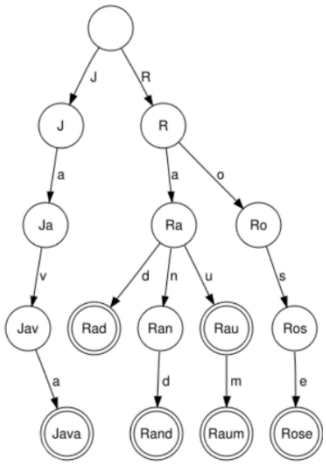
Vlastnosti:

- Na žiadnej ceste nie sú dva červené vrcholy za sebou
- Dĺžka cesty z koreňa do najvzdialenejšieho listu nie je viac ako dvakrát dlhšia ako cesta do najbližšieho listu
- Každý vnútorný vrchol má dvoch potomkov
-
- **Čierna výška vrcholu** = počet čiernych vrcholov na ceste z vrcholu do listu
- Najhoršia výška stromu s n listami: **2 log n – O(1)**

Trie – dynamická množina reťazcov

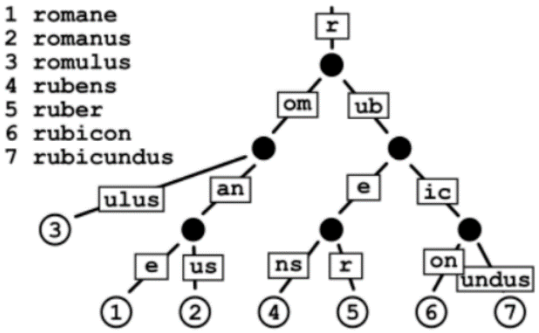
Je to strom prefixov reťazcov. Vrcholy zodpovedajú prefixom hodnôt prvkov v množine, nasledovníci vrcholu majú spoločný prefix.

- Koreň je prázdny reťazec
- Nie každý vrchol zodpovedá reťazcu z množiny
- Každý z listov zodpovedá nejakej hodnote v množine
- Každá hrana (prechod) má priradené písmenko



Radixový strom (radix tree)

- Priestorovo optimalizovaný trie
- Také vrcholy, ktoré sú jediné dieťa svojho rodiča sú spojené s rodičom do jedného vrcholu
- Hrany môžu mať priradený reťazec



Prednáška 06 & 07 – hashovanie

Základná myšlienka hashovania je hashovacou funkciou h zobraziť kľúč x do rozsahu indexov poľa – vložiť prvok s kľúčom x na index poľa h(x): h(Milan) = 2

Zovšeobecnený prístup, keď:

- Kľúče nemusia byť rôzne
- Rozsah (univerzum) kľúčov môže byť veľký (v porovnaní s počtom prvkov)
- Kľúče nemusia byť celé čísla

Očakávaná zložitosť všetkých operácií (insert,search, delete): **O(1)**

Pamäťová zložitosť: **O(n)**

Spôsoby riešenia kolízií

Synonymá – prvky, ktoré majú rôzne kľúče x1 ≠ x2, ale rovnaké hashovacie hodnoty h(x1) = h(x2), teda vkladám prvok x, ale miesto h(x) je už obsadené. Existujú dva **spôsoby riešenia kolízií**:

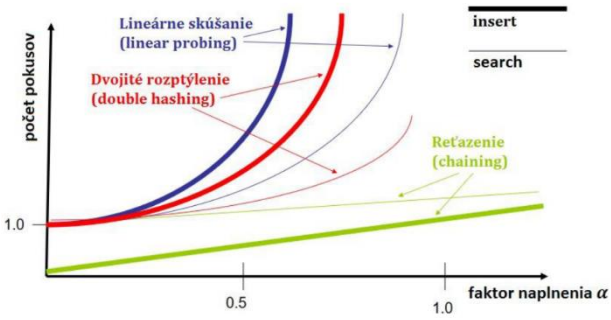
- **Reťazenie** (chaining) – umožníme,aby v každom mieste tabuľky mohlo byť aj viacero prvkov (napr. v spájanom zozname)
    - Políčko tabuľky nazývame vedierko (bucket)
        - zvyčajne spájaný zoznam
        - môže byť aj iné, napr binárny vyhľadávací strom (usporiadaný
    - Podľa nejakého sekundárneho kľúča)
    - Odhad zložitosti:
        - uvažujme, že v tabuľke je N prvkov v M vedierkach faktor naplnenia α = N/M (priemerný počet prvkov vo vedierku)
        - čas potrebný pre výpočet h(x): O(1)
        - očakávaný čas na vyhľadanie prvku vo vedierku: O(α)
        - ak je počet vedierok úmerný počtu prvkov N=O(M): α = N/M = O(M)/M = O(1)

- **Otvorené adresovanie** (open addressing) – v každom mieste tabuľky môže byť len jeden prvok (musíme nájsť nejaké alternatívne umiestnenie prvkov, ktoré majú rovnakú h(x) hodnotu)
    - Rôzne algoritmy sa líšia spôsobmi, ako prvky s rovnakou h(x) umiestnime tak, aby sme ich neskôr vedeli vyhľadať
    - Najjednoduchší prístup je prvok umiestniť na najbližšie voľné miesto v tabuľke:
        - h(Milan) = 2, h(Peter) = 2 => Peter pôjde do okienka 3
    - Dva hlavné prístupy k skúšaniu:
        - **Lineárne skúšanie**: h(x,i) = (h(x) + i) mod N
            - Nevýhoda lineárneho skúšania je, že prvky sa zoskupujú do súvislých obsadených postupností – tzv. strapcov (klastrov), čo významne spomaľuje vykonávanie operácií
            - Vzniká tzv. primárne klastrovanie – dlhé strapce zvyšujú priemerný čas vyhľadania
        - **Dvojité rozptýlenie**: h(x,i) = (h(x) + i · g(x)) mod N
            - Posun vypočítame druhou hash funkciou g
            - V praxi sa blíži k ideálnemu rovnomernému rozptýleniu

Reťazenie vs. otvorené adresovanie

Reťazenie:

- Nutný priestor navyše pre smerníky – zlá lokalita v pamäti
- Počet prvkov v tabuľke nie je obmedzený ($\alpha > 1$)
- Lineárne klesá výkonnosť pri zväčšujúcom sa α



Otvorené adresovanie:

- Prvky sú priamo v tabuľke – nie sú nutné smerníky navyše
- Obmedzený celkový počet prvkov ($\alpha \leq 1$)
- Rovnaké množstvo pamäti môže mať väčšiu tabuľku ako pri zreťazení
- Dobrá lokalita v pamäti – výhodné pre cachovanie prístupov do pamäti
- Faktor α značne ovplyvňuje výkonnosť
- Pri nízkych $\alpha < 0.5$ rýchlejšie ako reťazenie (netreba prechádzať smerníky)
- Výrazné spomalenie pre α blízke 1
- Nepodporuje delete (odstránenie), resp. použitie deleted symbolu výrazne spomaľuje vyhľadanie aj pri nízkych α

Hashovacia (rozptylová) funkcia

```
int hash(char *str)
{
    int i, len = strlen(str), h = 0;
    for (i = 0; i < len; i++)
        h = 31*h + str[i];
    return h;
}
```

- Výpočet by mal byť rýchly
- Dobrá hashovacia funkcia:
  - minimalizuje počet kolízií
  - rovnomerne rozptyľuje prvky do celej tabuľky
  - pravdepodobnosť, že $h(x)=i$ je $1/n$ pre každé $i \in \{0, 1, ..., N-1\}$
- Zvyčajne ako kompozícia dvoch funkcií $h(x) = h2(h1(x))$:
  - výpočet hashkódu h1: kľúč → celé číslo
    - zobrazuje kľúč x na celé číslo
    - snažíme sa navrhnúť výpočet hashkódu tak, aby čo najlepšie predchádzal kolíziám
  - kompresná funkcia h2: celé číslo → {0, 1, ..., N-1}
    - kompresná funkcia nemá ako „opraviť“ kolíziu v hashkódoch
- Obe funkcie (h1 a h2) by mali byť navrhnuté pre celkovú minimalizáciu počtu kolízií
- Polynomiálna akumulácia

Univerzálne hashovanie (universal hashing)

Základné pozorovanie: najlepšie ako rozptýliť prvky v tabuľke je náhodne, čo však nemôžeme použiť, lebo by sme ich tam nevedeli (opakovane) vyhľadať. Takže by sme chceli, aby h (hashovacia funkcia) bola **pseudonáhodná.**

Chceli by sme čo najlepší najhorší prípad. Skúsme randomizovať konštrukciu hashovacej funkcie. Funkcia h bude deterministická, ale ak ju vyberieme takto „pravdepodobnostne“, tak pre ľubovoľnú postupnosť insert a search operácií bude **očakávateľne dobrá**.

Randomizovaný algoritmus H pre konštrukciu hashovacích funkcií h: U → {1, ..., M} je univerzálny ak pre každé x ≠ y v U platí:

$$\Pr_{h \leftarrow H}[h(x) = h(y)] \leq \frac{1}{M}$$

Očakávaný počet kolízií kľúča x s inými je N/M.

Perfektné hashovanie (perfect hashing)

- Uvažujme, že množina kľúčov je statická
  - Tabuľka symbolov prekladača
  - Množina súborov na C D
- Všetky vyhľadania budú mať konštantný čas
- Priestorová zložitosť **O(N 2)**
  - Zvoľme veľkosť tabuľky M = N 2
- Uvažujme, triedu univerzálnu hashovacích funkcií H, náhodne vyberme h ∈ H, pravdepodobnosť kolízie:
  - Počet dvojíc 2N ,pre dvojicu pravdepodobnosť kolízie je ≤ 1/M, celková pravdepodobnosť kolízie ≤ 2N /M < 1/2
- Ak pre zvolenú h ∈ H máme kolízie, vyberieme znovu :)

Perfektné hashovanie – O(n) metóda

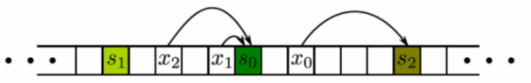
- Zlepšíme priestorovú zložitosť
- Použijeme dve úrovne hashovacích tabuliek:
- Hashovanie na prvej úrovni rozptýli prvky na triedy (môže sa ešte vyskytnúť kolízia)
- Druhá úroveň – v každom vedierku je ďalšia hashovacia tabuľka (i-te vedierko obsahuje n i prvkov), ktorá využíva O(N 2) metódu
- Platí:

$$Pr\left[\sum_i (n_i)^2 > 4N\right] < \frac{1}{2}$$

- Skúšame náhodné h, až kým nájdeme takú, že σ i (ni) 2 < 4N, a potom nájdeme sekundárme hashovacie funkcie h1, h2, ..., hN

Konzistentné hashovanie (consistent hashing)

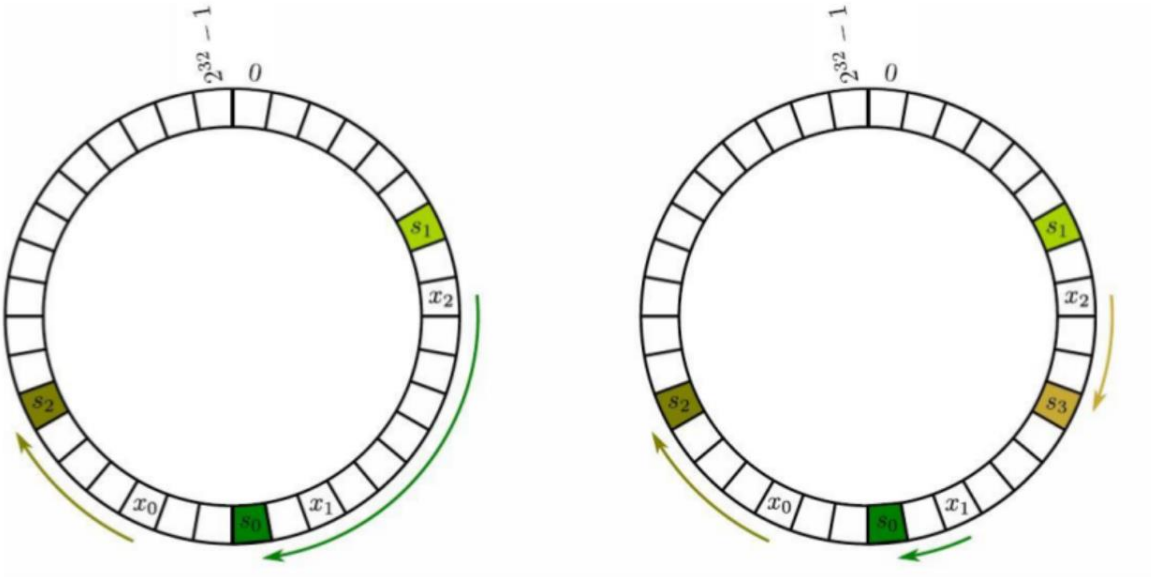
Je to také hashovanie, keď pri zmene veľkosti tabuľky zostane väčšina kľúčov na rovnakom mieste – napríklad webové cacheovanie.



Napr. prvok x_0 je v s_2, a prvky x_1 a x_2 su v s_0

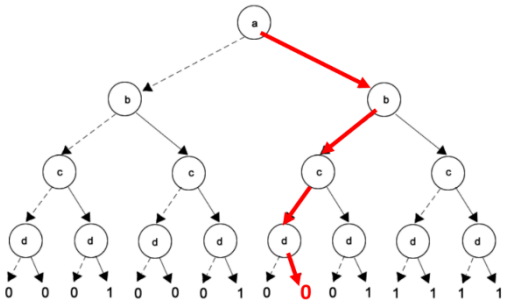
Hlavná myšlienka: okrem hashovania stránok, budeme do rovnakej tabuľky hashovať aj uzly (servery) cache. Prvok (stránku) x pridelíme tomu serveru, ktorý v tabuľke nasleduje najbližšie (doprava)

Predpokladajúc rovnomerné rozptýlenie (zaťaženie jedného uzlu = 1/n stránok), keď pridáme nový server s, presunieme len prvky uložené v s:



Prednáška 08 – binárne rozhodovacie diagramy

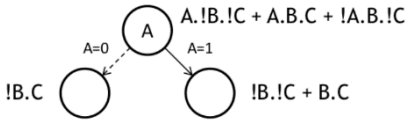
Binárny rozhodovací diagram



- Binary Decision Diagram (skratka BDD)
- Dátová štruktúra
- Má tvar ako binárny strom (len rozhodovací strom nemusí byť binárny)
- Neslúži ako ADT slovník/dynamická množina, ale slúži na rozhodovanie = rozhodovací strom
- Rozhodovanie prebieha prechodom od začiatku stromu (koreň) do konca (list)
- Každý vnútorný uzol predstavuje jedno čiastkové rozhodnutie, list = výsledné rozhodnutie
- Celkové rozhodnutie = celá cesta koreň → list
- Redukciou sa dostávame z **exponenciálnej** zložitosti až na **lineárnu**

Zostrojenie nového BDD

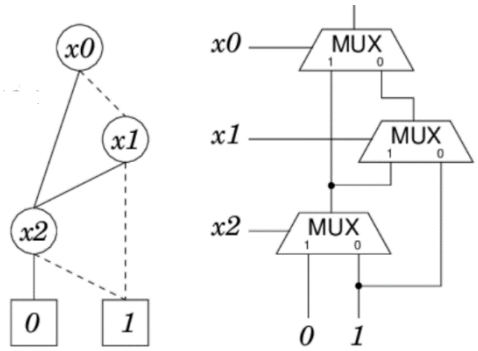
- Nový BDD sa môže zostrojiť relatívne jednoducho z ostatných spôsobov opisu Booleovskej funkcie
- Bez ohľadu na to, akú konkrétnu Booleovskú funkciu opisujeme
- Dva prístupy:
  - **Zhora nadol**
    - postupnou dekompozíciou (rozkladom) podľa jednotlivých premenných
    - každá premenná predstavuje jednu úroveň v BDD
    - napr. Shannonova dekompozícia



  - **Zdola nahor**
    - postupným skladaním výstupov
    - každá premenná predstavuje jednu úroveň v BDD

Multiplexorový strom

Čím menej uzlov má BDD, tým menej multiplexorov potrebujeme na realizáciu obvodu. Menšia plocha čipu, lacnejší, spoľahlivejší, s nižšou spotrebou energie



- Koreň BDD → výstup obvodu
- Uzol → MUX
- Hrana → vodič

Prednáška 09 – grafové algoritmy

Dijkstrov algoritmus – implementácia

- Najkratšie cesty z u = strom najkratších ciest z u
- Predpoklad: nezáporné váhy hrán w(e) ≥ 0

```
Dijkstra(G, w, u)
1 Inicializuj-počiatok(G, u)
2 S ← ∅
3 PQ ← V(G)
4 while PQ ≠ ∅
5 do x ← Extrakt-Min(PQ)
6  S ← S ∪ {x}
7       for každý vrchol y ∈ Neigh(x)
8            do Relax(x, y, w)
```

- Modifikácia – najkratšia u-v cesta v orientovanom grafe
- Predpoklad: nezáporné váhy hrán w(e) ≥ 0

```
Dijkstra(G, w, u, v)
1 Inicializuj-počiatok(G, u)
2 S ← ∅
3 PQ ← V(G)
4 while PQ ≠ ∅
5  do x ← Extrakt-Min(PQ)
6 if x = v then STOP.
7     S ← S ∪ {x}
8     for každý vrchol y ∈ Neigh(x)
9          do Relax(x, y, w)
```

PQ – min-prioritný rad

Dijkstrov algoritmus – odhad zložitosti

- Počet vrcholov N=|V|, počet hrán M=|E|
- **Q vo vektore:**
  - Extract-Min O(N), opakuje sa N krát, spolu O(N 2)
  - Každý vrchol sa vkladá do S práve raz.
  - Každá hrana sa relaxuje (v jednom smere) práve raz
  - **O(N 2 + M) = O(N 2)**

- **Q v binárnej halde:**
  - Extract-Min O(log N), opakuje sa N krát
  - Vytvorenie binárnej haldy O(N)
  - Relaxovanie sa zrealizuje pomocou operácie Decrease-Key O(log N)
  - Stále je M opakovaní
  - **O((N + M) * log N) = O(M * log N)** ak sú všetky vrcholy dosiahnuteľné z východiska

Bellman-Fordov algoritmus

Ohodnotenia hrán **môžu byť záporné**. Algoritmus zistí, ak existuje v grafe záporný cyklus dosiahnuteľný z počiatočného vrcholu, inak (ak neexistuje) nájde strom a dĺžky najkratších ciest z počiatočného vrcholu.

1) inicializácia
2) relaxácia: N-1 prechodov cez všetky hrany grafu
3) test na záporné cykly (ešte jeden pokus o relaxáciu; ak sa podarí, znamená to, že existuje záporný cyklus)

```
Bellman-Ford(G, w, s)
1 Inicializuj-počiatok(G, s)
2 for i ← 1 to N - 1
3   do for každú hranu e=(u,v) ∈ E
4         do Relax(u, v, w)
5 for každú hranu e=(u,v) ∈ E
6   do if t[v] > t[u]+w(e)
7         then return CONTAINS_NEGATIVE_CYCLE
8 return OK
```

- inicializácia (riadok 1) potrebuje O(N)
- každý z N - 1 prechodov (riadky 2–4) potrebuje O(M)
- záverečný test (riadky 5–8) potrebuje O(M).
- **celkovo O(N*M)**

Kruskalov algoritmus

```
MST-Kruskal(G, w)
1 A ← ∅
2 for každý vrchol v ∈ V
3   do Make-Set(v)
4 usporiadaj hrany v H v neklesajúcom poradí podľa váhy w
5 for každú hranu (u, v) ∈ E v poradí podľa neklesajúcej váhy
6  do if Find-Set(u) ≠ Find-Set(v)
7       then A ← A ∪ {(u, v)}
8            Union(u, v)
9  return A
```

Kritické je usporiadanie hrán O(M log M), potom M krát voláme operácie union-find. Dobrá implementácia union-find má zložitosť operácií „skoro konštantnú“. Celkovo teda **O(M log M).**

Primov (Jarníkov) algoritmus

Pri tvorbe minimálnej kostry sa udržuje rez medzi spracovanými (kostrou) a ešte nespracovanými vrcholmi

1) Inicializácia – vybrať ľubovoľný vrchol a označiť ho ako spracovaný
2) **Z rezu vybrať najlacnejšiu hranu e a vložiť ju do vytváranej minimálnej kostry.**
3) Nespracovaný vrchol hrany e označiť ako spracovaný.
4) Opakovať krok 2 pokým nie sú spracované všetky vrcholy.

```
          MST-Prim (G, w, r)
   O(N)   1 for každý vrchol u ∈ V
          2    key_u ← ∞, p_u ← NIL
   O(N)   3 PQ ← V
          4 key_r ← 0          (úprava priority v min-halde)
   O(N)   5 while PQ ≠ ∅
O(NlogN)  6    u ← Extract-Min (PQ)
  O(N+M)  7    for každý v ∈ Neigh(u)
  O(N+M)  8       if v ∈ PQ and w(u, v) < key_v
          9          p_v ← u,
O(MlogN)  10         key_v ← w(u, v)      (úprava min-haldy)
```

Q je min-halda, key_v priorita vrcholu v min-halde

Využitím vektoru s priamym prístupom O(N2):

- Riadok 6 bude O(N)
- Riadok 10 bude O(M) = O(N 2)

Prednáška 10 a 11 – triedenie

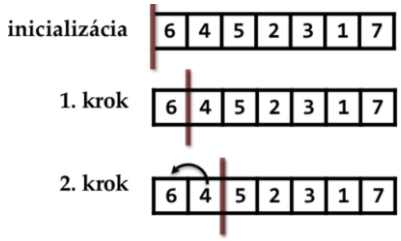
Odhady zložitosti algoritmov – opakovanie

- Analýza najhoršieho prípadu
- Použitie O-notácie pre asymptotický horný odhad
- Klasifikujeme algoritmy podľa týchto zložitostí
- Nevýhoda tohto prístupu: nemôžeme použiť na predvídanie výkonu alebo porovnanie algoritmov!
  - Quicksort – počet porovnaní v najhoršom prípade **O(N 2)**
  - Mergesort – počet porovnaní v najhoršom prípade **O(N log N)**
- V praxi je však Quicksort zvyčajne dva krát rýchlejší a používa polovičné množstvo pamäti...

Triedenie priamym vkladaním (Insert sort)

Insert sort spracúva vstupnú postupnosť postupne tak, že po jednom pridáva prvky na správne miesto do výslednej usporiadanej postupnosti (ktorá je najskôr prázdna a postupne sa rozširuje).

```
int* insert_sort(int *input, int n)
{
  int i, result[n];
  for (i = 0; i < n; i++)
    insert(input[i], result);
  return result;
}
```



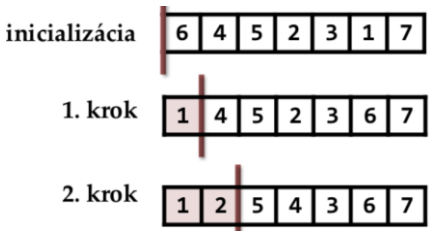
Vyžaduje rádovo L operácií, kde L je dĺžka poľa.

Analýza zložitosti (Insert sort)

- **Najlepší prípad**: prvky sú už usporiadané
  - Procedúra insert vykoná O(1) presunov
  - Celkovo – N krát insert O(1) = O(N) operácií
- **Najhorší prípad**: prvky sú usporiadané opačne
  - Volania procedúry insert vykonajú koľko presunov?
  - 0 + 1 + 2 + ... + N-1 = (N-1)*N/2
  - Celkovo O(N 2) operácií
- **Priemerný prípad**: prvky sú náhodne usporiadané
  - Volania procedúry insert vykonajú koľko presunov?
  - Asi polovicu ako pri najhoršom prípade
  - Celkovo O(N 2) operácií

Triedenie výberom (Select sort)

- Najjednoduchší-najprirodzenejší algoritmus
- **Algoritmus:**
  - Najmenší prvok môžeme zaradiť na začiatok vstupného poľa (najmenší vymeníme s prvkom, ktorý je nazačiatku)
  - Najmenší prvok zo zvyšku poľa bude druhý najmenší, atď.
- Označujeme aj MinSort / MaxSort



Usporadúvanie výmenami (Bubble sort)

Pri usporadúvaní porovnáva dva susedné prvky a ak nie sú v správnom poradí, vymenia sa. Procedúra sa opakuje, až kým nie sú prvky usporiadané (nie sú potrebné ďalšie výmeny).

Analýza zložitosti (Bubble sort)

- Jeden prechod = presun najväčšieho prvku na koniec
- i-tý prechod: n-i+1 operácií
- **Najlepší prípad**: 1 prechod **O(n)**
- **Najhorší prípad**: (n – 1) + (n – 2) + ... + 1 = (n – 1) * n / 2 = **O(n2)**
- Implementačne jednoduchý ale výpočtovo neefektívny

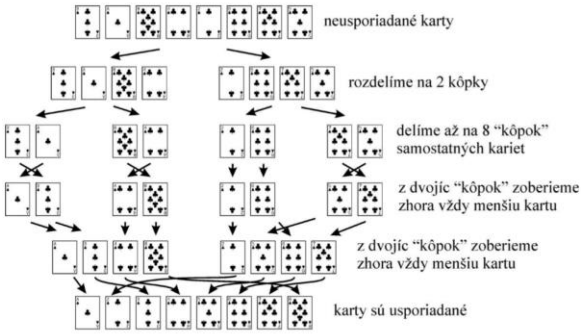
Triedenie zlučovaním (Merge sort)

Merge sort vstupnú postupnosť rozdelí na dve polovice, každú rekurzívne utriedi, no a výslednú usporiadanú postupnosť všetkých prvkov určí zlúčením týchto menších usporiadaných postupností.

Procedúra merge(input, left, middle, right) pomocou cyklu spojí usporiadané postupnosti prvkov input[left, ..., middle] a input[middle+1, ..., right] do jednej usporiadanej postupnosti.

```
int* merge_sort(int *input, int left, int right)
{
  int mid = (left+right)/2;
  merge_sort(input,left,mid);
  merge_sort(input,mid+1,right);
  return merge(input,left,mid,right);
}
```

Vyžaduje rádovo right-left (dĺžka poľa vstupujúceho do operácie) operácií.



Shellsort

- Usporadúvanie vkladaním so zmenšovaním prírastku
- Zovšeobecnenie triedenia vkladaním (Insert sort) a bublinkového (Bubble sort)
- Dobrá implementácia je jedna z najrýchlejších pre usporiadanie kratších postupností (do 1000 prvkov)
- Netriedi naraz celú postupnosť, ale pre prírastok h utriedi Insert sort-om vybranú podpostupnosť prvkov vzdialených h (pre všetky možné začiatky i):

```
for(h = n/2; h > 0; h = h/2) // zmensujuce sa prirastky
  for (i = 0; i < h; i++)
    insert_sort(a[i,i+h,i+2*h,...]);
```

- Postupnosť zmenšujúcich sa prírastkov, posledný h = 1

1. krok, prírastok 4 (n/2),
(vyznačené čísla sa usporiadajú vkladaním)

6 4 5 2 **8** 3 1 7 –> **6** 4 5 2 **8** 3 1 7
6 **4** 5 2 8 **3** 1 7 –> 6 **3** 5 2 8 **4** 1 7
6 3 **5** 2 8 4 **1** 7 –> 6 3 **1** 2 8 4 **5** 7
6 3 1 **2** 8 4 5 **7** –> 6 3 1 **2** 8 4 5 **7**

2. krok, prírastok 2

6 3 **1** 2 **8** 4 **5** 7 –> **1** 3 **5** 2 **6** 4 **8** 7
1 **3** 5 **2** 6 **4** 8 **7** –> 1 **2** 5 **3** 6 **4** 8 **7**

3. krok, prírastok 1

1 2 5 3 6 4 8 7 –> 1 2 3 4 5 6 7 8

Rýchle usporiadanie (Quicksort)

- Quicksort alebo usporadúvanie rozdeľovaním je jeden z najrýchlejších známych algoritmov založených na porovnávaní prvkov
- Priemerná doba výpočtu Quicksort-u je najlepšia zo všetkých podobných algoritmov
- Nevýhodou je, že pri nevhodnom usporiadaní vstupných dát môže byť časová aj pamäťová náročnosť omnoho väčšia
  - Quicksort – počet porovnaní v najhoršom prípade O(N 2)
  - Mergesort – počet porovnaní v najhoršom prípade O(N log N)
  - V praxi je však Quicksort zvyčajne dva krát rýchlejší a používa polovičné množstvo pamäti...

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[|(hi + lo) / 2|]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot
        do
            j := j - 1
        while A[j] > pivot
        if i ≥ j then
            return j
        swap A[i] with A[j]
```

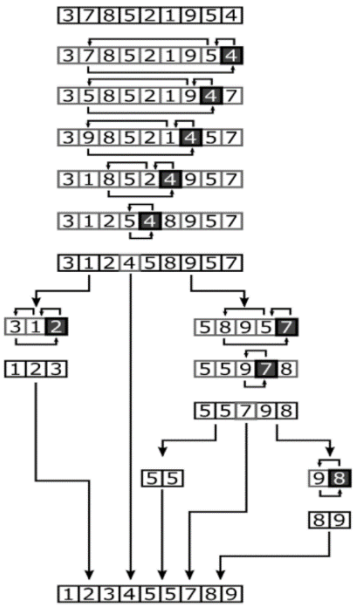
Quicksort – hlavná myšlienka

- Jeden prechod = rozčlenenie prvkov na dve podpostupnosti podľa pivota x:
  - prvky < x, prvky > x
- Rekurzívne usporiadať podpostupnosti
- Lomuto schéma

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

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5            then i ← i + 1
6                 exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

Analýza rýchleho usporiadania



- **Najhorší prípad**: vždy zle vyvážené rozčlenenie
  - Usporiadaná postupnosť :(
  - T(1) = Θ(1)
  - T(n) = T(n - 1) + Θ (n)
  - T(n) = Θ (n2)
- **Najlepší prípad**: vždy dokonale vyvážené rozčlenenie
  - T(n) = 2T(n/2) + Θ (n)
  - T(n) = Θ (n lg n)
- **Priemerný prípad**: náhodný
  - Napr. aj pre rozčlenenie 9 ku 1
  - T(n) = T(9n/10) + T(n/10) + n
  - T(n) = Θ (n lg n)

Výber pivota

- Krajný prvok – konštantný čas O(1) nemusí byť vhodný
- Ktorý by bol najlepší?
  - Taký, pre ktorý je počet prvkov rozčlenených podpostupností rovnaký (alebo čo najbližšie k sebe)
  - Medián
- Dobrý algoritmus: vybrať náhodný pivot

Nájdenie k-teho najmenšieho prvku

Daná postupnosť A[1..n] (neusporiadaných) čísel a celé číslo k (1 ⍰ k ⍰ n). Úloha je nájsť k-te najmenšie číslo v A.

Špeciálne prípady

- pre k=1 ide o nájdenie najmenšieho prvku,
- pre k=n ide o nájdenie najväčšieho prvku,
- ak n je nepárne, k=(n+1)/2 dá medián
- ak n je párne, podľa dohody je medián niečo medzi prípadmi k=floor((n+1)/2) a k=ceiling((n+1)/2)

Prvý nápad na riešenie:

- Usporiadať pole A a vybrať A[k]
  - Usporiadať vieme na mieste O(n log n).Výber A[k] je O(1). Spolu **O(n log n).**

Dá sa to rýchlejšie?

- Ak máme už dané k, tak usporiadaním sa urobilo viac práce ako je potrebné na určenie k-teho najmenšieho prvku. Prečo?
- Ak by k nebolo vopred dané, tak by práve usporiadané pole dávalo k-ty najmenší prvok pre ľubovoľné k.

Druhý nápad:

- Nájsť najmenší prvok v poli A a odstrániť ho. Pokračovať nájdením najmenšieho prvku vo zvyšku poľa A, odstránením atď. Opakovať k-krát.
- Nájsť minimum a odstrániť ho vieme O(n). **Spolu O(k*n).**

Je to rýchlejšie?

- závisí od porovnania k a log(n).
- pre veľké k (väčšie ako log n) je lepší prvý nápad
- pre malé k je druhý nápad lepší

Dá sa to rýchlejšie?

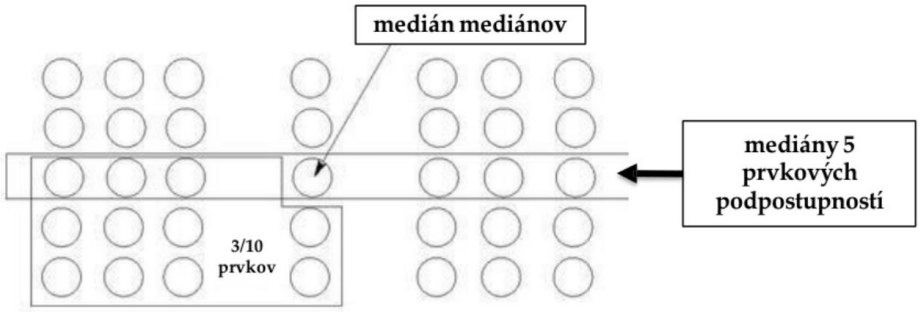
- Všimnime si, že v prvom aj druhom prípade dostaneme na výstupe k najmenších prvkov usporiadaných.
- Ale toto (usporiadanie) týchto k prvkov nepotrebujeme! -> Robíme robotu navyše.
- Výzva je určiť LEN k-ty prvok a urobiť to v čase **O(n).**

Blum, Floyd, Pratt, Rivest a Tarjan, 1973 Algoritmus s mediánom mediánov ako pivotom:

```
Select (A, k)
1.  x = median(A) //akurát, že zatiaľ nevieme ako v O(n)
2.  rozčleň A podľa pivota x. Nech je m-1 prvkov takých, že A[i]<x.
    Potom bude A[m]=x a n-m prvkov bude takých, že A[i]>x.
3.  if k=m then return x
                else if k<m then Select (A[1..m-1], k)
                        else Select (A[m+1..n], k-m)
```

Medián mediánom z piatich

1) Rozdeľ vstupnú postupnosť n prvkov do skupín po piatich (a možno jednej zvyškovej)
2) Nájdi medián každej skupiny (usporiadaním alebo natvrdo tretí najmenší) – dostaneš n/5 mediánov
3) Rekurzívne Select(„n/5 mediánov“, n/10)



Zložitosť: $T_{select}(n) = T_{select}(n/5) + T_{select}(7n/10) + n$ – *Celkovo O(n)*

Triedenie binárnym vyhľadávacím stromom

- Štruktúra vrcholov v BVS umožňuje skonštruovať jednoduchý algoritmus usporadúvania tzv. Treesort
- In-order prehľadávanie – usporiadaný výpis obsahu BVS
    - Zložitosť **O(n),** kde n je počet prvkov v strome
- Máme teda porovnávací algoritmus, ktorý dokáže usporiadať n čísel rýchlejšie ako O(n log n)?
    - Nie, pretože vytvorenie BVS trvá O(n log n)
    - Všimnime si, že štruktúra prvkov v BVS je „ekvivalentná“ samotnému problému usporiadania, pretože keď už máme BVS, tak dokážeme výsledné poradie získať v O(n)

Tree sort a Heap sort

- Heapsort používa binárnu haldu ako dátovú štruktúru
- Naplníme binárnu haldu všetkými údajmi, ktoré chceme zoradiť
- Potom postupne voláme Extract-Min (alebo Max) až kým nevyberieme všetky prvky
- Po každej extrakcii použijeme Select sort
- Tree sort používa binárny strom ako dátovú štruktúru
- Naplníme binárny strom všetkými údajmi, ktoré chceme zoradiť
- Potom použijeme in-order prehľadávanie

Halda (heap)

- Využíva špeciálny typ binárneho stromu
    - Tzv. úplný binárny strom (complete binary tree)
    - Na každej úrovni je úplne naplnený, okrem možno poslednej úrovne (rozdiel oproti plnému binárnemu stromu)
- Halda poskytuje len obmedzené operácie
    - **insert** (pridať prvok),
    - **delete** (odstrániť prvok)
    - **getmax** (vyhľadať najväčší prvok)
- Implementácia pomocou **BVS:**
    - insert /delete **O(n),**
    - getmax **O(1)**
- Cieľ je vyváženejšia zložitosť:
    - insert /delete /getmax O(log n)

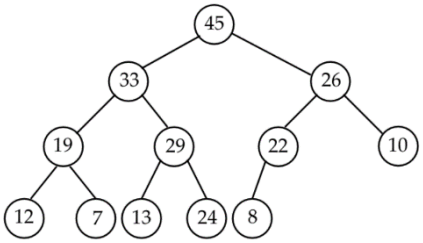
ADT Prioritný rad / front (Priority queue)

- Množina prvkov, ktorým je pridelená priorita (kľúč) – je ich možné podľa priority porovnávať.
- Prvky je možné vkladať v akomkoľvek poradí s rôznou prioritou avšak, pri výbere sa vyberá vždy len prvok s najvyššou prioritou.
- Operácie:
    - **insert**(S,x) – vloženie prvku x do množiny S (pom. spájaného zoznamu O(1))
    - **maximum**(S) – vrátenie prvku s najväčším kľúčom (pom. spájaného zoznamu O(n))
    - **removeMax**(S) – odstránenie prvku s najväčším kľúčom (O(n))

Implementácia pomocou binárnej haldy

Binárna halda je úplný binárny strom, pre ktorý platí, že hodnota kľúča vo vrchole je väčšia alebo rovná hodnotám kľúčov jeho nasledovníkov. **Vlastnosti**:

- Binárna halda má voľnejšie pravidlá usporiadania kľúčov (umiestnenie prvkov) ako binárny vyhľadávací strom
- Nemusí platiť, že ľavý podstrom obsahuje prvky s nižšími hodnotami kľúčov ako pravý podstrom!
- Platí tzv **haldová vlastnosť:**
  - kľúč(PARENT(i)) ≥ kľúč(i) pre všetky vrcholy i okrem koreňa



Dôsledok: koreň stromu (binárnej haldy) má vždy najväčšiu hodnotu kľúča (≥ ako ostatné vrcholy).

Binárna halda – Implementácia vektorom

- Koreň stromu na 1. pozícii heap[1]
- Nasledovníky vrchola zapísaného na i-tej pozícii vektora sú (ak existujú):
  - left(i) = 2*i
  - right(i) = 2*i + 1
  - parent(i) = ⌊i/2⌋
- heap[i..j], kde i>=1, je binárna halda práve vtedy, ak každý prvok nie je menší ako jeho nasledovníky.
- Operácia maximum – prvok heap[1] = **O(1)**

Binárna halda – Operácia insert

1) Vytvorí sa nový vrchol na najnižšej úrovni, označme v
2) Ak je v koreň stromu (haldy), končíme.
3) Ak kľúč(v) ≤ kľúč(parent(v)), končíme.
4) Inak (ak kľúč(v) > kľúč(parent(v))), vymeníme vrchol v so svojím rodičom, a pokračujeme na krok 2 pre v ← parent(v)

5) (Ak je kľúč vrchola v väčší ako kľúč nového rodiča, vymení sa aj s ním, ... opakujeme, kým nie je strom opäť haldou – spĺňa haldovú vlastnosť)

Zložitosť je O(log n), kde n je počet prvkov v halde, pretože: úplný binárny strom s n prvkami má hĺbku O(log n).

```
Heap-INSERT(heap, key):
  heap-size (heap) = heap-size(heap) +
  1   i = heap-size (heap)
  while i > 1 and heap[PARENT(i)] <
     key  do heap[i] = heap[PARENT(i)]
          i = PARENT(i)
  heap[i] = key
```

Odstránenie najväčšieho prvku z binárnej haldy

- Odstránime koreň haldy
- Odstránime najpravejší vrchol na najnižšej úrovni (jeho kľúč označme P) a hodnotu P zapíšeme do koreňa
- Mohli sme porušiť haldovú vlastnosť!
- Obnovíme haldovú vlastnosť smerom dole
    - Ak P je list, končíme.
    - Označme Q najväčšiu z hodnôt priamych potomkov P
- Ak Q ≤ Pteda v potomkoch nie sú väčšie kľúče, končíme.
- Inak (ak Q > P) vymeníme vrchol P s vrcholom Q, pokračujeme na krok 1 pre nižšie umiestnený vrchol P

```
Heap-EXTRACT-MAX(heap)
  if heap-size(heap) < 1
     then error
  max = heap[1]
  heap[1] = heap[heap-size(heap)]
  heap-size(heap) = heap-size(heap)-1
  HEAPIFY(heap, 1)
  return max
```

Binárna halda – heapify (pseudokód)

- Zložitosť **O(log n),** kde n je počet prvkov v halde

```
HEAPIFY(heap, i)
  lavy = left(i)
  pravy = right(i)
  if lavy <= heap-size(heap) and heap[lavy] > heap[i]
      then largest = lavy
      else largest = i
  if pravy <= heap-size(heap) and heap[pravy] > heap[largest]
      then largest = pravy
  if largest <> i
      then exchange (heap[i], heap[largest])
            HEAPIFY(heap, largest)
```

Binárna halda – vytvorenie haldy

- Z vektora heap[1..n], kde n=length(heap)
- Všetky prvky v podvektore heap[(⌊n/2⌋+1..n] sú listy a teda aj 1-prvkové haldy
- Zložitosť - vo výške h je najviac n / 2^{h+1} vrcholov, heapify haldy výšky h trvá **O(h)**

```
BUILD-HEAP(heap):
  heap-size(heap) = length(heap)
  for i = ⌊length[heap] / 2⌋ downto 1
     do HEAPIFY(heap, i)
```

Usporadúvanie haldou (Heapsort)

- Pomocou haldy dokážeme spraviť efektívny triediaci algoritmus, tzv Heapsort
- Postup: Vytvorím haldu a postupne z nej vyberiem všetky prvky
- Zložitosť – vytvorenie (**O(n)**) a n krát vybratie max (n*O(log n)) = **O(n log n)**

```
HEAP-SORT(A):
    BUILD-HEAP(A);
    for i = length(A) downto 2 do
    { A[1] ↔ A[i];
      heap-size(A) = heap-size(A)-1;
      HEAPIFY(A, 1)
    }
```

Porovnávacie algoritmy

- Algoritmus usporadúvania, ktorý prechádza vstupné kľúče a na základe operácie porovnávania rozhoduje, ktorý z dvoch prvkov sa má v usporiadanom poli objaviť ako prvý
- Operácia porovnávania musí mať tieto **vlastnosti**:
    - Ak a ≤ b a b ≤ c, tak a ≤ c
    - Pre všetky a a b, buď a ≤ b alebo b ≤ a
- Základným limitom je dolné ohraničenie počtu potrebných porovnávaní **Ω(n log n)**, ktoré je v najhoršom prípade potrebné na usporiadanie postupnosti

Výhody porovnávacích algoritmov

- Výhody porovnávacách algoritmov
    - Použiteľné as-is pre rôzne dátové typy
    - Čísla, reťace, ...
    - Jednoduchá implementácia porovnávania n-tíc v lexikografickom usporiadaní
    - Reverzná funkcia porovnávania = reverzne usporiadaná postupnosť

- Ako prekonať teoretický limit Ω(n log n) porovnávacích algoritmov?
    - Zbaviť sa porovnávania prvkov :) (budeme vyšetrovať štruktúru hodnôt kľúčov)
    - Obetovať priestorovú zložitosť

Usporadúvanie spočítavaním (Counting sort)

- Usporadúvanie výpočtom poradia
  - Neporovnávame kľúče!
  - Pokúsime sa priamo určiť jeho poradie v postupnosti
- **Vstup**: n čísel v rozsahu 0..k-1
- Určuje počet prvkov menších ako prvok x, pomocou čoho zistí správnu pozíciu prvku x vo vstupnom poli
- Určuje počet prvkov menších ako prvok x, pomocou čoho zistí správnu pozíciu prvku x vo vstupnom poli
- Algoritmus pracuje s tromi poliami:
  - Pole a[0..n-1] obsahuje údaje, ktoré sa majú usporiadať
  - Pole b[0..n-1] obsahuje konečný usporiadaný zoznam údajov
  - Pole c[0..k-1] je použité na počítanie počtu prvkov

```
// pocet vyskytov konkretnej hodnoty
for(i = 0; i < n; i++)
  c[a[i]]++;

// prefixove sucty: urcime index posledneho prvku s hodnotou j
for(j = 1; j < k; j++)
  c[j] = c[j] + c[j-1];

// prvky z pola a vložíme na prislusny index v poli b
for(i = n-1; i >= 0; i--)
  b[--c[a[i]]] = a[i];
```

- Koľko operácií algoritmus vykoná?
  - rádovo n+k+n
- Koľko pomocnej pamäte potrebuje?
  - pole veľkosti n a pole veľkosti k
- Vhodný len pre malé k << n
- Pre veľký rozsah (int) je potrebné veľa pomocnej pamäte

Stabilný algoritmus

- Algoritmus usporadúvania je stabilný, ak vždy zachová pôvodné poradie prvkov s rovnakými kľúčmi
- Ak prvky s rovnakými kľúčmi sú neodlíšiteľné, tak nie je potrebné sa zaoberať stabilitou algoritmu (napr ak kľúčom je samotný prvok)
- Zachovať pôvodné poradie prvkov 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 pôvodné poradie prvkov a pri zhodných kľúčoch sa berie do úvahy toto poradie
- Viacnásobné usporiadanie je možné obísť vytvorením jedného kľúča usporiadania, ktorý je zložený z primárneho, sekundárneho, atď.
- Takéto úpravy nestabilných algoritmov majú negatívny vplyv na výpočtovú zložitosť

- **Príklad** – dvojice (kľúč,prvok):
  (4, 5) (2, 7) (2, 3) (5, 6)
- Dve možné usporiadania:
  (2, 7) (2, 3) (4, 5) (5, 6) – zachované poradie prvkov s kľúčmi 2 – stabilné usporiadanie
  (2, 3) (2, 7) (4, 5) (5, 6) – zmenené poradie prvkov s kľúčmi 2 – nestabilné usporiadanie

- Príklad na viacnásobné usporiadanie – dvojice (kľúč 1, kľúč 2):
  (4, 5) (2, 7) (2, 3) (4, 6)
- Usporiadanie najprv podľa kľúča 2, potom podľa kľúča 1:
  (2, 3) (4, 5) (4, 6) (2, 7) – podľa kľúča 2
  (2, 3) (2, 7) (4, 5) (4, 6) – podľa kľúča 1
- Usporiadanie najprv podľa kľúča 1, potom podľa kľúča 2:
  (2, 7) (2, 3) (4, 5) (4, 6) – podľa kľúča 1
  (2, 3) (4, 5) (4, 6) (2, 7) – podľa kľúča 2 – narušené poradie

- Pre zachovanie stability viacnásobného usporadúvania je potrebné usporadúvať postupne podľa kľúčov so zvyšujúcou sa prioritou

Radixové usporadúvanie

- Spracovanie sčítania ľudu USA 1880 trvalo skoro 10 rokov (robí sa každých 10 rokov)
- Herman Hollerith (1860-1929)
- Ako prednášateľ na MIT navrhol prototyp strojov na spracovanie diernych štítkov, doba spracovania ďalšieho sčítania ľudu v 1890 sa tým skrátila na 6 týždňov
- **Základná myšlienka:**
  - začni triediť podľa najnižšieho rádu

```
RADIX-SORT(A, d)
    for i ← 1 to d
        do stabilné usporadúvanie(A) podľa i-tej číslice
```

- **Radixové usporadúvanie** neporovnáva dva celé kľúče, ale spracúva a porovnáva len časti kľúčov
- Kľúče považuje za čísla zapísané v číselnej sústave so základom k (radix, koreň), pracuje s jednotlivými číslicami:

$$\text{hodnota} = x_{d-1}k^{d-1} + x_{d-2}k^{d-2} + \ldots + x_2k^2 + x_1k^1 + x_0k^0$$

- Dokáže usporadúvať čísla, znakové reťazce, dáta, ... (počítače reprezentujú všetky údaje ako postupnosti 1 a 0 – binárna sústava => 2 je základ)
  - Uvažujme problém: usporiadať milión 64-bitových čísiel
  - Prvé riešenie: 64 prechodov cez milión čísiel?
  - Lepšie riešenie: interpretovať ich ako čísla v sústave so základom (radixom) 216, budú to najviac 4-miestne čísla ...vtedy to algoritmus usporiada len v 4 prechodoch!
- Usporiada množinu čísiel vo viacerých prechodoch, začínajúc od číslic najnižšieho (jednotkového) rádu, potom usporiada podľa číslic najbližšieho vyššieho (desiatkového) rádu atď.

```
ace        now        now        tag
bet        sky        hut        tip
```

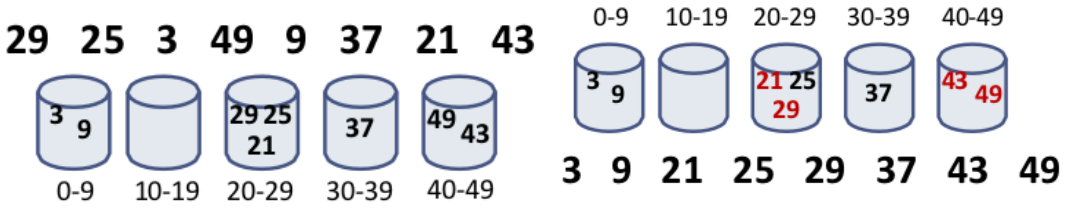
- **LSD Radix sort** (least significant digit) – usporadúvanie podľa číslic postupuje od poslednej číslice (s najmenšou váhou) k prvej číslici (s najväčšou váhou)– stabilný.
- **MSD Radix sort** – od prvej číslice k poslednej – lexikografické usporiadanie – nestabilný
- Je dôležité na samotné usporadúvanie podľa jednotlivých číslic použiť nejaký stabilný algoritmus, aby sa nemenilo poradie prvkov s rovnakými číslicami jednej váhy pri usporadúvaní podľa inej váhy
- Keďže počet možných číslic (ak k=10) je len 10, tak na usporiadanie podľa nich je výhodné použiť usporadúvanie spočítavaním.

Vedierkové usporadúvanie (Bucket sort)

- Predpokladá, že vstup je akoby generovaný náhodným procesom, ktorý prvky distribuuje rovnomerne na celom intervale
- Rozdelí interval na n rovnako veľkých disjunktných podintervalov (vedierok) a potom do nich rozmiestni vstupné čísla
- Osobitne v každom vedierku sa potom tieto čísla usporiadajú

Vedierkové usporadúvanie – príklad

- Vytvoria sa prázdne vedierka veľkosti M/n (M – maximálna hodnota vstupného poľa, n – počet prvkov vstupného poľa)
- Rozptýlenie – prechádzanie vstupným poľom a rozmiestnenie každého prvku do prislúchajúceho vedierka



- Usporiadanie naplnených vedierok
- Zreťazenie vedierok – postupné prechádzanie usporiadaných vedierok a presúvanie prvkov späť do vstupného poľa

Analýza zložitosti (Bucket sort)

- Jednotlivé vedierka väčšinou predstavujú spájaný zoznam, do ktorého sa na správne miesto presúvajú prvky zo vstupného poľa (insert sort)
- Činnosti ako vytvorenie vedierok, určenie prislúchajúceho vedierka, presunutie prvku do vedierka a zreťazenie vedierok do výslednej postupnosti trvajú **O(n)**
- Výpočtová zložitosť usporiadania prvkov vo vedierkach Insert sortom **$O(n^2)$**
- Výsledná časová zložitosť závisí od rozloženia prvkov vo vedierkach. Ak sú prvky rozmiestnené nerovnomerne a v niektorých vedierkach ich je veľmi veľa, **tak časová zložitosť Insert sortu $O(n^2)$ prevažuje nad lineárnou zložitosťou** a predstavuje výslednú zložitosť celého usporadúvania. Takýto stav sa môže vyskytnúť ak rozsah prvkov m je oveľa väčší ako ich počet
- Preto sa niekedy celková zložitosť značí podobne ako pri Counting sorte **O(n+m).** Ak m=O(n), tak výsledná časová zložitosť je O(n). Ak sa počet vedierok rovná počtu vstupných prvkov, tak v priemere to vychádza na jeden prvok v každom vedierku, a **preto sa za priemernú zložitosť berie O(n)**

Prednáška 12 – výpočtovo intenzívne úlohy a HW akcelerácia

- Algoritmy s vysokou výpočtovou **zložitosťou**
  - $O(N)$, $O(N.\log(N))$, $O(N^2)$, $O(N^3)$, $O(2^N)$, …
- Operácie, ktoré sa používajú **často**
- Možnosti riešenia príliš vysokej záťaže:
  - Použiť algoritmus s nižšou zložitosťou (ak existuje)
  - Optimalizovať implementáciu (zvyčajne nevýrazné)
  - Znížiť frekvenciu použitia alebo množstvo dát
  - Aplikovať algoritmy AI, evolučné algoritmy, štatistické modely, negarantovať správny výsledok na 100% alebo vedome znížiť kvalitu výsledku
  - Použiť výkonnejší HW
  - Paralelizmus, HW akcelerácia

Paralelizmus

- Existujú rôzne formy:
  - Viac-vláknovosť (multithreading)
  - Viac-úlohovosť (multiprocessing)
  - GPU
  - HW akcelerácia
- Namiesto toho, aby sme program vykonali inštrukciu za inštrukciou (sekvenčne), vykonáme ho (alebo nejakú jeho časť) paralelne
- Väčší problém rozdelíme na menšie (ideálne rovnako veľké) časti a každé vlákno/proces/jadro GPU vyrieši jednu časť

HW akcelerácia

- Nahradenie softvérovej implementácie vybranej funkcionality (algoritmu) implementáciou hardvérovou
  - Môže byť úplné alebo čiastočné (HW/SW Co-Design)
- SW sa vykonáva sekvenčne, krok za krokom, alebo inštrukcia za inštrukciou
- HW dokáže využívať paralelizmus v najväčšej možnej miere
- Možnosti realizácie:
  - **ASIC** (Application Specific Integrated Circuit) – vlastný čip vyrobený na mieru podľa návrhu
  - **FPGA** (Field Programmable Gate Array) – existujúci čip, ktorý sa konfiguruje podľa návrhu

Výhody a nevýhody HW akcelerácie na čipe

Výhody:

- Najvyššia priepustnosť
- Najnižšia latencia
- Najnižšia spotreba energie
- Možnosť dosiahnutia lepších výsledkov/rozhodnutí vďaka realizácii komplexnejších a robustnejších algoritmov, ktoré môžu vo forme SW byť príliš pomalé

Nevýhody:

- Vývoj je náročnejší, zdĺhavejší a drahší
- Nižšia flexibilita na zmeny požiadaviek (len ASIC)
- HW navyše (niečo stojí)

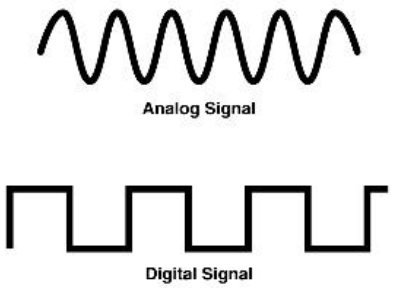
HW/SW Co-Design

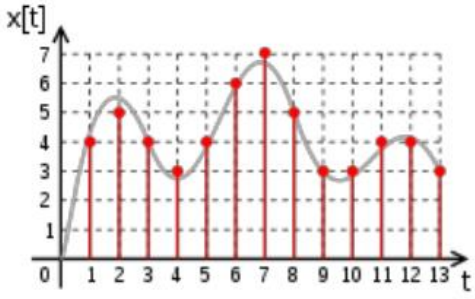
Možnosti kombinácie HW a SW:

- **ASIC** pozostávajúci z CPU a koprocesora (HW akcelerátor) – najdrahšie, ale najefektívnejšie riešenie
- **FPGA** pozostávajúci zo soft-core CPU IP jadra a koprocesora – najlacnejšie riešenie, avšak CPU realizované v FPGA nedosahuje štandardný výkon (100-400 MHz)
- **FPGA SoC** – hard-core CPU (zvyčajne ARM) a FPGA na jednom čipe – kompromis ASIC a FPGA
- **HW** akcelerátor na samostatnom čipe (ASIC/FPGA) a existujúci CPU čip, spolu na jednej doske plošných spojov – pomalšia komunikácia, vysoká latencia

Integrované obvody

- Integrovaný obvod = mikročip
- Mikročipy z hľadiska signálu môžu byť:
    - Digitálne (digital design) – jednotky a nuly, čísla
    - Analógové (analog design) – napätie, prúd
    - Zmiešané (mixed-signal design) – kombinácia oboch svetov na jednom čipe





Číslicový návrh a formy jeho opisu

- Pri návrhu sa často používajú diagramy, podobne ako pri návrhu softvéru, kde sa používajú najmä UML diagramy
    - **Blokové diagramy** – pre dekompozíciu časti dizajnu (modulu) na komponenty (submoduly)
    - **Stavové diagramy** – pre opis stavových automatov
    - **Diagramy aktivít** – pre vyjadrenie rozhodovacej logiky
    - **Časové diagramy** – pre opis komunikačných protokolov
- Okrem diagramov sa opisuje číslicový systém v podobe RTL kódu (podobne ako pri programovaní, len opisujeme HW, nie SW):
- VHDL, Verilog, SystemVerilog, Chisel, SpinalHDL, ...

Zoraďovanie údajov

- Vstup: množina viac-bitových prvkov (items)
    - Pridávané prvkov paralelne (naraz)
    - Alebo sekvenčne (za sebou)
- Prvok (item) sa skladá z 2 častí:
    - Kľúč (data)
    - Payload (napr. ID)
- Výstup:
    - Zoradená množina prvkov
    - Alebo MIN/MAX prvok
- Realizovateľné softvérovo alebo hardvérovo

Dôležité parametre pri HW zoraďovaní

- Maximálny počet prvkov, ktoré je možné zoradiť
- Veľkosť prvkov = počet bitov na 1 prvok
- Smer zoradenia, t.j. MIN alebo MAX
- Či je vstup 1 prvok (sekvenčne) alebo všetky prvky (paralelne)
- Či je výstup len MIN/MAX prvok (sekvenčne) alebo kompletná množina prvkov (paralelne)
- Či je možné vybrať (odstrániť) len MIN/MAX prvok alebo ľubovoľný prvok podľa ID

Dôležité vlastnosti

Výkon, ktorý pozostáva z:

- Max. pracovná frekvencia
- Priepustnosť
- Latencia

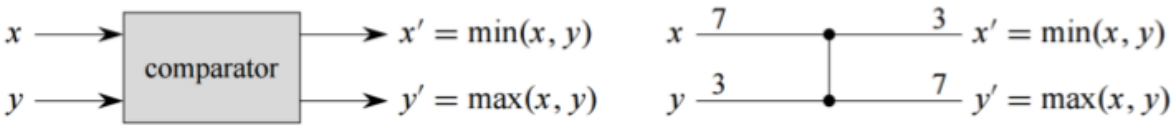
Množstvo potrebného hardvéru (plocha čipu)

- Cena (výrobné náklady)
- Veľkosť (fyzické rozmery)
- Spoľahlivosť (odolnosť voči poruchám)

Spotreba energie

- Absolútna / relatívna (na výkon)
- Statická / dynamická

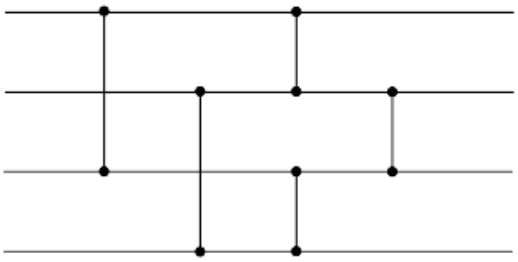
Sorting node

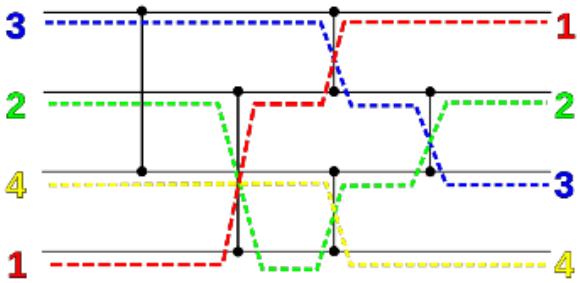


Sortovacia sieť (Sorting net)

- 1954 – Armstrong, Nelson and O'Connor
- Sieť sortovacích uzlov (sorting nodes)
- Vhodné na realizáciu paralelného sortovania t.j. keď je vstup paralelný – keď sa môžu meniť všetky vstupy v rovnakom čase

Existujú rozličné topológie





Optimálna sortovacia sieť

- Minimálny počet sortovacích uzlov
- Závisí od počtu vstupov

Sekvenčný vstup

- Keď sa mení max. 1 prvok (vstup) v čase
  - inštrukcia INSERT (vloží nový prvok) alebo REMOVE (vymaže prvok podľa ID)
- Existujú viaceré architektúry
- Každá ma svoje výhody a nevýhody

Najpopulámejšie architektúry:

- **FIFO Array**
  - Pre každú možnú hodnotu existuje 1 FIFO buffer
  - Hĺbka (kapacita) každého FIFO buffra je rovná maximálnemu množstvu prvkov, ktoré chceme zoradiť
  - Pridávanie prvkov:
    - Nový prvok sa vždy vloží do FIFO buffra prislúchajúceho hodnote (data) prvku
  - Výstup:
    - Použije sa výstup prvého neprázdneho FIFO buffra
    - FIFO buffre sú fixne usporiadané
  - Vhodné pre malý počet možných hodnôt

- **Sorting Tree**
  - V podstate sortovacia sieť, ktorá má tvar binárneho stromu
  - Každý vstup je prvok pre jedno konkrétne ID, ktorý je uložený v D-FF
  - Vhodné pre väčší rozsah hodnôt, ale malý počet prvkov

- **Pipelined Sorting Tree**

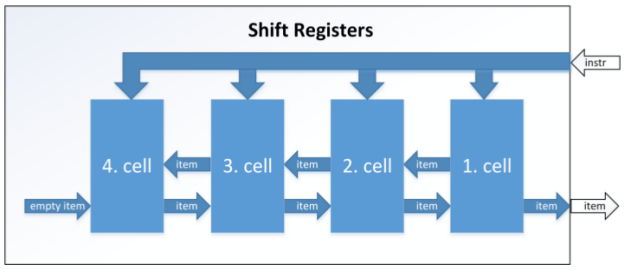
  To isté ako Sorting Tree, ale každý sortovací uzol je registrovaný

  Vyriešený problém dĺžky kritickej cesty (funguje aj pre vyššie pracovné frekvencie)

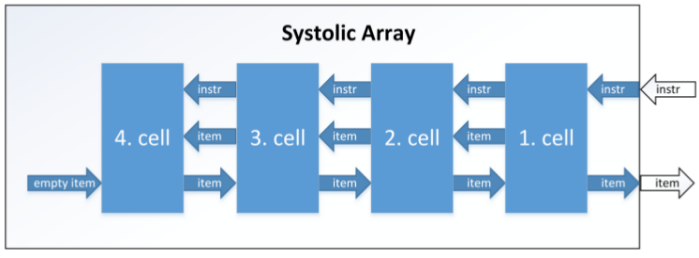
  Vyššia spotreba registrov (väčšia plocha aj spotreba energie)

  Spracovanie vstupu už netrvá 1 takt, ale $log_2(N)$

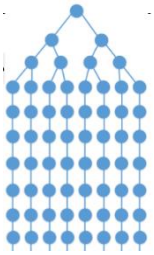
- **Shift Registers**
  - Bunky so spoločnou zbernicou na vstupe
  - Výmena prvkov medzi susednými bunkami tak, aby boli prvky zoradené



- **Systolic Array**
  - Ako Shift Registers, len nemá spoločnú zbernicu
  - Inštrukcia s novým prvkom (alebo mazanie) sa posúva postupne, o 1 bunku ďalej v každom takte



- **Rocket Queue**
  - Vstup aj výstup zhora
  - 2 typy buniek: duplikujúce a zlúčené
  - Organizované do levelov
  - Bunky rovnakého levelu zdieľajú komparátor
  - Inštrukcia prechádza zhora nadol - 1 level za 1 takt
  - Pridávanie prvkov je vyvažované



- **Heap Queue**

  Modifikácia Rocket Queue

  Používajú sa iba duplikujúce bunky

  Prvky v bunkách sú uložené do SRAM pamäte

  Možnosť odstrániť len MIN/MAX prvok (na samotnom vrchole Heap Queue)

  Ideálna architektúra pre zoraďovanie veľkého množstva dát (t.j. veľká kapacita) a pre veľký rozsah dát

  Kapacita môže byť len $2^N - 1$

Celkové porovnanie

	ID based Remove	Stable Sorting	Linear	Memory Type	Stable Clock Freq.	Latency (clock cycles)
Heap Queue	No	No	No	RAM	Yes	2
Rocket Queue	Yes	No	No	Reg	Yes	2
Systolic Array	Yes	Yes	Yes	Reg	Yes	2
Shift Registers	Yes	Yes	Yes	Reg	No	2
Sorting Tree	Yes	No	No	Reg	No	2
Pipelined Sorting Tree	Yes	No	No	Reg	Yes	$log_2(N)+1$
Software Min Max Heap	No	No	No	-	Yes	$A.\ log_2(N)+B$
Software Select Sort	Yes	Yes	Yes	-	Yes	$C.\ N+D$