Pokročilá algoritmizace

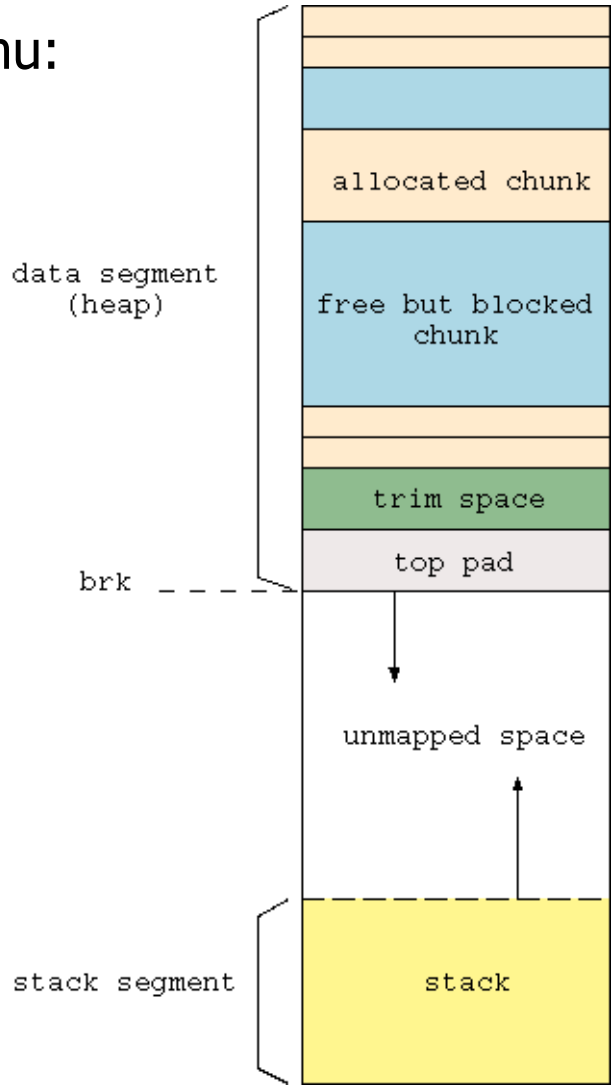
dynamické datové struktury, garbage collector, haldy

Jiří Vyskočil, Marko Genyg-Berezovskyj

2009

Dynamické datové struktury

Reprezentace datové paměti programu:

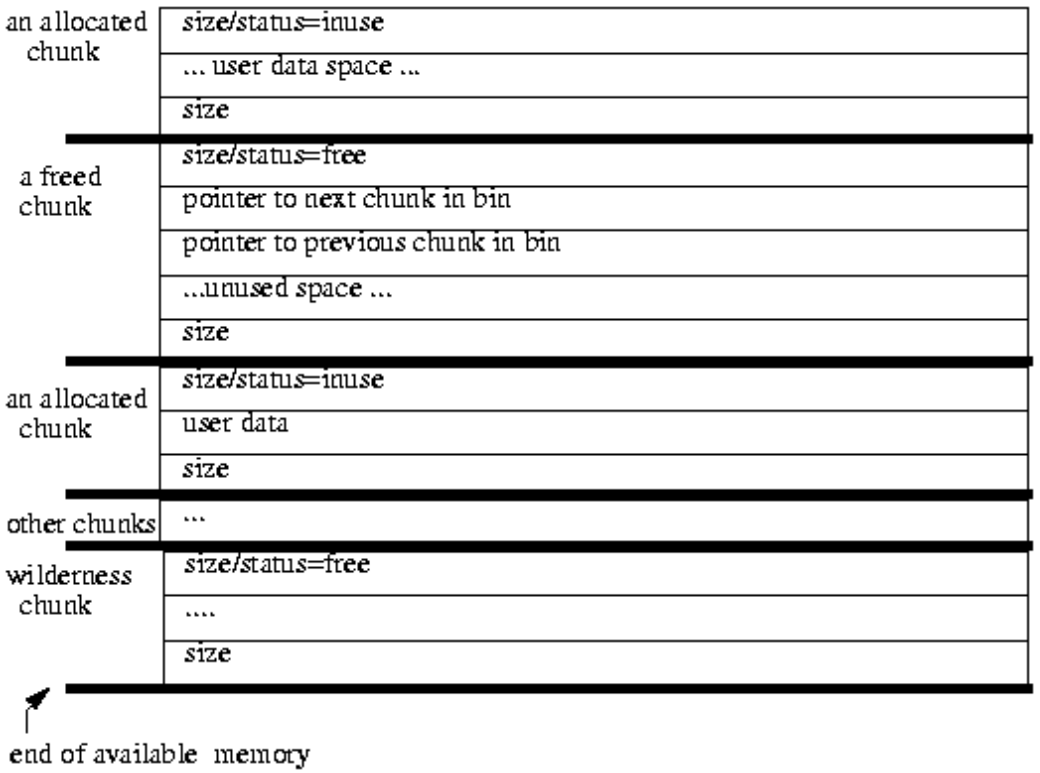


Dynamické datové struktury

- cíle alokátoru dynamické paměti
  - Minimalizovat čas alokace a dealokace (případně ještě realokace).
  - Minimalizovat prostor, včetně minimalizace fragmentace.
  - Maximalizovat lokalitu – alokovat objekty, které se užívají často společně, blízko u sebe. To snižuje počet přestránkování a výpadků na cache paměti.

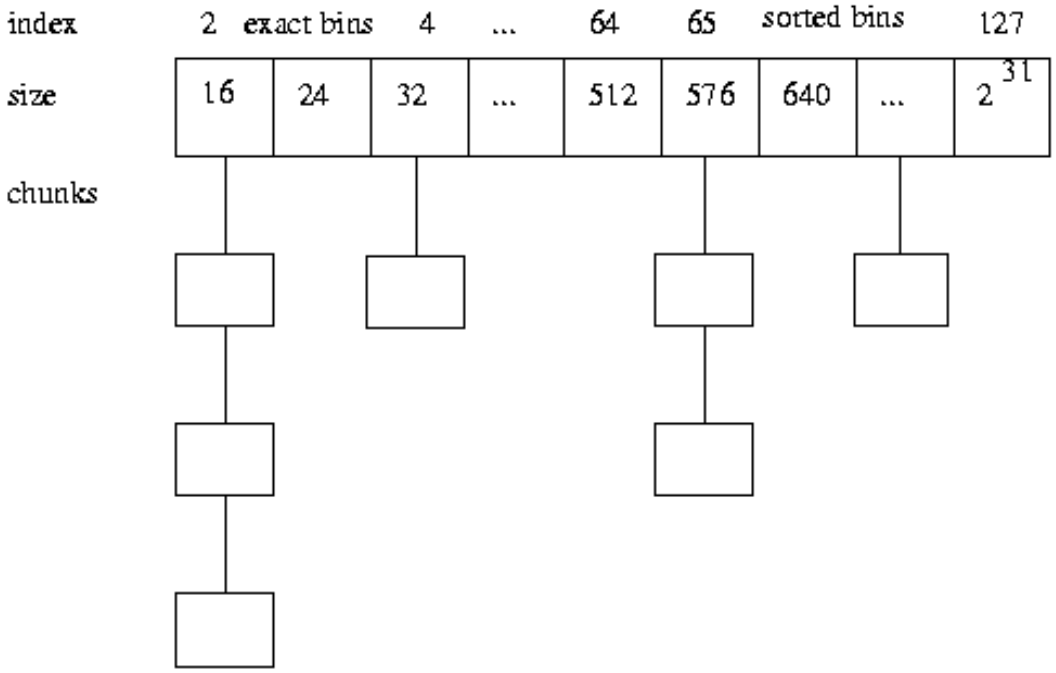
Dynamické datové struktury

- reprezentace dynamické paměti
  - Dva bloky vedle sebe volných *bloků paměti* (chunk) mohou být spojeny do jednoho.
  - Jednotlivé bloky paměti mohou být procházeny oběma směry.
  - Volné bloky paměti jsou mezi sebou propojeny (viz dále).



Dynamické datové struktury

- Volné bloky paměti jsou rozděleny do skupin (bins) podle velikosti.
- Velikost bloků je zarovnaná (některé architektury to přímo vynucují).
- Do určité velikosti jsou skupiny zastoupeny pro každou velikost.
- Při alokaci objektu se začne hledat ve skupině s nejbližší stejnou nebo vyšší velikostí (tzv. best-fit order).



Dynamické datové struktury

- dealokace probíhá následovně:
  - Podíváme se před a za dealokovaný blok. Pokud před ním nebo za ním existuje volný blok, spojíme takové bloky do jednoho. (nejdříve samozřejmě odpojíme tyto krajní volné bloky z příslušné skupiny). Tato operace zamezuje fragmentaci bloků.
  - Výsledný blok přeznačkujeme na volný blok a připojíme do skupiny odpovídající velikosti tohoto bloku.
- Díky mnoha potřebným informacím ve volném bloku je jeho minimální velikost poměrně velká.
- Lokalitu alokace lze řešit procházením vedlejších volných bloků (nearest-fit) často aproximovaná nalézáním dalšího volného bloku (next-fit). Tyto strategie vedou často k vysoké fragmentaci.
- Fragmentaci lze obecně obejít alokací paměti přímo přes operační systém (pokud implementuje stránkování). Je to ovšem pomalejší a minimální velikost bloku je stejná jako velikost stránky. Tyto techniky se často používají při ladění (např. knihovna efence) přidáním volných stránek před nebo za alokované bloky.

Garbage collector

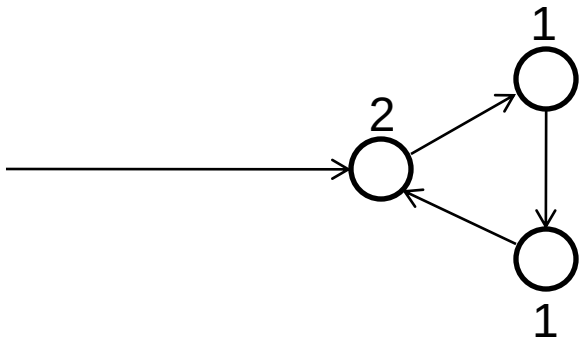
- **garbage collector (GC)**
  - je označení pro metodu automatické správy paměti programu.
- základní princip
  1. V programu se vyhledají takové datové objekty, které již nebudou v budoucnu použity.
  2. Navrátí se zdroje tam, kde se nalezené objekty vyskytovaly.
- Výhody
  - programátor se nemusí starat o rušení objektů
  - zabraňuje chybám ukazatelů
    - přístup k objektu, který byl již zrušen
    - dvojité zrušení již zrušeného objektu
    - zapomínání rušení již nepotřebných objektů
- Nevýhody
  - GC vyžaduje pro svou práci další výpočetní prostředky
  - během své činnosti vytváří v programu různě dlouhé běhové prodlevy, takže není vhodný pro nasazení v real-time systémech.
  - přídavné informace, které GC potřebuje pro svůj běh zvyšují nároky na paměť
  - sémantický GC je neřešitelný problém (většina současných GC pracuje na syntaktické úrovni)

Garbage collector - počítání referencí

- **algoritmus počítání referencí**
  - Vůbec první algoritmus pro GC se jmenoval *počítání referencí* (reference counting) :
    - Ke každému objektu je přiřazen čítač referencí.
    - Když je objekt vytvořen, jeho čítači je nastavena hodnota 1.
    - V okamžiku, kdy si nějaký jiný objekt nebo kořen programu (kořeny jsou hledány v programových registrech, v lokálních proměnných uložených v zásobnících jednotlivých vláken a ve statických proměnných) uloží referenci na tento objekt, hodnota čítače je zvětšena o 1.
    - Ve chvíli, kdy je reference mimo rozsah platnosti (např. po opuštění funkce, která si referenci uložila), nebo když je referenci přiřazena nová hodnota, čítač je snížen o 1.
    - Jestliže je hodnota čítače některého objektu nulová, může být tento objekt uvolněn z paměti.
    - Když je uvolňován z paměti, všem objektům, na něž má objekt referenci, se sníží hodnota o 1 - tedy uvolnění jednoho objektu může vést k uvolnění dalších objektů.

Garbage collector - počítání referencí

- **algoritmus počítání referencí**
  - Nevýhoda této metody spočívá ve faktu, že neumí detekovat cykly. Cyklus nastává v okamžiku, kdy dva a více objektů ukazují samy na sebe. Takové dva objekty nebudou mít nikdy čítač roven nule, přestože jsou nedosažitelné z kořene programu.



  - Další nevýhoda spočívá v režii, která je nutná pro zvyšování a snižování čítačů u každého objektu.
  - Kvůli těmto nedostatkům se reference counting v dnešní době nepoužívá jako univerzální GC.

Garbage collector - Mark & Sweep

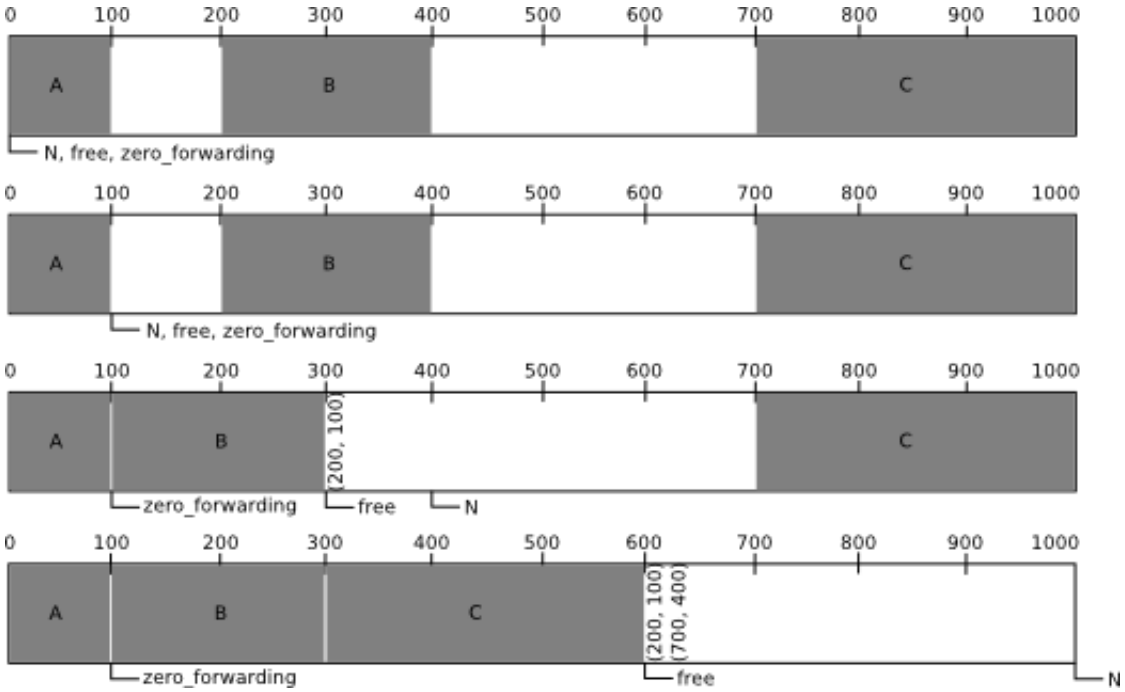
- **algoritmus Mark & Sweep**
  - Patří mezi *trasovací algoritmy.* Ty fungují tak, že tzv. „zastaví svět“ (v tomto smyslu tedy běh programu) a začnou vyhledávat objekty. Začínají v kořenové množině programu a pokračují po referencích, dokud neprozkoumají všechny dosažitelné objekty. Algoritmy, založené na tomto principu, se používají téměř výlučně pro implementaci GC v dnešních programovacích jazycích.
- popis
  - Algoritmus nejdříve nastaví všem objektům, které jsou v paměti, speciální příznak *navštíven* na hodnotu *ne*.
  - Poté projde všechny objekty, ke kterým se lze dostat. Těm, které takto navštívil, nastaví příznak na hodnotu *ano*.
  - V okamžiku, kdy se už nemůže dostat k žádnému dalšímu objektu, znamená to, že všechny objekty s příznakem *navštíven* majícím hodnotu *ne* jsou odpad - a mohou být tedy uvolněny z paměti.
- nevýhody
  - Největší nevýhodou je, že při GC je přerušen běh programu. To znamená, že se programy pravidelně na předem neznámou dobu „zmrazí“. Tato nevýhoda lze řešit inkrementální verzí s třemi stavy u každého objektu tzv. Tri-color Marking algoritmem.
- výhody
  - Lze realizovat s konstantní paměťovou složitostí.

Garbage collector - kopírovací

- **kopírovací algoritmus**
  - Tento algoritmus nejprve rozdělí prostor na haldě na dvě části, kdy jedna je aktivní a s druhou se nepracuje.
  - Vždy můžeme alokovat objekty v celkové velikosti, která je poloviční velikost haldy.
  - Pokud se při alokaci nevejdeme do místa na části haldy, je potřeba provést úklid.
  - Úklid spočívá v prohození aktivní a neaktivní části.
  - Do nově aktivní části se překopírují živé objekty ze staré, již neaktivní, části.
  - Mrtvé objekty nekopírujeme, ale při dalším prohození aktivní a neaktivní části je jednoduše přepíšeme.
- nevýhody
  - Kopírovací algoritmus probíhá zdlouhavě, protože se objekty musí přesouvat z částí haldy. Kvůli náročnosti celého přesunu tedy mohou být prodlevy ještě znatelnější než při použití mark & sweep.
- výhody
  - Nenastává fragmentace jako u Mark & Sweep. Objekty jsou totiž udržovány na spodku právě používaného prostoru, proto stačí k jejich adresovaní mnohem menší rozptyl adres.
  - Alokoace nových objektů je výrazně jednodušší a rychlejší než u standardních alokátoru (ve srovnání například s C/C++).

Garbage collector – Mark & Compact

- **algoritmus Mark & Compact**
  - Odstraňuje problém fragmentace Mark & Sweep algoritmu.
  - Odstraňuje problém dvojnásobné potřeby paměti u kopírovacího algoritmu.
  - Odstraňuje problém zbytečného kopírování velkých dlouhodobých objektů u kopírovacího algoritmu.
  - 1. fáze je stejná jako u Mark & Sweep, ale v druhé provede při odstraňování objektu navíc defragmentaci.



Garbage collector - generační

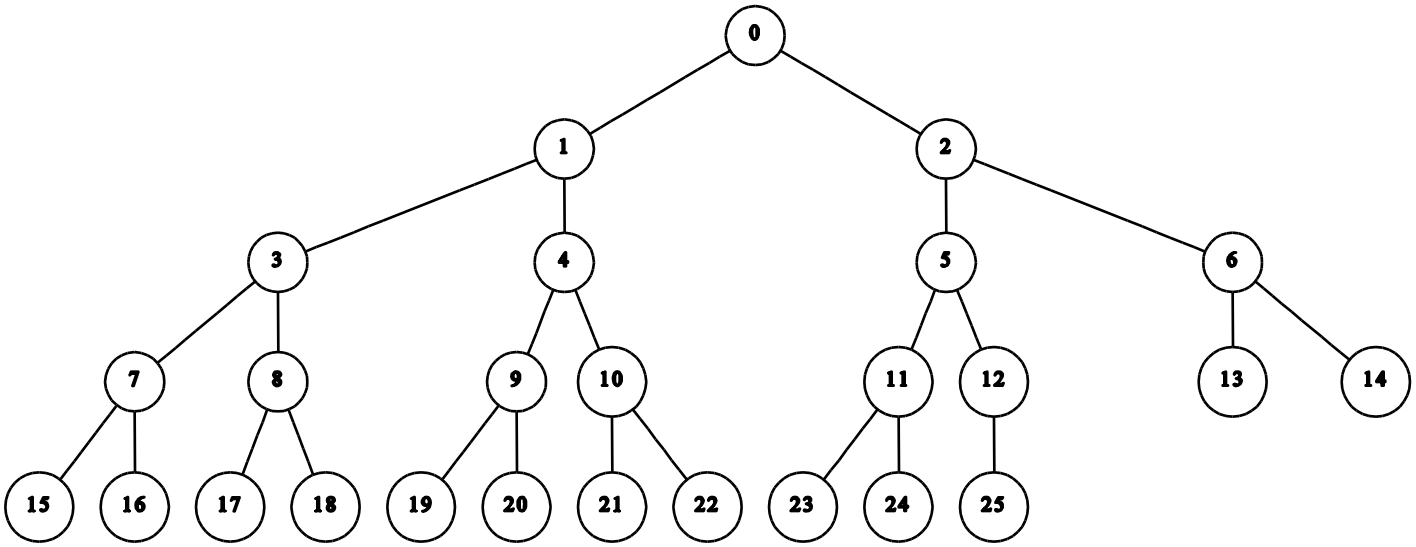
- **generační algoritmus**
  - Při použití GC se dají empiricky vypozorovat dva důležité fakty.
    - mnoho objektů se stane odpadem krátce po svém vzniku.
    - malé procento referencí ve „starších“ objektech ukazuje na objekty mladší.
  - U trasovacích algoritmů, kde se využívá celá halda, musel GC při každém spuštění procházet mezi objekty a všechny živé objekty buďto překopírovat do jiné části haldy, nebo je označit a dále projít celou haldu a uvolnit mrtvé. A právě z důvodu brzkého *úmrtí* většiny objektů je tato metoda velice neefektivní.
  - *Generační* GC využívá těchto skutečností a rozděluje si paměť programu do několika částí, tzv. *generací*.
  - Objekty jsou vytvářeny ve spodní (nejmladší) generaci a po splnění určité podmínky, obvykle stáří, jsou přeřazeny do starší generace.
  - Pro každou generaci může být úklid prováděn v rozdílných časových intervalech (obvykle nejkratší intervaly budou pro nejmladší generaci) a dokonce pro rozdílné generace mohou být použity rozdílné algoritmy úklidu.
  - V okamžiku, kdy se prostor pro spodní generaci zaplní, všechny dosažitelné objekty v nejmladší generaci jsou zkopírovány do starší generace. I tak bude množství kopírovaných objektů pouze zlomkem z celkového množství mladších objektů, jelikož většina z nich se již stala odpadem.

Haldy

- halda (anglicky heap)
  - datová struktura (obvykle stromová struktura s vlastností: Pokud A je potomek B, potom B<=A), která je specializovaná na provádění následujících operací (vhodná pro reprezentaci prioritní fronty):
    - **Insert** (*x*)
      - Vloží prvek *x* do haldy.
    - **AccessMin**
      - Vrátí nejmenší prvek haldy.
    - **DeleteMin**
      - Odstraní z haldy nejmenší prvek (obvykle kořen).
    - **DecreaseKey** (*x*,*d*)
      - Zmenší hodnotu prvku *x* o *d*.
    - **Merge** (H_1, H_2)
      - Sloučí haldy H_1 a H_2 do jedné, kterou vrátí.
    - **Delete** (*x*)
      - Odstraní prvek *x* z haldy.

Binární halda

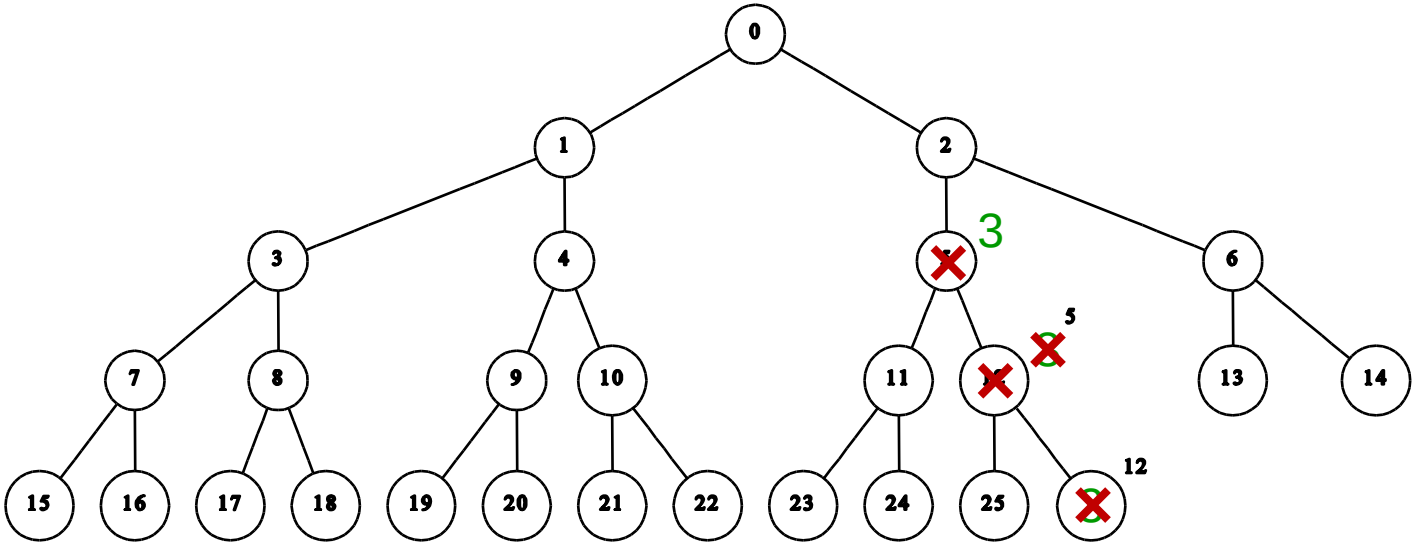
- binární halda
  - Je binární strom s následujícími vlastnostmi:
    - Je vyvážený až do předposledního patra. Poslední patro je postupně zaplněno zleva doprava.
    - Každý vrchol je menší nebo roven všem svým potomkům.



Binární halda - Insert

- Insert (x)
  1. Přidáme prvek x na konec haldy;
  2. **while** (parent(x)† > x) **{**
  3. Prohodíme umístění prvku x s prvkem parent(x);
  4. **}**

†parent(x) vrací rodiče x. Pro prvek x, který nemá rodiče vrací parent(x) x.



Binární halda

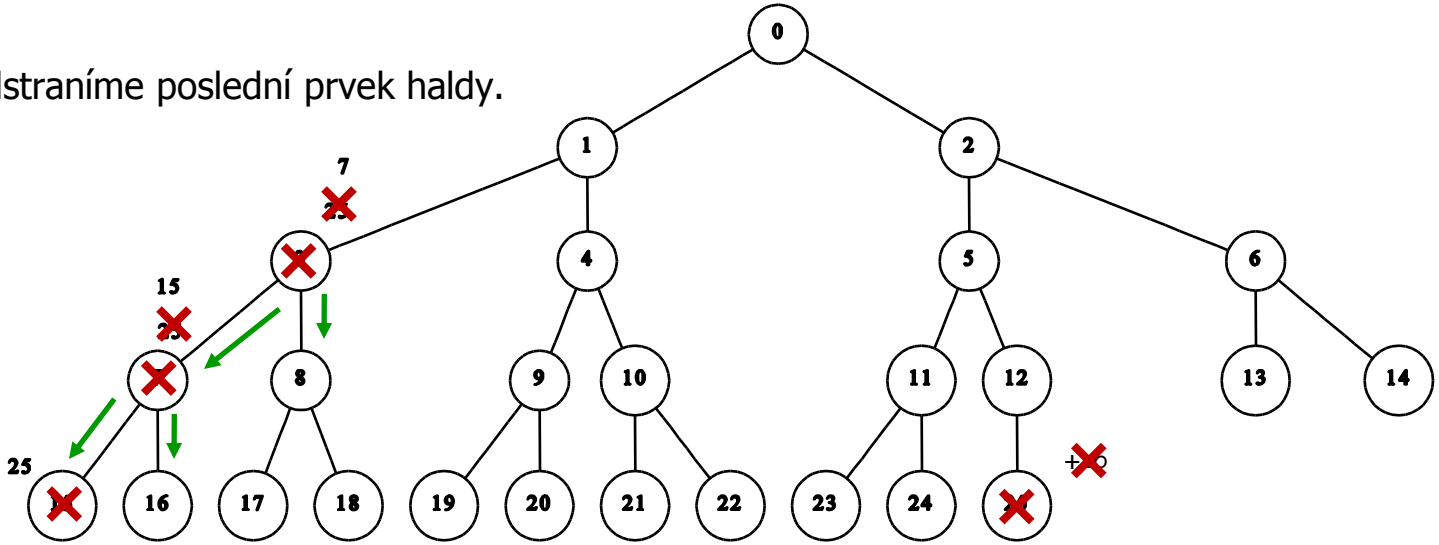
- **AccessMin**
  - Je kořen binárního stromu haldy.
- **DeleteMin**
  1. $\&x$ = umístění kořene haldy;
  2. $x = +\infty$;
  3. $\&y$ = umístění posledního prvku haldy;
  4. **do {**
  5. Prohodíme umístění prvku x s prvkem y;
  6. $\&x = \&y$;
  7. **for each** $z \in \text{descendants}(x)^{\dagger}$ **do**
  8. **if** $(y > z)$ **then** $\&y = \&z$;
  9. **}**
  10. Odstraníme poslední prvek haldy.

  $^{\dagger}\text{descendants}(x)$ vrací všechny potomky x.
- **DecreaseKey** (x , d)
  - Zmenšíme hodnotu prvku x o d a pak aplikujeme analogický postup jako u operace Insert.

Binární halda - Delete

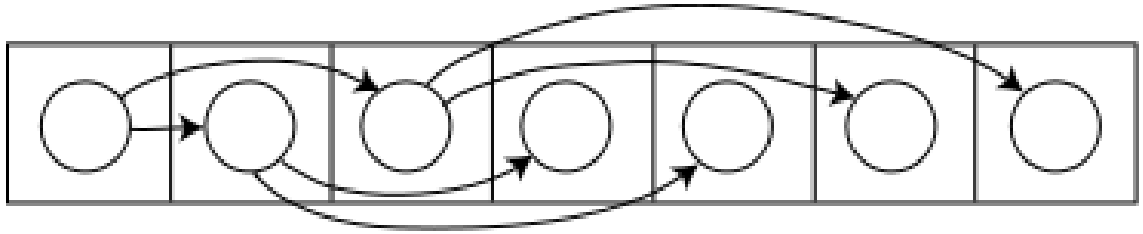
- Delete (&x)

1. $x = +\infty$; &y = umístění posledního prvku haldy;
2. **do {**
3. Prohodíme umístění prvku x s prvkem y;
4. &x = &y;
5. **for each** z ∈ descendants(x) **do**
6. **if** ($y > z$) **then** &y = &z;
7. **} while** ($x \neq y$);
8. **while** (parent(x) > x) **{**
9. Prohodíme umístění prvku x s prvkem parent(x);
10. **}**
11. Odstraníme poslední prvek haldy.



Binární halda - reprezentace

- **reprezentace v paměti**
  - **stromovou dynamickou datovou strukturou s ukazateli v obou směrech**
  - **polem (s kořenem s indexem 1)**



2. syn
$(2*n)+1$
nebo také $(n << 1) + 1$

otec
n div 2
nebo také $n >> 1$

prvek
n

1. syn
$2*n$
nebo také $n << 1$

Binární halda - složitost

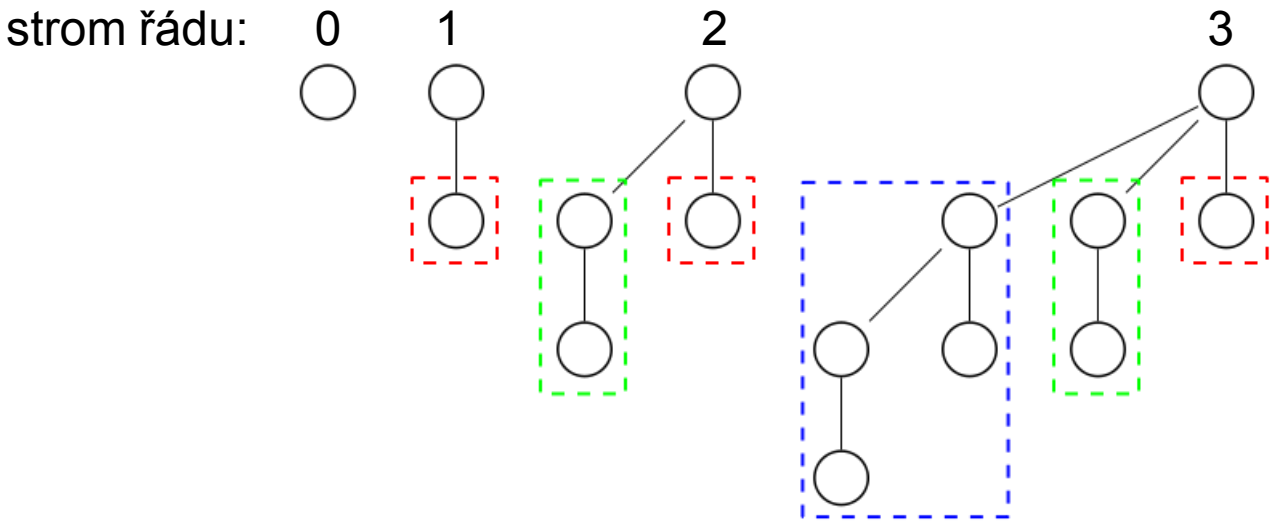
- **Insert**
  - O(log(n))
- **Delete**
  - O(log(n))
- **AccessMin**
  - O(1)
- **DeleteMin**
  - O(log(n))
- **DecreaseKey**
  - O(log(n))
- **Merge**
  - Lze provést v O(n) novým vybudováním haldy.

d-regulární halda

- Pro d=2 je d-regulární halda právě binární halda.
- Parametr d udává stupeň štěpení stromu haldy.
- Operace nad d-regulární haldou jsou analogické jako v případě binární haldy.
- Asymptotická složitost je stejná jako u binární haldy.
- Přesná složitost se liší základem logaritmu (základ je d) a pro operaci Delete navíc ještě s násobkem d.
- Pro rychlou implementaci je výhodné za d volit mocniny 2. Pak lze použít reprezentaci polem s bitovými posuny.

Binomiální halda

- **binomiální halda**
  - Binomiální halda reprezentující množinu S, kde $|S| = n$ je množina tzv. *binomiálních stromů* řádů $i=1,\ldots,\lfloor \log(n) \rfloor$. Každý řád je přitom zastoupen nejvýše jedním stromem.
- **binomiální stromy** se konstruují následujícím způsobem:
  - strom řádu 0 je samostatný vrchol
  - strom řádu i (pro $i \geq 1$) se skládá z kořene a i synů, což jsou binomiální stromy řádu 0 až $i - 1$.



Binomiální halda – binomiální stromy

- Pro binomiální strom řádu *i* platí
  - každý vrchol je menší nebo roven všem svým potomkům
  - má 2^i vrcholů
  - jeho hloubka je *i*
  - jeho kořen má *i* synů
  - pro $\forall j \leq i$ existuje syn kořene, který je binomiálním stromem řádu *j*

- Binomiální stromy lze definovat i jiným způsobem:
  - Strom řádu *i* vznikne ze dvou stromů řádu *i - 1* tak, že ke stromu, jehož kořen reprezentuje menší prvek se jako další syn připojí strom s větším kořenem. Tak je zachována lokální podmínka na uspořádání haldy.

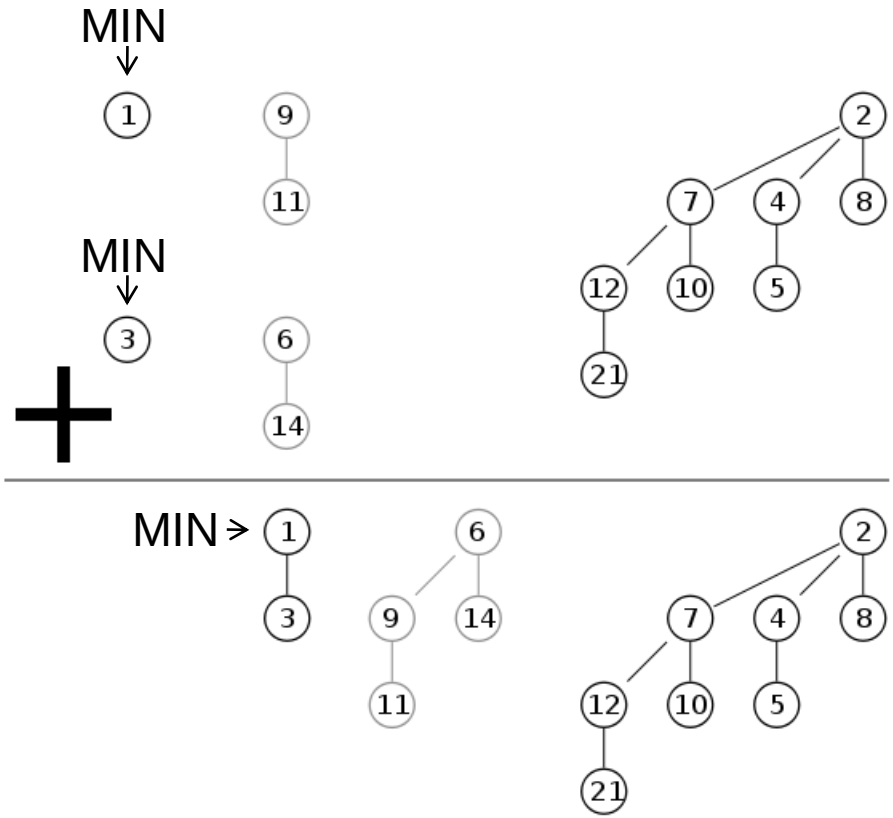
Binomiální halda – reprezentace

- Binomiální halda se standardně implementuje pomocí pole ukazatelů, v němž *i*-tý ukazatel je buď null nebo ukazuje na kořen stromu řádu *i*.
- Kromě ukazatelů na jednotlivé stromy existuje v haldě ještě zvláštní ukazatel na strom, v jehož kořeni je uložen nejmenší prvek z celé haldy. Tento tzv. *MIN ukazatel* se aktualizuje při každé operaci.

Binomiální halda – Insert, AccessMin, Merge

- **Insert** (x)
  - Vytvoří se strom řádu 0, tedy jediný vrchol reprezentující přidávaný prvek x. Tento strom je vlastně velmi jednoduchá binomiální halda.
  - Poté se zavolá **Merge** na obě haldy.
- **AccessMin**
  - Vrátí prvek reprezentovaný kořenem stromu, na nějž ukazuje *MIN ukazatel*.
- **Merge** (H_1, H_2)
  - Tato operace probíhá analogicky ke sčítání binárních čísel. Obě haldy se prochází podle řádu stromů od nejmenšího. Pokud je příslušný řád zastoupen pouze v jedné haldě, překopíruje se do výsledné haldy. Pokud je zastoupen v obou, spojí se do jednoho stromu vyššího řádu a ten se umístí do „přenosového stromu“ (analogie přenosového bitu). Ten se případně spojuje s dalšími stromy z původních hald. Do výsledné haldy se přidá *MIN ukazatel*, jehož hodnota odpovídá minimu z obou vstupních hald.

Binomiální halda - Merge



Binomiální halda - DeleteMin

1. **procedure** DeleteMin(binomial_heap H)
2. tree_with_minimum = H.*MIN*;
3. **for each** tree ∈ tree_with_minimum.subTrees **do {**
4. tmp.addTree(tree);
5. **}**
6. H.removeTree(tree_with_minimum)†;
7. H = **Merge**(H, tmp);

† Technicky tato operace smaže pouze kořen stromu. Všichni potomci kořene se dále používají v tmp haldě, která se následně sloučí operací Merge.

Binomiální halda – DecreaseKey, Delete

- **DecreaseKey**
  - Funguje analogicky jako v případě binární haldy.

- **Delete** (x)
  - Nejprve pomocí operace DecreaseKey snížíme hodnotu prvku x na -∞.
  - Potom odstraníme tento prvek pomocí operace DeleteMin.

Binomiální halda – složitost

- **Merge**
  - O(log(n))
- **Insert**
  - O(log(n))
  - Amortizovaná složitost je konstantní. Operace je analogická k inkrementaci binárního čítače.
- **AccessMin**
  - O(1)
- **DeleteMin**
  - O(log(n))
- **DecreaseKey**
  - O(log(n))
- **Delete**
  - O(log(n))

Fibonacciho halda

- **Fibonacciho halda** je druh haldy, který Principiálně vychází z binomiální haldy.
- Hlavní výhodou Fibonacciho haldy je nízká asymptotická složitost prováděných algoritmů.
- Operace Insert, AccessMin a Merge probíhají v O(1).
- Operace DecreaseKey probíhá v amortizovaném konstantním čase.
- Operace Delete a DeleteMin pracují s amortizovanou časovou složitostí O(log(n)).
- Užití Fibonacciho haldy není vhodné pro real-time systémy, protože některé operace mohou mít v nejhorším případě lineární složitost.

Fibonacciho halda

- Fibonacciho haldu tvoří skupina stromů vyhovující lokální podmínce na uspořádání haldy, která vyžaduje, aby pro každý uzel stromu platilo, že prvek, který reprezentuje, je menší než prvek reprezentovaný jeho potomky. Z této podmínky vyplývá, že minimálním prvkem je vždy kořen jednoho ze stromů.
- Vnitřní struktura Fibonacciho haldy je v porovnání s binomiální haldou daleko více flexibilní. Jednotlivé stromy nemají pevně daný tvar a v extrémním případě může každý prvek haldy tvořit izolovaný strom nebo naopak všechny prvky mohou být součástí jediného stromu hloubky N. Tato flexibilní struktura umožňuje velmi jednoduchou implementaci operací s haldou.
- Operace, které nejsou potřebné, odkládáme a vykonáváme je až v okamžiku, kdy je to nevyhnutelné, například spojení nebo vložení nového prvku se jednoduše provede spojením kořenových seznamů (s konstantní náročností) a jednotlivé stromy spojíme až při operaci snížení hodnoty klíče.