PDF verze

Úvod

Algoritmus

- Konečná, uspořádaná množina úplně definovaných pravidel pro vyřešení nějakého problému
- Posloupnost výpočetních kroků, které transformují vstup na výstup

Heuristika

- Postup, který nedává vždy přesné řešení problému.
- Ve většině případů dává dostatečně přesné řešení v rozumném čase.
- Nezaručuje nalezení přesného řešení.
- Použijeme tehdy, pokud pro daný problém neexistuje přesný algoritmus, nebo jeho použití je neekonomické.

Asymptotická časová zložitost

Odvozena od počtu tzv. elementárních operací: sčítání, násobení, porovnání, skoky, atd.

Používají se tři různé složitosti: - O – Omikron (velké O, Ø, big O) – horní hranice chování - Ω – Omega – dolní hranice chování - Θ – Théta – třída chování

Prostorová složitost

- Měří paměťové nároky algoritmu
- Kolik nejvíce elementárních paměťových buněk algoritmus použije.
- Elementární paměťová buňka: proměnná typu integer, float, byte apod.

Lineární abstraktní datové typy

Abstraktní datový typ (ADT)

Abstraktní datový typ (ADT) je definován množinou hodnot, kterých smí nabýt každý prvek tohoto typu, a množinou operací nad tímto typem.

ADT TList

Rekurzivní definice

Délka seznamu Je-li seznam prázdný, má délku nula. V jiném případě je jeho délka 1 plus délka zbytku seznamu.

Ekvivalence dvou seznamů Dva seznamy jsou ekvivalentní, když jsou oba prázdné nebo když se rovnají jejich první prvky a současně jejich zbytky.

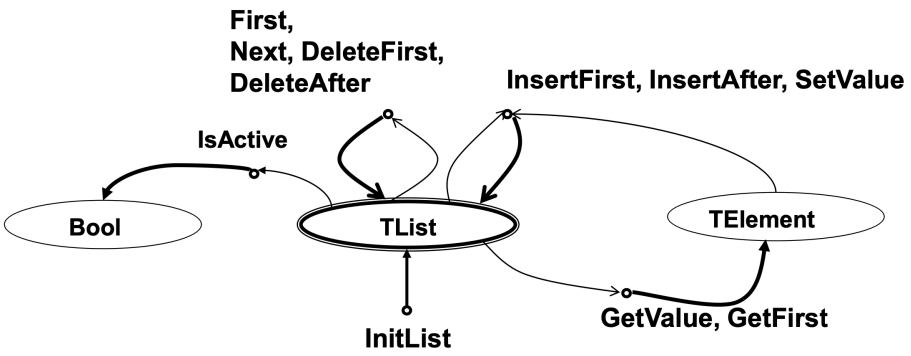


Figure 1: ADT TList

Zásobník a fronta

Převod infixové notace na postfixovou

1. Zpracovávej vstupní řetězec položku po položce zleva doprava a vytvářej postupně výstupní řetězec.
2. Je-li zpracovávanou položkou operand, přidej ho na konec vznikajícího výstupního řetězce.
3. Je-li zpracovávanou položkou levá závorka, vlož ji na vrchol zásobníku.
4. Je-li zpracovávanou položkou operátor, pak ho na vrchol zásobníku vlož v případě, že:
   - zásobník je prázdný
   - na vrcholu zásobníku je levá závorka
   - na vrcholu zásobníku je operátor s nižší prioritou

   Je-li na vrcholu zásobníku operátor s vyšší nebo shodnou prioritou, odstraň ho, vlož ho na konec výstupního řetězce a opakuj krok 4, až se ti podaří operátor vložit na vrchol.
5. Je-li zpracovávanou položkou pravá závorka, odebírej z vrcholu položky a dávej je na konec výstupního řetězce, až narazíš na levou závorku. Levou závorku odstraň ze zásobníku. Tím je pár závorek zpracován.
6. Je-li zpracovávanou položkou omezovač =, pak postupně odstraňuj prvky z vrcholu zásobníku a přidávej je na konec řetězce, až zásobník zcela vyprázdníš, a na konec přidej rovnítko.

Prioritní fronta

- Prvkům fronty je navíc přiřazena priorita.
- Prvky s vyšší prioritu přeskakují prvky s nižší prioritou a jsou obsluhovány

ADT TStack – diagram signatury

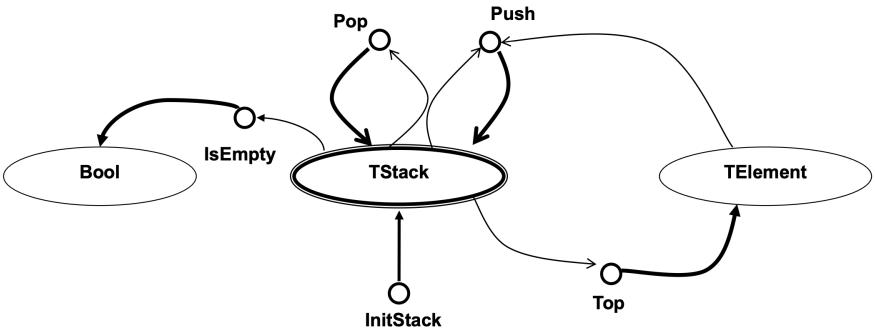


Figure 2: ADT TStack

ADT TQueue – diagram signatury

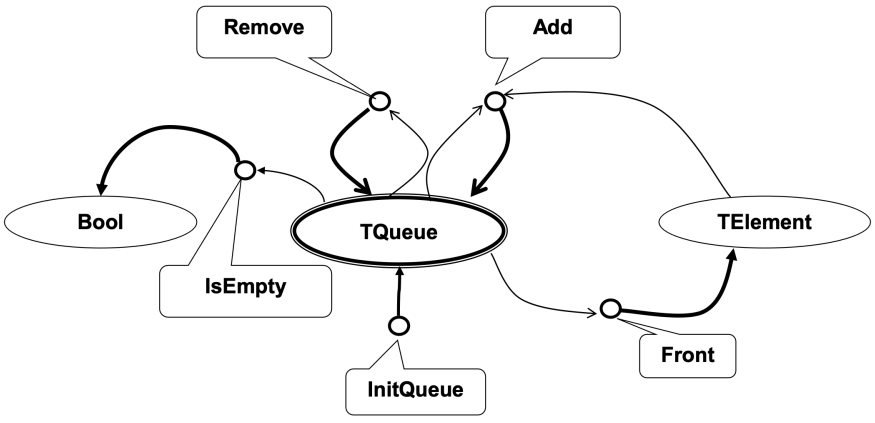


Figure 3: ADT TQueue

dříve než prvky s nižší prioritou.
- Jako první opouští frontu nejstarší prvek s nejvyšší prioritou.

Mapovací funkce

- Převádí n-tici indexů prvku n-dimenzionálního pole na jeden index jednorozměrného pole.
- Závisí na tom, jak je n-dimenzionální pole uloženo v paměti (po řádcích nebo po sloupcích).

Stromové datové struktury

Kořenový strom

Kořenový strom je souvislý acyklický graf, který má jeden zvláštní uzel, který se nazývá kořen (angl. root).

- Kořen je takový uzel, že platí, že z každého uzlu stromu vede jen jedna cesta do kořene.

Výška stromu

- výška prázdného stromu je 0,
- výška stromu s jediným uzlem (kořenem) je 1,
- výška jiného stromu je počet hran od kořene k nejvzdálenějšímu uzlu + 1.

Rekurzivní definice binárního stromu

Binární strom je buď prázdný, nebo sestává z jednoho uzlu zvaného kořen a dvou binárních podstromů – levého a pravého.

Binární strom sestává z: - **kořene**, - **neterminálních** (vnitřních) uzlů, které mají ukazatel na jednoho nebo dva uzly synovské a - **terminálních** uzlů (listů), které nemají žádné potomky.

Vyváženost stromu

- Binární strom je váhově vyvážený, když pro každý jeho uzel platí, že počty uzlů jeho levého a pravého podstromu se rovnají a nebo se liší právě o 1.
- Binární strom je výškově vyvážený, když pro každý jeho uzel platí, že výška levého podstromu se rovná výšce pravého podstromu a nebo se liší právě o 1.
- Maximální výška vyvážených stromů: **c.log(n)**

Výška stromu – rekurzivně

```
void HeightBT (TNode *ptr, int *max)
{
    int hl,hr;
```

```
    if (ptr != NULL){
        HeightBT(ptr->left,&hl);
        HeightBT(ptr->right,&hr);
        if (hl > hr) {
            *max = hl+1;
        } else {
            *max = hr+1;
        }
    } // if ptr != NULL
    else {
        *max = 0;
    }
}
```

nebo

```
int max (int n1, int n2)
{ // funkce vrátí hodnotu většího ze dvou parametrů
    if (n1 > n2) {
        return n1;
    } else {
        return n2;
    }
}

int Height (TNode *ptr)
{
    if (ptr != NULL) {
        return max(Height(ptr->left),Height(ptr->right))+1;
    } else {
        return 0;
    }
}
```

Ekvivalence (struktur) dvou BS

```
bool EQTS (TNode *ptr1, TNode *ptr2)
{
    if ((ptr1 == NULL)||(ptr2 == NULL)){
        return ptr1 == ptr2;
    } else {
        return (EQTS(ptr1->left,ptr2->left) &&
        EQTS(ptr1->right,ptr2->right));
        // && (ptr1->data == ptr2->data) pro ekvivalenci BS
    }
}
```

Kopie BS – rekurzivně

```
TNode * CopyR (TNode *orig)
{
    TNode *copy;
    if (orig != NULL){
        copy = (TNode *) malloc(sizeof(TNode));
        // zkontrolovat úspěšnost operace malloc
        copy->data = orig->data;
        copy->left = CopyR(orig->left);
        copy->right = CopyR(orig->right);
        return copy;
    } else {
        return NULL;
    }
}
```

Test váhové vyváženosti BS

```
bool TestWBT (TNode *ptr, int *count)
{
    bool left_balanced, right_balanced;
    int left_count, right_count;
    if (ptr != NULL){
        left_balanced = TestWBT(ptr->left,&left_count);
        right_balanced = TestWBT(ptr->right,&right_count);
        *count = left_count + right_count + 1;
        return  (left_balanced && right_balanced &&
                (abs(left_count - right_count) <= 1));
    } else {
        *count = 0;
        return true;
    }
}
```

Level-order průchod

```
void LevelOrder (TDLList *l, TNode *ptr)
{ /* globální fronta ukazatelů */
    InitQueue(&q1);
    Add(&q1,ptr);
    while (!IsEmpty(&q1)) {
        TNode *aux = Front(&q1);
        Remove(&q1);
        if (aux != NULL) {
            DLL_InsertLast(l,aux->data);
            Add(&q1,aux->left);
```

```
            Add(&q1,aux->right);
        }
    } //while
}
```

Vyhledávací tabulky

- Každá položka má zvláštní složku – klíč
- V tabulce s (ostrým) vyhledáváním je hodnota klíče jedinečná (neexistují dvě či více položek se stejnou hodnotou klíče).

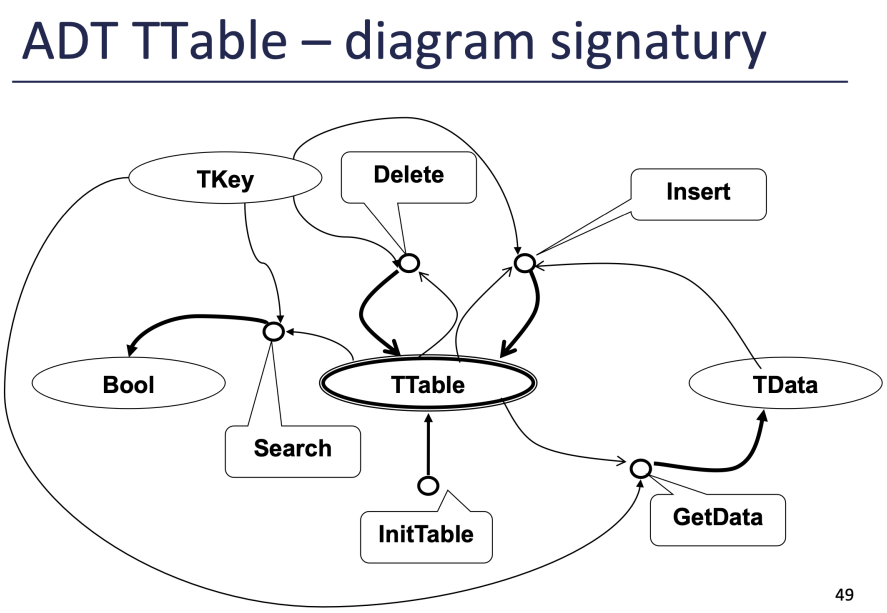


Figure 4: ADT TTable

Sekvenční vyhledávání

```
bool function Search (TTable t, TKey k)
    found ← false
    i ← 0
    while not found and (i < t.n):
        if k = t.array[i].key:
            found ← true
        else:
            i ← i + 1
    return (found)
```

Sekvenční vyhledávání se zarážkou

Zarážka (sentinel, guard, stop-point): - Dovoluje vynechat test na konec pole. - Sníží efektivní kapacitu tabulky o jednu položku. - Vynecháním testu na konec se algoritmus zrychlí.

```
bool function SearchG (TTable t, TKey k)
    i ← 0
    t.array[t.n].key ← k // vložení zarážky
    while k != t.array[i].key:
        i ← i + 1
    return (i != t.n)
    // když našel až zarážku, tak vlastně nenašel ...
```

Binární vyhledávání

- Lze provést nad seřazenou množinou klíčů ve struktuře s náhodným přístupem (v poli).
- Připomíná metodu půlení intervalu pro hledání jediného kořene funkce v daném intervalu
- Výhoda: časová složitost vyhledávání je v nejhorším případě logaritmická: $\mathbf{log_2(n)}$

```
left ← 0 right ← t.n-1 // levý index
// pravý index
do:
    middle ← (left+right) div 2
    if k < t.array[middle].key:
        // hledaná položka je vlevo
        right← middle - 1
    else:
        // hledaná položka je vpravo
        left ← middle + 1
while (k != t.array[middle].key) and (left   right)
return (k = t.array[middle].key)
```

Dijkstrova varianta Dijkstrova varianta umožňuje existenci více prvků se shodným klíčem

```
left ← 0
right← t.n-1
while right != (left+1):
    middle←(left+right) div 2
    if t.array[middle].key   k:
        left ← middle
    else:
        right← middle
return ((k = t.array[left].key), left)
```

Dijkstrova varianta končí **vždy za stejnou dobu**, určenou hodnotou dvojkového logaritmu počtu prvků.

Vyhledávání v binárním stromu

- Je-li vyhledávaný **klíč roven kořeni**, vyhledávání končí úspěšným vyhledáním.
- Je-li klíč **menší**, pokračuje vyhledávání v **levém podstromu**, je-li **větší**, pokračuje v **pravém podstromu**.
- Vyhledávání končí neúspěšně, pokud je prohledávaný (pod)strom **prázdný**.

```
bool function Search (TNode *rootPtr, TKey k)
if rootPtr = NULL:
    return (false) // nenašli jsme
else:
    if rootPtr->key = k:
        return (true) // našli jsme
    else:
    if k < rootPtr->key: // hledáme v levém podstromu
        return (Search(rootPtr->lPtr,k))
    else: // hledáme v pravém podstromu
        return (Search(rootPtr->rPtr,k))
```

BVS – Insert (rekurzivní zápis)

```
TNode* function Insert (TNode *rootPtr, TKey k, TData d)
    if rootPtr = NULL: // vytvoření nového uzlu
        return CreateNode(k,d)
    else:
        if k < rootPtr->key: // jdeme vlevo
            rootPtr->lPtr ← Insert(rootPtr->lPtr,k,d)
        else:
            if rootPtr->key < k: // jdeme vpravo
                rootPtr->rPtr ← Insert(rootPtr->rPtr,k,d)
            else: // přepíšeme stará data novými
                rootPtr->data← d
        return rootPtr
```

BVS Rušení uzlu – operace Delete Uzel nezrušíme fyzicky, ale přepíšeme hodnotou takového uzlu, který lze zrušit snadno, a při přepisu nedojde k porušení uspořádání BVS.

Vhodný uzel: - **nejpravější uzel levého podstromu rušeného uzlu** (maximum v levém podstromu) nebo - **nejlevější uzel pravého podstromu rušeného uzlu** (minimum v pravém podstromu).

```
TNode* function BVSMin (TNode *rootPtr)
// funkce vrátí ukazatel na nejlevější uzel v daném
// neprázdném(!) stromu
    if rootPtr->lPtr = NULL: // další levý už neexistuje
        return rootPtr
    else: // pokračujeme vlevo
        return BVSMin(rootPtr->lPtr)

TNode* function BVSDelete (TNode *rootPtr, int k)
    if rootPtr = NULL: // prázdný (pod)strom
        return NULL
    else:
        if k < rootPtr->key: // rušený klíč je v levém podstromu
            rootPtr->lPtr← BVSDelete(rootPtr->lPtr,k)
            return rootPtr
        else:
            if rootPtr->key < k: // rušený klíč je v pravém podstromu
                rootPtr->rPtr← BVSDelete(rootPtr->rPtr,k)
                return rootPtr
            else: // nalezen uzel s daným klíčem
                if (rootPtr->lPtr = NULL) and (rootPtr->rPtr = NULL):
                    free(rootPtr) // rušený nemá žádného syna
                    return NULL
                else:
                    if (rootPtr->lPtr != NULL) and (rootPtr->rPtr != NULL):
                        // rušený má oba podstromy
                        TNode *min ← BVSMin(rootPtr->rPtr) // najdi minimum
                        rootPtr->key← min->key // nahraď
                        rootPtr->data ← min->data
                        rootPtr->rPtr← BVSDelete(rootPtr->rPtr,min->key)
                        return rootPtr
                    else: // rušený má pouze jeden podstrom
                        if rootPtr->lPtr = NULL: // rušený nemá levého syna
                            TNode *onlyChild← rootPtr->rPtr
                        else: // rušený nemá pravého syna
                            TNode *onlyChild← rootPtr->lPtr
                        free(rootPtr)
                        return onlyChild
```

AVL stromy

- **Výškově vyvážený strom**
- Je maximálně o 45 % vyšší než váhově vyvážený strom.
- Výškově vyvážený binární vyhledávací strom je strom, pro jehož každý uzel platí, že výška jeho dvou podstromů je stejná nebo se liší o 1.
- **Kritický uzel** – nejvzdálenější uzel od kořene, v němž je v důsledku

vkládání nebo rušení porušená rovnováha.

Každému uzlu přiřadíme váhu takto: - 0: zcela vyvážený uzel - -1: výška levého podstromu je o jedna větší - 1: výška pravého podstromu je o jedna větší

Pokud v rámci operace Insert nebo Delete dojde ke změně váhy na hodnotu -2/2, je potřeba situaci napravit. Mohou nastat 4 různé situace, které se napravují různými způsoby: - **LL**: kritický uzel je příliš těžký vlevo a jeho levý syn je těžký vlevo - **LR**: kritický uzel je příliš těžký vlevo a jeho levý syn je těžký vpravo - **RR**: kritický uzel je příliš těžký vpravo a jeho pravý syn je těžký vpravo - **RL**: kritický uzel je příliš těžký vpravo a jeho pravý syn je těžký vlevo

- Situaci LL opravíme pravou rotací
- Situaci LR opravíme dvojitou rotací – levá rotace následovaná pravou rotací
- Situaci RR opravíme levou rotací
- Situaci RL opravíme dvojitou rotací – pravá rotace následovaná levou rotací

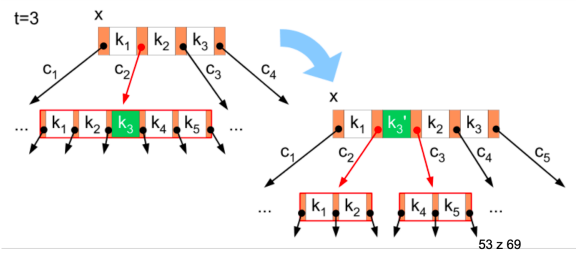
(a,b)-stromy

(a,b)-strom pro parametry a 2, b 2a-1 je obecný vyhledávací strom, pro který navíc platí: 1. Kořen má 2 až b synů, ostatní vnitřní vrcholy a až b synů. 2. Všechny vnější vrcholy jsou ve stejné hloubce.

(a,b)-strom s n klíči má hloubku **Θ(log n)**. Časová složitost: **Θ(log n)** (délka všech cest od kořene k listům je stejná) Obvykle se používají (a, 2a-1) nebo (a,2a)-stromy, časté parametry: **(2,3)** nebo **(2,4)**

Vkládání do (a,b)-stromu

- Nevkládáme nový list (porušení pravidla o stejné hloubce vnějších uzlů)
- Jde-li vložit další klíč do příslušného uzlu na nejnižší hladině, aniž by došlo k přeplnění uzlu – vložíme.
- Pokud by mělo dojít k přeplnění uzlu, uzel rozštěpíme, prostřední klíč vložíme do nadřazeného uzlu (abychom mohli připojit 1 syna navíc) a zbývající klíče přiřadíme do nových vrcholů.
- Přidáním klíče do nadřazeného vrcholu posuneme problém štěpení uzlu o úroveň výš.
- Bude-li potřeba rozštěpit kořen, vytvoříme nový kořen s jediným klíčem a celý strom se o hladinu prohloubí.



53 z 69

Varianta: zcela naplněné uzly jsou štěpeny už cestou dolů stromem, při vyhledávání místa, kam má být nový prvek vložen.

Mazání v (a,b)-stromu

- Klíč na nejnižší hladině lze smazat přímo, ale nesmí vzniknout uzel s nedostatečným počtem synů.
- Klíče na vyšších hladinách nelze smazat přímo, nahradíme jejich hodnotu např. nejnižším klíčem z nejlevějšího vrcholu pravého podstromu a ten potom smažeme.
- Řešení nedostatečného počtu synů s využitím bratra:
  - Má-li bratr (lze vybrat levého i pravého) pouze *a* synů, sloučíme podměrečný uzel s bratrem a doplníme uzel klíčem z otce (možný problém nedostatku synů se přesune na otce).
  - Má-li bratr více než *a* synů, odpojíme od něj nejpravějšího syna *c* a největší klíč *m*. Klíč *m* přesuneme do otce, z otce příslušný klíč přesuneme do podměrečného uzlu a před něj připojíme syna *c*.
- Pokud zmizí z kořene všechny klíče, je kořen smazán, čímž se sníží výška stromu.

56 z 69

Figure 5: Mazání v (a,b)-stromu

LLRB stromy

LLRB strom je binární vyhledávací strom s vnějšími vrcholy, jehož hrany jsou obarveny červeně a černě. Přitom platí následující axiomy: 1. Neexistují dvě červené hrany bezprostředně nad sebou. 2. Jestliže z vrcholu vede dolů jediná červená hrana, pak vede doleva. 3. Hrany do listů jsou vždy obarveny černě. (To se hodí, jelikož listy jsou pouze virtuální, takže do nich neumíme barvu hrany uložit.) 4. Na všech cestách z kořene do listu leží stejný počet černých hran.

LLRB strom – překlad (2,4) stromu na BVS s **logaritmickou hloubkou** a možností vyvažování.

Překlad (2,4)-stromu na LLRB Každý vrchol (2,4)-stromu nahradíme konfigurací jednoho nebo více binárních vrcholů.

Pro zachování korespondence mezi stromy zavedeme 2 barvy hran: - Červené hrany – spojují vrcholy tvořící 1 konfiguraci - Černé hrany – hrany mezi konfiguracemi (hrany původního stromu)

Vrcholy označujeme dle počtu synů jako 2-vrchol, 3-vrchol, 4-vrchol.

Transformace 3-vrcholu – nahradíme 2 vrcholy a červená hrana musí vždy vést **doleva**.

Vkládání v LLRB

- Vyváženost stromu je udržována rotacemi, a to jen **červených** hran.
- Nový uzel vkládáme na nejnižší hladinu, připojujeme ke stromu pomocí červené hrany a v případě potřeby (červená hrana vedoucí doprava nebo 2 červené hrany nad sebou) rotujeme.
- Při cestě stromem dolů **štěpíme zcela zaplněné uzly** (4-vrcholy)
- Štěpení je realizováno pomocí **přebarvení** – tím se uzel rozštěpí a prostřední klíč se stane součástí nadřazeného vrcholu (víme jistě, že se tam vleze, protože všechny 4-vrcholy rovnou štěpíme).
- Na nejnižší úrovni vložíme uzel.
- Štěpení může zanechat ve stromu špatné konfigurace červených hran (červená hrana vedoucí doprava, nebo 2 červené hrany nad sebou) – opravujeme pomocí **rotací při cestě stromem zpět ke kořeni** (jednoduché při využití rekurze)

Tabulka s přímým přístupem (TPP)

- Implementace vyhledávací tabulky polem, ve které jsou klíče mapovány na indexy pole:
  - Ideální struktura z pohledu vyhledávání
  - Bohužel obvykle nerealizovatelná

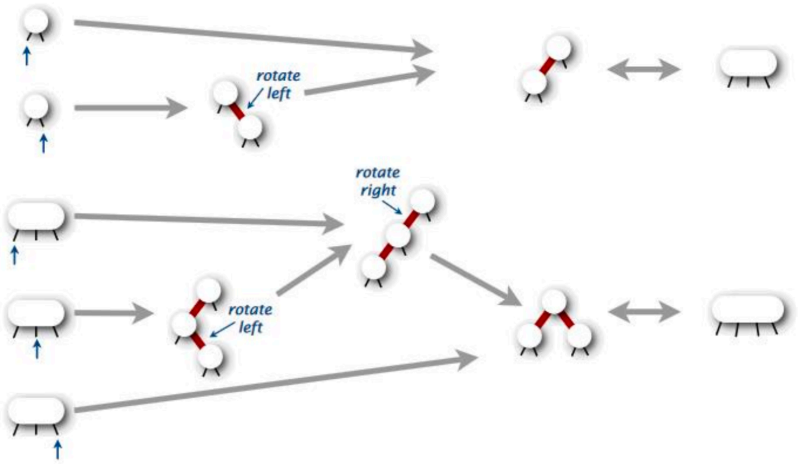


Figure 6: Vkládání v LLRB

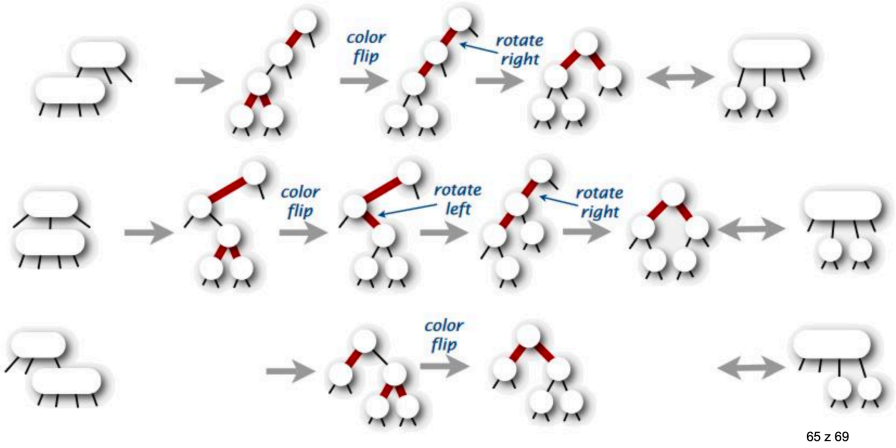


Figure 7: Vkládání v LLRB

Mazání v LLRB

- Mazání vnitřních uzlů se opět řeší náhradou hodnoty z vhodného uzlu na nejnižší hladině a smazáním tohoto uzlu – tedy mažeme buďto minimum z pravého podstromu nebo maximum z levého podstromu.
- Mazání minima: pokud do uzlu na nejnižší hladině vede červená hrana, lze smazat přímo (odpovídá to mazání klíče z 3-vrcholu).
- Problém: pokud v okolí vrcholu není žádná červená hrana (mazání klíče z 2-vrcholu – uzel by si musel půjčit klíč od souseda nebo se s ním spojit).
- Řešení: cestou stromem dolů provádíme úpravy tak, aby aktuální uzel nebyl 2-vrchol.
- Pomocí úprav mohou vzniknout nekorektní 3-vrcholy nebo 4-vrcholy – ty jsou upraveny při návratu z rekurze (cestou stromem zpět ke kořeni).

66 z 69

Figure 8: Mazání v LLRB

- Vyžaduje **vzájemně jednoznačné zobrazení (bijekce)** mapující každý prvek množiny klíčů K do množiny indexů pole H (sousedních adres v paměti).
- Vyhledávání: spočívá v přímém zjištění, zda na pozici klíče (indexu) dané tabulky je nebo není obsazeno.
- Časová složitost přístupu v TPP: $\mathbf{\Theta(1)}$
- **Obtíž**: nalezení **vhodné mapovací funkce**.

Mapovací funkce

- Nalezení vzájemně jednoznačného zobrazení (mapovací funkce) je velmi obtížné => je potřeba počítat s tím, že běžná mapovací funkce může **různým klíčům přiřadit stejnou hodnotu** (stejné místo v paměti).
- **Kolize** – dva různé klíče jsou namapovány do stejného místa.
- **Synonyma** – dva nebo více klíčů, které jsou namapovány do téhož místa.
- Nechť je dáno mapovací pole s rozsahem [0…N] nebo [1…N].
- Mapovací funkce transformuje klíč na index v daném rozsahu.
- Typicky lze rozdělit do dvou etap:
  - převod klíče na přirozené číslo (N>0),

– převod přirozeného čísla na hodnotu spadající do intervalu (nejčastěji s použitím operace modulo).

Mapovací funkce – požadavky

- **Determinismus** - Pro daný klíč vrátí vždy stejnou hodnotu.
- **Rovnoměrné (uniformní) rozložení** - Na každé místo se mapuje přibližně stejně velké množství klíčů.
- **Využití celých vstupních dat**
- **Vyhnutí se kolizím podobných klíčů** - V praxi bývá řada klíčů velice podobných.
- **Rychlý výpočet**

Ukázka mapovací funkce – BKDR

```
unsigned int BKDRHash(char* str, unsigned int length)
{
    unsigned int seed = 131;
    unsigned int hash = 0;
    unsigned int i = 0;
    for (i = 0; i < length; str++, i++)
    {
        hash = (hash * seed) + (*str);
    }
    return hash;
}
```

Ukázka mapovací funkce – DJB

```
unsigned long DJBHash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;
    while (c = *str++)
        // hash * 33 + c
        hash = ((hash << 5) + hash) + c;
    return hash;
}
```

Tabulka s rozptýlenými položkami

Tabulka s rozptýlenými položkami (TRP) sestává: - z mapovacího prostoru (pole) a - ze seznamů synonym.

Seznam synonym (i prázdný) začíná na každém prvku mapovacího pole. - **Explicitní zřetězení** – adresa následníka je obsažena v jeho předchůdci (zřetězení záznamů). - **Implicitní zřetězení** – adresa následníka se získá pomocí funkce z adresy předchůdce (otevřená adresace).

Princip vyhledávání v TRP spočívá ve dvou krocích: 1. **Nalezení indexu prvku v poli** k danému klíči pomocí mapovací funkce (na tomto indexu začíná seznam synonym, které se namapovaly do tohoto místa). 2. **Sekvenční průchod** tímto seznamem synonym (vyhledáváme položku s daným klíčem).

Vyhledávání v TRP má **index-sekvenční** charakter.

TRP s explicitním zřetězením synonym

- Seznam synonym je obvykle realizován jako **lineární seznam**.
- Maximální doba vyhledávání je pak dána délkou nejdelšího seznamu synonym – **O(n)**.
- Místo lineárních seznamů pro uložení synonym lze použít **vyvažované binární vyhledávací stromy**.
- Pak je časová složitost v nejhorším případě $\mathbf{O(\log_2 n)}$.

TRP s implicitním zřetězením synonym

- TRP implementovaná polem, ve kterém jsou uloženy jak první prvky seznamů synonym, tak jejich další položky.
- Pro přístup k synonymům existují různé metody pro určení kroku:
  - Lineární: `h(k, i) = (h(k) + C*i) % (Max+1)`
  - Kvadratická: `h(k, i) = (h(k) + C1*i + C2*i2) % (Max+1)`
  - S dvojí rozptylovací funkcí: `h(k, i) = (h1(k) + h2(k)*i) % (Max+1)`

kde i = 0, 1, 2, … – pokusy o vložení C, C1, C2– konstanty Max+1 – velikost pole

Implicitní zřetězení s pevným krokem

- Krok = 1: **a(i+1) = a(i) + 1**
- Konec seznamu synonym je dán **prvním volným prvkem**, který se najde se zadaným krokem.
- Nové synonymum se vloží na první volné místo (**na konec seznamu**).
- Tabulka (pole) musí obsahovat **alespoň jeden volný prvek**. Efektivní kapacita je o 1 menší než počet položek.
- Tabulka je implementovaná **kruhovým polem**.

Velikost rozptylovacího pole

- Krok s hodnotou 1 má tendenci vytvářet shluky (angl. **cluster**).
- Výhodnější je krok **větší než 1**.
- Kdyby měl krok hodnotu prvočísla, které je nesoudělné s jakoukoli velikostí pole, pak by mohl postupně projít všemi prvky pole.
- Výhodnější ale je, aby **hodnotu prvočísla měla velikost mapovacího pole**. Pak jakýkoli krok dovolí projít všemi prvky mapovacího pole.

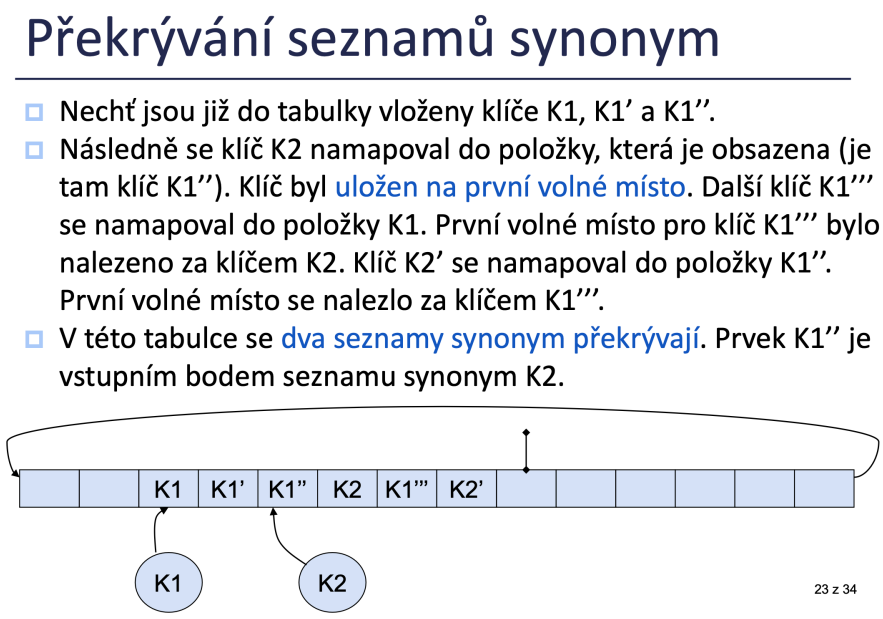


Figure 9: Překrývání seznamů synonym

- Je vhodné dimenzovat velikost mapovacího pole TRP tak, aby bylo rovno prvočíslu.

TRP s dvojí rozptylovací funkcí

Brentova varianta

- Brentova varianta je **varianta metody TRP se dvěma rozptylovacími funkcemi**.
- Brentova varianta provádí **při vkládání rekonfiguraci prvků** pole s cílem **investovat do vkládání** a získat lepší průměrnou dobu vyhledání.

Hodnocení TRP s implicitním zřetězením

- Operaci **Delete** lze řešit pomocí **zaslepení** – vložením klíče, který nebude nikdy vyhledáván.
- TRP s implicitním zřetězením je vhodná v aplikacích, v nichž se **operace Delete nepoužívá příliš často**.
- Maximální kapacita TRP pro rozsah pole <0..Max> je **Max** (o 1 menší než počet prvků pole) – alespoň jeden prvek musí zůstat jako **zarážka vyhledávání**.

TRP s dvojí rozptylovací funkcí

- Metoda s **dvojí rozptylovací** (hashovací) **funkcí**:
  krok v rozptylovacím poli je určen za běhu
  druhou rozptylovací funkcí.
- Nechť má rozptylovací pole rozsah <0..Max>, (kde hodnota Max+1 je prvočíslo) a nechť KInt je klíč transformovaný na celou nezápornou hodnotu.
- ***První rozptylovací funkce*** vrací index z intervalu <0..Max>:

$$\mathbf{ind_0 = h_1(KInt) = KInt \bmod (Max+1)}$$

- ***Druhá rozptylovací funkce*** vytváří krok z intervalu <1..Max>:

$$\mathbf{krok = h_2(KInt) = KInt \bmod Max + 1}$$

Figure 10: TRP s dvojí rozptylovací funkcí

Prvek K1 se přesune na první volné místo s krokem $h_2(K1)$
a na jeho místo se vloží prvek K.

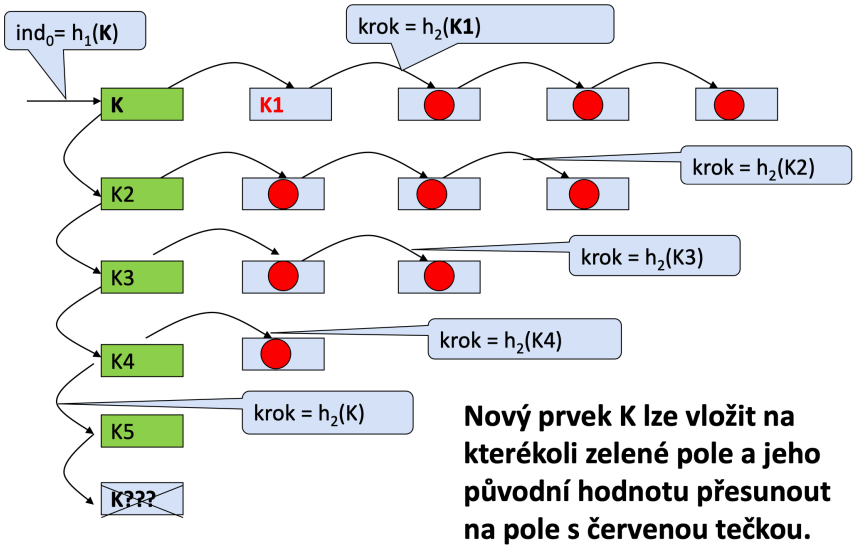


Figure 11: Brentova varianta

Hodnocení metod vyhledávání

Metoda vyhledávání	Časová složitost
Sekvenční vyhledávání	n
Binární vyhledávání v seřazeném poli	$log\ \ n$
Binární vyhledávací strom	$log\ \ n$
BVS při degradaci na seznam	n
Vyvažovaný BVS (např. AVL)	$log\ \ n$
TRP	1
TRP při maximální kolizi klíčů	n
TRP při maximální kolizi a vyváženém stromu	$log\ \ n$

Hashovací funkce

Vlastnosti: - Vstup **libovolné délky** transformuje na výstup **fixní délky** - **Determinismus** – pro stejný vstup vrací vždy stejný výstup - **Rychlost** – není výpočetně náročné funkci vyčíslit - **Malá změna na vstupu** (např. jednoho bitu) způsobí **velkou změnu na výstupu** (tzv. Avalanche Effect) - Navržena tak, aby měla **co nejméně kolizí**.

Využití hashovacích funkcí: - **Zajištění integrity dat** – kontrolní součty (sítě, archivy aj.) - **Zajištění nepopiratelnosti dat** – elektronický podpis = hash zprávy zašifrovaný privátním klíčem podepisujícího - **Zajištění důvěrnosti dat** - Ukládání hesel – operační systémy, informační systémy, aplikace (při přihlášení se hash zadaného hesla porovná s uloženým hashem) - Součást kryptografických protokolů (šifrovací protokoly TLS/SSL) - **Rychlá identifikace souborů** – souborové systémy, distribuované systémy, forenzní analýza digitálních dat - **Tabulky s rozptýlenými položkami** (hashovací tabulky)

Kryptografické hashovací funkce Aby byla funkce použitelná pro kryptografické účely, musí být výpočetně nezvládnutelné v „rozumném čase“: - Z výstupu spočítat původní vstup (**1st Preimage Resistance**) - Pro daný hash najít další vstup, který povede na stejný hash (**2nd Preimage Resistance**) - Najít dva vstupy, které povedou ke kolizi – stejnému hashi (**Collision resistence**)

Další typy hashovacích funkcí

- **Fuzzy hashing / Similarity hashing** – analýza podobnosti: Je naopak žádoucí, aby dva podobné vstupy měly podobný hash
  - SSDEEP, sdhash, TLSH
- **Klouzavé hashovací funkce** (rolling hash functions) – Efektivní výpočet hodnot posouvajícího se okna nad vstupními daty
  - Adler32, CRC, Rabin-Karpův hash, Spamsum
- **Percepční hashování** (perceptual hashing) – detekce podobných multimediálních souborů (obrázky, zvuk)

- pHash, dHash, aHash

Řazení

Třídění (**sorting**) položek neuspořádané množiny je uspořádání do tříd podle hodnoty daného atributu – klíče položky.

Řazení (**ordering**, **sequencing**) je uspořádání položek podle **relace lineárního uspořádání** nad klíči.

Setřídění (**merging**) je vytváření souboru seřazených položek sjednocením několika souborů položek téhož typu, které jsou již seřazeny.

Vlastnosti řadicích algoritmů

- **Přirozenost** – algoritmus se chová přirozeně pokud:
  - je doba potřebná k seřazení náhodně uspořádaného pole větší, než k seřazení již uspořádaného pole
  - a doba potřebná k seřazení opačně seřazeného pole je větší, než doba k seřazení náhodně uspořádaného pole.
  - Jinak říkáme, že se algoritmus nechová přirozeně.
- **Stabilita** vyjadřuje, zda mechanismus algoritmu zachovává relativní pořadí klíčů se stejnou hodnotou.

Řazení podle více klíčů

Problém lze řešit třemi způsoby: - Složená relace uspořádání - Opakované řazení - Aglomerovaný klíč

Aglomerovaný klíč Uspořádaná N-tice klíčů se konvertuje na vhodný typ, nad nímž je definována relace uspořádání.

Příklad aglomerovaného klíče: Rodné číslo

Řazení polí bez přesunu položek

V případě dlouhých položek jsou přesuny časově velmi náročné => řazení polí bez přesunu položek.

Implementace: - K řazenému poli vytvoříme **pomocné pole** (tzv. pořadník, location). - Po dokončení řazení pořadník udává, v jakém pořadí by měly být seřazeny položky původního pole (na první pozici pořadníku je index prvního prvku seřazeného pole atd.).

Chceme-li mít na konci **seřazené pole**: - Přeskládáme prvky do výstupního pole s využitím pořadníku. - Prvky zřetězíme a přeskládáme do výstupního pole, nebo přeskládáme v poli samotném.

Klasifikace algoritmů řazení

- Podle **přístupu k paměti**:
  - metody vnitřního řazení (**řazení polí**) – přímý (náhodný) přístup
  - metody vnějšího řazení (**řazení souborů a seznamů**) – sekvenční přístup
- Podle **typu procesoru**:
  - **sériové** (jeden procesor) – jedna operace v daném okamžiku
  - **paralelní** (více procesorů) – více souběžných operací
- Podle **principu** řazení:
  - Princip **výběru** (selection) – přesouvají maximum/minimum do výstupní posloupnosti.
  - Princip **vkládání** (insertion) – vkládají postupně prvky do seřazené výstupní posloupnosti.
  - Princip **rozdělování** (partition) – rozdělují postupně množinu prvků na dvě podmnožiny tak, že prvky jedné jsou menší než prvky druhé.
  - Princip **slučování** (merging) – setřiďují se postupně dvě seřazené posloupnosti do jedné.
  - Jiné principy …

Řazení na principu výběru (Select sort)

- Jádrem metody je nalezení extrémního prvku v zadaném segmentu pole a jeho výměna na konec (začátek) seřazené části pole.
- Takto je nalezeno *MAX-1* minim (maxim), která jsou umístěna na svoji pozici.

```
procedure SelectSort (TArray A)
    for i ← (0, MAX-2):
        indexMin ← i // Poloha pomocného minima
        min ← A[i] // Pomocné minimum
        for j ← (i+1, MAX-1):
            if min > A[j]:
                min ← A[j]
                indexMin ← j
        A[i] <-> A[indexMin]
```

- Metoda je **nestabilní**. Vyměněný první prvek se může dostat za prvek se shodnou hodnotou.
- Má **kvadratickou časovou složitost**.

Metoda bublinového výběru – Bubble sort

- Princip stejný jako u metody Select sort.
- Liší se metodou nalezení extrému a jeho přesunu:
  - Porovnává se každá dvojice a v případě obráceného uspořádání se přehodí.

```
procedure BubbleSort (TArray A)
    // průchod zprava - minimum doleva
    i ← 1
    do:
        finish ← true
        for j ← (MAX-1, i)-1: // bublinový cyklus
            if A[j-1] > A[j]:
                A[j-1] <-> A[j]
                finish ← false
        i ← i+1
    while (not finish) and (i < MAX)
```

```
procedure BubbleSort2 (TArray A)
    // průchod zleva - maximum doprava
    auxN ← MAX-1
    continue ← true
    while continue and (auxN > 0):
        continue ← false
        for i ← (0, auxN-1): // bublinový cyklus
            if A[i+1] < A[i]:
                A[i+1] <-> A[i]
                continue ← true // výměna - nelze skončit
        auxN ← auxN-1
```

- Bublinový výběr je metoda **stabilní** a přirozená. Je to jedna z mála metod použitelná pro vícenásobné řazení podle více klíčů!
- Má časovou složitost **kvadratickou**.
- Je to nejrychlejší metoda v případě, že pole je již seřazené!

Bubble sort – varianty Od Bubble sortu byla odvozena řada vylepšených variant: - **Ripple sort**: pamatuje si polohu první výměny a je-li větší než 1, neprochází dvojicemi, u nichž je jasné, že se nebudou vyměňovat. - **Shaker sort**: střídá směr probublávání zleva a zprava (používá houpačkovou metodu) a skončí uprostřed. - **Shuttle sort**: zavede při výměně dvojice menší prvek na své místo a teprve pak pokračuje dál. Končí tím, že nevymění nejpravější dvojici.

Řazení hromadou – Heap sort

Hromada (halda, heap) je struktura stromového typu, pro niž platí, že mezi otcovským uzlem a všemi jeho synovskými uzly platí **stejná relace uspořádání**.

Nejčastější případ hromady je **binární hromada**, která je založená na binárním stromu, pro který navíc platí: - Všechny hladiny kromě poslední jsou plně obsazené. - Poslední hladina je zaplněna zleva.

Rekonstrukce hromady Významnou operací nad hromadou je její **rekonstrukce** poté, co se poruší pravidlo hromady v jednom uzlu.

Nejvýznamnějším případem je porušení v kořeni.

Operace **Sift** (prosetí nebo také zatřesení hromadou): - Operace, která znovuustaví hromadu porušenou v kořeni. - Prvek z kořene se postupnými výměnami **propadne** na své místo a do kořene se dostane prvek splňující pravidla hromady. - Operace má v nejhorším případě složitost $\mathbf{\log_2 n}$.

Implementace hromady polem Protože musí být zaplněny všechny hladiny kromě poslední a poslední musí být zaplněna zleva, můžeme strom ukládat do pole **po hladinách**.

Pak **platí pro otcovský a synovské uzly vztah**: když je otcovský uzel na indexu *i*, pak je levý syn na indexu *2i+1* a pravý syn na indexu *2i+2*.

Vytvoření hromady

- Začneme s **nejnižším a nejpravějším otcovským uzlem** – ten je kořenem hromady (podstromu), která je porušená v kořeni. Operací Sift opravíme.
- Dále **postupujeme po všech otcovských uzlech doleva a nahoru** až k hlavnímu kořeni.

Má-li pole *MAX* prvků (indexováno od *0* do *MAX-1*), pak nejnižší a nejpravější otcovský uzel odpovídající hromady má index: *(MAX div 2) – 1*. Následující otcovské uzly leží na **předchozích** indexech.

Celkem musíme opravit n/2 hromad, celé ustavení hromady zvládneme v čase **n/2*log$_2$n**.

```
procedure HeapSort (TArray A)
    // ustavení hromady
    left ← (MAX div 2)-1 // nejnižší a nejpravější otec
    right← MAX-1
    for i ← (left, 0):
        SiftDown(A,i,right)

    // vlastní cyklus Heap-sortu
    for right ← (MAX-1, 1):
        A[0] A[right]
        // výměna kořene s akt. posledním prvkem
        SiftDown(A,0,right-1) // znovuustavení hromady

procedure SiftDown (TArray A, int left, int right)
    // left je index kořenového uzlu, který porušuje heap,
    // right je index posledního prvku heapu
    i ← left
```

```
j ← 2*i+1 // index levého syna
temp ← A[i] // pomocná proměnná
continue ← j   right // řídicí proměnná cyklu
while continue:
    if j < right: // uzel má oba syny
        if A[j] < A[j+1] // pravý syn je větší
            j← j+1 // pokračujeme tedy s ním
    if temp   A[j]: // temp našel své místo = konec
        continue ← false
    else: // temp padá níž, A[j] jde o úroveň výš
        A[i]← A[j]
        i ← j // syn je otcem v dalším cyklu
        j← 2*i+1 // nový levý syn
        continue ← j   right // pokračujeme až na list
A[i]← temp // konečná pozice „propadajícího" kořene
```

Zhodnocení

- Heap sort je řadicí metoda s **linearitmickou složitostí**, protože sift umí rekonstruovat hromadu (najít extrém mezi N prvky) s logaritmickou složitostí.
- Heap sort je **nestabilní** a **nechová se přirozeně**.