- Úvod
- Lineární abstraktní datové typy
- Zásobník a fronta
- Stromové datové struktury
- Vyhledávací tabulky
- Řazení

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, Ơ, 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.

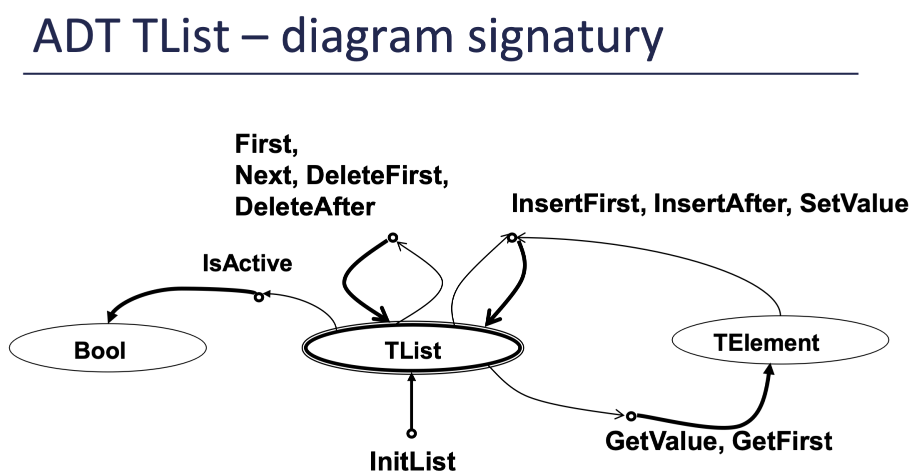


Figure 1: ADT TList

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.

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

ADT TStack – diagram signatury

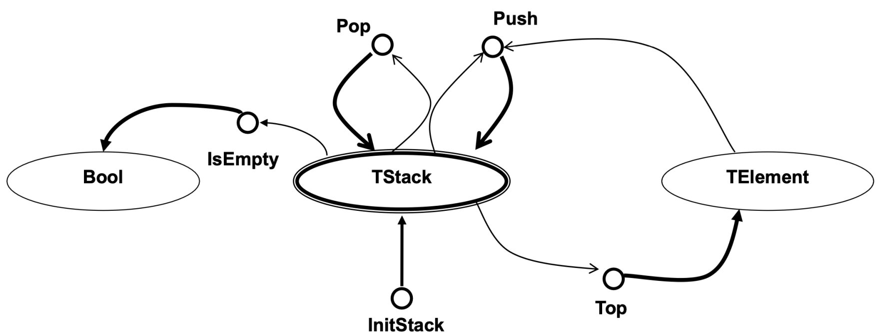


Figure 2: ADT TStack

ADT TQueue – diagram signatury

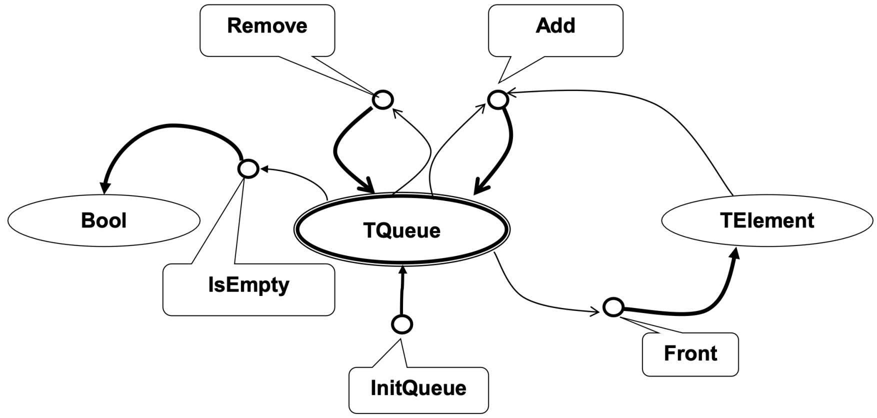


Figure 3: ADT TQueue

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 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).

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

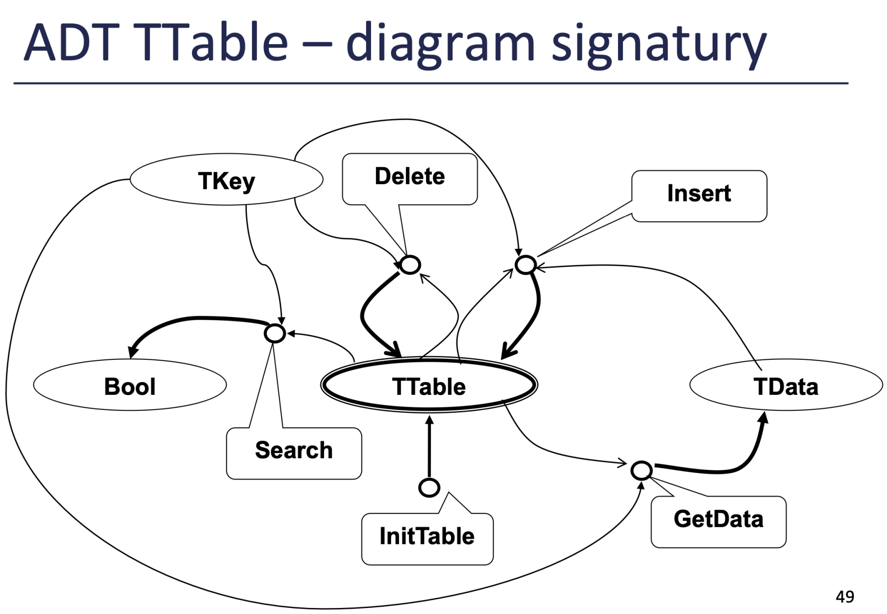


Figure 4: ADT TTable

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

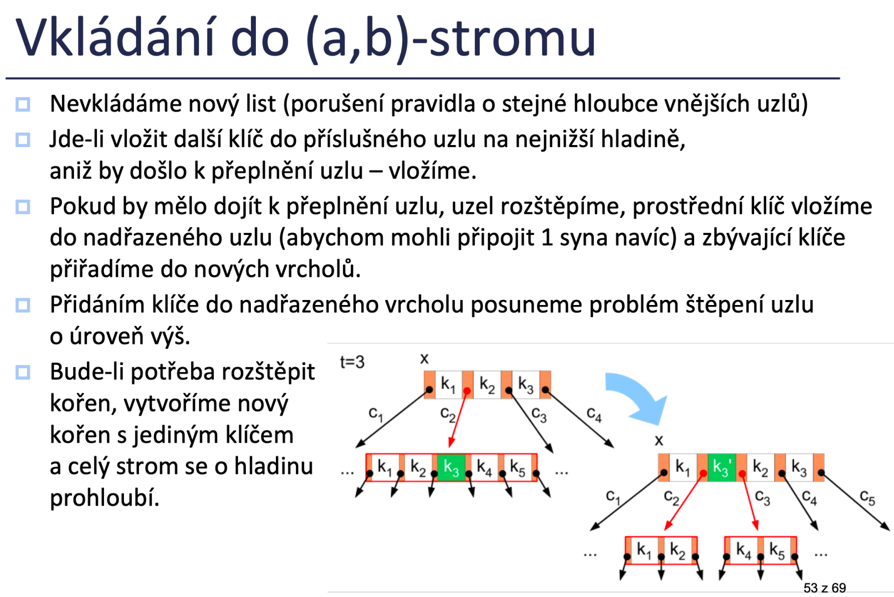
Pokud v rámci operace Insert nebo Delete dojde ke změně váhy na hodnotu -2/2, je potřeba situaci napravit. Mohou nastat 4 různé situace, které se napravují různými způsoby: - **LL**: kritický uzel je příliš těžký vlevo a jeho levý syn je těžký vlevo - **LR**: kritický uzel je příliš těžký vlevo a jeho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



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

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)

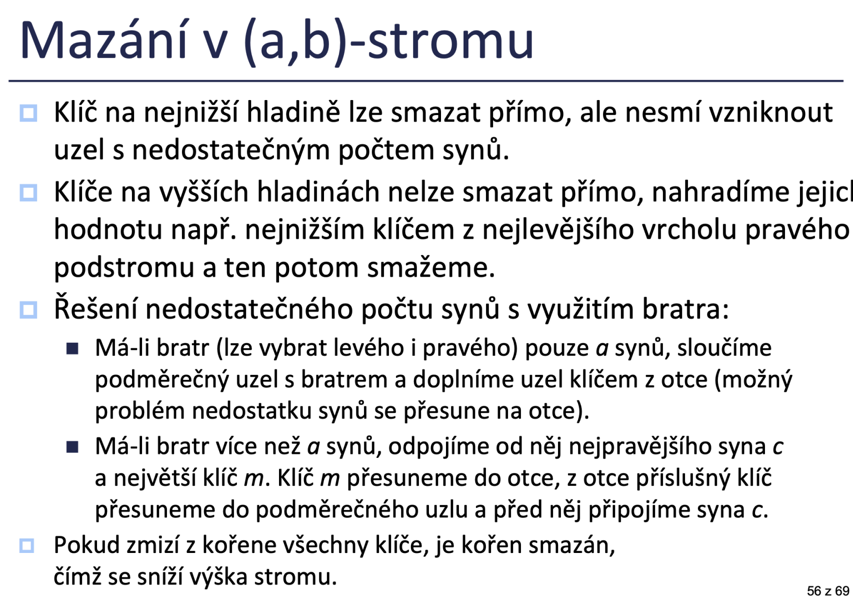


Figure 5: Mazání v (a,b)-stromu

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)

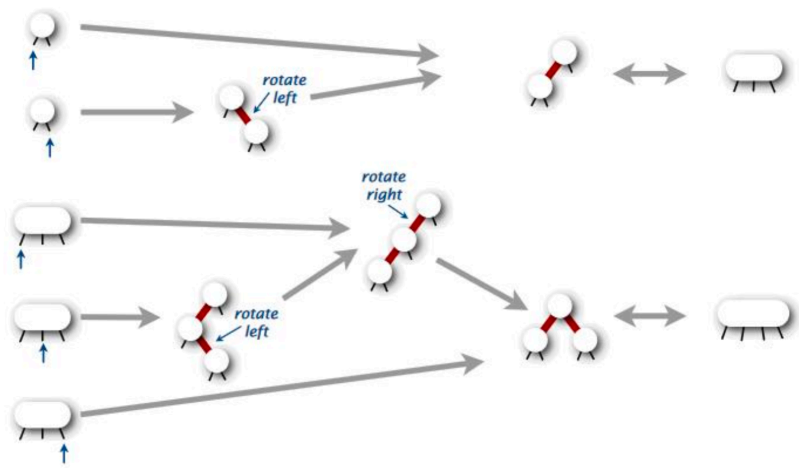


Figure 6: Vkládání v LLRB

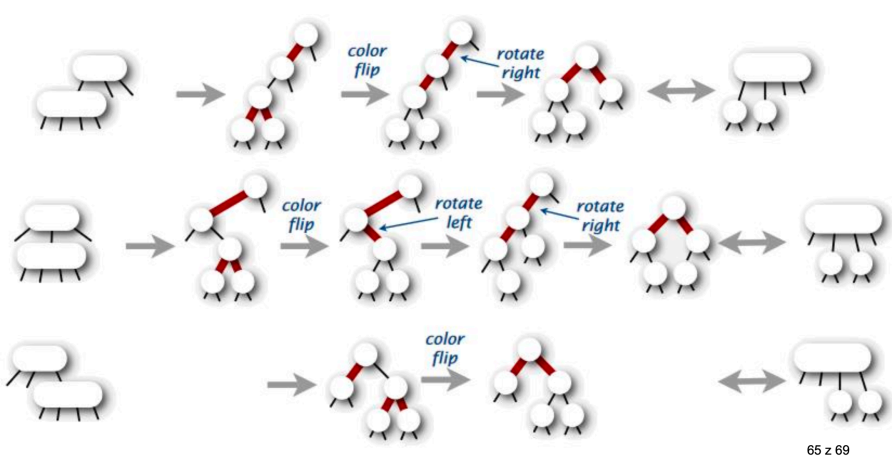


Figure 7: Vkládání v LLRB

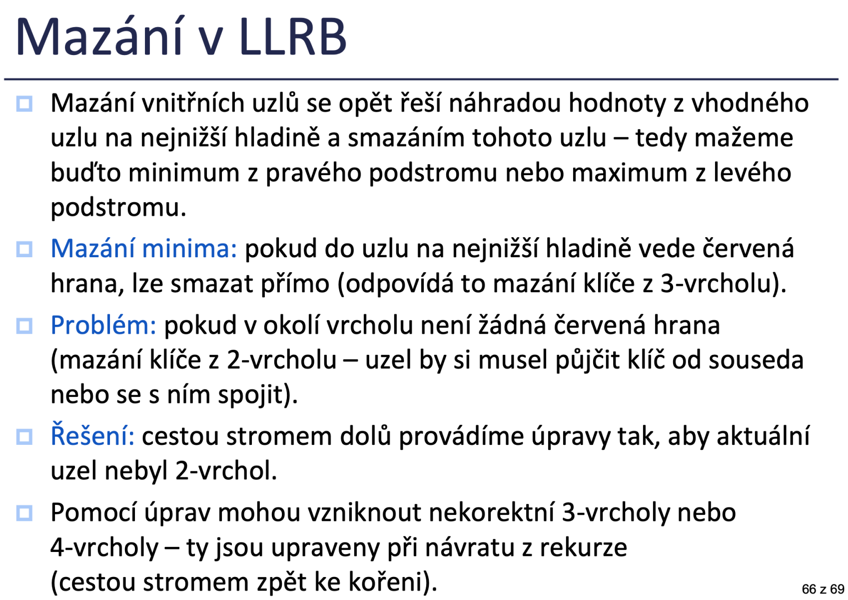


Figure 8: Mazání v LLRB

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
- 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

```c
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

```c
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 – $\mathbf{O(n)}$.
- 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: $\mathbf{a(i+1) = a(i) + 1}$
- 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**).

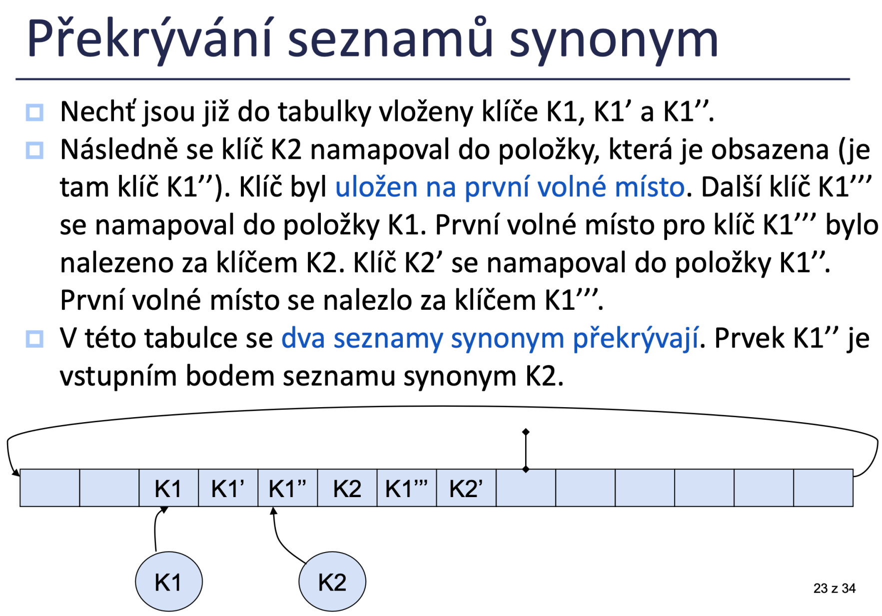


Figure 9: Překrývání seznamů synonym

- 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.
- 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**.

Figure 10: TRP s dvojí rozptylovací funkcí

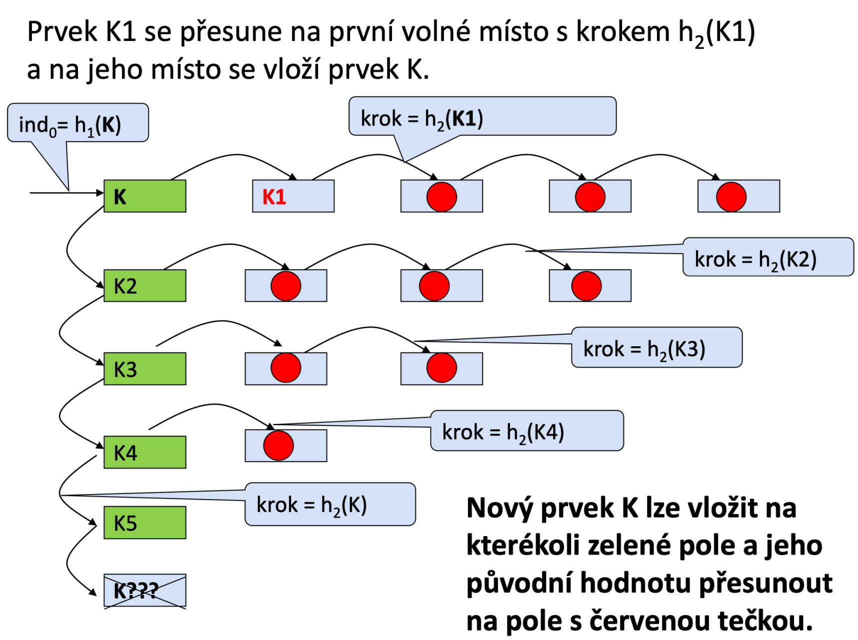


Figure 11: Brentova varianta

- 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**.

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.

Select sort

```
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**.

Další využití hromady Prioritní frontu lze implementovat binární hromadou. V kořeni bude vždy prvek s maximální/minimální prioritou.

Řazení na principu vkládání

Pole dělíme na dvě části: - Levou – seřazenou a pravou – neseřazenou. - Levou část tvoří na začátku první prvek.

Algoritmus řazení má následující strukturu:

```
for i ← (1, MAX-1):
// najdi v levé části index k, kam se má zařadit prvek A[i]
// posuň část pole od k do i-1 o jednu pozici doprava
// vlož na A[k] hodnotu zařazovaného prvku
```

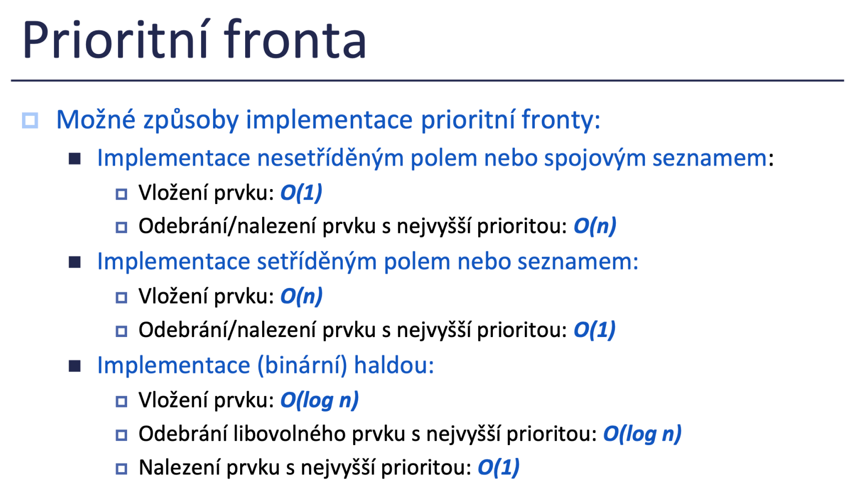


Figure 12: Prioritní fronta

Bubble-insert sort

Kombinuje vyhledání místa pro vkládání i posun segmentu pole do jednoho cyklu postupným porovnáváním a výměnou dvojic prvků.

```
procedure BubbleInsertSort (TArray A)
    for i ← (1, MAX-1):
        tmp ← A[i]
        j ← i-1
        while j   0 and tmp < A[j]: // zkrat. vyhodnocování!
            // najdi a posuň prvek
            A[j+1]← A[j]
            j← j-1
        A[j+1] ← tmp // konečné vložení na místo
```

- Metoda je **stabilní** – je vhodná pro vícenásobné řazení podle více klíčů.
- Chová se **přirozeně** a pracuje **in situ**.
- Má **kvadratickou** časovou složitost.

Binary-insert sort

Pro vložení prvku vyhledáváme místo v **seřazené posloupnosti** – lze využít **binární vyhledávání**.

V případě shodných klíčů musí metoda nalézt místo **za nejpravějším ze shodných klíčů** – varianta Dijkstrovy metody binárního vyhledávání.

```
procedure BinaryInsertSort (TArray A)
```

```
for i ← (1, MAX-1):
    tmp ← A[i]
    // hranice již seřazené části
    left ← 0
    right← i-1
    while left  right:
        m ← (left+right) div 2 // binární vyhledání
        if tmp < A[m]:
            right← m-1
        else:
            left← m+1 // ale skončíme těsně za
    for j ← (i-1, left):
        A[j+1]← A[j] // posun segmentu pole doprava
    A[left] ← tmp // prvek z pozice i na své místo
```

- Metoda je **stabilní**.
- Chová se **přirozeně** a pracuje **in situ**.
- Má **kvadratickou** časovou složitost.

Řazení rozdělováním

Quick sort

Představme si algoritmus, který umí (rychle) rozdělit množinu položek na **dvě podmnožiny**: - jedna by obsahovala všechny **prvky s klíčem menším** (nebo rovným) jisté hodnotě - druhá by obsahovala všechny **prvky s klíčem větším** (nebo rovným) téže hodnotě

Mechanismu rozdělení říkáme **partition**.

Medián – prvek z množiny hodnot, pro který platí: - Polovina prvků je menší než medián. - Polovina prvků je větší než medián.

Při znalosti mediánu je snadné implementovat proceduru `partition`, která rozdělí pole na dvě části: - Procházíme pole současně zleva (index i) a zprava (index j). - Zleva hledáme prvek větší nebo roven mediánu, zprava prvek menší nebo roven mediánu. - Nalezené prvky vyměníme a hledáme další prvky pro výměnu. - Proces ukončíme až se dvojice indexů překříží. - ia jjsou návratové hodnoty funkce, definující intervaly *left..j* (prvky menší nebo rovny mediánu) a *i..right* (prvky větší nebo rovny mediánu).

Mechanismus partition I. Protože hledání mediánu je časově náročné, použijeme tzv. **pseudomedián**: - Libovolná hodnota z daného souboru čísel - Vhodnou hodnotou je číslo ze středu intervalu: `(left+right) div 2` - Experimentálně je prokázáno, že toto číslo splní svou roli velmi podobně jako medián.

Abychom nemuseli při hledání hodnot pro výměnu kontrolovat hranice pole, používáme pseudomedián jako **zarážku**.

```
(int, int) function partition (TArray A, int left, int right)
    i ← left // inicializace i
    j ← right // inicializace j
    PM ← A[(i+j) div 2] // ustavení pseudomediánu
    do
        while A[i] < PM:
            i ← i+1 // první i zleva, pro A[i]>=PM
        while A[j] > PM:
            j ← j-1 // první j zprava pro A[j]<=PM
        if i   j:
            A[i] <-> A[j] // výměna nalezených prvků
            i ← i+1
            j ← j-1
    while i   j
    return (i, j)

procedure QuickSort (TArray A, int left, int right)
    // Při volání má left hodnotu 0 a right hodnotu MAX-1
    i, j ← partition(A, left, right)
    if left < j:
        QuickSort(A, left, j) // Rekurze doleva
    if i < right:
        QuickSort(A, i, right) // Rekurze doprava
```

Mechanismus partition II.

- Jako pseudomedián (pivot) je volen nejpravější prvek
- Pole procházíme postupně zleva doprava (index j) a ve zpracované části udržujeme vlevo prvky menší nebo rovny pivotu (do indexu i) a vpravo prvky větší než pivot.
- Vždy když narazíme na prvek menší než pivot, vyměníme ho s prvním prvkem, který je větší než pivot
- Nakonec je pivot vyměněn s prvním prvkem části pole s prvky většími než pivot.
- Partition vrací index nové pozice pivota, rekurzivní volání pokračují vlevo a vpravo od tohoto prvku
- Méně efektivní než předchozí mechanismus.

```
int function partitionII (TArray A, int left, int right)
    i ← left - 1
    PM ← A[right] // ustavení pseudomediánu
    for j ← (left, right-1): // projdi pole zleva
        if A[j]   PM: // menší musí do levé části
            i ← i+1 // za poslední prvek
            A[i]   A[j] // výměna nalezených prvků
    A[i+1]   A[right]
```

```
    return i+1

procedure QuickSortII (TArray A, int left, int right)
    if left < right:
        q ← partitionII (A, left, right)
        QuickSortII(A, left, q-1) // rekurze doleva
        QuickSortII(A, q+1, right) // rekurze doprava
```

Quick sort – nerekurzivní zápis

```
procedure NonRecQuicksort (TArray A, int left, int right)
    InitStack(s)
    Push(s,left) // uložení hranic celého pole
    Push(s,right)
    while not IsEmpty(s): // vnější cyklus
        right ← Top(s)
        Pop(s) // čtení v opačném pořadí
        left ← Top(s)
        Pop(s)
        while left < right: // dokud je co dělit
            i,j ← Partition(A, left, right)
            Push(s,i) // interval pravé části do zás.
            Push(s,right)
            right ← j // pravý index pro další cyklus
```

Zhodnocení

- Quick sort patří **mezi nejrychlejší** algoritmy pro řazení polí.
- Quick sort je **nestabilní** a **nepracuje přirozeně**.
- Časová složitost je **linearitmická** pro vhodně zvolený pseudomedián.
- V nejhorším případě – při špatné volbě pseudomediánu (vždy minimum nebo maximum), je časová složitost **kvadratická**.
    - zlepšení volby pseudomediánu – výběr mediánu ze tří náhodně vybraných hodnot

Shell sort

Metoda řazení se **snižujícím se přírůstkem**. Metoda používá **opakované průchody polem**, ve kterých řadí vždy jen určitou **podposloupnost** původní sekvence: - Původní sekvence je rozdělena na několik podposloupností, do kterých jsou vybrány **prvky vzdálené od sebe o určitý krok**. - Prvky v každé podsekvenci jsou uspořádány jedním **bublinovým průchodem** (Bubble-insert sort). - Po seřazení všech podposloupností se **krok zmenší** a **opakuje se řazení** pro nové podsekvence. - V **poslední etapě** řazení je krok roven jedné, všechny prvky jsou v jedné podposloupnosti, a řazení je dokončeno posledním bublinovým průchodem.

```
procedure ShellSort (TArray A)
    step ← MAX div 2 // první krok - polovina délky pole
    while step > 0:
        for i ← (step, MAX-1): // cykly pro paralelní n-tice
            j ← i-step
            while (j   0) and (A[j] > A[j+step]): //bubl.ins.
                A[j]   A[j+step]
                j← j-step // snížení indexu o krok
        step ← step div 2 // půlení kroku
```

- Shell sort je **nestabilní** metoda.
- Pracuje **in situ**.
- Časová složitost závisí na zvolené řadě snižujících se kroků:
  - Pro uvedenou verzi (n/2, n/4, …, 1) je v nejhorším případě časová složitost $\mathbf{n^2}$.
  - Existují řady, pro které je časová složitost $\mathbf{n^{3/2}}$ nebo **n*log$_2$ n**.

Řazení na principu slučování

Řazení setřiďováním – Merge sort

- Pole rozdělujeme do tzv. běhů – souvislých **úseků, které už jsou setříděny (seřazeny)**.
- Na začátku budou všechny běhy jednoprvkové.
- Poté budeme dohromady **slévat vždy dva sousední** běhy do jediného setříděného běhu o délce dané součtem počtu prvků slévaných běhů, který bude ležet na místě obou vstupních běhů.
- Po poslední iteraci bude posloupnost sestávat z jediného běhu, a bude tudíž setříděná (seřazená).

Metoda **vyžaduje pomocné (nebo pomocná) pole**, pro uložení setříděné posloupnosti (nebo uložení vstupních posloupností).

Rekurzivní varianta – metoda postupně volá sebe sama pro levou a pravou polovinu zadané části pole a při návratu z rekurze slévá již setříděné posloupnosti

```
procedure MergeSort (TArray A, int left, int right)
    // Při volání má left hodnotu 0 a right hodnotu MAX-1
    if (left < right):
        q ← (left + right) div 2
        MergeSort(A, left, q)
        MergeSort(A, q+1, right)
        Merge(A, left, q, right)
```

```
procedure Merge (TArray A, int left, int mid, int right)
    left_count ← mid - left + 1 // počet prvků levé posloupnosti
    right_count ← right - mid // počet prvků pravé posloupnosti
    for i ← (0,left_count-1): // levá posloupnost do pom. pole
        L[i] ← A[left+i]
```

```
for j ← (0,right_count-1):
    R[j] ← A[mid+1+j] // začíná na indexu mid+1
L[left_count] ← MaxInt // ustavení zarážek
R[right_count] ← MaxInt
i ← 0
j ← 0
for k ← (left,right): // slévání a vkládání do pole A
    if L[i]   R[j]: // vyber menšího
        A[k] ← L[i] // menší byl v levé posloupnosti
        i ← i+1
    else:
        A[k] ← R[j]  // menší byl vpravo
        j ← j+1
```

- Jedná se o **stabilní** metodu.
- Potřebuje pomocné pole o stejné velikosti jako je zdrojové pole – tzn. **nepracuje in situ**.
- Časová složitost je **linearitmická**.

Sequence-merge sort

Řazení **setřiďováním posloupností** – sekvenční metoda využívající přímý přístup k prvkům pole. - Postupuje polem zleva a současně zprava a setřiďuje dvě **proti sobě** postupující neklesající posloupnosti. Výsledek se ukládá do cílového pole. - Počet vzniklých posloupností se počítá v počitadle. - Algoritmus **končí**, vznikne-li jen **jedna cílová posloupnost**.

Zhodnocení

- Významným rysem algoritmu je jeho **houpačkový mechanismus**:
  - automaticky střídá pozici zdrojového a cílového pole i krok postupující proti sobě orientovanými slučovanými neklesajícími posloupnostmi.
- Metoda Sequence-merge sort je **nestabilní**.
- **Nechová se přirozeně** a **nepracuje in situ**.
- Asymptotická časová složitost je **linearitmická**.

List-merge sort

- Řazení polí **setřiďováním seznamů** – pracuje na principu **slučování** metodou **bez přesunů položek**.
- K základnímu poli je nezbytné vytvořit stejně velké pomocné pole *Ptr* indexových ukazatelů, které **zřetězí neklesající posloupnosti**.
- Jádrem algoritmu je **setřídění dvou seznamů** zřetězených v pomocném poli indexovými ukazateli.

Princip **První krok**: zřetězení neklesajících posloupností do seznamu a vložení jejich začátků do dvojsměrného seznamu začátků.

Následující **cyklus**: - V každé iteraci se vyzvednou ze seznamu začátky **dvou** zřetězených neklesajících posloupností. - **Setříděním** těchto posloupností vznikne jedna zřetězená neklesající posloupnost, jejíž začátek se vloží na konec seznamu. - Cyklus se **ukončí**, je-li v seznamu již jen začátek jedné neklesající zřetězené posloupnosti.

Zhodnocení

- List-merge sort je algoritmus pracující **bez přesunu položek**.
- Je **potenciálně stabilní**.
- Stabilita se zajistí např. tím, že se začátky vkládají do dvojsměrného seznamu (na pozici vyjmutých začátků) a při setřiďování se u shodných prvků musí do výstupní posloupnosti vložit prvek první posloupnosti.

Tim sort

- **Kombinuje** *Merge sort* a *Insert sort*.
- *Merge sort* je použit na **setřiďování neklesajících posloupností**.
- Pokud jsou neklesající posloupnosti příliš **krátké**, jsou metodou *insert sort* prodlouženy.
- Jsou setřiďovány vždy dvě sousední podposloupnosti – **stabilní** metoda.
- Nalezené/vytvořené podposloupnosti nemusí být setříděny hned, ale mohou být odloženy na **zásobník**. Díky tomu dochází k setřídění podobně dlouhých podposloupností.
- Použití dalších technik pro **zlepšení výkonnosti**:
  - Binary-search (pro nalezení první/poslední pozice, které se dotkne vkládání),
  - galloping mode (při vkládání více prvků za sebou ze stejné podposloupnosti),
  - detekce klesajících posloupností,
  - velikost běhů atd.
- Časová složitost je **linearitmická**, **nepracuje in situ**.

Řazení tříděním podle základu

Řazení tříděním podle základu je počítačová verze procesu řazení na děrnoštítkových třídicích strojích.

Řazení tříděním podle základu je jednou z verzí tzv. *přihrádkového třídění* (bucket sort), které lze použít i na jiné než číselné klíče.

Řazení tříděním lze implementovat tak, aby šlo o metodu pracující **bez přesunu položek**.

Radix sort

Radix sort využívá pomocné datové struktury: - Seznamy (příp. fronty) prvků pro stejnou cifru. - Pole pro uchování začátků jednotlivých seznamů.

Implementace s využitím přihrádek (např. s využitím jednosměrných seznamů). Po každém roztřídění prvků jsou prvky znovu spojeny do jedné posloupnosti.

```
Vstupní posloupnost vlož do seznamu S
for j← (1, POCCIF) do
    // inicializace přihrádek
    // třídění prvků ze seznamu S do přihrádek dle j-té číslice
    // vytvoření prázdného seznamu S
    // postupné připojení všech přihrádek do seznamu S
end for
```

Hodnocení

- Radix sort je stabilní metoda.
- Stav uspořádání nemá podstatný vliv na čas a proto se jeví, jako by se nechoval přirozeně.
- Metoda nepracuje in situ.
- Časová složitost je linearitmická.

Zhodnocení řadicích metod

Algoritmus	Časová složitost	Pomocná paměť	Stabilita
Bubble sort	$O(n^2)$	-	ano
Heap sort	O(n log n)	-	ne
Insert sort	$O(n^2)$	-	ano
Quick sort	O(n log n)	O(log n)	ne
Shell sort	$O(n^2)$	-	ne
Merge sort	O(n log n)	Θ(n)	ano
Radix sort	O(n log n)	Θ(n)	ano

Pozn.: Quick sort má uvedenou složitost pro vhodně zvolený medián.

Figure 13: Zhodnocení řadicích metod