IAL – 4. přednáška

Stromové datové struktury

Vyhledávací tabulky I.

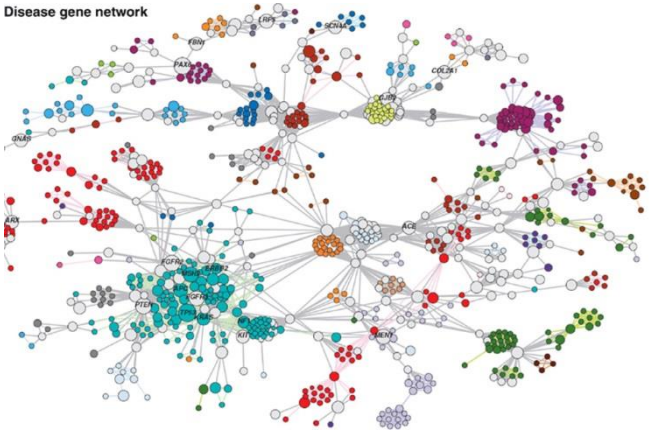
13. a 19. října 2021

Obsah přednášky

- Stromové datové struktury
  - Kořenový strom
  - Binární strom
- Vyhledávací tabulky
  - Hodnocení a klasifikace metod
  - Sekvenční vyhledávání

Nelineární datové typy

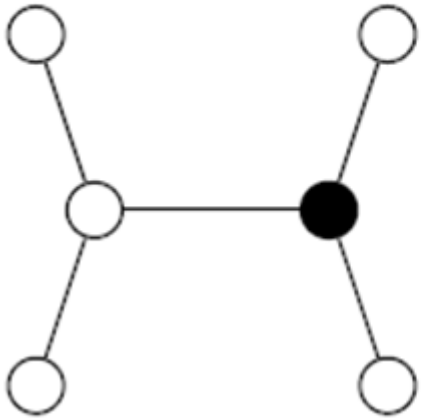
- Data nejsou uspořádána postupně – obecně jeden prvek může být připojen k libovolnému počtu prvků.
- Tyto struktury umožňují modelovat složitější vztahy mezi daty.
- Typické nelineární struktury:
  - Graf
  - Kořenový strom (speciální případ grafu)
  - Binární strom
    (speciální případ kořenového stromu)
  - Halda (speciální případ kořenového stromu)

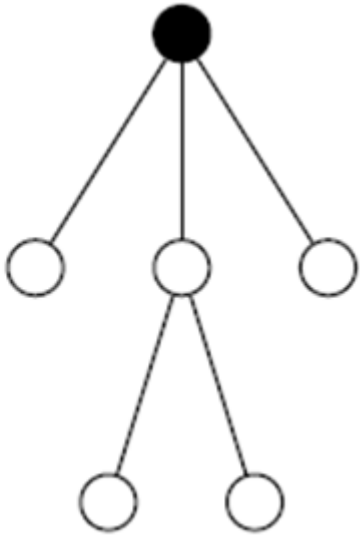


- *Pozn.:* Grafům bude věnována samostatná přednáška v závěru semestru.

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.
- Z každého uzlu vede jen jedna hrana směrem ke kořeni do uzlu, kterému se říká *otcovský* uzel, a libovolný počet hran k uzlům, kterým se říká *synovské*.
- Uzly bez potomků označujeme jako *listy* (listové uzly), uzly s potomky jako *vnitřní uzly*.





Vlastnosti stromů

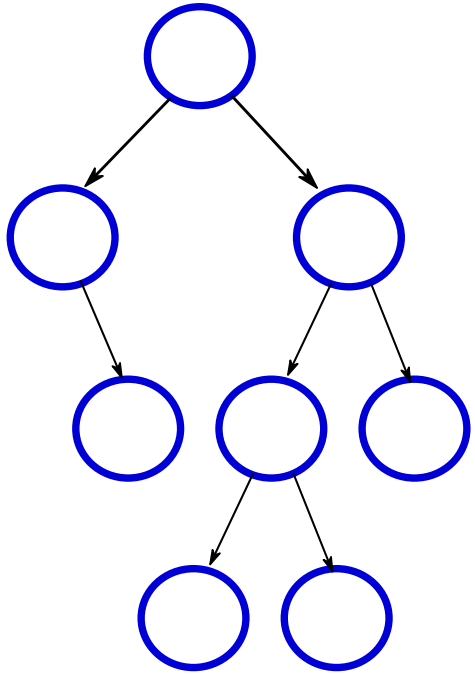
- **Stupeň uzlu *u*** – počet potomků uzlu *u*.

  Často definujeme maximální možný počet potomků – speciální typy stromů:

  - Binární stromy
  - 2-3 stromy
  - B-stromy
- **Seřazený strom** – je kořenový strom, ve kterém jsou potomci každého uzlu mezi sebou seřazeni
- **Cesta k uzlu *u*** – posloupnost všech uzlů od kořene k uzlu *u*
- **Délka cesty** – počet hran, které cesta obsahuje (počet uzlů-1)
- **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.
- **Průchod stromem** – posloupnost všech uzlů stromu, v níž se žádný uzel nevyskytuje dvakrát

Binární strom (BS)

- **Rekurzivní definice binárního stromu:**
  Binární strom je buď prázdný, nebo sestává z jednoho uzlu zvaného kořen a dvou binárních podstromů – levého a pravého.



Vlastnosti binárních stromů

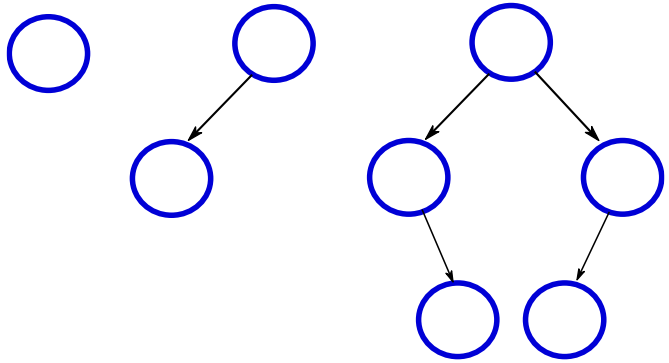
- Binární strom sestává z:
  - kořene,
  - neterminálních (vnitřních) uzlů, které mají ukazatel na jednoho nebo dva uzly synovské a
  - terminálních uzlů (listů), které nemají žádné *potomky*.

Vlastnosti binárních stromů

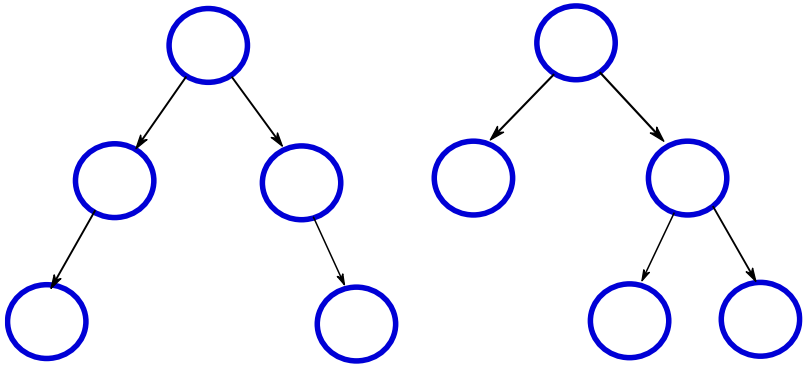
- Vyváženost stromu:
  - Binární strom je váhově vyvážený, když pro každý jeho uzel platí, že počty uzlů jeho levého a pravého podstromu se rovnají a nebo se liší právě o 1.
  - Binární strom je výškově vyvážený, když pro každý jeho uzel platí, že výška levého podstromu se rovná výšce pravého podstromu a nebo se liší právě o 1.
  - Maximální výška vyvážených stromů: ***c.log(n)***
  - Při zajištění vyváženosti nemůže dojít k degradaci stromu na seznam.

Příklad: (ne)vyvážené stromy

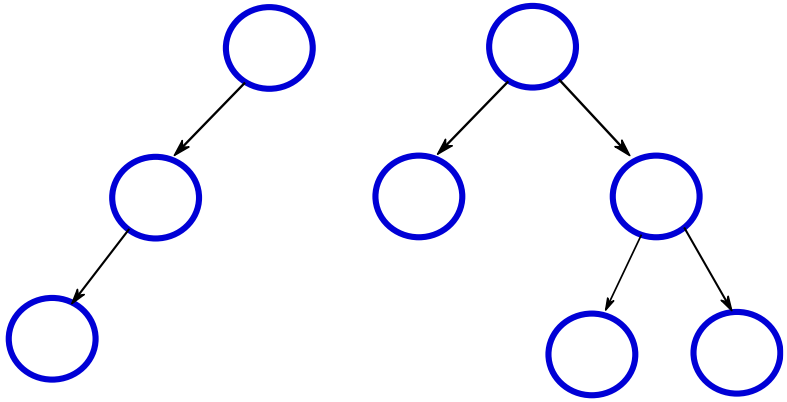
- Váhově vyvážené stromy



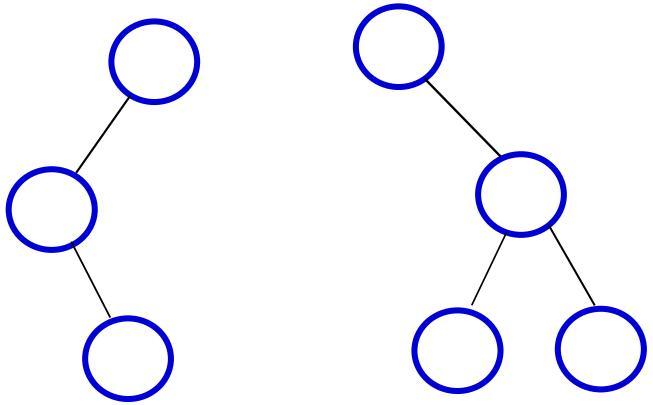
- Výškově vyvážené stromy



- Váhově nevyvážené stromy



- Výškově nevyvážené stromy



Operace nad binárním stromem

- Má smysl zavádět obecný ADT binární strom?
  - Museli bychom zavést mnoho operací, které by umožňovaly manipulaci s daty v libovolném uzlu stromu (mnoho možností, příliš složité řešení).
  - ADT (operace) zavedeme až pro konkrétní typ stromu (např. binární vyhledávací strom), kdy způsob použití tohoto stromu omezí počet operací, které budeme potřebovat.

Operace nad binárním stromem

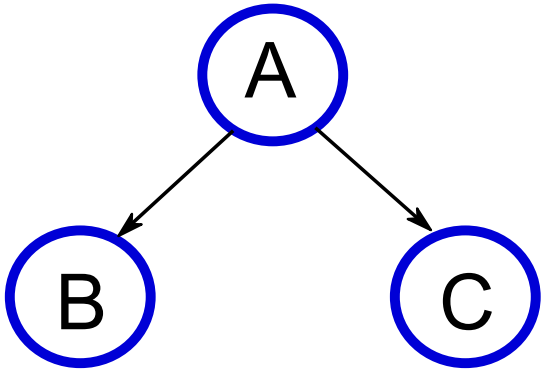
- Vkládání
  - Vložený uzel je třeba navázat na jeho otce a případně na vkládaný uzel správně navázat jeho syny.
- Rušení
  - Rušení listu (jednoduché, korekce ukazatele v otci).
  - Rušení uzlu s jedním synem (také jednoduché, korekce ukazatele v otci).
  - Rušení uzlu s dvěma syny (obtížnější).
- Vkládání/rušení
  - Může porušit uspořádanost nebo vyváženost stromu.
  - Konkrétní způsoby implementace těchto operací probereme u jednotlivých typů stromů.

Operace nad binárním stromem

- Průchody stromem (základ mnoha dalších algoritmů):
  - Do šířky (level-order)
  - Do hloubky (pre-order, in-order, post-order)
- Další operace dle typu a určení stromu:
  - Binární vyhledávací strom jako vyhledávací tabulka
  - operace **InitTable**, **Insert**, **Search**, **GetData**, **Delete**
- Další možné operace nad BS:
  - výška BS
  - ekvivalence (struktur) dvou BS
  - kopie BS
  - zrušení BS
  - váhová/výšková vyváženost stromu

Procházení BS do hloubky

- Mějme následující binární strom:
- Jednotlivé průchody zpracují uzly v následujícím pořadí:
  - PreOrder: A, B, C
  - InOrder: B, A, C
  - PostOrder: B, C, A



- Inverzní průchody (obrácené pořadí synovských uzlů):
  - InvPreOrder: A, C, B
  - InvInOrder: C, A, B
  - InvPostOrder: C, B, A
- *Pozn.:* PreOrder je reverzí InvPostOrder , PostOrder je reverzí InvPreOrder.

Rekurzivní průchody BS

```
typedef struct tnode
{
  TData data;
  struct tnode *lPtr;
  struct tnode *rPtr;
}TNode;

void PreOrder (TDLList *l, TNode *rootPtr)
            // Seznam l byl inicializován před voláním
{
  if (rootPtr != NULL){
    DLL_InsertLast(l,rootPtr->data);
    PreOrder(l,rootPtr->lPtr);
    PreOrder(l,rootPtr->rPtr);
  }
}
```

Rekurzivní průchody BS

- Záměnou pořadí rekurzivního volání a zpracování prvku v podmíněném příkazu if získáme další průchody:

```
// InOrder
      InOrder(l,rootPtr->lPtr);
      DLL_InsertLast(l,rootPtr->data);
      InOrder(l,rootPtr->rPtr);

// PostOrder
      PostOrder(l,rootPtr->lPtr);
      PostOrder(l,rootPtr->rPtr);
      DLL_InsertLast(l,rootPtr->data);
```

- *K procvičení:* Po záměně pořadí rekurzivního zpracování levého a pravého syna pak dostaneme odpovídající inverzní průchody.

Nerekurzivní PreOrder 1/2

```
void LeftMostPre (TNode *ptr, TDLList *l)
/* s1 - globální zásobník ukazatelů */
{
  while (ptr != NULL){
    Push(&s1,ptr);
    DLL_InsertLast(l,ptr->data);
    ptr = ptr->lPtr;
  }
}
```

Nerekurzivní PreOrder 2/2

```
void NRPreOrder (TDLList *l, TNode *ptr)
{
    DLL_InitList(l);
    InitStack(&s1);
    LeftMostPre(ptr,l);
    while (!IsEmpty(&s1)){
      ptr = Top(&s1);
      Pop(&s1);
      LeftMostPre(ptr->rPtr,l);
    }
}
```

Nerekurzivní InOrder 1/2

```
void LeftMostIn (TNode *ptr)
/* s1 - globální zásobník ukazatelů */
{
  while (ptr != NULL){
    Push(&s1,ptr);
    ptr = ptr->lPtr;
  }
}
```

Nerekurzivní InOrder 2/2

```
void NRInOrder (TDLList *l, TNode *ptr)
{
  DLL_InitList(l);
  InitStack(&s1);
  LeftMostIn(ptr);
  while (!IsEmpty(&s1)){
     ptr = Top(&s1);
     Pop(&s1);
     DLL_InsertLast(l,ptr->data);  // změna od PreOrder
     LeftMostIn(ptr->rPtr);
  }
}
```

Nerekurzivní PostOrder

- PostOrder se vrací k *otci* dvakrát:
  - poprvé zleva, aby šel doprava,
  - podruhé zprava, aby zpracoval otcovský uzel.
  - Pro rozlišení obou návratů použijeme zásobník booleovských hodnot.

```
void LeftMostPost(TNode *ptr)
/* s1 - globální zásobník ukazatelů
   sb1 - globální zásobník booleovských hodnot */
{
   while (ptr != NULL){
      Push(&s1,ptr);
      B_Push(&sb1,true);
      ptr = ptr->lPtr;
   }
}
```

```
void NRPostOrder (TDLList *l, TNode *ptr)
{
  bool fromLeft;
  DLL_InitList(l);
  InitStack(&s1);
  B_InitStack(&sb1);
  LeftMostPost(ptr);
  while (!IsEmpty(&s1)) {
    ptr = Top(&s1);
    fromLeft = B_Top(&sb1);
    B_Pop(&sb1);
    if (fromLeft){    // přichází zleva, půjde doprava
      B_Push(&sb1,false);
      LeftMostPost(ptr->rPtr);
    } else {      // zprava, odstraní a zpracuje uzel
      Pop(&s1);
      DLL_InsertLast(l, ptr->data);
    } //if
  }   //while
}
```

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

Výška stromu – rekurzivně (v1)

```
void HeightBT (TNode *ptr, int *max)
{
  int hl,hr;
  if (ptr != NULL){
     HeightBT(ptr->lPtr,&hl);
     HeightBT(ptr->rPtr,&hr);
     if (hl > hr) {
       *max = hl+1;
     } else {
       *max = hr+1;
     }
  } // if ptr != NULL
  else {*max = 0;}
}
```

Výška stromu – rekurzivně (v2)

```
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->lPtr),Height(ptr->rPtr))+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->lPtr,ptr2->lPtr)&&
                      EQTS(ptr1->rPtr,ptr2->rPtr));
   // && (ptr1->data == ptr2->data) pro ekvivalenci BS
   }
}
```

Kopie BS – rekurzivně

```
TNode * CopyR (TNode *origPtr)
{
  TNode *copyPtr;
  if (origPtr != NULL){
    copyPtr = (TNode *) malloc(sizeof(TNode));
            // zkontrolovat úspěšnost operace malloc
    copyPtr->data = origPtr->data;
    copyPtr->lPtr = CopyR(origPtr->lPtr);
    copyPtr->rPtr = CopyR(origPtr->rPtr);
    return copyPtr;
  } else {
    return NULL;
  }
}
```

Kopie BS – nerekurzivně 1/3

```
TNode * LeftMostCopy (TNode *origPtr)
{
  if (origPtr == NULL)
  {
    return NULL;                              // není co kopírovat
  }
  else
  {
    TNode *newPtr = (TNode *) malloc(sizeof(TNode));
                    // zkontrolovat úspěšnost alokace paměti
    newPtr->data = origPtr->data;
    Push(&s1,origPtr);
    Push(&s2,newPtr);
    origPtr = origPtr->lPtr;       // posun po diagonále orig.
    TNode * tmpPtr = newPtr;       // newPtr se bude vracet
```

Kopie BS – nerekurzivně 2/3

```
    while (origPtr != NULL){
        tmpPtr->lPtr = (TNode *) malloc(sizeof(TNode));
              // zkontrolovat úspěšnost alokace paměti
        tmpPtr = tmpPtr->lPtr;          // po diagonále kopie
        tmpPtr->data = origPtr->data;
        Push(&s1,origPtr);
        Push(&s2,tmpPtr);
        origPtr = origPtr->lPtr; // po diagonále originálu
    }   // while
    tmpPtr->lPtr = NULL;
    return newPtr;
  } // else
}   // LeftMostCopy
```

Kopie BS – nerekurzivně 3/3

```
TNode * CopyNR (TNode *origPtr)
{// s1 a s2 - inicializované globální zásobníky ukazatelů
  TNode *copyPtr;
  TNode *copyAuxPtr;
  TNode *origAuxPtr;

  copyPtr = LeftMostCopy(origPtr);
  while (!IsEmpty(&s1)){
                                  // LeftMostCopy pro pravé syny
     origAuxPtr = Top(&s1);
     Pop(&s1);
     copyAuxPtr = Top(&s2);
     Pop(&s2);
     copyAuxPtr->rPtr = LeftMostCopy(origAuxPtr->rPtr);
  }
  return copyPtr;
}
```

Zrušení BS – rekurzivně

```
void DestroyR (TNode *ptr)
{
  if(ptr != NULL)
  {
    DestroyR(ptr->lptr);
    DestroyR(ptr->rptr);
    free(ptr);
  }
}
```

Zrušení BS – nerekurzivně (v1)

```
void DestroyNR (TNode *ptr)
{
   InitStack(&s1);                    // s1 - zásobník ukazatelů
   do {
     if (ptr == NULL){              // vezmu uzel ze zásobníku
       if (!IsEmpty(&s1)) {
         ptr = Top(&s1);
         Pop(&s1);
       }
     } else {
       if (ptr->rPtr != NULL){ // pravého dám do zásobníku
         Push(&s1,ptr->rPtr);
       }
       TNode *auxPtr = ptr;
       ptr = ptr->lPtr;                        // jdu doleva
       free(auxPtr);                  // zruším aktuální uzel
     }  //else
   } while((ptr != NULL)||(!IsEmpty(&s1)));
}
```

Zrušení BS – nerekurzivně (v2) 1/2

```
void LeftMostDestroy (TNode *ptr)
{
  while (ptr != NULL){
    Push(&s1, ptr);
    ptr = ptr->lPtr;
  }
}
```

Zrušení BS – nerekurzivně (v2) 2/2

```
void DestroyWithLeftMost (TNode *ptr)
{
  InitStack(&s1);
  LeftMostDestroy(ptr);
  while(!IsEmpty(&s1)) {
    ptr = Top(&s1);          // nejlevější na diagonále
    Pop(&s1);
    if (ptr->rPtr != NULL){              // pravá větev
      LeftMostDestroy(ptr->rPtr);
    }
    free(ptr);
  }
}
```

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->lPtr,&left_count);
    right_balanced = TestWBT(ptr->rPtr,&right_count);
    *count = left_count+right_count+1;
    return (left_balanced && right_balanced &&
           (abs(left_count-right_count)<=1));
  }
  else{
    *count = 0;
    return true;
  }
}
```

Vytvoření BS z prvků pole

- Vytvoření váhově vyváženého binárního stromu ze seřazeného pole – rekurzivně

```
void TreeFromArray(TNode **ptr, int left, int right, int array[])
{
   if (left <= right){
     int middle = (left+right)/2;
     *ptr = (TNode *) malloc(sizeof(TNode));
            //zkontrolovat úspěšnost operace malloc
     (*ptr)->data = array[middle];
     TreeFromArray(&(*ptr)->lPtr,left,middle-1,array);
     TreeFromArray(&(*ptr)->rPtr,middle+1,right,array);
     }
   else {
     (*ptr) = NULL;
   }
}
```

K procvičení

- Vytvořte nerekurzivní funkci, která vhodným parametrem určí, zda se zadaný průchod binárním stromem do zadaného seznamu uloží v podobě PreOrder, InOrder nebo PostOrder.
- Napište nerekurzivní funkci PostOrder pomocí inverzního PreOrderu s jedním zásobníkem.
- Implementujte funkci pro výšku stromu nerekurzivně.
- Implementujte funkci pro ekvivalenci (struktur) dvou stromů nerekurzivně.
- Implementujte funkci pro test výškové vyváženosti stromu.

K procvičení

- Vytvořte funkci, která zjistí počet listů BS.
- Vytvořte funkci, která spočítá průměrnou vzdálenost a rozptyl vzdáleností listů od kořene BS.
- Vytvořte funkci, která nalezne a do výstupního seznamu uloží nejdelší cestu od kořene k listu.

Obsah přednášky

- Stromové datové struktury
  - Kořenový strom
  - Binární strom
- **Vyhledávací tabulky**
  - **Hodnocení a klasifikace metod**
  - **Sekvenční vyhledávání**

Základní pojmy a klasifikace

- Přístupová doba (angl. *Access Time*)
- Doba vyhledání
  - minimální
  - maximální
  - průměrná/střední
  - **při úspěšném vyhledání**
  - **při neúspěšném vyhledání**
- Vyhledávání v datové struktuře
  - s přímým přístupem
  - se sekvenčním přístupem

Metody implementace tabulky (1/2)

- Sekvenční vyhledávání v neseřazeném poli
- Sekvenční vyhledávání v neseřazeném poli se zarážkou
- Sekvenční vyhledávání v seřazeném poli
- Sekvenční vyhledávání v poli seřazeném podle pravděpodobnosti vyhledání klíče
- Sekvenční vyhledávání v poli s adaptivním uspořádáním podle četnosti vyhledání

Metody implementace tabulky (2/2)

- Binární vyhledávání v seřazeném poli
  - normální binární vyhledávání
  - Dijkstrova varianta binárního vyhledávání
- Binární vyhledávací stromy (BVS)
- AVL stromy
- Stromy s více klíči ve vrcholech
- Tabulky s rozptýlenými položkami (angl. *Hashing tables*) – TRP

Sekvenční vyhledávání

- Dohoda:
  - Pro typ klíče budeme používat nejčastěji identifikátor `TKey`,
  - pro název klíčové složky položky tabulky identifikátor `key`,
  - a pro hodnotu vyhledávaného klíče identifikátor `k`.
- Nad typem `TKey` rozlišujeme dva typy relací:
  - Relace rovnosti
  - Relace uspořádání
- Pro sekvenční vyhledávání stačí, aby nad typem `TKey` byla definována relace rovnosti.

Vyhledávací algoritmus

- Základní struktura vyhledávacího algoritmu:

```
found ← false
while (not found and <množina prvků není vyčerpána>) do
      <prozkoumej další prvek, a je-li to hledaný,
      proveď found ← true>
end while
Search ← found
```

Vyhledávací algoritmus

- Pozor na ošetření konce cyklu – možné řešení:

```
found ← false
i ← 0
while not found and (i < max) do
    if k = array[i]
      then found ← true
      else i ← i+1
    end if
end while
Search ← found
```

Vyhledávací algoritmus

- Pozor na ošetření konce cyklu – nevhodné řešení:

```
i ← 0
while (k <> array[i]) and (i < max) do
      i ← i+1
end while
Search ← k = array[i]
```

- Pokud hledaný klíč v poli není, dojde k přístupu na adresu za hranicí pole!
  - Toto řešení lze použít pouze pokud prohodíme použité podmínky a použijeme zkratové (neúplné) vyhodnocování booleovských výrazů.

Vyhledávací algoritmus

- Zkratové vyhodnocování booleovských výrazů:
  - `B1 and B2 and B3 and ... and BN`
    ⇒ je-li B1 *false*, vše je *false*
  - `B1 or B2 or B3 or ... or BN`
    ⇒ je-li B1 *true*, vše je *true*
- Vyhledávací algoritmus pak může mít tvar:

```
i ← 0
// zkratové vyhodnocení Booleovského výrazu
while (i < max) and (k <> array[i]) do
    i ← i+1
end while
Search ← i < max
```

Vyhledávací algoritmus

- Ošetření konce cyklu v seznamu – možné řešení:

```
found ← false
while not found and ptr <> NULL do
    if k = ptr->key
        then found ← true
        else ptr ← ptr->nextPtr
    end if
end while
```

- Využití neúplného vyhodnocení logického výrazu:

```
... while (ptr <> NULL) and (k <> ptr->key) do ...
```

Vyhledávací algoritmus

- Špatné ošetření konce cyklu:

```
ptr ← l.first
while (k <> ptr->key) and (ptr <> NULL) do
        // chybná reference
     ptr ← ptr->nextPtr
end while
```

Sekvenční vyhledávání – implementace

- Definujeme následující datové typy:

```
#define MAX …

typedef struct telem
{                       // typ položky tabulky
    TKey key;
    TData data;
}TElem;


typedef struct ttable
{                       // typ tabulka implementovaná polem
    TElem array[MAX]; // pole tabulky
    int n;              // aktuální počet prvků v tabulce
}TTable;
```

Operace Search

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

Varianta Search pro vkládání

- Varianta operace Search, která vrací polohu (index) hledaného prvku:

```
(bool, int) function SearchInd (TTable t, TKey k)
  found ← false
  i ← 0
  while not found and (i < t.n) do
        if k = t.array[i].key
          then found ← true
          else i ← i+1
        end if
  end while
  where ← i          // pro not found je i nedefinováno
  return (found, where)
end function
```

Operace Insert

```
bool function Insert (TTable t, TElem el)
  overflow ← false                    // příznak plné tabulky
  found, where ← SearchInd(t, el.key)
  if found
    then t.array[where] ← el  // přepsání staré položky
    else
      if t.n < MAX          // je místo - vkládáme
        then
          t.array[t.n] ← el
          t.n ← t.n+1
        else overflow ← true // nelze vložit - přetečení
      end if
  end if   // found
  return (not overflow)
end function
```

Operace Delete

```
procedure Delete (TTable t, TKey k)
  found, where ← SearchInd(t,k)
  if found
    then
                              // rušený je přepsán posledním
      t.array[where] ← t.array[t.n-1]
      t.n ← t.n-1
  end if
end procedure
```

- Operaci **Delete** lze také implementovat zaslepením:
  - Klíč rušené položky se přepíše hodnotou, která se nikdy nebude vyhledávat.
  - Snižuje aktivní kapacitu tabulky!

Hodnocení sekvenčního vyhledávání

- Minimální čas úspěšného vyhledání: 1
- Maximální čas úspěšného vyhledání: n
- Průměrný čas úspěšného vyhledání: n/2
- Čas neúspěšného vyhledání: n

- Nejrychleji jsou vyhledány položky, které jsou na počátku tabulky.

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 do
      i ← i+1
  end while
  return (i <> t.n)
      // když našel až zarážku, tak vlastně nenašel ...
end function
```

Sekvenční vyhledávání v seřazeném poli

- Pole je seřazeno podle velikosti klíče:
  - Nad typem klíč musí být definována relace uspořádání.
- Operace **Search**:
  - Skončí neúspěšně, jakmile narazí na položku s klíčem, který je větší než hledaný klíč.
  - Urychlí se pouze neúspěšné vyhledávání.
- Operace **Insert** a **Delete**:
  - Musí zachovat uspořádání pole.
  - Vyžadují proto posuny segmentů pole.

Sekvenční vyhledávání v seřazeném poli

- Operace **Insert**:
  - Musí najít správné místo pro vložení prvku.
  - Segment pole od nalezeného místa se musí posunout o jednu pozici vpravo.
- Operace **Delete**:
  - Segment pole vpravo od mazaného prvku se musí posunout o jednu pozici vlevo – přepíše rušený prvek.

Posuny segmentů pole

- Posun segmentu doprava se provede cyklem zprava (od konce pole) – posun segmentu `[low..(high-1)]` o jednu pozici doprava:

```
for i ← (high, low+1)^-1 do
  t.array[i] ← t.array[i-1]
end for
```

Seřazení pole – četnost vyhledávání

- V praxi jsou často některé položky vyhledávány mnohem častěji než ostatní – při použití sekvenčního vyhledávání se vyplatí tyto položky umístit na začátek pole, aby byly nalezeny rychleji.
- Seřazení pole podle četnosti vyhledávání:
  - Jednou za čas – lze realizovat pomocí počitadla, které se aktualizuje po každém přístupu k položce.
  - Průběžně – adaptivní rekonfigurace podle četnosti vyhledávání – při každém přístupu k položce se položka vymění se svým levým sousedem (pokud existuje).

Seřazení pole – četnost vyhledávání

- Při adaptivní rekonfiguraci je součástí operace **Search** při úspěšném vyhledání příkaz:

```
if where > 0
then t.array[where] ↔ t.array[where-1]
```

- *Pozn.:* zápis `A ↔ B` označuje operaci výměny hodnot, kterou je obvykle potřeba realizovat trojicí příkazů a pomocnou proměnnou. Tuto operaci je v rámci IAL možné použít vždy pro usnadnění zápisu.