

IAL – 4. přednáška



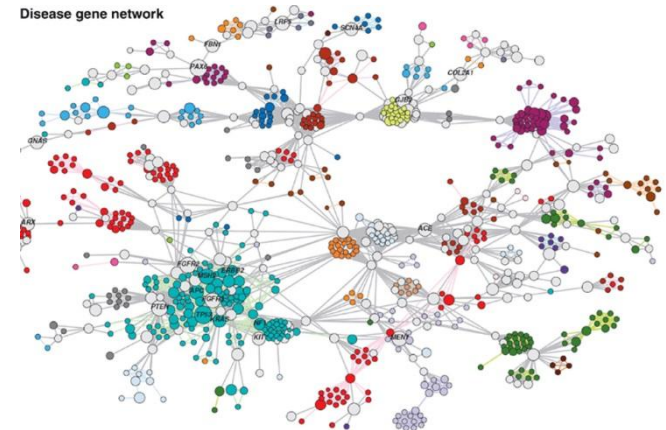
Stromové datové struktury

Obsah přednášky

- Stromové datové struktury
 - Kořenový strom
 - Binární strom
 - Průchody binárním stromem
 - Algoritmy nad binárním stromem

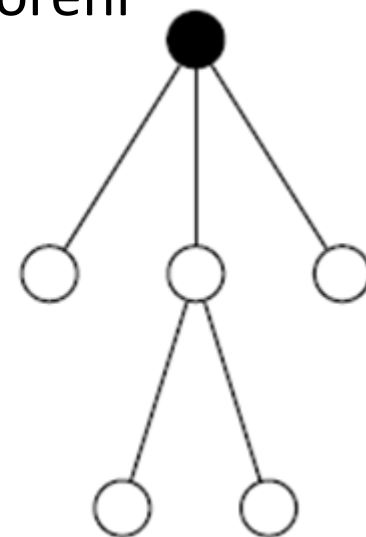
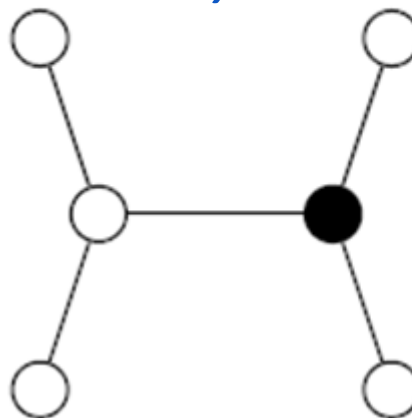
Nelineární datové typy

- 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:**

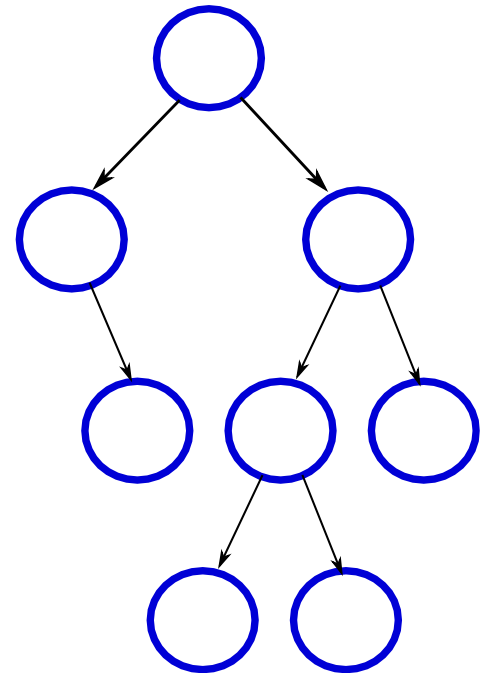
- výška prázdného stromu je 0,
- výška stromu s jediným uzlem (kořenem) je 1,
- výška jiného stromu je počet hran od kořene k nejvzdálenějšímu uzlu + 1.

- **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*.

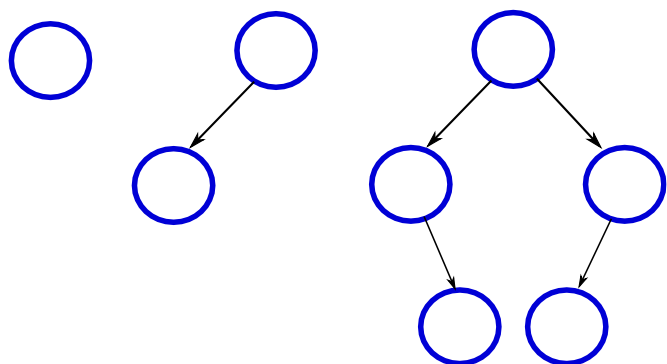
Vlastnosti binárních stromů

□ **Vyváženost** stromu:

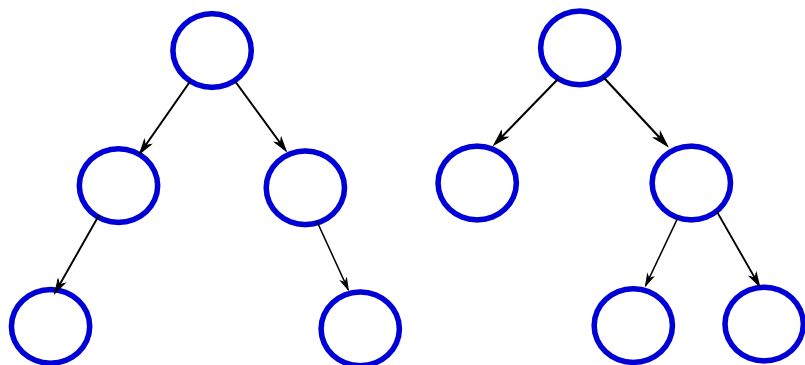
- Binární strom je **váhově vyvážený**, když pro každý jeho uzel platí, že **počty uzlů** jeho levého a pravého podstromu se rovnají a nebo se liší právě o 1.
- Binární strom je **výškově vyvážený**, když pro každý jeho uzel platí, že **výška** levého podstromu se rovná výšce pravého podstromu a nebo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

Příklad: (ne)vyvážené stromy

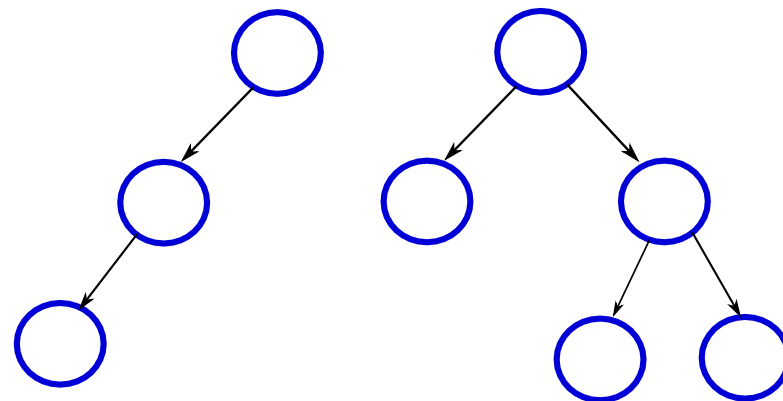
□ Váhově vyvážené stromy



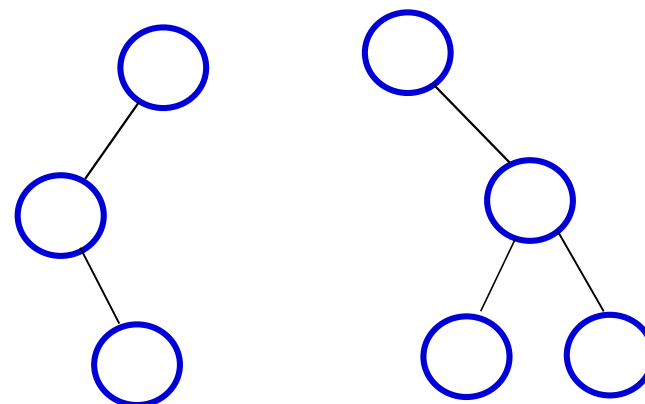
□ Výškově vyvážené stromy



□ Váhově **ne**vyvážené stromy



□ Výškově **ne**vyvážené 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 **složitě** 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

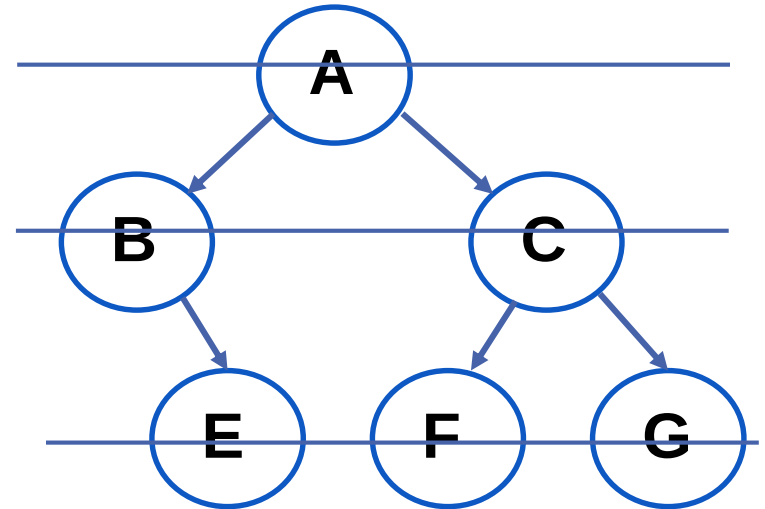
- Může porušit uspořádanost nebo vyváženost stromu.
- Konkrétní způsoby implementace těchto operací probereme u jednotlivých typů stromů.

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 šířky

- *Level-order, Breadth First Search, BFS*
- Průchod stromem po hladinách (od kořene):
 - A, B, C, E, F, H
- Reverzní průchod do šířky – průchod po hladinách od nejnižší hladiny směrem ke kořeni:
 - E, F, H, B, C, A

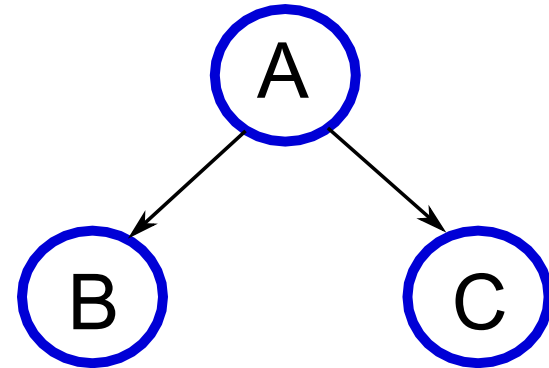


Procházení BS do hloubky

□ Mějme následující binární strom:

□ Jednotlivé průchody zpracují uzly v následujícím pořadí:

- PreOrder: A, B, C
- InOrder: B, A, C
- PostOrder: B, C, A

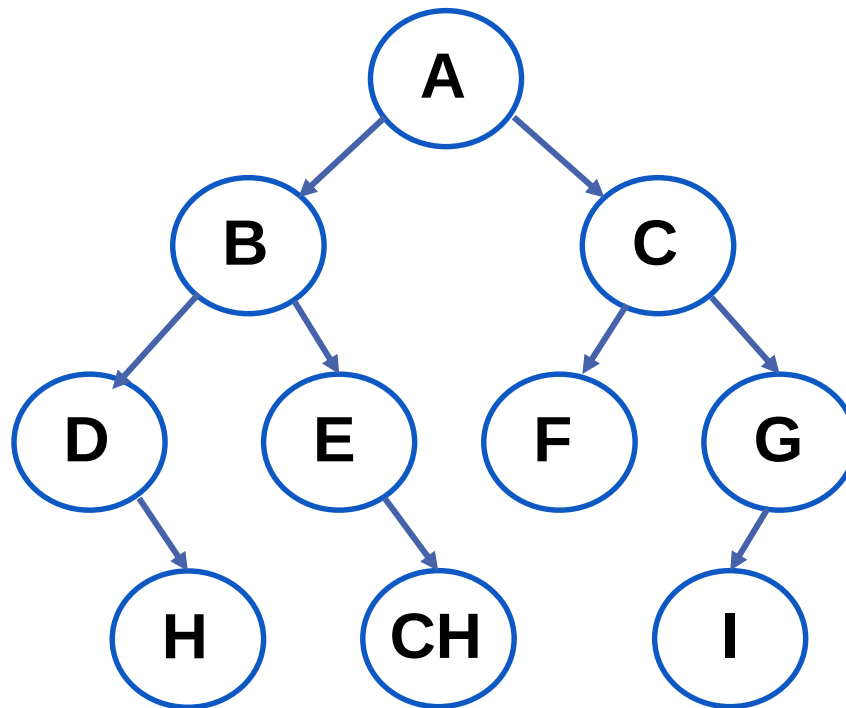


□ Inverzní průchody (obrácené pořadí synovských uzlů):

- InvPreOrder: A, C, B
- InvInOrder: C, A, B
- InvPostOrder: C, B, A

□ *Pozn.:* PreOrder je reverzí InvPostOrder , PostOrder je reverzí InvPreOrder.

Příklad: Průchody stromem



Rekurzivní průchody BS

```
typedef struct tnode
{
    TData data;
    struct tnode *left;
    struct tnode *right;
} TNode;

void PreOrder (TDLLList *l, TNode *root)
    // Seznam l byl inicializován před voláním
{
    if (root != NULL) {
        DLL_InsertLast(l, root->data);
        PreOrder(l, root->left);
        PreOrder(l, root->right);
    }
}
```

Rekurzivní průchody BS

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

// InOrder

```
InOrder(l, root->left);  
DLL_InsertLast(l, root->data);  
InOrder(l, root->right);
```

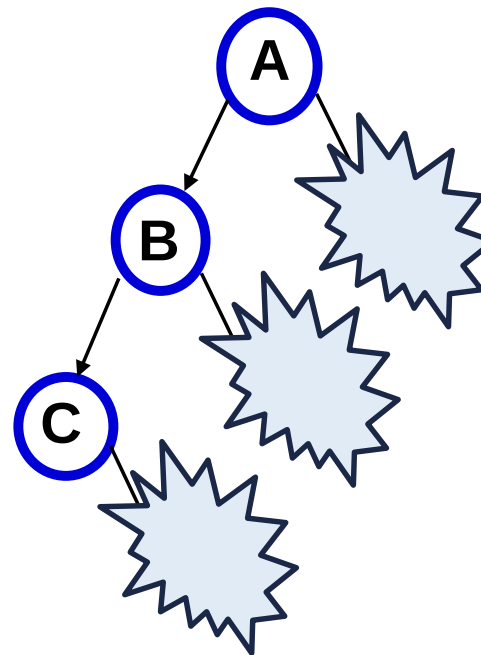
// PostOrder

```
PostOrder(l, root->left);  
PostOrder(l, root->right);  
DLL_InsertLast(l, root->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, TDLLList *l)
/* s1 - globální zásobník ukazatelů */
{
    while (ptr != NULL) {
        Push(&s1, ptr);
        DLL_InsertLast(l, ptr->data);
        ptr = ptr->left;
    }
}
```



Nerekurzivní PreOrder 2/2

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

Nerekurzivní InOrder 1/2

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

Nerekurzivní InOrder 2/2

```
void NRInOrder (TDLLList *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->right);
    }
}
```

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->left;
    }
}
```

```

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->right);
        } else {          // zprava, odstraní a zpracuje uzel
            Pop(&s1);
            DLL_InsertLast(l, ptr->data);
        } //if
    } //while
}

```

Level-order průchod

```
void LevelOrder (TDLLList *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
}
```

Výška stromu – rekurzivně (v1)

```
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; }
}
```

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->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;
    }
}
```

Kopie BS – nerekurzivně 1/3

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

Kopie BS – nerekurzivně 2/3

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

Kopie BS – nerekurzivně 3/3

```
TNode * CopyNR (TNode *orig)
{ // s1 a s2 - inicializované globální zásobníky ukazatelů
    TNode *copy;
    TNode *copyAux;
    TNode *origAux;

    copy = LeftMostCopy(orig);
    while (!IsEmpty(&s1)) {
        // LeftMostCopy pro pravé syny
        origAux = Top(&s1);
        Pop(&s1);
        copyAux = Top(&s2);
        Pop(&s2);
        copyAux->right = LeftMostCopy(origAux->right);
    }
    return copyPtr;
}
```

Zrušení BS – rekurzivně

```
void DestroyR (TNode *ptr)
{
    if (ptr != NULL)
    {
        DestroyR(ptr->left);
        DestroyR(ptr->right);
        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->right != NULL) { // pravého dám do zásobníku
                Push(&s1, ptr->right);
            }
            TNode *aux = ptr;
            ptr = ptr->left; // jdu doleva
            free(aux); // 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->left;
    }
}
```

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->right != NULL) { // pravá větev
            LeftMostDestroy(ptr->right);
        }
        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->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;
    }
}
```

Vytvoření BS z prvků pole

- Vytvoření váhově vyváženého binárního stromu ze seřazeného pole (rekurzivně)

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