13 Pole, seznam, zásobník, fronta, strom

Data, se kterými pracují programy (algoritmy) jsou uchovávána v **datových strukturách** a lze je prezentovat **různými datovými typy**.

Pro efektivní práci s daty je nutné zvolit:

- odpovídající datový typ
- vhodnou datovou strukturu.

DATOVÉ STRUKTURY MOHOU BÝT:

Samostatné hodnoty (**Jednoduché typy**) – charakterizují množiny hodnot, které lze nazvat hodnotami elementárními, a jsou tvořeny jednoduchými datovými typy, jako jsou čísla, písmena, znaky apod.

Strukturované údaje (**Strukturované typy**) – charakterizují množiny struktur (uskupení) a mají charakter záznamů. Záznamy většinou popisují nějaké objekty. Je v nich více položek - **datových členů**, přičemž každý člen popisuje nějaký rys nebo hodnotu daného objektu. Například bude-li záznam popisovat osobu, pak typicky bude obsahovat její rodné číslo, jméno a příjmení, jednotlivé části adresy bydliště (název ulice s popisným číslem, PSČ a název města) atd.

Pro popis algoritmů většinou není důležité, pro jaké datové typy budou prakticky použity, zda pro jednoduché nebo pro strukturované datové typy. Budou-li pracovat s nějakými prvky z nějaké množiny dat. Nicméně aby algoritmy mohly prvky vyhledávat nebo je srovnávat, potřebují, aby u každého prvku byla stanovena hodnota, která ho reprezentuje. U prvků, jež jsou tvořeny **jednoduchými datovými typy**, **je touto hodnotou samotná hodnota prvku**. **U strukturovaného typu je touto hodnotou většinou nějaký jeho člen (nebo několik jeho členů)**, a to takový, že jeho hodnota prvek jednoznačně určuje. Hodnotě, která prvek reprezentuje (určuje), budeme říkat **klíč prvku**.

ZÁKLADNÍ ROZDĚLENÍ DATOVÝCH STRUKTUR:

1. **Základní datové struktury** – vyskytují se téměř ve všech programovacích jazycích. Za běhu programu nemění svůj rozsah. Např. **proměnná**, **pole**, **záznam** a **objekt**.
2. **Odvozené datové struktury** – nazývány jako abstraktní datové struktury. Často implementovány jako objekty. Za běhu programu mohou měnit svůj rozsah. Např. **seznam**, **zásobník, fronta a strom**.

POLE

Pole je lineární (lineární = data jdou po sobě) **datová struktura** a je dostupná ve všech běžných programovacích jazycích.

Pole je posloupnost (řada) proměnných stejného datového typu (tzv. složek), uložených v operační paměti v nepřetržité řadě zasebou a chápaných jako jeden celek.

Poznámka: paměť počítače má lineární uspořádání, proto je pole nejpřirozenější a nejsnadněji realizovatelná datová struktura pro uložení údajů (dat) do paměti.

Vytváří se na základě deklarace, ve které určíme její **jméno** (*identifikátor*), **rozsah** (*typ*), **rozměr** (dimenzi) a **počet složek** (*rozsah indexu*).

Každé pole je tedy charakterizováno:

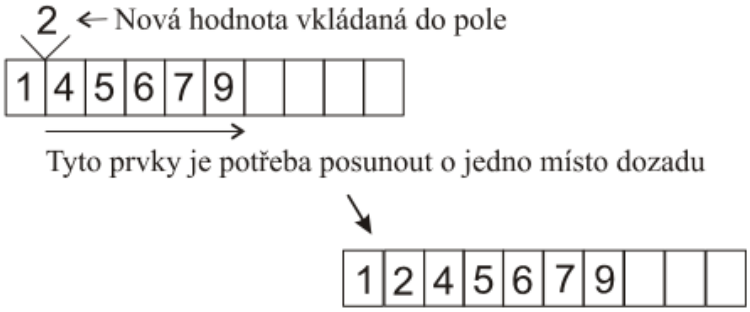
1. **rozsahem** – počtem prvků (složek), které vkládáme do tohoto pole,
2. **rozměrem** – např. jednorozměrné, dvourozměrné atd.,
3. **indexováním** prvků – indexování určuje pořadí (umístění) prvků v poli, indexovat lze od 0 (nejčastější způsob indexování, v programovacím jazyce), nebo od 1 nebo si počáteční hodnotu indexu zvolí programátor. (Indexy jsou přirozená čísla.)

Na obrázku je vidět jednorozměrné pole osmi čísel

V paměti počítače je **adresována** pouze první složka (prvek) pole, s ostatními prvky je pracováno na základě **indexu**.

Použití pole je velmi efektivní, pokud jsou splněny určité předpoklady:

- Potřebujeme předem vědět, kolik hodnot do pole budeme ukládat, pokud potřebujeme hodnoty do pole ukládat postupně, budeme to dělat tak, že je budeme ukládat za již uložené hodnoty – na konec. Obdobným způsobem budeme hodnoty odebírat – od konce. Pokud nebudeme vědět, kolik hodnot budeme do pole ukládat, může se nám stát, že dojde k **přetečení paměti** (vyčerpání pole).
- Problémem je případ, kdy hodnoty v poli jsou již vloženy a my potřebujeme vložit/odebrat prvek z „prostředku" pole. V tomto případě musíme všechny hodnoty od místa vložení/odebrání posunout o 1 místo doprava/doleva.



Obdobná situace nastane, pokud z pole odstraníme hodnotu, jež v něm není poslední (posuneme doleva). Obě operace jsou z hlediska práce počítače neefektivní. Pokud příslušný algoritmus vyžaduje operace zvětšení rozsahu, vkládání „doprostřed"uložených hodnot, odstraňování z míst, jež jsou „uprostřed"uložených hodnot, pak je účelnější pro uložení hodnot použít **lineární dynamickou datovou strukturu** (např. dynamicky alokované pole, seznam,…).

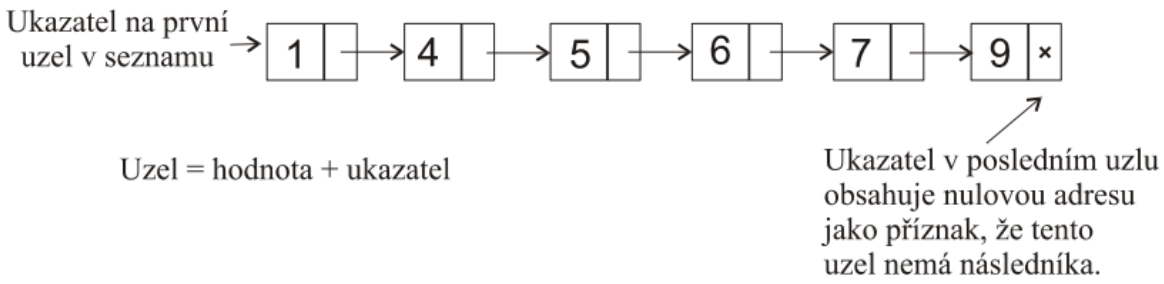
Použití tzv. **dynamicky alokovaných polí** nám umožní provést „dynamické"zvětšení rozsahu pole, nemusíme předem znát počet prvků vkládaných do pole a nestane se, že by došlo k přetečení pole. Tato operace je spojena s **realokací paměti** pro pole – přesunem již uložených hodnot v poli na jiné místo v operační paměti, což je časově poměrně náročná operace a proto se snažíme, aby ke zvětšování/zmenšování (přidávání/odebíraní hodnot) z pole docházelo co nejméně.

Programovací jazyky se velmi liší v tom, jak s polem pracují. V některých jazycích (zejména starších, kompilovaných) nebylo možné za běhu programu vytvořit pole s dynamickou velikostí (např. mu dát velikost dle nějaké proměnné).

SEZNAM

Seznam je lineární dynamická datová struktura. Přívlastek **dynamická** zde vyjadřuje skutečnost, že seznam je budován postupně, paměť pro něj není na rozdíl od pole přidělena najednou, ale je **alokována samostatně** pro každou ukládanou hodnotu. Tím na rozdíl od pole není na začátku zapotřebí stanovit počet hodnot, které budou do seznamu ukládány. A protože je paměť pro každou ukládanou hodnotu přidělována samostatně, nejsou na rozdíl od pole hodnoty do paměti uloženy za sebou, ale víceméně „nahodile“ na různých místech. Tím ovšem ztrácíme základní výhodu pole plynoucí z uložení hodnot v paměti za sebou, a tou je možnost přejít k další hodnotě v poli prostým zvýšením hodnoty indexu.

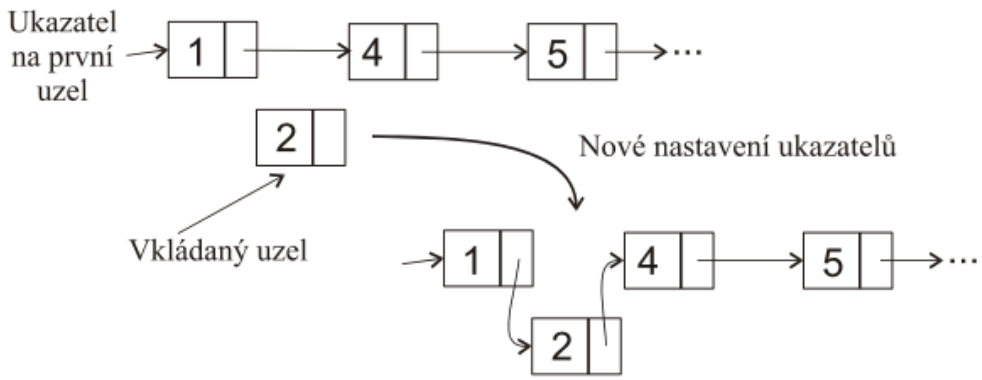
Abychom i v seznamu mohli přejít k následující hodnotě, ukládáme spolu s každou hodnotou ještě **ukazatel** na místo v paměti, kde je uložena následující hodnota. Této dvojici hodnota (prvek) + ukazatel v seznamu říkáme **uzel**. Ukazatel v posledním uzlu obsahuje nulovou adresu jako příznak, že tento uzel nemá následníka. Abychom mohli se seznamem pracovat, musíme si navíc někde uchovat **ukazatel (adresu)** na první uzel v seznamu.



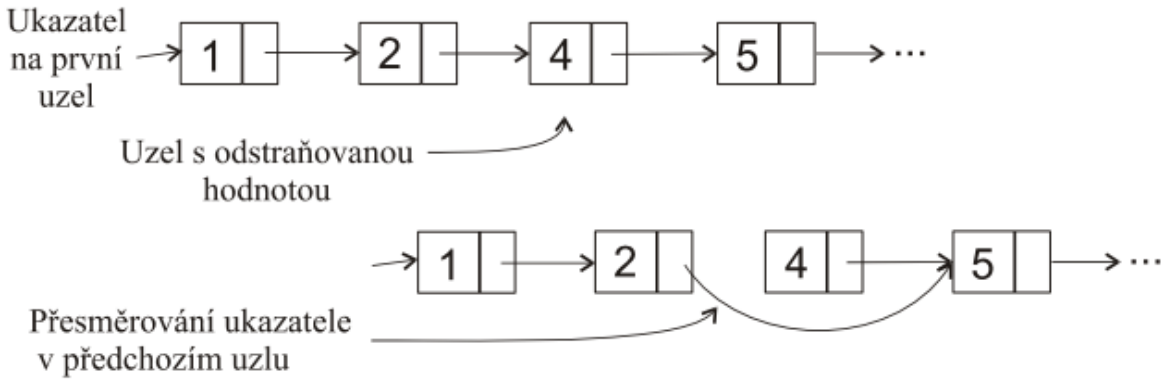
Termín ukazatel je používán v programovacích jazycích pro označení proměnné, do které ukládáme adresu nějakého místa v paměti. U seznamu máme ukazatel na první uzel, což je adresa prvního uzlu seznamu v paměti, a každý uzel obsahuje ukazatel na následující uzel, tedy adresu následujícího uzlu v paměti.

Výhody:

- Do seznamu lze snadno kamkoliv vkládat nové prvky – vytvoříme nový uzel, do něj přidáme hodnotu, najdeme v seznamu místo k vložení a přesměrujeme ukazatele.

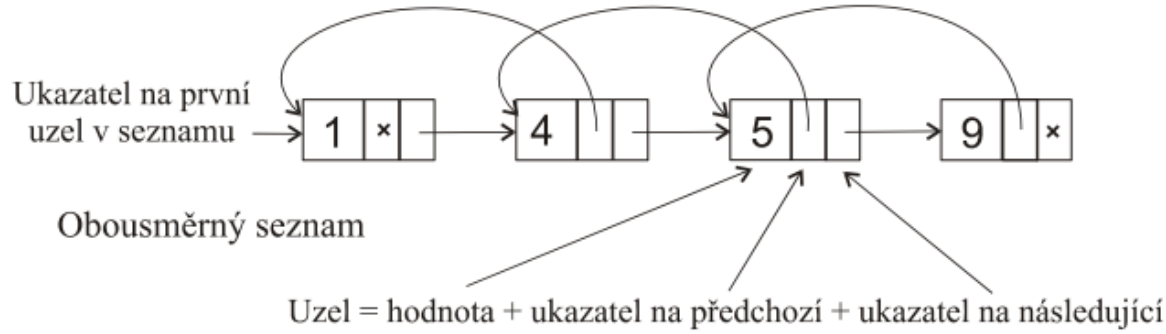


- V seznamu lze snadno odstranit kterýkoliv prvek – ukazatele v předchozím uzlu přesměrujeme na uzel, jenž v seznamu následuje za uzlem zrušenou hodnotu.



Nevýhody

- Na rozdíl od pole zde není přímý přístup k libovolné uložené hodnotě – máme jen ukazatele na první uzel, nezbývá tedy, než projít postupně všechny uzly v seznamu (od prvního) až k uzlu s požadovanou hodnotou,
- další nevýhodou je, chceme-li přejít k hodnotě v předcházejícím uzlu, musíme opět začít seznam procházet od prvního uzlu, protože v uzlu máme jen ukazatel na následující uzel, nikoliv na předchozí – to znamená, že seznam je jen **jednosměrný**. Řešením je použití tzv**. obousměrného seznamu**, který má v uzlu dva ukazatele, jeden ukazuje na předchozí uzel a druhý na následující => plýtvání operační paměti (uchovávání adres uzlů).

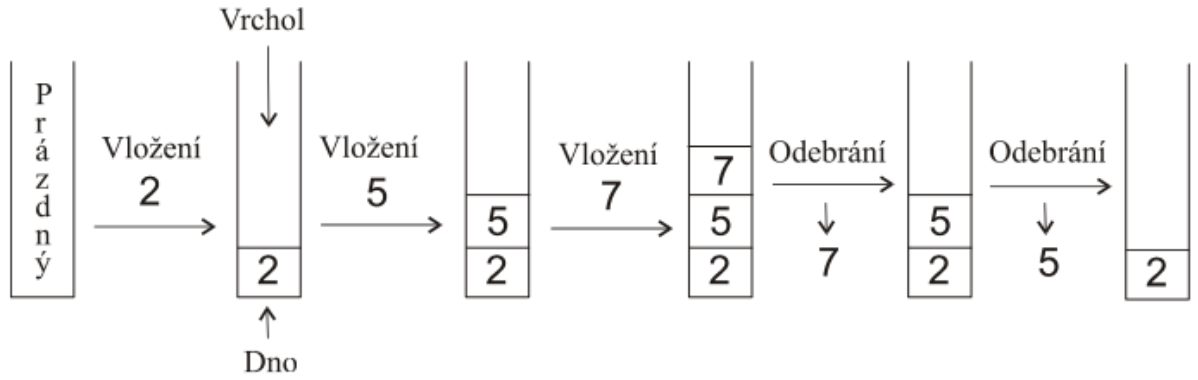


Závěr:

Lze říct, že práce se seznamem je komplikovanější než s polem (musíme udržovat ukazatele, ...), proto seznamy používáme méně než pole. Používáme je v případech, kdy použití pole je problematické.

ZÁSOBNÍK

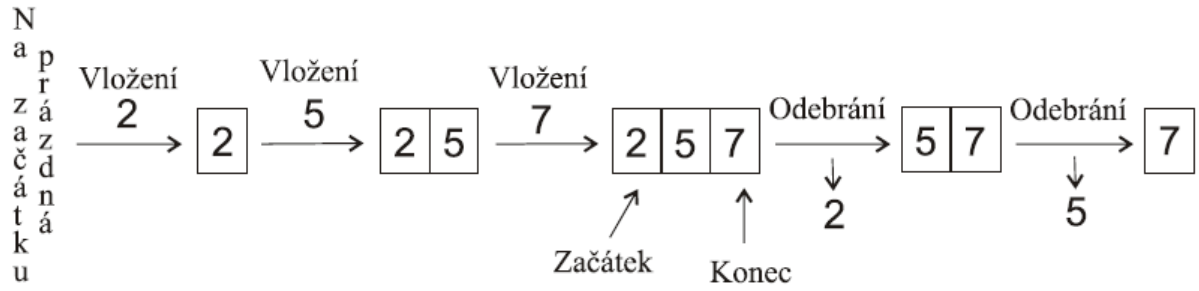
Velmi významnou a hojně používanou dynamickou datovou strukturou je zásobník. **Je opět lineární datovou strukturou**. Jeden konec této struktury je fixní a označujeme ho jako dno zásobníku. Druhý konec tvoří vrchol zásobníku. Zásobník má definovány dvě základní operace: **uložení** datového prvku na zásobník a **odebrání** prvku ze zásobníku. Operace uložení probíhá tak, že datový prvek se uloží (přidá) na vrchol zásobníku. Operace odebrání naopak odebere prvek z vrcholu zásobníku. Tudíž operace na zásobníku pracují výlučně s jeho vrcholem. Ostatní datové prvky uložené na zásobníku krom prvku na vrcholu zásobníku nejsou přímo dostupné. Na začátku je zásobník prázdný, tj. jeho vrchol je totožný s jeho dnem. Postupným přidáváním prvku na zásobník se jeho vrchol posouvá směrem ode dna, naopak odebíráním prvků se vrchol posouvá směrem ke dnu. Je zřejmé, že prvky jsou ze zásobníku odebírány v opačném pořadí, než v jakém byly do něho ukládány. **Jde o datovou strukturu typu LIFO** (Last In First Out = Poslední dovnitř - první ven).



Zásobník patří mezi abstraktní datové struktury. U abstraktních datových struktur definujeme jen jejich vlastnosti a operace, nezabýváme se přitom jejich implementací**. Pokud bychom zásobník v praxi potřebovali implementovat, nejsnadnější je to pomocí pole**. Začátek pole bude tvořit dno zásobníku, poslední zaplněný prvek pole bude vrcholem zásobníku. Jediným problematickým rysem zde může být stanovení velikosti pole tak, aby nedošlo k přeplnění zásobníku.

FRONTA

Velmi významnou dynamickou datovou strukturou, i když ne tak častou používanou jako zásobník, je fronta**. Je také lineární datovou strukturou** a má opět definovány dvě základní operace: **vložení** prvku do fronty a **odebrání** prvku z fronty. Operace vložení probíhá tak, že datový prvek se uloží na konec fronty. Operace odebrání naopak odebere prvek ze začátku fronty. Na začátku je fronta prázdná. Postupným přidáváním prvků do fronty se její konec vzdaluje od začátku, naopak odebíráním prvků se začátek přibližuje ke konci. Je zřejmé, že prvky jsou z fronty odebírány ve stejném pořadí, v jakém byly do fronty vkládány. **Jde o datovou strukturu typu FIFO** (First In First Out = První dovnitř - první ven).



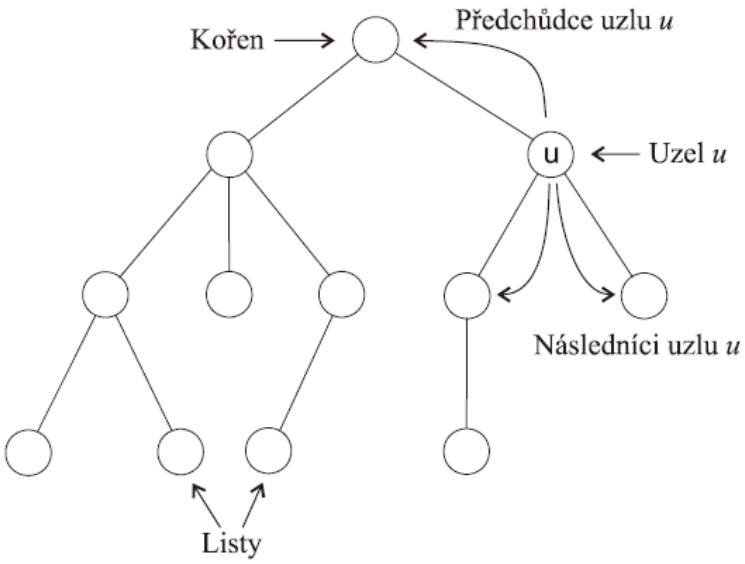
Fronta je rovněž abstraktní datová struktura. V jejím popisu opět není uvedeno, jak ji implementovat. V praxi, pokud budeme potřebovat použít frontu, lze ji také vytvořit pomocí pole, i když ne tak snadno jako zásobník. Problém je v tom, že při odebírání se začátek fronty v poli postupně posunuje dozadu. To vyřešíme cyklickým přechodem z posledního prvku pole na jeho první prvek. Dalším problematickým rysem je zde stejně jako u zásobníku stanovení velikosti pole tak, aby nedošlo k přeplnění fronty.

STROMY

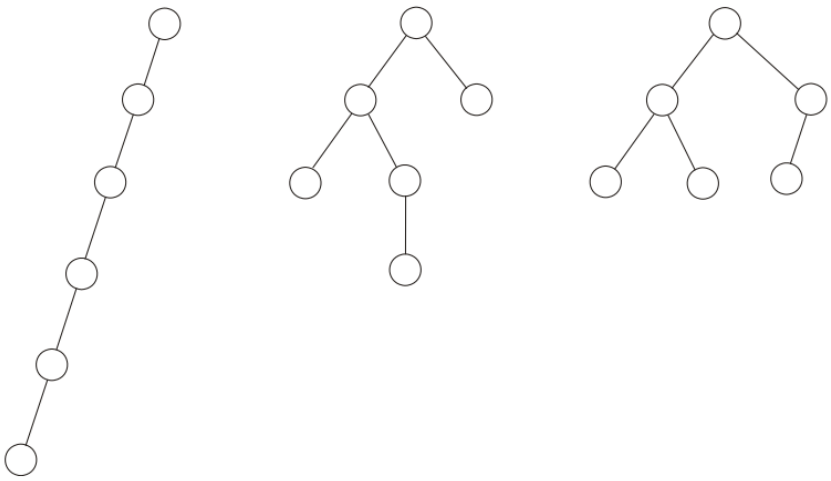
Z **nelineárních datových struktur** jsou v algoritmech **nejpoužívanější stromy**. **Stromy jsou specifickým případem matematických grafů**. Terminologií grafů bychom je popsali jako obyčejné acyklické grafy. **Skládají se z uzlů a hran**. **V uzlech** jsou při použití stromů v algoritmech **uloženy datové prvky**, nad kterými algoritmus probíhá. **Hrany reprezentují vztahy mezi uzly (prvky)**, které jsou základem daného algoritmu.

Strom se kreslí směrem shora-dolů(někdy také zleva-doprava, je-li příliš široký). Zcela nahoře je první uzel stromu, který se nazývá **kořen**. Pod ním jsou uzly, které jsou jeho **následníci**. Jsou s ním spojeny **hranami**. Tyto uzly mohou mít rovněž následníky, které jsou opět pod nimi a jsou s nimi opět spojeny hranami. Uzly, které nemají žádné následníky, nazýváme **listy**. Každý uzel vyjma kořene je hranou spojen s právě jedním uzlem, který je nad ním. Tento uzel je jeho předchůdce.

Na kreslení uzlů a hran nejsou žádná zvláštní omezení. Uzly většinou kreslíme jako kružnice, hrany jako rovné čáry.



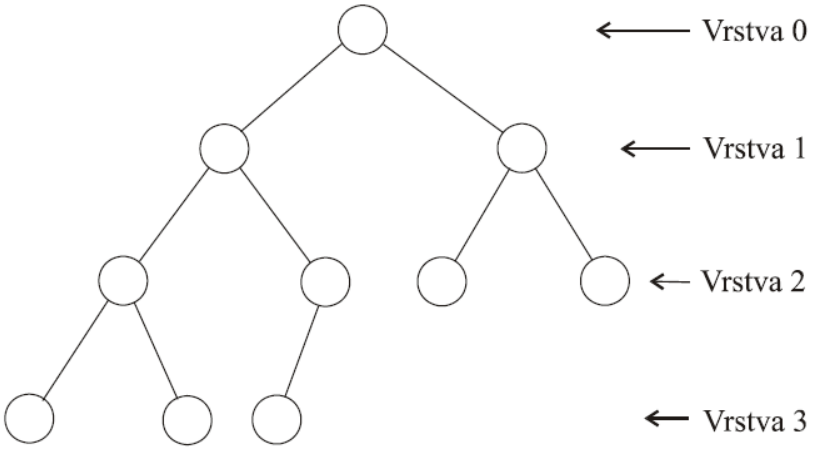
Běžně používané stromy mají zpravidla omezeno, kolik může mít uzel následníků. V tomto ohledu nejjednodušší stromy jsou stromy, v nichž každý uzel může mít nejvýše dva následníky. Používají se poměrně často a říkáme jim **binární stromy**. Při použití stromů v algoritmech si uchováváme ukazatel (adresu) na kořenový uzel. K libovolnému uzlu se pak dostaneme tak, že začneme od kořene a postupně po hranách přecházíme k nižším uzlům, až dojdeme k žádanému uzlu. **Přechod od nějakého uzlu po hraně k jeho následníkovi považujeme za jednu základní operaci**. Počet operací potřebných k tomu, abychom se od kořene dostali k danému uzlu, je roven počtu hran, které jsou na cestě od kořene k tomuto uzlu. Tento počet nazýváme **vzdáleností uzlu od kořene**. Maximum ze vzdáleností uzlu od kořene stromu nazveme **výškou stromu**. **Výška stromu** je tedy vzdálenost kořene od listů, které jsou ve stromu nejníže. Zřejmě čím má strom při daném počtu uzlů menší výšku, tím je to výhodnější, nebo tím nižší je maximální délka cesty od kořene k uzlům stromu. Na následujícím obrázku jsou tři binární stromy. Všechny mají stejný počet uzlů – šest.



Levý strom je nejméně výhodný. V podstatě je to seznam a o seznamu víme, že časová složitost přístupu k jeho uzlům je lineární. Naproti tomu strom zcela vpravo má tu vlastnost, že má při daném počtu uzlů nejmenší možnou výšku. Je to **vyvážený binární strom.**

Vyvážený binární strom je takový binární strom, který má ve všech vrstvách maximální možný počet uzlů vyjma poslední vrstvy, která může být zaplněna jen zčásti. Vrstvou přitom tady rozumíme všechny

uzly, které mají stejnou vzdálenost od kořene, tedy mají stejnou vodorovnou úroveň při běžném nakreslení stromu shora-dolů. Tuto vzdálenost nazveme číslem vrstvy. Podíváme-li se na následující příklad vyváženého binárního stromu



můžeme z něho odvodit, jaké počty uzlů budou obecně v jednotlivých vrstvách vyváženého binárního stromu výšky h.

ZÁVĚR

Ukládání a vyhledávání může být prováděno nad daty uloženými v hlavní paměti nebo v sekundární paměti; podle typu paměti volíme vhodné **datové struktury** a algoritmy. Při vývoji softwaru závisí složitost implementace a rychlost práce výsledného programu na správném výběru **datových** struktur.

Kritéria pro návrh datových struktur:

- rychlost čtení (včetně nalezení dat),
- rychlost zápisu (operace vložení, mazání, aktualizace),
- paměťová náročnost,
- náročnost implementace (čím komplikovanější algoritmus, tím větší pravděpodobnost chyby).

Existuje celá řada dalších datových struktur, než ty, které byly uvedeny v tomto textu. Mnohé „klasické“ datové struktury jsou obsaženy buď ve standardních knihovnách programovacích jazyků, nebo vestavěny přímo v programovacích jazycích.