Softwarové inženýrství

Petr Horáček

Datové struktury

Obsah

1. Úvod

Datová struktura je pojem z teorie programování, který vyjadřuje jakým způsobem jsou data v paměti uložena a organizována. Způsob uložení dat v paměti zároveň určuje způsob jak s nimi pracovat. Datové struktury je tedy potřeba navrhnout tak, aby práce s nimi byla optimální, to znamená, snadná, rychlá, ...

Nelze vytvořit univerzální datovou strukturu, protože každá situace vyžaduje jiný přístup při ukládání dat. Je tedy třeba pro každou situaci navrhnout specifickou datovou strukturu, která optimálně využívá výpočetních a paměťových zdrojů daného počítače.

I když je třeba pro každou situaci vytvořit specifickou datovou strukturu, lze při tom využít některé principy, nebo kombinace obecných datových struktur. Tyto datové struktury popisují účel a způsob použití, které je možné využít v určitých situacích.

Při vytváření (navrhování) datových struktur je přihlíženo k fungování reálného světa. Při programování se většinou řeší problémy (algoritmy) inspirované fungováním z reálného světa a proto i u většiny datových struktur je možné najít vhodné přirovnání v reálném světě.

2. Statické a dynamické datové struktury

2.1 Statické datové struktury

2.2 Dynamické datové struktury

Dynamické datové struktury jsou takové datové struktury, které v průběhu vykonávání programu mohou měnit svůj rozsah - velikost (počet datových položek). Dynamické datové struktury se používají v takových případech kdy není dopředu možné odhadnout kolik datových polože bude potřeba k uložení dat a nebo se toto množství výrazně mění v průběhu vykonávání programu. Nevýhodou dynamických datových struktur oproti dynamickým datovým strukturám je jejich nižší rychlost v náhodném přístupu k datovým položkám. Proto jsou dynamické datové struktury stále vyvíjeny tak, aby se co nejvíce snížil počet operací potřebných pro získání požadovaných dat.

Datové struktury jsou tvořeny proměnnými - statické i dynamické. Pro manipulaci s dynamickými datovými strukturami je nutné přistupovat k paměťové adresy na které jsou uloženy. K tomu slouží datový typ **ukazatel**.

3. Pole

Pole je datová struktura, která umožňuje přistupovat k řadě paměťových míst (proměnných stejného datového typu) pomocí jednoho identifikátoru (ukazatele). Každá proměnná v poli se nazývá prvek pole a je identifikována pomocí unikátního čísla - index prvku pole. Je důležité, že indexy pole jsou číslovány od nuly (kvůli výpočtu adresy jednotlivých prvků pole). Velikost pole vyjadřuje počet prvků, které obsahuje (nikoli množství paměti, které zabírá). Pole se používá pro uložení sady hodnot, dat, ke kterým je třeba dynamicky přistupovat. Pole poskytuje širší možnosti než samostatné proměnné. Jedná se o silný nástroj pro práci s velkým množstvím dat.

Pole je dále základem mnoha složitějších datových struktur, které jsou využívány pro ukládání a třídění dat. Typickým příkladem je dynamické pole, statická fronta, ... Pole (statické) je velice datová struktura, která umožňuje velice rychlý zápis a čtení dat, ale má jednu velkou nevýhodu (která nemusí být vždy na obtíž) - jeho velikost je statická, to znamená, že má pevný počet prvků a nelze je měnit.

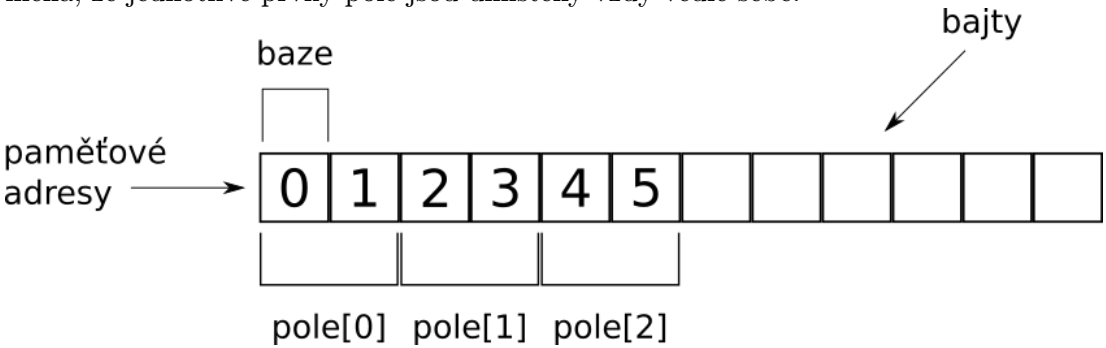
3.1 Popis struktury pole

Pole vnitřně tvoří ukazatel na paměť, respektive na datový typ prvků pole. Specifikace datového typu prvků pole je důležitý. Vyjadřuje totiž, kolik bajtů každý prvek v paměti zabírá. Tato informace je následně využívána při výpočtu adresy pro přístup k jednotlivým prvkům pole. Při výpočtu adresy jednotlivých prvků pole je použit vzorec:

$$adresa = baze + index \cdot datový\_typ$$

kde *adresa* označuje výslednou adresu prvku pole v paměti, *baze* označuje adresu prvního prvku pole, tato adresa je uložena v ukazateli na pole, *index* je indexem konkrétního prvku pole a datový_typ označuje počet bajtů, které zabírá konkrétní datový typ.

V paměti počítače je pole reprezentováno jako lineární datová struktura. To znamená, že jednotlivé prvky pole jsou umístěny vždy vedle sebe.



4. Struktury a třídy

5. Dynamické pole

Dynamické pole je datový kontejner (struktura určená k uchovávání dat) postavená na statickém poli. Dynamické pole umožňuje přidávat libovolný počet datových prvků, čímž odstraňuje nevýhody klasického statického pole jehož velikost je fixní.

Dynamické pole vnitřně ukládá prvky do statického pole fixní délky. V případě, že dojde k vyčerpání kapacity statického pole dojde k alokaci nového pole větší délky než původní (většinou dvojnásobné délky). Prvky ze starého pole se překopírují do pole nového a staré pole se uvolní z paměti (dealokuje). Prvky z dynamického pole lze také mazat. V případě, že je pole příliš prázdné (množství nezabraných prvků překročí určitou mez) dojde k alokaci menšího pole a překopírují se do něj prvky z pole starého. Tím je šetřena paměť v počítači, která tak může být využita jinde.

Problémem při relokaci většího (menšího) statického pole je zvolit o kolik má být nové pole větší. Protože alokace nového pole, kopírování prvků a uvolňování paměti je z výpočetního hlediska náročná operace je třeba zvolit kompromis mezi výkonem a efektivním využití paměti počítače. V případě, že je relokováno pole po menších částech, může docházet často k potřebě relokovat staré pole. V případě, že je relokováno po větších částech, může dojít k neefektivnímu využití paměti.

Dynamické pole je poměrně rychlá datová struktura, ale její slabina tkví v potřebě jednou za čas relokovat, zvětšit statické pole, které vnitřně využívá. Z tohoto důvodu se nehodí do real-time systémů, kde by tato prodleva mohla způsobit nežádoucí zpoždění.

Dynamické pole se hodí všude tam, kde je třeba rychlý přístup k uloženým datům, a dynamicky přidávat nové prvky - není předem znám jejich počet pro vytvoření statického pole.

5.1 Výpočetní složitost relokace

Největší slabina dynamického pole je v potřebě jednou za čas relokovat statické pole. Výkon dynamického pole tedy závisí na tom jak často je třeba jej relokovat. Při implementaci dynamického pole je třeba zvolit nejoptimálnější způsob jak (o kolik) při relokaci zvětšovat statické pole. Je nutné zvolit kompromis mezi výkonem a využitím paměti. Podle toho zda je třeba optimalizovat dynamické pole pro výkon nebo pro optimální využití paměti je třeba zvolit metodu relokace, respektive rychlost zvětšování dynamického pole.

5.2 Implementace dynamického pole

Dynamické pole je vnitřně tvořeno několika proměnnými většinou uzavřenými do struktury, nebo do třídy. Tyto proměnné jsou:

- **iSize** - pole typu int, které určuje aktuální velikost statického pole.
- **iIndex** - pole typu int, ukazující na nový prvek dynamického (statického) pole.
- **iList** - statické pole, které tvoří dynamické pole.

Dále je tvořeno několika funkcemi (metodami), které pracují s vnitřními proměnnými. Tyto proměnné jsou zvnějšku nepřístupné, aby uživatel dynamického pole nemohl ovlivnit funkčnost dynamického pole. Těmito funkcemi jsou:

- **init** - jedná se o obdobu konstruktoru, v této funkci se inicializují vnitřní proměnné.

• **deInit** - jedná se o obdobu destruktoru, v této funkci se uvolňuje paměť vnitřních proměnných.
• **add** - funkce přidá hodnotu do statického pole a inkrementuje hodnotu proměnné index. V případě, že je statické pole již plné, zavolá funkci relok.
• **removeItem** - opak funkce add. Dekrementuje se hodnota proměnné index. V případě, že počet nezaplněných položek statického pole překročí určitou mez, relokuje se statické pole na menší velikost, aby se ušetřilo místo v paměti.
• **get** - funkce vrací prvek ve statickém poli na daném indexu.
• **size** - funkce vrací počet zaplněných položek ve statickém poli.
• **relok** - chráněná funkce, která slouží ke změně velikosti statického pole.

5.2.1 Vytváření a mazání dynamického pole

Při vytváření dynamického pole se musí nejprve vytvořit datová struktura s proměnnými a následně se musejí jednotlivé proměnné inicializovat na počáteční hodnotu. Odkaz na danou strukturu se vrátí jako výsledek dané funkce. V případě konstruktoru třídy se vytvoření proměnných děje automaticky.

```
iArrayList* temp = (iArrayList*) malloc(sizeof(iArrayList));
temp -> iIndex = 0;
temp -> iSize = 4;
temp -> iList = (int32_t*) malloc(sizeof(int32_t)*temp -> iSize);
return temp;
```

Při mazání dynamického pole je třeba uvolnit zabranou paměť, aby v programu nedošlo k únikům paměti. Zároveň je z bezpečnostních důvodů vymazána adresa na paměť na které se nacházel objekt dynamického pole z jeho ukazatele. K tomu je třeba předat do funkce deinit ukazatele na ukazatele na objekt dynamického pole. V případě třídy se toto děje v destruktoru.

```
free((*list)->iList); //uvolnění statického pole
free((*list)); // uvolnění objektu dynamického pole
(*list) = NULL; // vymazání adresy dynamického pole z jeho ukazatele
```

5.2.2 Práce s prvky dynamického pole

Aby byla zajištěna automatická relokace statického pole je třeba k dynamickému (statickému) poli přistupovat pomocí obslužných funkcí (metod). Funkce add slouží k přidávání prvků do pole. Funkce zkontroluje zda není statické pole již plné a vloží uloží do něj nová data. V případě, že je již zaplněno je třeba zavolat funkci relok, která zvětší kapacitu statického pole.

```
if(list->iIndex >= list->iSize)
{
```

```
  relok(list, list->iSize*2);
}
list->iList[list->iIndex] = iData;
list->iIndex++;
```

Funkce get slouží pro přístupu k prvkům dynamického pole. Funkce přijímá index adresující prvek statického pole. Funkce get jednoduše zkontroluje zda je hodnota indexu v přípustném intervalu a jako výsledek své operace vrátí hodnotu uloženou v poli na daném indexu. V případě, že index není v přípustném intervalu, funkce vrací hodnotu NULL.

```
if(iIndexOut <= list->iIndex)
{
  return list->iList[iIndexOut];
}
else
{
  printf("Bad index size!!!\n");
  return NULL;
}
```

Mazání dat z dynamického pole probíhá tak, že je dekrementována proměnná iIndex. Tím se zvýší počet volných prvků dynamického pole. Tím je ale vymazán prvek, který se nachází na konci statického pole. Aby byl smazán prvek umístěný na libovolném místě v poli, je třeba prvky uložené v poli setřásat. Jednoduše se všechny ostatní prvky nacházející se za vymazaným prvkem přesunou o jeden prvek směrem k menšímu indexu.

```
if(iIndexRem < list->iIndex)
{
  uint32_t i;
  for(i = iIndexRem; i < list->iIndex; i++)
  {
    list->iList[i] = list->iList[i+1];
  }
  list->iIndex--;
  /*v případě, že je v poli obsazených pouze polovina prvků, je pole zmenšeno aby se
ušetřila paměť*/
  if(((list->iSize+1)/list->iIndex) == 2)
  {
    relok(list, list->iSize/2);
  }
}
else
```

```
{
  printf("Bad index size!!!\n");
}
```

Při relokaci dochází ke změně velikosti statického pole. Velikost statického pole se může změnit směrem nahoru a nebo směrem dolů. Směrem nahoru se mění v případě, že dojde místo ve statickém poli. Směrem dolů se mění v případě, že statické pole má příliš mnoho neobsazených položek. Množství neobsazené paměti se kontroluje ve funkci removeItem. Při relokaci se vytvoří nové pole, které je buď větší a nebo menší než pole původní. Do tohoto nového pole se nakopírují položky z původního pole a následně se původní pole uvolní z paměti (zruší se). Nakonec se pouze přepíše adresa původního pole adresou nového pole. Vnitřně se pouze změní velikost pole potře aktuální potřeby, bez jakéhokoli zásahu z venčí.

```
//vytvořit pomocné pole
int32_t* pom = (int32_t*) malloc(sizeof(int32_t)* iNewSize);
//nakopírovat prvky do nového pole
uint32_t i;
for(i = 0; i < list->iIndex; i++)
{
  pom[i] = list->iList[i];
}
//uvolnit paměť
free(list->iList);
list->iList = pom;
list->iSize = iNewSize;
```

6. Spojový seznam

Spojový seznam je dynamická datová struktura, která primárně slouží pro uložení dat a je základem mnoha složitějších datových struktur. Spojový seznam funguje podobně jako pole hodnot, ale odstraňuje jeho hlavní nevýhodu - statický počet prvků. Využívá se všude tam kde je nutné ukládat proměnné množství dat. Jeho jedinou nevýhodou je nižší rychlost přístupu k uloženým datům.

6.1 Popis spojového seznamu

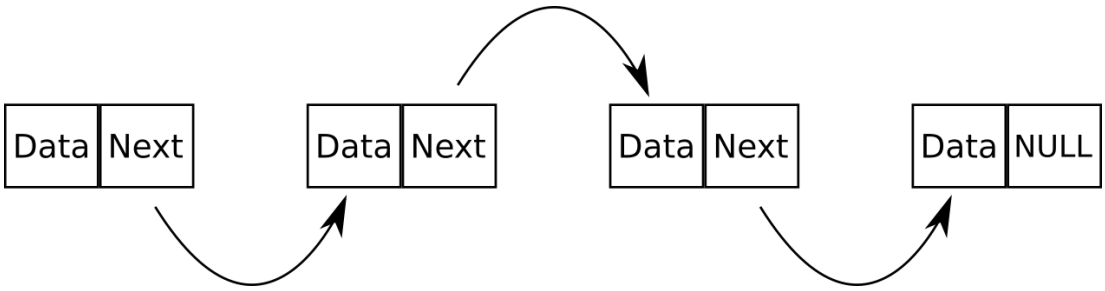
Spojový seznam je tvořen jednotlivými prvky (podobně jako v případě pole), které se nazývají uzly. V základu každý uzel obsahuje data a adresu následujícího uzlu v seznamu. Díky tomu lze přidat další uzel pouze přepsáním adresy posledního prvku v seznamu. Dále lze jednoduše (téměř stejným způsobem) vkládat nové prvky doprostřed seznamu. Protože každý uzel je samostatná datová struktura, která dohromady tvoří komplexní datové úložiště. V paměti pak nejsou většinou jednotlivé uzly uloženy vedle sebe ale náhodně v rámci paměti procesu. Při přístupu k datům uložených v konkrétním uzlu je nutné projít všechny uzly, které danému uzlu předcházejí. To zvyšuje výpočetní náročnost a zvyšuje čas přístupu k datům. Tento nedostatek je částečně řešen několika způsoby.

Rozlišuje se několik druhů spojových seznamů:

- Jednosměrný spojový seznam
- Obousměrný spojový seznam
- Rozšířený obousměrný spojový seznam

6.1.1 Jednosměrný spojový seznam

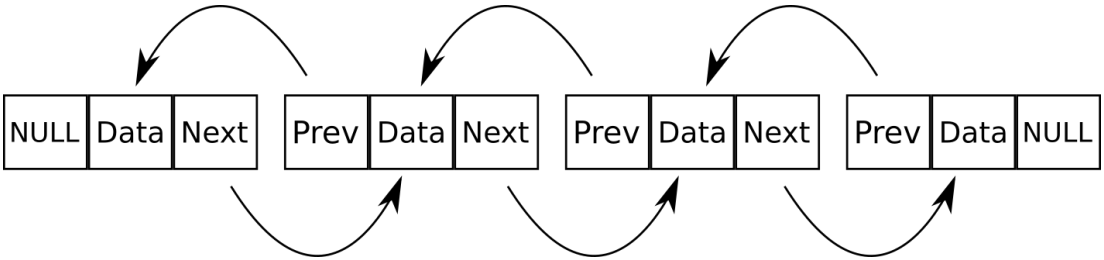
Jednosměrný spojový seznam je základním a nejjednodušším typem spojového seznamu. Bohužel je také nejméně efektivní v přístupu k datům (především u většího počtu uzlů). V jednosměrném spojovém seznamu obsahuje každý uzel kromě dat pouze adresu následujícího uzlu v řadě. Jako výchozí pozice je brán první uzel v seznamu a je nutné pro nalezení adresy uzlu na daném indexu projít všechny předcházející uzly.



6.1.2 Obousměrný spojový seznam

Uzly obousměrného spojového seznamu na rozdíl od jednosměrného spojového se-

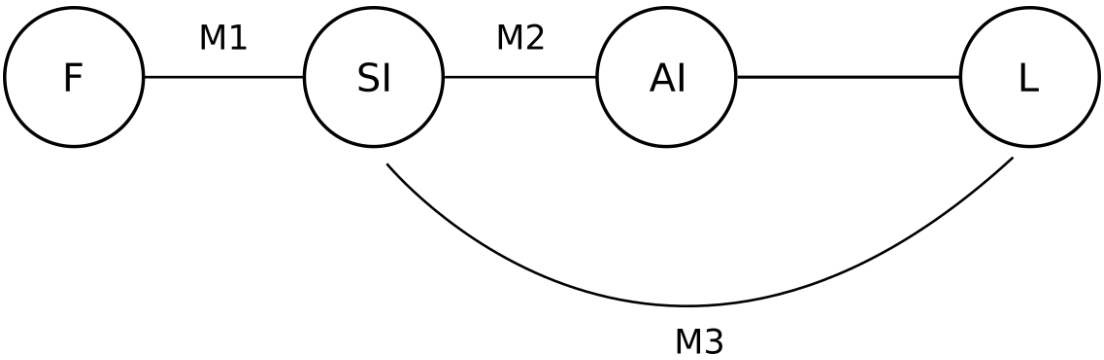
znamu obsahují kromě dat adresu následujícího uzlu a adresu předchozího uzlu. Díky tomu lze hledat konkrétní uzel v seznamu oběma směry. Pro orientaci v seznamu se využívá celočíselná proměnná, která uchovává index (pořadí) uzlu na který se aktuálně ukazuje (na který je dostupná paměťová adresa). Na základě aktuálního indexu na který spojový seznam ukazuje a indexu hledaného uzlu je rozhodnuto zda se bude daný uzel hledat směrem dopředu nebo dozadu. Protože počáteční pozice při hledání daného uzlu může být kdekoli v seznamu, znamená to, že počet průchodů uzly, které předcházejí hledanému je menší než v případě jednosměrného spojového seznamu. Spojový seznam následně vždy ukazuje na poslední uzel na který bylo přistupováno.



6.1.3 Rozšířený obousměrný spojový seznam

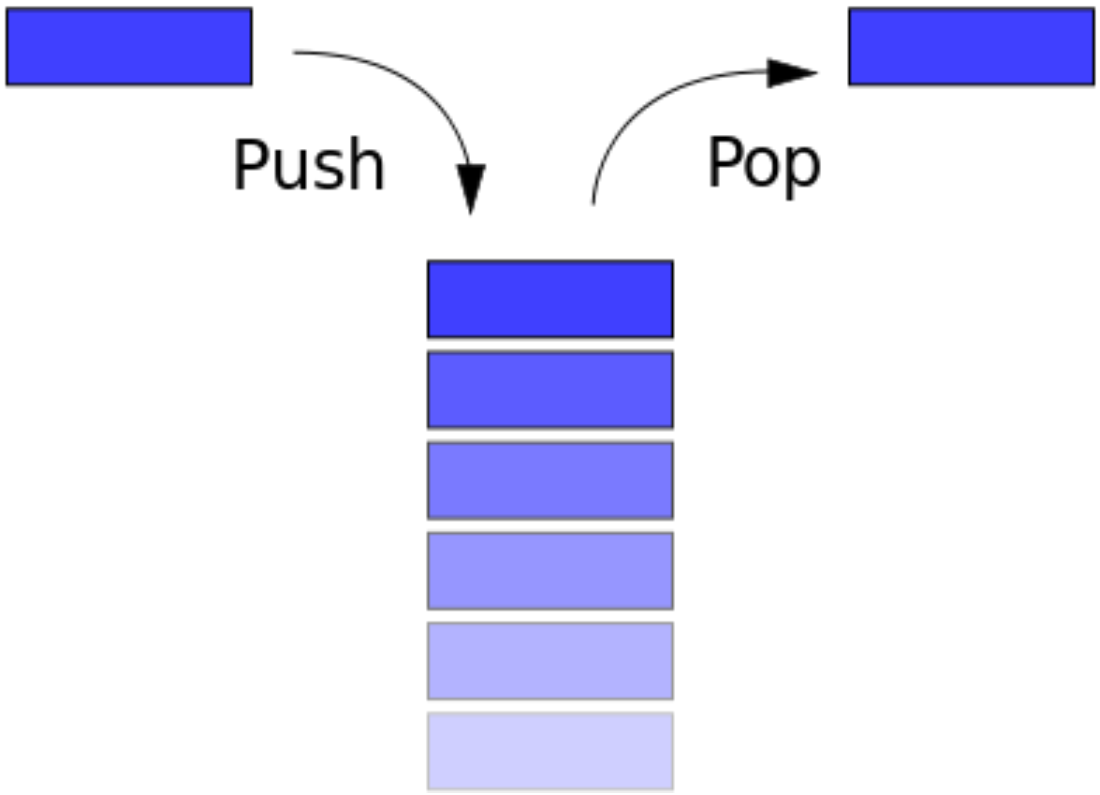
Rozšířený obousměrný spojový seznam je zabalen v další datové struktuře, která uchovává kromě indexu aktuálního uzlu a jeho adresy také adresu prvního a posledního uzlu ve spojovém seznamu. Díky tomu je možné využít rozšířené možnosti vyhledávání konkrétního uzlu podle indexu. Při vyhledávání je využita teorie grafu. Pokud je struktura spojového seznamu zobrazena do grafu, jsou pro vyhledávání uzlů důležité tři metrické údaje:

- Vzdálenost hledaného uzlu na daném indexu od prvního uzlu seznamu - M1.
- Vzdálenost mezi uzlem na indexu na který seznam aktuálně ukazuje a indexu hledaného uzlu - M2.
- Vzdálenost hledaného uzlu na daném indexu od posledního uzlu v seznamu M3.



7. Zásobník

Zásobník je datová struktura, která se používá pro dočasné uložení dat. Zásobník je charakteristický způsobem manipulace s daty typu LIFO - Last In, First Out. To znamená, že poslední vložená data jsou přečtena jako první. Při práci se zásobníkem se používají dvě základní operace - PUSH a POP. Operace **PUSH** slouží k vložení dat do zásobníku a operace **POP** k přečtení dat uložených v zásobníku. Důležité je, že zásobníková datová struktura neumožňuje náhodný přístup k datům jako v případě statického nebo dynamického pole. Datová struktura typu LIFO v praxi znamená, že aby mohla být přečtena konkrétní data, musejí být nejprve přečtena všechna data, která byla po těchto datech do zásobníku vložena.



Zásobník pro ukládání a čtení používá celočíselná hodnota (proměnná), které se nazývají **ukazatel na vrchol zásobníku**. Ukazatel na vrchol zásobníku v sobě uchovává adresu naposledy vložených dat v paměti. Při ukládání dat do zásobníku se ukazatel na vrchol zásobníku inkrementuje na následující volnou datovou položku v paměti a do ní se uloží nová data. V případě čtení se nakopírují data na vrcholu zásobníku do místa čtení zásobníku a ukazatel na vrchol zásobníku dekrementuje. Přečtená data se

sice ze zásobníku nevymažou, ale je na ně vymazaná reference a následující uložení nových dat stará data přepíše.

Statické polo je charakteristické tím, že umožňuje uložit pouze omezené množství datových prvků. Toho je využíváno v situacích, kdy je

7.1 Implementace statického zásobníku

Statické pole je vnitřně tvořeno statickým polem o určité velikosti. Pomocí proměnné, která tvoří ukazatel na vršek zásobníku je toto pole indexováno a zároveň kontrolováno, zda již pole není plné. Proměnná ukazatel na vršek zásobníku je celočíselná proměnná, která vnitřně uchovává indexy jednotlivých prvků ve statickém poli. V případě, že je pole již plné, není možné do něj zapsat další hodnoty a je nutné tuto událost v programu ošetřit. Pro kontrolu obsazenosti pole je pouze porovnávána hodnota ukazatele na vrchol zásobníku a konstanty udávající velikost statického pole tvořící zásobník.

8. Fronta

Fronta je datová struktura typu FIFO - Firest In Firest Out. To znamená, že první vložená data jsou také jako první přečtena. V programování se nejčastěji využívá jako synchronizační primitivum. Data, která jsou posílána v určitém pořadí jsou díky frontě také ve stejném pořadí zpracována.

9. Strom

Strom je datová struktura využívající principy z **teorie grafů**. Jedná se o datovou strukturu používanou k ukládání a rychlému vyhledávání uložených dat. Jedná se o podobnou datovou strukturu jako spojový seznam.

9.1 Části binárního stromu

Pro popis datové struktury stromu se používají některé pojmy, které jsou inspirovány zkutečným stromem. Jednotlivé části stromu tvoří **větve stromu**. Větev je spojením mezi dvěma body tvořící strom. **Uzel** je termín používaný k popisu ukončovacího bodu větve. V binárním stromu existují tři typy uzlů:

• **Počáteční uzel** - nazývá se také **kořenový uzel** a nachází se ve stromové struktuře na nejvyšší úrovni.
• **Uzel větve** - z kořenového uzlu vedou větve k uzlům větve
• **Koncový uzel** - jedná se o uzel, který se nevětví na další větve. Takový uzel se také nazývá **listový uzel**, protože na konci každé větve se u zkutečného stromu nachází list. Takový uzel se nachází v dolní části stromu.

Uzel větve představuje jakési rozcestí, které spojuje daný uzel se dvěma větvemi. Každá větev je zakončena podřízeným uzlem. Tyto větve jsou označovány jako *pravá větev* a *levá větev*.

U datové struktury stromu je důležitý vztah nadřízenosti (nadřazenosti) a podřízenosti mezi jednotlivými uzly. Tento vztah je relativní. Všechny uzly s výjimkou kořenového uzlu mají svůj nadřízený uzel. Některé uzly ale nemají ani žádné podřízené uzly. Pro určení vztahu mezi uzly je nutné vzít jeden uzel, který je označován jako **aktuální uzel**. Uzel ze kterého vzniká aktuální uzel se označuje jako **nadřazeným uzlem** aktuálního uzlu. Uzel nebo uzly, které vznikají z aktuálního uzlu jsou označovány jako **podřízené uzly**. Podřízený uzel může být ještě označován jako **pravý** nebo **levý uzel** podle toho kterým směrem se uzel větví z aktuálního uzlu.

9.2 Hloubka a velikost stromu

Binární strom lze popsat pomocí dvou rozměrů: **hloubka stromu** a **velikost stromu**.

Hloubka stromu je počet úrovní stromu. Každé rozvětvení aktuálního uzlu k podřízenému uzlu vytváří novou úroveň.

Velikost stromu je počet uzlů ve stromu. Velikost stromu s daným počtem úrovní nelze přesně určit, ale lze ji odhadnout, určit maximální množství uzlů, které strom s daným počtem úrovní může obsahovat. To je možné pomocí vzorce:

$$velikost \approx 2^{hloubka}$$

Velikost stromu je jen přibližná, protože strom nemusí být vyvážený. **Vyvážený strom** je binární strom, jehož každý uzel má dva podřízené uzly. U **nevyváženého stromu** je možné najít u několika uzlů méně než dva podřízené uzly.

9.3 Klíč

Každý uzel stromu obsahuje klíč, který je spojen s konkrétní uloženou hodnotou uvnitř stromu. Klíč je hodnota, která je porovnávána s kritérii vyhledávání. Pokud se index a kritéria vyhledávání shodují, dojde k načtení uložených dat z uzlu odpovídají daném klíči. Jako klíč může být použit jakýkoli typ dat.

10. Hašovací tabulka