1. Základní pojmy

Cílem této úvodní kapitoly je představit základní programovací prostředky pro reprezentaci množin a připomenout pojmy, s nimiž se budeme setkávat při návrzích datových struktur a při jejich analýze. V první sekci připomeneme základní programovací techniky pro ukládání množin a pro práci s nimi. Zadefinujeme základní operace a popíšeme reprezentace množin pomocí polí a spojových seznamů. Dále připomeneme struktury zásobník a fronta. Druhá sekce bude věnována základům pravděpodobnosti, která hraje významnou roli při analýze datových struktur. Připomeneme si nejen samotné pojmy, ale i některá důležitá fakta a obraty, které budeme později používat. Obsahem třetí sekce budou grafy. Tato kombinatorická struktura tvoří základ mnoha datových struktur. Seznámíme se zde s výchozími pojmy, přičemž zvláštní pozornost bude věnována stromům, kterými se budeme dále podrobně zabývat v samostatných kapitolách. Konečně poslední sekce bude věnována nástrojům pro porovnávání algoritmů – teorii složitosti. Řekneme si něco o abstraktním modelu, ve kterém počítáme složitost algoritmu, a o jeho praktické použitelnosti.

1.1 Reprezentace množin

V celé knize (až na výjimky) budeme předpokládat, že reprezentované množiny jsou uloženy v interní paměti počítače, která je navíc homogenní (tj. neobsahuje cache s vyšší rychlostí přístupu k datům než má zbytek paměti). Dále pro jednoduchost budeme vždy ztotožňovat celé záznamy s jejich klíči. To není ve skutečnosti žádné omezení, protože pokud je záznam podstatně větší než klíč, pak jeho další složky lze uložit na libovolné jiné místo v paměti a klíč s nimi spojit ukazatelem, takže případná změna pozice klíče nezmění pozici zbytku záznamu.

Časovou složitost algoritmů budeme vyjadřovat počtem provedných kroků, a to tak, že vždy nalezneme klíčovou akci (v našem případě to obvykle bude počet porovnání) a odhadneme, kolikrát se během výpočtu opakuje. Celkový počet těchto akcí budeme obvykle shora odhadovat asymptoticky, tj. až na multiplikativní konstantu. Protože ale pro praktické použití algoritmu je důležité znát co nejpřesnější odhad, budeme pro nejpoužívanější techniky uvádět i konkrétní hodnoty parametrů, ovšem jen těch, které jsou nezávislé na použitém počítači. Pokud

nebude řečeno jinak, budeme předpokládat 'rozumně velké' hodnoty argumentů operací, takže všechny operace stejného typu budou stejně 'drahé' bez ohledu na jejich argument.

Problém odhadu složitosti formálně rozebereme v sekci 1.4. Obsahem této sekce je popis základního problému datových struktur a základních technik jeho řešení (další komplikovanější, ale efektivnější strategie budou obsahem dalších kapitol knihy).

Základní problém studovaný v teorii datových struktur je slovníkový problém. Tento problém se týká reprezentace množin a operací s nimi a lze ho zformulovat takto:
Je dáno **konečné** univerzum U a úkolem je reprezentovat množinu $S \subseteq U$. Cílem je spolu s reprezentací navrhnout algoritmy realizující následující operace:

- (•) **MEMBER**(x, S) – zjistí, zda prvek $x \in U$ patří do reprezentované množiny S, a oznámí výsledek, případně vrátí pozici x v reprezentaci S,
- (•) **INSERT**(x, S) – vytvoří reprezentaci množiny $S \cup \{x\}$,
- (•) **DELETE**(x, S) – vytvoří reprezentaci množiny $S \setminus \{x\}$.

Operace **INSERT** a **DELETE** se nazývají aktualizační operace. Jejich implementace musí zajistit i správnou funkci algoritmů pro nové reprezentované množiny. Zdůrazňujeme, že při řešení slovníkového problému budeme obvykle předpokládat, že univerzum je sice konečné, ale o hodně větší než je reprezentovaná množina.

Tuto základní verzi slovníkového problému řeší například hašování, které podrobně popíšeme ve druhé kapitole. Ve třetí kapitole budeme vyšetřovat tzv. uspořádaný slovníkový problém, což je modifikace pro univerza, která jsou totálně uspořádaná. Tohoto uspořádání lze využít při konstrukci struktur (např. vyhledávacích stromů) a kromě toho umožňuje ještě další operace, třeba hledání k-tého nejmenšího prvku, rozdělení množiny na prvky menší a větší než je zadaná hodnota apod.

Nyní se seznámíme s elementárními datovými strukturami pro řešení slovníkového problému – jsou to pole a spojový seznam.

1.1.1 Pole hodnot

Množinu $S = \{s_1, s_2, \ldots, s_n\}$ reprezentujeme polem $P[1..n]$ tak, že $P(i) = s_i$ pro každé $i = 1, 2, ..., \ldots n$. Vyhledávání v tomto poli probíhá tak, že postupně porovnáváme jeho prvky s hledanou hodnotou, dokud nenajdeme shodu nebo nenarazíme na konec pole. Formálně tento algoritmus vypadá takto:

MEMBER(x, S):
for $i := 1$ **to** $|S|$ **do**
  if $P(i) = x$ **then Výstup**: $x \in S$, konec **endif**
enddo
Výstup: $x \notin S$

Ke zjištění, že hledaný prvek se v množině nevyskytuje, musí algoritmus otestovat všechny hodnoty v poli. Ale i v případě, že se v ní hledaný prvek nachází, bude třeba projít téměř celé pole, pokud očíslování prvků množiny bude nepříznivé vzhledem k hodnotě hledaného argumentu. Lze ukázat, že když množina S bude vybrána z U náhodně podle rovnoměrného rozdělení a očíslování zvolíme rovněž náhodně, pak v průměrném případě navštívíme přibližně $\frac{|S|}{2}$ hodnot pole. V dalších kapitolách ukážeme, že existují očíslování, která výrazně urychlí práci algoritmu.

Při implementaci aktualizačních operací **INSERT** a **DELETE** v této struktuře se musí řešit problém, že tyto operace mění velikost reprezentované množiny, zatímco velikost pole je pevně dána jeho deklarací. Proto se obvykle pracuje s polem $P[1..m]$, kde $m \geq n = |S|$, ale nesmí být příliš velké vzhledem k n. I přesto se může stát, že operaci **INSERT** nelze provést, protože nový prvek už není kam uložit, nebo naopak že v důsledku operace **DELETE** se zmenší velikost S tak, že m překročí vzhledem k n akceptovatelný poměr. Standardním řešením tohoto problému je realokace pole P, při níž se v průběhu výpočtu podle aktuální situace vytvoří nové pole dvojnásobné, resp. poloviční velikosti, do kterého se prvky reprezentované množiny uloží (tento postup se také používá např. v hašování, jak uvidíme později). To ale vyžaduje určité množství času navíc. Proto se pole používá hlavně jako statická, nikoli dynamická struktura.

Výhodou pole je okamžitý přístup k prvku na zadané pozici, čehož využívají například některé třídicí algoritmy (heapsort), ale i algoritmy pro řešení slovníkového problému (vyhledávání v uspořádaném poli popsané ve třetí kapitole). Protože obecně nejsou dány žádné požadavky na způsob očíslování prvků množiny S, můžeme tuto strukturu výhodně použít v uspořádaném univerzu, když budeme volit očíslování podle rostoucího (klesajícího) pořadí hodnot.

1.1.2 Booleovské pole

V zadání slovníkového problému jsme vyslovili předpoklad, že množina, se kterou pracujeme, je částí nějakého většího univerza U. Je-li univerzum celočíselné, můžeme pro reprezentaci množiny S použít booleovské pole $BP[1..|U|]$, v jehož složkách budou uloženy hodnoty charakteristické funkce množiny S v U, tj. $BP(x) = 1$ pro $x \in S$, $BP(x) = 0$ pro $x \notin S$. Vyhledávání spočívá pouze ve zjištění, zda na příslušné pozici v poli je 1 nebo 0, přidání prvku x do množiny S znamená nastavení hodnoty $BP(x) = 1$ a odebrání prvku x znamená naopak nastavení hodnoty $BP(x) = 0$. Implementace těchto operací vyžaduje konstantní počet kroků. Bohužel nároky na paměť jsou značné, velikost požadované paměti je rovna velikosti univerza a to může být v některých případech neúnosné. Pro příklad – podnikatel, který má několik desítek zaměstnanců a vede si jejich evidenci podle rodných čísel, by musel pro tuto reprezentaci použít univerzum skládající se (skoro) ze všech možných desetimístných čísel. Další nevýhodou této struktury je neefiktivnost operací spojených s uspořádáním univerza. Např. hledání nejmenšího prvku vyžaduje prohlédnutí tolika hodnot pole, kolik je velikost nejmenšího prvku. Protože je přirozené

předpokládat, že $|S| << |U|$, bude v průměrném případě zapotřebí $\frac{|U|}{|S|}$ kroků, ale v nejhorším případě až $|U| - |S|$ kroků.

1.1.3 Spojové seznamy

Pole hodnot, které jsme popsali v podsekci 1.1.1, je jednou z možných implementací seznamu $s_1, s_2, \ldots, s_n$, kde prvek s_{i+1} je následníkem a prvek s_{i-1} předchůdcem prvku s_i (s výjimkou posledního prvku, který nemá následníka, a prvního, který nemá předchůdce). Protože prvky pole jsou uloženy v paměti počítače v daném pořadí bezprostředně za sebou, není k nalezení následníka či předchůdce zapotřebí žádný ukazatel, a prvek tak může být reprezentován jen svojí hodnotou. Za tuto jednoduchost se ovšem platí problémy při vkládání a mazání. Jinou implementací, jejímž hlavním cílem je umožnit jednoduchou a přirozenou realizaci operací **INSERT** a **DELETE**, je spojový seznam. Tato struktura na rozdíl od reprezentace polem nemusí mít předem alokovanou paměť a její velikost se proto může dynamicky měnit. Charakteristické pro ni je, že k prvkům se přistupuje pomocí ukazatelů. Hodnotou ukazatele je buď adresa, na které je v paměti uložen prvek, na nějž ukazuje, nebo speciální 'prázdná' hodnota NIL. Popíšeme dvě její verze – jednosměrný spojový seznam a dvousměrný spojový seznam.

Ve spojovém seznamu je každý prvek reprezentován záznamem, který je tvořen klíčem *key*, ukazatelem *next* na následující prvek v seznamu (u posledního prvku je $next = \text{NIL}$) a položkou *data*, která obsahuje buď další data spojená s daným klíčem nebo ukazatel na místo, kde jsou uložena (v našich algoritmech ji nebudeme používat). Ve dvousměrném seznamu je ještě navíc ukazatel *previous* na předchozí prvek (u prvního prvku je $previous = \text{NIL}$). Kromě toho je se jménem seznamu vždy spojen ukazatel $first$ na první prvek v seznamu a ve dvousměrném spojovém seznamu navíc ukazatel *last* na poslední prvek seznamu. V případě, že seznam je prázdný, je $first = \text{NIL}$, u dvousměrného seznamu i $last = \text{NIL}$.

Vyhledávání probíhá tak, že podle ukazatele $first$ najdeme začátek seznamu, který pak procházíme pomocí ukazatele *next*, dokud nenastane shoda klíče s hodnotou hledaného prvku nebo se nedojde na konec seznamu (dvousměrný seznam lze procházet i od konce daného ukazatelem *last* pomocí ukazatele *previous*). Pokud hledaný prvek v seznamu není, může být přidán operací **INSERT** tak, že se pro něj naalokuje paměť, vytvoří se záznam s příslušnou hodnotou klíče a pomocí nastavení ukazatelů se do seznamu zařadí (obvykle na poslední místo). Pokud prvek v seznamu je a chceme ho smazat, 'přeskočíme' přenastavením ukazatelů sousedních záznamů jeho místo v paměti, které následně uvolníme (záznam fyzicky smažeme).

V praxi se lze setkat i s obecnějším způsobem vyhledávání. Hledáme první (respektive poslední) prvek v seznamu podle daného klíče při současném splnění dané podmínky P, jejíž tvar musí zároveň zajistit ukončení vyhledávání v případě, kdy prvek v seznamu není. Prohledávání provádíme tak, že kontrolujeme, zda je podmínka P splněna a zda klíč má předepsanou hodnotu. Skončíme, když nalezneme prvek se zadaným klíčem, nebo když není splněna P. Protože neúspěšné hledání

prvku v tomto případě může skončit i na jiném místě než na konci seznamu, vkládá se při operaci **INSERT** nový záznam s příslušnou hodnotou klíče na toto místo přenastavením ukazatelů sousedních záznamů.

Příklady podmínky P:

- (•) při hledání záznamu s klíčem x prohledáváním celého seznamu má podmínka P tvar $t.next \neq \text{NIL}$ a prohledávání pokračuje, dokud je $t.key \neq x$ a $t.next \neq \text{NIL}$ (kde t ukazuje na prvek seznamu, první část testu kontroluje hodnotu klíče, druhá splnění podmínky P) – navíc musíme testovat, zda seznam není prázdný (to testuje podmínka $first \neq \text{NIL}$);
- (•) jsou-li záznamy v seznamu uspořádány v rostoucím pořadí podle klíčů, pak podmínka P má tvar $t.key < x$ a $t.next \neq \text{NIL}$, tedy prohledávání pokračuje, dokud $t.key \neq x$ a $t.key < x$ a $t.next \neq \text{NIL}$ (navíc test na neprázdnost seznamu $first \neq \text{NIL}$);
- (•) jsou-li záznamy v seznamu uspořádány v rostoucím pořadí podle klíčů a chceme-li najít největší prvek s klíčem menším nebo rovným x, pak podmínka P je $t.next \neq \text{NIL}$ a prohledávání pokračuje, dokud je $t.key \leq x$ a $t.next \neq \text{NIL}$ (na začátku $first \neq \text{NIL}$). Hledaný prvek je tedy buď na konci seznamu, nebo je to předchůdce záznamu, na němž se vyhladávání zastavilo;
- (•) chceme-li zjistit, zda existuje záznam s klíčem x mezi prvními k prvky seznamu, tak nejprve otestujeme neprázdnost seznamu testem $first \neq \text{NIL}$ a pak položíme $i = 1$ (při každém testu na splnění P zvětšíme i o 1), podmínka P bude $i \leq k$ a $t.next \neq \text{NIL}$ a prohledávání bude pokračovat, dokud bude splněno $t.key \neq x$ a ($i \leq k$ a $t.next \neq \text{NIL}$);

Všimněme si rozdílu mezi druhým a třetím příkladem – testy jsou v obou případech téměř stejné, ale liší se jejich zápis, protože se liší cíl vyhledávání.

Následující formální zápis algoritmů je platný pro jednosměrný seznam a pro prohledání celého seznamu (tj. pro první podmínku P z předchozího příkladu).

MEMBER(x, S):
if $S.first = \text{NIL}$ **then**
  Výstup: $x \notin S$
else
  $t := S.first$
  while $t.next \neq \text{NIL}$ a $t.key \neq x$ **do**
    $t := t.next$
  enddo
  if $t.key = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**
endif

INSERT(x, S):
if $S.first = \text{NIL}$ **then**
  vytvoř nový záznam r
  $r.key := x$, $r.next := \text{NIL}$, $S.first := r$
else