

Třetí skupina zadání projektů do předmětu Algoritmy II, letní semestr 2013/2014

doc. Mgr. Jiří Dvorský, Ph.D.

29. dubna 2014

Verze zadání

29. dubna 2014 První verze

1 Entropie slov v textu

1.1 Problém

Máte k dispozici text napsaný v nějakém přirozeném jazyce, například v angličtině. Vaším úkolem je změřit kolik informace tento text obsahuje. Také informace je veličina, která se dá změřit. Správně se množství informace, které je svázáno s nějakým slovem, písmenem, symbolem nazývá *entropie* a jednotka se nazývá *bit* (ano, je to ten samý bit známý z počítačové techniky). Entropie slova A se označuje symbolem $H(A)$ a závisí na pravděpodobnosti $p(A)$ s jakou se slovo A v textu vyskytuje:

$$H(A) = -\log_2 p_A$$

Pravděpodobnost slova určíme jako podíl četnosti daného slova ku celkové četnosti všech slov. Celkové množství informace v textu je rovnou součtu množství informace v jednotlivých slovech.

1.2 Ukázka

Je dán tento vstupní text:

```
computer mouse monitor.
computer, mouse; mouse
mouse monitor. computer
```

Nejprve text rozebereme na slova a zjistíme jejich četnost. V textu se vyskytují tato unikátní slova: **monitor** 3×, **mouse** 4× a **computer** 2×. Celková četnost všech slov je 9. Z četností jednotlivých slov vypočteme jejich pravděpodobnosti a logaritmováním pak entropie jednotlivých slov:

Slovo	Četnost	Pravděpodobnost	Entropie [bity]
computer	3	$\frac{3}{9}$	1,584962501
mouse	4	$\frac{4}{9}$	1,169925001
monitor	2	$\frac{2}{9}$	2,169925001

Celkovou entropii textu potom můžeme vypočítat podle tabulky.

Slovo	Četnost	Entropie	Entropie všech výskytů slova
computer	3 ×	1,584962501	= 4,754887503
mouse	4 ×	1,169925001	= 4,679700004
monitor	2 ×	2,169925001	= 4,339850002
Celková entropie			13,77443751

1.3 Implementace

- Vstupní text je v angličtině, znaky s diakritikou se nebudou vyskytovat, velká a malá písmena se rozlišují.
- Slovo je definováno jako souvislá posloupnost znaků z množiny $\{a, \dots, z, A, \dots, Z\}$.
- Jméno vstupního textového souboru zadáte z klávesnice.
- Na výstup vypíšete jednak seznam všech slov, ke každému slovu jeho četnost a entropii. Na konec vypíšete celkovou entropii daného vstupního textu.
- Pro úspěšné vyřešení tohoto příkladu je vhodné slova a jejich četnosti ukládat do binárního stromu. Pokud by měl někdo zájem, lze tento příklad úspěšně řešit i s využitím hašovací tabulky.
- Výpočet entropie je možné provést až nakonec, kdy už jsou známy konečné hodnoty jednotlivých četností.
- Pro čtení a zápis do souboru můžete použít třídy `ifstream` (čtení) a `ofstream` (zápis) z hlavičkového souboru `fstream`.

2 Vyhodnocení logického výrazu pomocí binárního stromu

2.1 Problém

Spolu s kolegy vytváříte interpret nového programovacího jazyka. Vaším úkolem v týmu je implementovat logické výrazy ve formě binárního stromu a naučit se tyto výrazy vyhodnocovat.

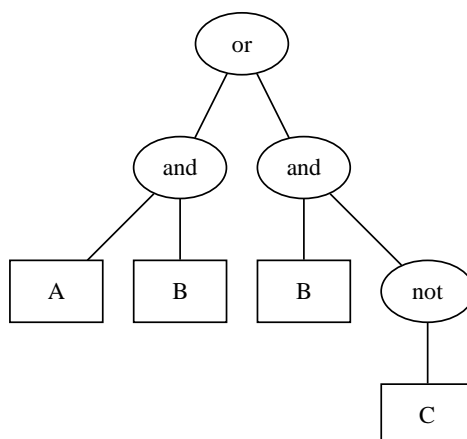
Logické výrazy můžeme totiž velice přehledně zachytit ve formě stromu. Každý uzel představuje jednu logickou operaci, podstromy tohoto uzlu představují podvýrazy. Listy stromu jsou potom logické proměnné, logické proměnné už nemají žádné další podvýrazy, parametry a podobně. Vyhodnocování takto reprezentovaného výrazu je pak velice snadné a lze na něj aplikovat rekurzi. Začneme od kořene stromu, který je v podstatě reprezentantem celého výrazu. Abychom mohli vyhodnotit kořen, musíme vyhodnotit jeho případné podstromy (podvýrazy), následně jejich případné podstromy a tak dále.

2.2 Ukázka

Pro ukázkou předpokládejme, že máme sestavit stromovou reprezentaci logického výrazu

$$A \text{ and } B \text{ or } B \text{ and not } C$$

Z priority operací je zřejmé, že jako poslední se bude vyhodnocovat *or*, to bude tvořit kořen stromu. Jeho podstromy budou podvýrazy *A and B* a *B and not C*. První uvedený podstrom bude mít v kořeni operátor *and* a dále už jen listy, logické proměnné *A* a *B*. Druhý podstrom bude mít v kořeni opět operátor *and*, jeden jeho potomek bude tvořen jen listem s logickou proměnnou *B*, ale druhý bude opět podstrom s operátorem *not* v kořeni a jediným potomkem, listem s logickou proměnnou *C*.



2.3 Implementace

- Logické proměnné při implementaci označte jednoznačnými čísly $0, 1, 2, \dots, N - 1$. V matematice bychom je označili $a_0, a_1, a_2, \dots, a_{N-1}$. Tyto proměnné uložte do tabulky proměnných. Tabulku implementujte jako třídu, hodnoty logických proměnných $a_0, a_1, a_2, \dots, a_{N-1}$ uvnitř třídy uložte do pole velikosti N . N považujte za globální konstantu. Pro přístup k jednotlivým proměnným implementujte metody **void** `SetVariable(const int IndexOfVariable, const bool NewValue)` a **bool** `GetVariableValue(const int IndexOfVariable)`. Pokud budeme chtít změnit hodnotu, kterékoliv proměnné ve výrazu, stačí ji změnit jen v této tabulce, ve stromu s výrazem bude uložen jen odkaz na tuto tabulku, nikoliv proměnná samotná.
- Jako operátory implementujte *and*, *or*, *xor*, *not*. Operátory mají následující prioritu:

Operátor	Priorita
<i>not</i>	nejvyšší
<i>and</i>	
<i>or</i> , <i>xor</i>	nejnižší

Operátory s vyšší prioritou jsou vyhodnocovány dříve než operátory s nižší prioritou.

- Při implementaci tohoto příkladu, lze velice výhodně využít dědičnost a polymorfismus. Vytvoříme abstraktní třídu **Node**, která bude mít čistě virtuální metodu **Evaluate**, která vyhodnotí daný uzel na základě vyhodnocení případných podstromů. Jednotlivé operátory můžeme implementovat jako potomky této třídy a metoda **Evaluate** bude vyhodnocovat *and*, *or*, *xor*, *not*. Každá tato třída si bude udržovat ukazatele na příslušné podstromy, *and*, *or*, *xor* na dva podstromy, *not* na jeden podstrom.
- Obdobně můžeme jako potomka třídy **Node** implementovat i třídu reprezentující logickou proměnnou. Ta bude mít jako atribut index proměnné z rozsahu $0, 1, 2, \dots, N - 1$ a ukazatel na tabulku proměnných. Metoda **Evaluate** bude prostě jen vracet hodnotu logické proměnné daného indexu.
- Logické výrazy, které bude Váš program vyhodnocovat, můžete vytvořit přímo v kódu programu. K tomuto účelu je vhodné implementovat konstruktory tříd s parametry pro přímé nastavení podstromů. Strom pro ukázkový příklad ze zadání můžeme vytvořit touto konstrukcí v C++:

```
VariableTable* table = new VariableTable(N); // N = 3
```

```
Node* root = new Or(  
    new And(  
        new Variable(0, table), // A  
        new Variable(1, table) // B  
    ),  
    new And(  
        new Variable(1, table), // B  
        new Not(  
            new Variable(2, table) // C  
        )  
    )  
);
```

Vyhodnocení pak proběhne zavoláním `root->Evaluate()`;

- V programu demonstrujete sestavení a vyhodnocení několika Vámi zvolených výrazů, zhruba tímto postupem:
 1. Do programu zakódujete pomocí stromu řekněme tři výrazy, můžou mít klidně společnou tabulku proměnných.
 2. Nastavíte proměnné na Vámi zvolené hodnoty, stačí nastavit v kódu programu.
 3. Vyhodnotíte výrazy a vypíšete výsledky.
 4. Změníte hodnoty vybraných proměnných.
 5. Vyhodnotíte výrazy a vypíšete výsledky, čímž by mělo být demonstrováno, že se vypočetlo něco jiného.

3 Tvorba konkordance

Konkordance (z latinského *concordia* = „shoda“, „svornost“) je shoda určitých sledovaných znaků. V literární vědě, zejména biblistice, se tak zpravidla označují i knihy obsahující seznam nebo soupis věcných nebo slovních podobností v různých částech knihy nebo různých knihách. Obvykle jsou konkordance slov sepisovány jako slovníky, tj. podle výrazů v abecedním pořádku. U každého z nich je udáno, kde jej v knize lze nalézt.

Vášim úkolem bude takovou konkordanci sestavit. Jinak řečeno pro zadaný textový soubor sestavíte seznam slov a ke každému slovu vypíšete seznam řádků na kterých se toto slovo vyskytuje. Slova jsou v konkordanci seříděna podle abecedy, řádky s výskyty jsou uváděny vzestupně. Vstupní text budete číst z textového souboru, výslednou konkordanci vypíšete opět do textového souboru.

3.1 Ukázka

Je dán tento vstupní text:

```
computer monitor.  
computer, mouse; mouse  
mouse monitor. computer
```

Výsledná konkordance bude vypadat takto:

```
computer - 0, 1, 2  
monitor - 0, 2  
mouse - 1, 2
```

3.2 Implementace

- Vstupní text je v angličtině, znaky s diakritikou se nebudou vyskytovat.
- Slovo je definováno jako souvislá posloupnost znaků z množiny $\{a, \dots, z, A, \dots, Z\}$.
- Písmena ve slovech je nutné převést na malá, lze využít funkci `tolower` z hlavičkového souboru `ctype.h`, viz zde <http://msdn.microsoft.com/en-us/library/8h19t214.aspx>.
- Číslování řádků začíná od 0.
- Pokud se slovo vyskytuje na jednom řádku vícekrát, počítá se jen jeden výskyt.
- Jméno vstupního textového souboru zadáte z klávesnice. Jméno výstupního textového souboru zadáte z klávesnice.
- Ze zadání je patrné, že prvním problémem je implementace algoritmu, který bude postupně číst znaky z textového souboru, převádět je na malá písmena a sestavovat z nich slova. Pokud načte znak konce řádku, zvýší počítadlo řádku o jedna. Druhou částí programu bude vhodná datová struktura, kde se budou uchovávat nalezená slova a ke každému slovu seznam řádků kde se dané slovo vyskytuje. Zároveň je tu požadavek vypsát slova seříděná podle abecedy. Tomuto požadavku vyhovuje binární vyhledávací strom, kde v uzlu stromu bude uloženo slovo z konkordance a, nejlépe, objekt reprezentující seznam řádků s výskytem slova. Seznam (implementováno jako spojový seznam tj. s využitím pointerů) řádků má být, pro dané slovo, vypsán vzestupně. Protože jsou řádky číslovány vzestupně, stačí nové řádky do příslušných seznamů vkládat nakonec – tím máme rovnou zajištěno, že se seznam bude udržovat seříděný, tak jak je specifikováno v zadání.
- Pro čtení a zápis do souboru můžete použít třídy `ifstream` (čtení) a `ofstream` (zápis) z hlavičkového souboru `fstream`.

4 Hádání zvířat

Napište program, který hádá zvířata na základě otázek a odpovědí ano/ne. Hra může probíhat například takto (odpovědi uživatele jsou vyznačeny tučným písmem):

Mysli si zvíře a já budu hádat.

Má nohy? **Ano**

Je to kočka? **Ano**

Vyhrál jsem! Pokračovat? **Ano**

Mysli si zvíře a já budu hádat.

Má nohy? **Ne**

Je to had? **Ano**

Vyhrál jsem! Pokračovat? **Ano**

Mysli si zvíře a já budu hádat.

Má nohy? **Ne**

Je to had? **Ne**

Jsem poražen! Co to bylo za zvíře? **Žížala**

Napiš mi, prosím, otázku na kterou je pro žížalu odpověď ano a pro hada ne:

Žije pod zemí?

Děkuji, pokračovat? **Ano**

Mysli si zvíře a já budu hádat.

Má nohy? **Ne**

Žije pod zemí? **Ne**

Je to had? **Ne**

Jsem poražen! Co to bylo za zvíře? **Ryba**

Napiš mi, prosím, otázku na kterou je pro rybu odpověď ano a pro hada ne:

Žije ve vodě?

Děkuji, pokračovat? **Ne**

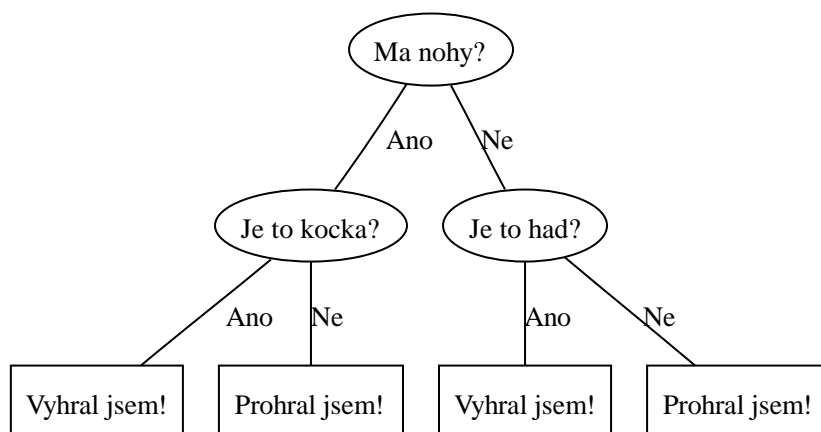
Ahoj

Na začátku program musí mít nějaké minimální znalosti. Ví, že kočka má nohy a had nohy nemá. V okamžiku, kdy program nesprávně hádá, že myšlené zvíře je had, program prohrál a zeptá se uživatele – soupeře, jak se myšlené zvíře jmenovalo – byla to žížala – a jak se dá rozlišit had od žížaly. V dalším kole už umí poznat o jedno zvíře navíc.

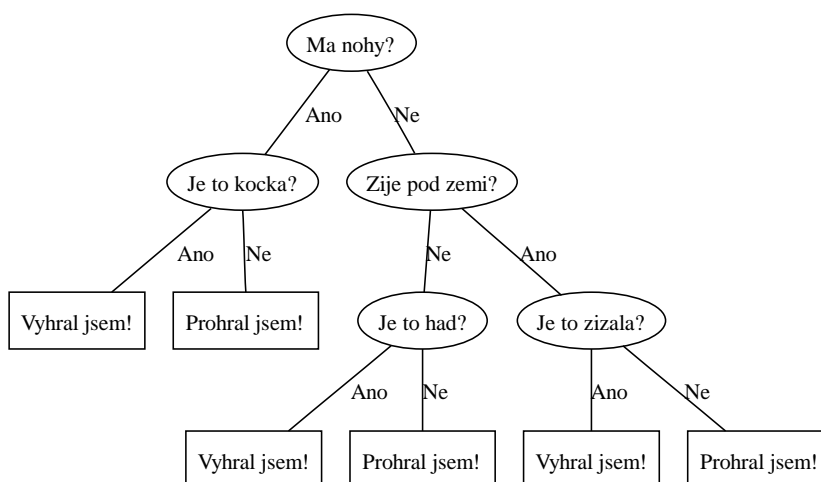
Poznámky

- Program běží v cyklu, hra končí až když uživatel napíše, že nechce pokračovat. Potom program skončí a naučená zvířata se zapomenou. Není nutné řešit ukládání znalostí.
- Nezapínejte se tím, že program nebude umět správně skloňovat. Zvířata se budou zadávat v prvním pádu čísla jednotného a v tomto tvaru se budou i používat.
- Jak hru implementovat? Program si buduje binární strom složený z otázek. První otázka je uložena v kořeni stromu, pomocí odpovědí ANE/NE sestupujeme k potomkům. Vede-li odpověď ANO k levému nebo pravému potomkovi je relativně jedno, zvolte si sami nějaký systém, usnadní Vám to implementaci¹.
- Program buduje binární strom, ne **binární vyhledávací strom**! V binárním stromu není definováno uspořádání nad daty v uzlech, jde tu o vzájemnou návaznost uzlů (otázek, zvířat), o návaznost odpovědí ano/ne, nikoliv o uspořádávání dat stylem „menší vlevo, větší vpravo“.
- Doporučení: zkuste si několik her, nejdříve nakreslit na papír – program není tak složitý, jak vypadá.
- Postupné vytváření stromu otázek můžete vidět na obrázcích 1, 2, 3 a 4.

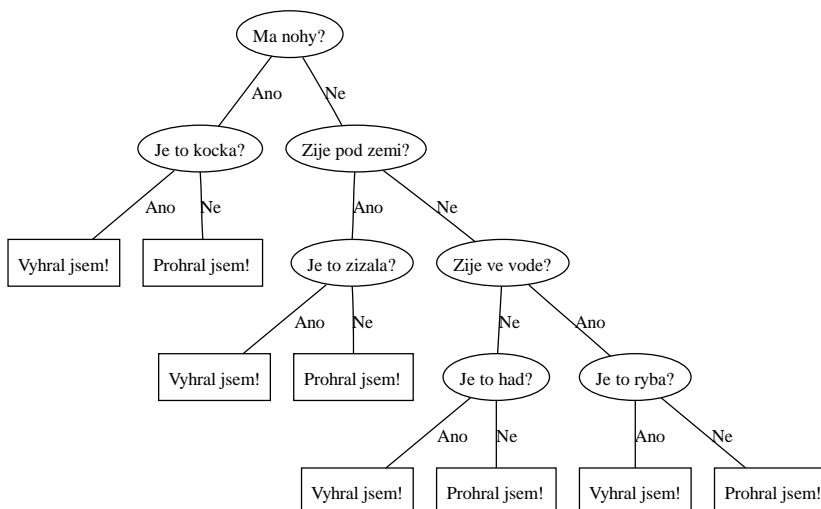
¹ Ukázkové obrázky generoval program, takže je většinou odpověď ANO vlevo, ale někdy se přece jen ANO zatoulalo doprava. Na významu stromu to však nic nemění.



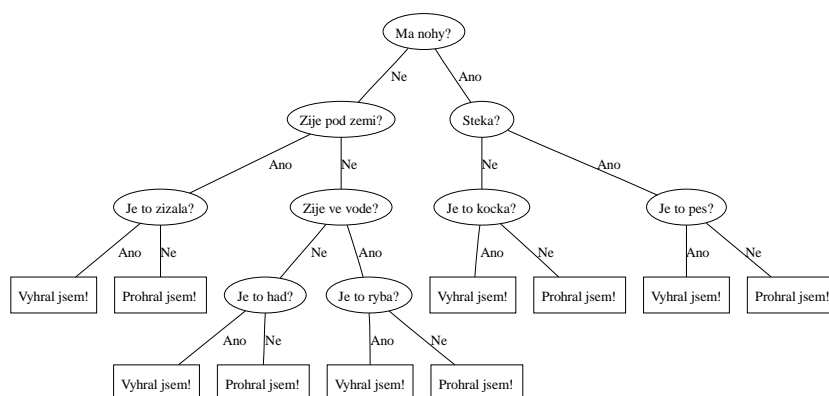
Obrázek 1: Výchozí strom otázek, program zná jen kočku a hada



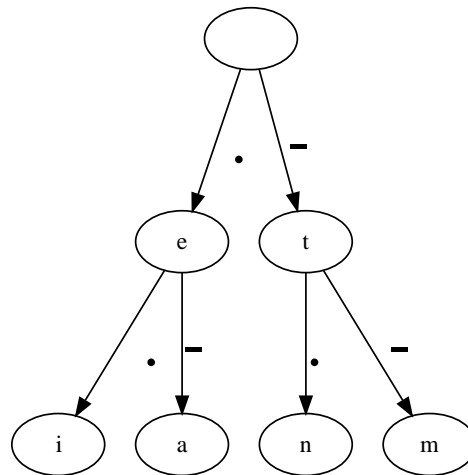
Obrázek 2: Strom otázek po přidání žížaly



Obrázek 3: Strom otázek po přidání ryby



Obrázek 4: Strom otázek po přidání psa



Obrázek 5: Strom pro dekódování morseovky

```
new MorseNode('n', NULL, NULL),  
new MorseNode('m', NULL, NULL)  
);  
);
```

- V uzlu si můžete ukládat další informace podle Vaší potřeby.
- Jakmile dekódujeme dva znaky / za sebou, jde o konec slova a do výstupního textu přidáme, za poslední dekódované písmeno, mezeru.
- Jakmile dekódujeme tři znaky / za sebou, jde o konec věty. V tomto případě přidáme za poslední dekódované písmeno tečku, pak mezeru. A navíc další dekódované písmeno bude velké, písmeno leží na začátku nové věty.
- Zprávu zakódovanou morseovkou bude Váš program číst z textového souboru.
- Dešifrovanou zprávu bude také zapisovat do textového souboru.
- Pro čtení a zápis do souboru můžete použít třídy `ifstream` (čtení) a `ofstream` (zápis) z hlavičkového souboru `fstream`.

6 Výška binárního vyhledávacího stromu

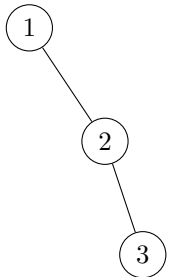
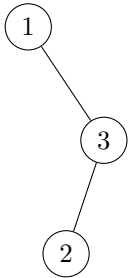
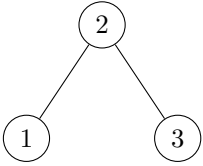
Toto zadání je poměrně prosté. Máte dáno přirozené číslo n . Vygenerujte všech $n!$ permutací čísel $0, 1, 2, \dots, n-1$ a pro každou takto vygenerovanou permutaci vytvořte binární vyhledávací strom. A u tohoto stromu změřte jeho výšku. Celkem dostanete $n!$ výšek (nemusí být pochopitelně všechny různé). Změřené výsledky zpracujte do tabulky.

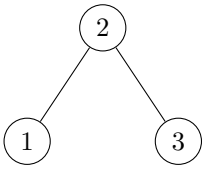
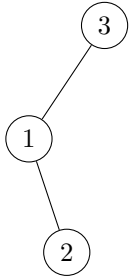
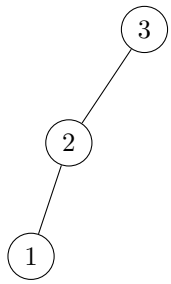
6.1 Implementace

- Jak je definována výška stromu se podívejte do skript.
- Pro systematické generování všech permutací lze vymyslet jednoduchý rekursivní algoritmus pracující s polem, viz například:
 - <http://en.wikipedia.org/wiki/Permutation>
 - <http://www.fit.vutbr.cz/study/courses/ICP/public/Prednasky/STL/node7.html>
 - <http://mff.lokiware.info/KombinatorickeAlgoritmy>
- Permutace není nutné nejprve vygenerovat a pak z nich vytvářet binární stromy. Proces vytváření stromu můžete spustit ihned, jakmile budete mít k dispozici další permutaci. Vytvoříte strom, změříte jeho výšku, kterou si zaznamenáte. Poté můžete strom smazat a přejít k vytváření další permutace.
- Strom z permutace vytvoříme tak, že prvky z permutace postupně vložíte do stromu ve stejném pořadí v jakém jsou zapsány v permutaci. Například, máme-li permutace (1342) vložíte do stromu nejprve prvek 1, pak 3, pak 4 a nakonec 2.
- Zdrojový kód k binárnímu vyhledávacímu stromu je v ukázkových příkladech na webu. Pokud budete implementovat strom vlastními silami, není potřeba implementovat například mazání prvku ze stromu. Naopak ale musíte vymyslet jak spočítat výšku stromu.
- Pro testování použijte základní nevyváženou variantu binárního stromu.
- Ukázka výpočtu pro $n = 3$ je v tabulce 2. Výsledky jsou v tabulce 1.
- Program otestujte aspoň pro $n = 10$, $n! = 3628800$.

Výška stromu	Četnost
1	2
2	4
Celkem	$6 = n!$

Tabulka 1: Výsledky pro $n = 3$

Permutace	Binární strom	Výška
		
(123)		2
		
(132)		2
		
(213)		1

Permutace	Binární strom	Výška
		
(231)		1
		
(312)		2
		
(321)		2

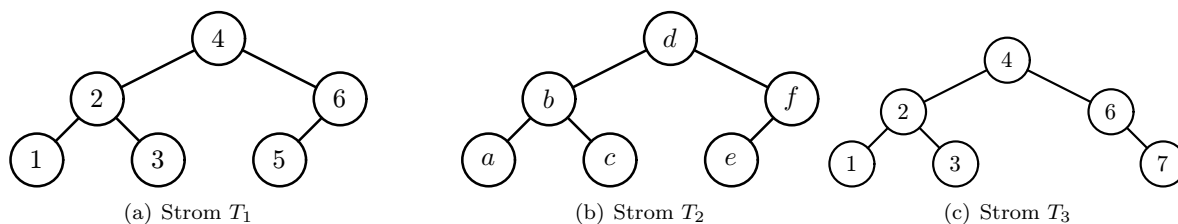
Tabulka 2: Ukázka výpočtu pro $n = 3$

7 Struktura stromů

Na vstupu jsou dány dva binární vyhledávací stromy A a B . Napište program, který rozhodne, zda dané dva stromy mají shodnou strukturu. *Struktura stromu je nezávislá na datech ve stromu uložených, jde pouze o existenci či neexistenci uzlů a hran mezi nimi.*

7.1 Ukázka

Uvažujme stromy na obrázku 6. Shodnou strukturu se stromem T_1 má strom T_2 , přestože obsahují různá data. Strom T_1 a strom T_3 nemají shodnou strukturu – z uzlu označeném číslem 6 vede v jednom případě hrana doleva v druhém případě vede doprava (lhostejno, že jeden uzel obsahuje číslo 5 a druhý 7). Pochopitelně, každý strom má shodnou strukturu sám se sebou.



Obrázek 6: Ukázky stromů

7.2 Implementace

- Předpokládejte, že oba stromy nebudou prázdné.
- Stromy budou obsahovat pouze celá čísla (typ `int`).
- Stromy budou zadány ve formě textového souboru jako posloupnost čísel, na každém řádku bude pouze jedno číslo.
- Stromy ze souborů získáte tak, že čísla načítaná postupně z textového souboru budete vkládat, pomocí obvyklého algoritmu, do binárního vyhledávacího stromu (jednoduchá, nevyvážená varianta).
- Jméno vstupního textového souboru zadáte z klávesnice.
- Jméno výstupního textového souboru zadáte z klávesnice.
- Výsledek vypíše Váš program na konzolu ve tvaru „Stromy mají/nemají shodnou strukturu“, žádná jiná informace se nepožaduje.
- V tomto příkladě máte opět několik možností řešení. Jednak můžete vhodným způsobem systematicky projít oba dva stromy, výsledky průchodů si vhodně zakódovat do řetězce a tyto řetězce pak porovnat. Druhou možností je systematicky procházet oba stromy najednou a porovnávat je „on-line“.
- Uvědomte si, že data uložená v uzlech Vás vlastně ani nemusí zajímat. Data načítaná ze vstupních souborů slouží jen k tomu, aby se dal binární strom nějakým (opakovatelným) způsobem poskládat.

8 Průnik dvou množin

Jsou dány dvě množiny $A = \{a_1, a_2, \dots, a_n\}$ a $B = \{b_1, b_2, \dots, b_m\}$, kde $a_i, b_i, n, m \in \mathbb{N}$. Napište program, který sestrojí průnik $C = A \cap B$.

Ukázka

Jestliže $A = \{2, 3, 5\}$ a $B = \{1, 3, 4, 5\}$ pak $C = A \cap B = \{3, 5\}$

Poznámky

- Vstupem do programu budou dva textové soubory, které budou obsahovat sekvence čísel, na každém řádku jedno číslo. Čísla budou v rozsahu $0 \leq x < 10^8$, čísla se mohou (a určitě budou) **opakovat**. Počet čísel bude v obou souborech shodný – 10^6 .
- Jak už to v životě bývá, data přicházející od uživatelů většinou nebývají ve formě, kterou bychom mohli ihned zpracovat. Množiny se kterými se bude pracovat je nutné nejprve z daných sekvencí vytvořit – odstranit duplicity.
- Výstupem z programu bude textový soubor, na každém řádku bude jedno číslo z Vaší výsledné množiny. Čísla budou seřazena vzestupně.
- Jména vstupních souborů a výstupního souboru se budou zadávat jako argumenty z příkazové řádky ve tvaru:

VasProgram Vstup1 Vstup2 Vystup

- Vzhledem k rozsahu množin není možné využít naivní algoritmus, spočívající v porovnání „každého prvku s každým“. Složitost tohoto algoritmu je $O(n^2)$, což pro $n = 10^6$ činí 10^{12} ! Výsledku byste se taky nemuseli dočkat.
- Klíčem k úspěchu je využití některého ze třídících algoritmů nebo datových struktur, možností řešení je relativně mnoho. Při některých způsobech řešení není dokonce nutné vytvářet množinu (odstraňování duplicit atd.) dopředu, je možné zpracovávat přímo sekvenci čísel ze vstupních souborů.