Databáze studentů

Termín odevzdání:	**22.04.2018 23:59:59**
Pozdní odevzdání s penalizací:	**01.07.2018 23:59:59** (Penále za pozdní odevzdání: 100.0000 %)
Hodnocení:	**5.0000**
Max. hodnocení:	**5.0000** (bez bonusů)
Odevzdaná řešení:	10 / 20 Volné pokusy + 20 Penalizované pokusy (-2 % penalizace za každé odevzdání)
Nápovědy:	2 / 2 Volné nápovědy + 2 Penalizované nápovědy (-10 % penalizace za každou nápovědu)

Úkolem je realizovat třídy, které implementují správu studentů na studijním oddělení.

Předpokládáme, že na studijním oddělení vedou agendu studentů. Pro jednoduchost je každý student reprezentovaný svým jménem (uvažujeme jeden řetězec pro příjmení a jméno či více jmen, jednotlivá slova jsou oddělená alespoň jednou mezerou), datem narození a rokem nástupu na fakultu. Chceme mít možnost studenty přidávat, odebírat a vyhledávat v databázi studentů. O studentech předpokládáme, že trojice údajů jméno + datum narození + rok nástupu je unikátní, v databázi se neopakuje. Samotná jména, data narození a roky nástupu se samozřejmě opakovat již mohou. Při vkládání a odebírání studenta zadáváme vždy již vyplněnou instanci studenta (obsahuje zmiňovanou trojici údajů).

Trochu komplikovanější je vyhledávání. Při vyhledávání chceme studenty filtrovat podle jména, data narození a roku nástupu na fakultu. Navrácený výsledek je seznam studentů, tento seznam chceme mít možnost seřadit podle zadaných kritérií (jméno, datum narození, rok nástupu do studia, pořadí registrace).

Poslední metodou rozhraní bude našeptávač - metoda, která pro zadanou část jména studenta vybere všechny studenty, kteří zadanému vzoru vyhovují.

Celá implementace je rozdělena do několika tříd. V přiloženém zdrojovém kódu naleznete základ řešení - implementaci některých tříd a požadované rozhraní zbývajících tříd. Jejich popis následuje:

Třída `CDate` reprezentuje jednoduché datum. Instance této třídy budou sloužit pro ukládání data narození studentů a pro filtrování podle data narození. Třída je implementovaná v testovacím prostředí a je vložená (v bloku podmíněného překladu) i v přiloženém zdrojovém souboru. Význam metod je následující:

konstruktor (y,m,d)
: inicializuje instanci zadaným rokem, měsícem a dnem.

CompareTo (date)
: metoda porovná `this` a instanci `date`. Výsledkem metody je jedna ze tří hodnot: -1 pokud je `this` menší než `date`, 0 pokud si jsou rovné a +1 pokud je `this` větší než `date`.

výstupní operátor <<
: zobrazí datum do zadaného proudu, funkce slouží zejména pro testování a ladicí výpisy.

poznámka
: třída je implementovaná v testovacím prostředí, implementace dodaná v přiloženém souboru slouží pouze pro Vaše ladění a při testování na Progtestu se neuplatní. Tedy nemá cenu zkoušet do rozhraní přidávat další metody, po odevzdání na Progtest nebudou k dispozici.

Výčtový typ `ESortKey` souží k identifikaci kritéria řazení. Výčtový typ je deklarován v testovacím prostředí, dodaná implementace je umístěná v bloku podmíněného překladu a slouží pouze pro lokální testování. Má následující přípustné hodnoty:

NAME
: řazení dle jména studenta,

BIRTH_DATE
: řazení dle data narození studenta,

ENROLL_YEAR
: řazení dle roku nástupu na fakultu.

poznámka
: jedná se o C++11 výčtový typ, kde kompilátor striktně trvá na tom, aby souhlasil typ proměnné a typ přiřazované hodnoty. S hodnotami je potřeba pracovat pomocí plně kvalifikovaného jména, např. `ESortKey::NAME`.

Třída `CStudent` reprezentuje jednoho studenta v databázi. Její implementace je na Vás. Povinné veřejné rozhraní je:

konstruktor (name, dateOfBirth, enrollYear)
: konstruktor inicializuje instanci pomocí zadaných parametrů.

operator ==
: porovná dvě instance `CStudent`. Za shodné je bude považovat, pokud se přesně shodují ve jménu, datu narození i roku nástupu na fakultu. Porovnání jména rozlišuje malá a velká písmena (case sensitive) a rozlišuje pořadí jmen (tedy např. "Jan Jakub Ryba" a "Jakub Jan Ryba" nejsou stejná jména).

operator !=
: porovná dvě instance `CStudent`. Za odlišné je bude považovat, pokud se přesně liší ve jménu, datu narození nebo roku nástupu na fakultu (v souladu s konvencí se tedy jedná o negaci výsledku operátoru ==).

další

Vaše implementace si do třídy může dodat další potřebné metody a členské proměnné, které budete potřebovat.

Třída `CFilter` reprezentuje kritéria pro filtrování studentů ve vyhledávání:

konstruktor
: konstruktor inicializuje prázdný filtr (takovému filtru vyhoví jakýkoliv záznam),

BornBefore
: přidá do filtru omezení, kterému vyhoví pouze studenti narození před zadaným datem (studenti narození v zadaný den již filtru nevyhoví),

BornAfter
: přidá do filtru omezení, kterému vyhoví pouze studenti narození po zadaném datu (studenti narození v zadaný den již filtru nevyhoví),

EnrolledBefore
: přidá do filtru omezení, kterému vyhoví pouze studenti zapsaní před zadaným rokem (studenti zapsaní v zadaném roce již filtru nevyhoví),

EnrolledAfter
: přidá do filtru omezení, kterému vyhoví pouze studenti zapsaní po zadaném roku (studenti zapsaní v zadaném roce již filtru nevyhoví),

Name
: přidá do filtru omezení na jméno (nebo více jmen) studentů. Chování se trochu liší od ostatních kritérií:

  - pokud metoda `Name` nebyla ve filtru použita, filtr neomezuje výsledek podle jmen studentů,
  - pokud je metoda `Name` zavolaná právě jednou, musí jména studentů odpovídat zadanému řetězci,
  - pokud je metoda `Name` zavolaná vícekrát, musí jména studentů odpovídat nějakému zadanému řetězci.

  Filtrování podle jména (jmen) má dále komplikovanější vlastní porovnávání: při filtrování porovnáváme jména bez rozlišování malých a velkých písmen, a dále při porovnávání považujeme za shodná jména, která se liší pořadím slov. Tedy například "Jan Jakub Ryba" a "Jakub Jan Ryba" a "RYBA jan JaKuB" v porovnání považujeme za shodná (ale např. "Jan Jakub Jan Ryba" se už liší).

další
: Vaše implementace si do třídy může dodat další potřebné metody a členské proměnné, které budete potřebovat.

Třída `CSort` reprezentuje kritéria pro řazení studentů ve výsledku vyhledávání. Řazení může využívat více kritérií pro porovnávání, každé kritérium může být vzestupné nebo sestupné. Předpokládejme, že máme řazení nastaveno na řazení podle jména (první kritérium) a podle data narození (druhé kritérium). Při porovnávání studentů se tedy nejprve srovnají jména, pokud se liší, je pořadí porovnávaných studentů dané. Pokud se jména neliší, použije se pro porovnávání datum narození. Opět, pokud se liší data narození, je pořadí dané, v opačném případě se použije případné další kritérium řazení. Pokud při řazení není k dispozici další kritérium řazení a pořadí stále není rozhodnuté, použije se pořadí registrace studenta v databázi (stabilní řazení). Všimněte si, že pokud nenastavíme žádné kritérium řazení, má být výsledek vyhledávání v pořadí registrace studentů.

konstruktor
: konstruktor inicializuje prázdný seznam kritérií řazení,

AddKey (key, ascending)
: metoda přidá další kritérium řazení. Kritérium je dané hodnotou výčtového typu `ESortKey` (první parametr), druhým parametrem je hodnota `true` pro vzestupné řazení podle zadaného kritéria nebo `false` pro sestupné řazení podle zadaného kritéria.

další
: Vaše implementace si do třídy může dodat další potřebné metody a členské proměnné, které budete potřebovat.

Třída `CStudyDept` reprezentuje databázi studentů. Databáze poskytuje následující rozhraní:

konstruktor
: konstruktor inicializuje prázdnou databázi,

AddStudent (student)
: metoda přidá studenta do databáze, vrací signalizaci úspěchu (`true`) nebo neúspěchu (`false` - duplicitní student),

DelStudent (student)
: metoda odebere studenta z databáze, vrací signalizaci úspěchu (`true`) nebo neúspěchu (`false` - student v databázi neexistoval),

Search (filter, sortOpt)
: metoda vyhledá v databázi studenty, kteří vyhovují kritériím filtru `filter` a vrátí jejich seznam seřazený v pořadí daném podmínkami v `sortOpt`,

Suggest (name)
: metoda slouží jako našeptávač pro uživatelské rozhraní databáze. Metoda vezme řetězec `name`, rozdělí jej na jednotlivá slova a najde registrované studenty takové, že jejich jméno obsahuje všechna takto získaná slova z `name`. Porovnávání slov nerozlišuje malá a velká písmena (case insensitive). Například pokud jsou v databázi registrovaní studenti "Jan Jakub Ryba", "Jan Ryba" a "Jan Novak", pak volání `Suggest("jan")` zahrne všechna tři tato jména, volání `Suggest("Ryba JaN")` vrátí pouze první dva z nich, volání `Suggest("Ryba Jakub")` vrátí pouze prvního z nich, `Suggest("Ryba Ja")` vrátí prázdnou množinu a `Suggest("Jan jan JAN")` vrátí všechna tři jména.

další

Vaše implementace si do třídy může dodat další potřebné metody a členské proměnné, které budete potřebovat.

Odevzdávejte zdrojový kód s implementací tříd `CStudent`, `CFilter`, `CSort` a `CStudyDept`. Za základ implementace použijte přiložený soubor s deklarací metod a se sadou základních testů. Pro implementaci se může hodit doplnit i další pomocné třídy.

Zadání se trochu podobá dřívější domácí úloze. Předpokládáme, že při implementaci použijete vhodné kontejnery z STL (je k dispozici téměř celá), dále předpokládáme, že Vaše implementace bude časově a paměťově efektivní. Veškeré vkládání a mazání by mělo být rychlejší než lineární, řazení pak `n log n`. Není rozumné na všechny vnitřní struktury používat kolekci `vector`. Pokud chcete využívat C++11 kontejnery `unordered_set / unordered_map`, pak hashovací funktor neodvozujte jako specializaci `std::hash`. Hashovací funkci/funktor deklarujte explicitně při vytváření instance `unordered_set / unordered_map`. (Specializace `std::hash` předpokládá opětovné otevření jmenného prostoru `std`. To se těžko realizuje, pokud jste uzavřeni do jiného jmenného prostoru. Návody dostupné na internetu (stack overflow, cpp reference) implicitně předpokládají, že jmenné prostory nepoužíváte, na nich doporučovaná řešení nejsou ideálně kompatibilní.)

V testech se kontroluje rychlost operace hledání. Řazení výsledků by mělo být časově optimální (n log n), ale rychlost významně ovlivní i vlastní filtrování výsledků. Mohlo by se zdát, že lineární filtrování výsledků bude dostatečně rychlé (vždyť řazení je pomalejší - n log n), ale zde každý z dílčích algoritmů pracuje s jiným objemem dat. Filtrování musí zpracovat celou databázi s řádově desítkami až stovkami tisíc údajů. Výsledkem filtrování je seznam, který bude často velmi krátký a který je třeba řadit. Tedy lineární algoritmus filtrování celkový čas často ovlivní více než algoritmus řazení (filtrování bude v nejhorším případě lineární algoritmus, ale často může fungovat výrazně lépe - pokud jsou vhodným způsobem zapojené podmínky filtru).

V této úloze není explicitní test kontrolující kopírování instancí vytvořených tříd, samotné testovací prostředí dokonce nikde instance Vašch tříd nekopíruje. Kopírování ale může být potřeba již uvnitř Vaší implementace. Obecně by to neměl být problém, pokud třídy vzniknou skládáním C++ řetězců, STL kontejnerů, Vašich kopírovatelných/přesouvatelných tříd a celých čísel, budou automaticky generované kopírující a přesouvací konstruktory/operátory = fungovat správně.