Systeemprogrammeren

Academiejaar 2018–2019

sysprog@lists.ugent.be

Oefeningenset 5
geavanceerde C++

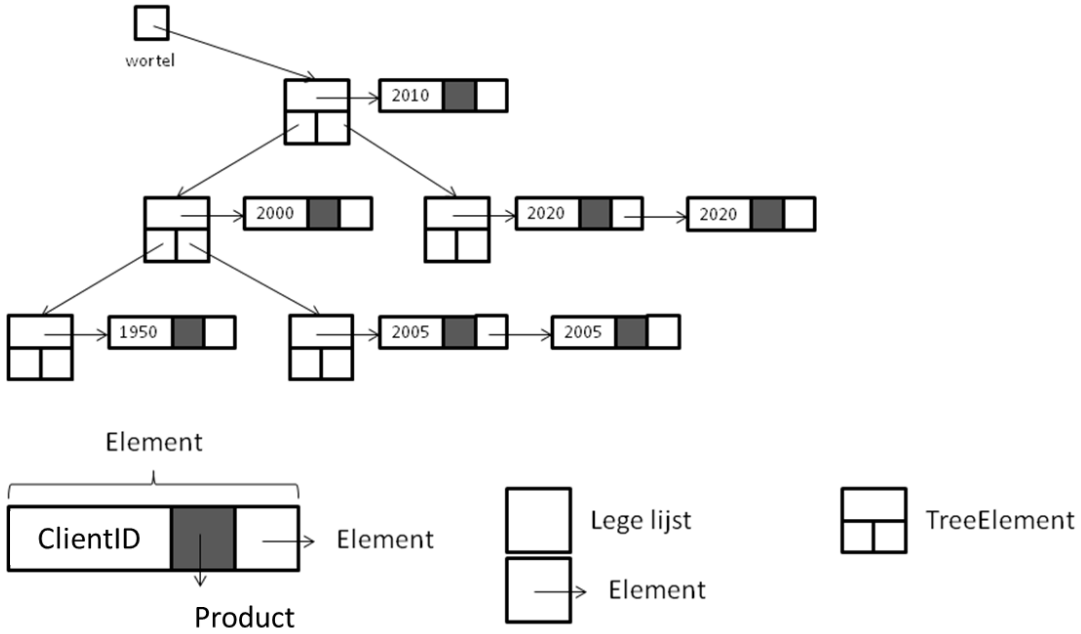
Bijhouden van klantengegevens

Situering

In deze opgave zullen verschillende geavanceerde principes van C++ aan bod komen: overerving en polymorfisme in klassen, definitie en gebruik van templates en ten slotte het gebruik van de Standard Template Library (STL).

De bedoeling is om met behulp van bovenstaande principes een datastructuur te implementeren die gebruikt zal worden om klanten van een winkel bij te houden.

Om de klanten en hun aankopen op te slaan, zullen we gebruik maken van een binaire zoekboom. Elke knoop in de boom stelt een klant voor en bevat 1 of meerdere aangekochte producten van die klant. Als een klant meerdere producten gekocht heeft, en dus meerdere producten in één knoop van de boom opgeslagen moeten worden, dan worden deze bijgehouden in een geschakelde lijst. Figuur 1 illustreert een voorbeeld van de binaire zoekboom met de verschillende elementen. De figuur toont een binaire zoekboom met 5 knopen (klanten) die elk 1 of meerdere producten aangekocht hebben. Een `Element` bevat een klantennummer (`ClientID`), een verwijzing naar het product (`Product`) en een verwijzing naar het volgende `Element`. Een knoop van de boom (`TreeElement`) bevat een verwijzing naar een `Element` en naar 2 andere knopen (`TreeElement`), het linker- en rechterkind.



Figuur 1: Visuele voorstelling van de binaire zoekboom

Opgave

Met deze opgave worden 4 bestanden geleverd:

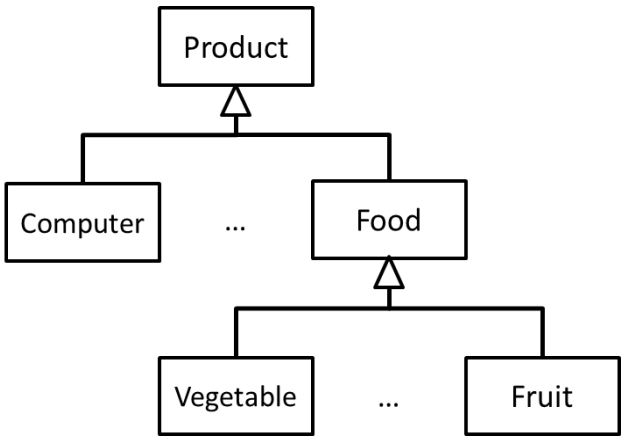
- `main.cpp`: dit bestand bevat de verschillende testmethodes die jouw geschreven code zullen aanroepen.
- `BinarySearchTree.h`: dit header-bestand bevat reeds de initiële structuur voor de binaire zoekboom.
- `BinarySearchTree.cpp`: dit implementatiebestand zal je zelf verder moeten aanvullen, maar er staan reeds wat declaraties van methodes klaar.

Het eerste deel van deze opgave gaat over het aanmaken van de vereiste datatypes om de producten van de winkel te beschrijven. Daarna zullen we in het tweede deel de binaire zoekboom uitwerken om in het derde deel de veralgemening van de zoekboom door te voeren aan de hand van templates, zodat niet alleen producten kunnen opgenomen worden, maar dat de ontwikkelde datastructuur gebruikt kan worden om generieke elementen bij te houden.

Overerving en polymorfisme

In de volgende paragraaf kan je een beschrijving vinden, die aangeeft welke functionaliteit moet voorzien worden in de klassen om de producten voor te stellen. Het doel van deze oefening is dat je aan de hand van de beschrijving de overervingstructuur van de klassen bepaalt, alsook welke attributen en methodes in welke klassen voorzien moeten worden. Jullie moeten m.a.w. zelf de declaraties van de klassen opmaken.

De producten (`Product`) die moeten opgeslagen worden, hebben een naam (`string name`) en een prijs (`long price`). Beide waarden moeten opgevraagd en aangepast kunnen worden. Verder is het mogelijk om voor elk product de categorie (`string getCategory()`) op te vragen. Er zijn 2 subklassen die zeker geïmplementeerd moeten worden: `Computer` en `Food`. De `getCategory()` methode zal bij `Computer` aangeven of de computer “high-end” (`Price>1000`),”mid”(`1000>=Price>500`) of “low-end”(`500>=Price>0`) is. Bij `Food` zal deze aangeven welke specifieke subklasse het is (bvb. `Fruit` of `Vegetable`). `Food` houdt ook nog een extra variabele bij, `Timestamp bestBefore`. Dit is een datum die aangeeft wanneer het product slecht wordt. `Timestamp` bevat zelf 3 `integers` die respectievelijk de dag, maand en jaar aangeven. Getters en setters moeten voor al deze variabelen ook geïmplementeerd worden. Je mag er vanuit gaan dat de waardes van dag, maand en jaar voor een `Timestamp` correct zijn. Ten slotte moet het gebruik van de “<<” operator bij een product een string uitprinten als volgt: “[<NAME> <PRICE>]” (bvb. [Banana 1])



Figuur 2: Voorbeeld overervingsstructuur

Je opdracht is nu om de nodige klassen uit te werken in C++, door gebruik te maken van polymorfisme en overerving. Zorg ervoor dat er geen dubbele code geprogrammeerd wordt. Houd voldoende rekening met eventuele virtuele methodes en abstracte basisklassen. Denk goed na welke variabelen en operaties je in de basisklasse zal steken, en welke in de kindklassen. Declareer de nodige C++ klassen en implementeer elke klasse in aparte .cpp en .h bestanden.

Binaire zoekbomen

Een binaire zoekboom wordt gebruikt om elementen gesorteerd op te slaan volgens het volgende principe:

- de waarde van de sleutel van iedere knoop is groter dan de waarde van alle sleutels in de linker subboom,
- de waarde van de sleutel van iedere knoop is kleiner (of gelijk indien de boom gelijkheid ondersteunt) dan de waarde van alle sleutels in de rechter subboom.

Zoeken in een binaire zoekboom – search

Vermits de elementen gesorteerd zijn opgeslagen, kan volgend algoritme toegepast worden om een element met een bepaalde waarde op te zoeken in de binaire zoekboom:

> vergelijk de waarde van het element met de waarde opgeslagen in de ouderknoop, indien deze kleiner is: ga naar de linker sub-boom en herhaal hetzelfde principe daar, indien de waarden gelijk zijn: element gevonden (wordt search hit genoemd), indien de waarde groter is: ga naar de rechter sub-boom en herhaal hetzelfde principe daar. Wanneer een gekozen sub-boom leeg is, stopt het algoritme: element niet gevonden (wordt search miss genoemd).

Er wordt dikwijls een recursieve implementatie van dit algoritme gekozen. Zorg dat je zeker ook weet hoe dit op een niet-recursieve manier kan aangepakt worden.

Toevoegen in de wortel - root insertion

Zoals in de syllabus beschreven, gebeuren alle toevoegingen in een binaire boom aan de toppen van de boom (i.e. onderaan). Voor sommige toepassingen is het aangewezen dat het nieuwe element zich in de wortel bevindt, vermits dit de toegangstijd verkort voor de meest recente knopen. Het is belangrijk om in te zien dat een nieuw element niet zomaar bovenaan kan toegevoegd worden. Daarom wordt de toevoeging onderaan gedaan, en aan de hand van rotaties wordt het nieuwe element gepromoveerd naar de wortel van de boom. Rotaties komen aan bod in de volgende sectie.

Rotatie in binaire zoekbomen - rotation

Een rotatie is een lokale transformatie op een deel van een binaire zoekboom die toelaat om de rol van de wortel en een van zijn kinderen om te wisselen, en dit terwijl de voorwaarden tussen de knopen van een binaire zoekboom behouden blijven. Rotatie is een basisoperatie die gebruikt wordt om de knopen in een binaire zoekboom te herschikken. Zoals hierboven vermeld wordt bij toevoegen van een element in de wortel (Eng.: root insertion) het element onderaan in de binaire zoekboom toegevoegd en dan de boom herschikt tot de nieuwe knoop in de wortel staat. Rotaties worden zeer dikwijls recursief toegepast.

Element verwijderen uit een binaire zoekboom - remove

Een element kan niet zomaar uit een binaire zoekboom verwijderd worden (omdat de voorwaarden voor een binaire zoekboom dan kunnen geschonden worden). De beste manier is de volgende:

- promoveer in de rechter sub-boom het kleinste element (i.e. meest linkse) naar de root van deze sub-boom,
- dit element krijgt dezelfde linker sub-boom als het verwijderde element,
- dit element komt in de volledige boom in de plaats van het verwijderde element.

De binaire zoekboom die hier gebruikt wordt organiseert `longs` (`ClientID`) met de producten van die klant. Deze binaire zoekboom bestaat uit een verzameling van boomelementen (`TreeElement`) die als waarde gelinkte lijsten van `Element` objecten hebben en bovendien twee `pointers`, een naar elke deelboom van de binaire zoekboom. De `Element` objecten bestaan op hun beurt uit de identifier van de klant, uit een `pointer` naar het volgende element in de lijst en een `pointer` naar het product (zie Figuur 1).

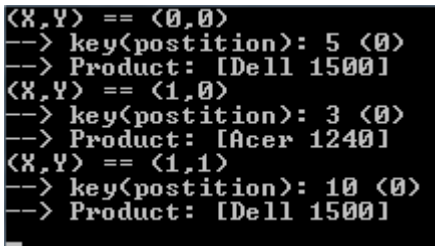
Deze binaire zoekboom gebruikt volgend principe; alle elementen waarvan de sleutel dezelfde sleutelwaarde heeft worden in dezelfde gelinkte lijst opgeslagen. De positie van die gelinkte lijst wordt uiteraard bepaald door de correcte positie in de zoekboom te vinden.

Gegeven de header `BinarySearchTree.h` en de bestanden `BinarySearchTree.cpp` en `main.cpp`.

Gevraagd: vul `BinarySearchTree.cpp` verder aan:

- `constructor:` alloceer de nodige ruimte voor de binaire zoekboom; de constructor zonder argument neemt een lege zoekboom als grootte voor de binaire zoekboom (alleen de constructor zonder argumenten wordt gevraagd, geen copy-constructor!).
- `destructor`: geeft alle geheugen vrij ingenomen door de binaire zoekboom; de product-objecten moeten hierbij NIET vrijgegeven worden.

- `void put(long clientID, Product* e)` voegt een product toe aan de binaire zoekboom volgens het volgende algoritme: aan de hand van het opgegeven klantennummer (`ClientID`) wordt de correcte top in de boom voor het toe te voegen product bepaald indien het klantennummer nog niet voorkomt in de binaire zoekboom (geval 1). Anders wordt de knoop gezocht die overeenkomt met het opgegeven klantennummer (geval 2).
    - Geval 1: er wordt een nieuwe gelinkte lijst gemaakt en het nieuwe product wordt daaraan toegevoegd (voor de structuur, zie Figuur 1). Pas bij het toevoegen de principes van rotatie toe.
    - Geval 2: de aanwezige gelinkte lijst wordt achteraan uitgebreid met het nieuwe product.
- `Product* get(long clientID)`: geeft een pointer terug naar het eerste voorkomen van een product waarvan de sleutel overeenstemt met de opgegeven sleutel. Anders wordt NULL teruggegeven.
- `Product* remove(long clientID)`: verwijdert een product uit de binaire zoekboom waarvan de sleutel overeenstemt met de opgegeven sleutel. Indien er meerdere producten zijn met die sleutel wordt het eerste dergelijke product verwijderd. Uiteraard moeten andere producten waarvan de sleutel dezelfde waarde heeft na verwijdering nog steeds vindbaar zijn. Bovendien moet de knoop uit de binaire zoekboom verwijderd worden wanneer het laatste product dat bij deze knoop hoort, verwijderd werd. Deze methode geeft ook de pointer terug naar het verwijderde product.
- `printTree()`: print alle (sleutel, waarde) paren af in de binaire zoekboom, voor de waarde moeten ook effectief de kenmerken van het product (zie Sectie 3) in de lijst uitgeprint worden en niet de pointer. Figuur 3 geeft een voorbeeld van een uitgeprinte binaire zoekboom met 3 knopen. Hiervoor kan je een recursieve benadering gebruiken die eerst de knoop zelf afprint en dan de knopen in de linkerboom en dan de rechterboom.



Templates

In de voorgaande opdracht heb je een binaire zoekboom geïmplementeerd die producten bijhoudt. Het vergt niet veel verbeelding om te begrijpen dat deze structuur ook perfect zou kunnen gebruikt worden voor het opslaan van arbitraire data, zolang deze maar bestaat uit de combinatie van een `long` met een waarde van een willekeurig type. Herwerk daarom de implementatie van de `Element`, `TreeElement` en `BinarySearchTree` klasse door gebruik te maken van templates om als dusdanig willekeurige waarden te kunnen opslaan in de binaire zoekboom. Meer informatie omtrent templates kan je vinden in de syllabus. Implementeer de template variant van de binaire zoekboom in afzonderlijke bestanden, nl. BinarySearchTreeTemplate.h en BinarySearchTreeTemplate.cpp.

Normaalgezien worden template klassen geïmplementeerd in een header file, omdat voor de compiler zowel de definitie als implementatie zichtbaar moeten zijn op het moment van instantiëring van een template klasse (hier dus in de `BinarySearchTree`, `TreeElement` en `Element` klasse) en omdat template klassen doorgaans klein in omvang zijn.

Hier is dit niet het geval en wordt de implementatie opgesplitst in header en cpp bestanden. Daarom moet je aan het einde van jouw cpp bestand het volgende toevoegen:

```
template class BinarySearchTreeTemplate<Product>;
```

STL

STL (Eng.: Standard Template Library) werd in 1994 ontwikkeld als de standaard bibliotheek voor C++: het bevat datatypes, algoritmen en hulpklassen. De belangrijkste onderdelen zijn:

- *containers*: dit zijn klassen, die dienen voor dataopslag en gebaseerd zijn op een bepaalde datastructuur. Er zijn zestien container-types mogelijk in STL, waaronder `vector`, `deque`, `list`, `set`, `multiset`, `map` en `multimap`.
- *iteratoren*: dit zijn klassen die dienen om efficiënt door alle aanwezige elementen in de containers te itereren.
- *algoritmen*: dit zijn methodes die een bepaald algoritme implementeren, bijvoorbeeld een sorteer-, zoek- of filteralgoritme.

STL is volledig gebaseerd op templates: de containers zijn allen klasse templates. De achterliggende code is ontwikkeld om zo snel mogelijke uitvoering toe te laten.

In de voorgaande stappen van deze opgave hebben jullie een eigen implementatie gemaakt van een gelinkte lijst. Hierin werden de verschillende producten van een bepaalde klant, d.m.v. `Element` of `ElementTemplate` objecten aan elkaar geschakeld.

We kunnen echter ook gebruik maken van een bestaande container implementatie uit de STL bibliotheek, nl. `list`. In dit laatste onderdeel wordt jullie gevraagd om in de template variant van de binaire zoekboom de eigen implementatie van de geschakelde lijst om te vormen naar een versie die gebruikt maakt van de `list` STL container om de verschillende parameterwaarden op te slaan. Bekijk ook alle methoden van de binaire zoekboom die over de elementen van de gelinkte lijsten itereren, en pas die aan zodat die nu

gebruik maken van STL iteratoren om doorheen te elementen te lopen. Pas ook zeker de implementatie van de methode `printTree()` aan, zodat die nu gebruik maakt van STL iteratoren. Onthoud dat een print-methode de uit te printen elementen niet mag/zal aanpassen. Implementeer je aanpassingen in een derde versie van de binaire zoekboom, met als naam `BinarySearchTreeSTL`.

Vergeet ook hier niet onderaan jouw cpp bestand het volgende toe te voegen:

```
template class BinarySearchTreeSTL<Product>;
```