Implementatie en toepassing van de trie

Tweede programmeeropdracht Datastructuren, najaar 2007

Een `Trie` is een `char`-aire boom, dwz. een boom waarvan knopen voor elk van de elementen van het type `char` ('character') een kind kunnen hebben (dat betekent maximaal 256 kinderen bij één knoop). Knopen kunnen informatie dragen van een type `InfoTp`. Dit type hangt af van de toepassing.

Een voorbeeld van een trie is de linker boom in Figuur 1. Een label `a` op een tak betekent dat het om een tak naar het kind `a` gaat. De waarde die bij het eindpunt van een tak staat is de waarde van het informatieveld. In het voorbeeld is het informatieveld een geheel getal.

Een toepassing van de trie die we hier gaan uitwerken is ZLW datacompressie.

Specificatie van de trie

De `Trie` is voorzien van een intern venster ('window') dat we door de boom kunnen bewegen. De meeste operaties vinden plaats ten opzichte van de knoop waar het venster zich bevindt: de *huidige* knoop.

Uitgaande van een `class Trie` beschouwen we de volgende basis-operaties:

`Trie ();`
: Constructor: creëer een nieuwe Trie, bestaande uit alleen een wortel.

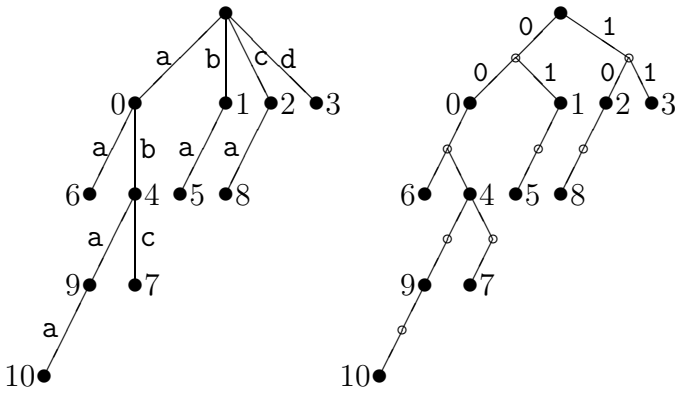
`~Trie ();`
: Destructor.

`bool AtLeaf ();`
: Test of het venster op een blad staat.

`void GoRoot ();`
: Plaats het venster op de wortel.

`bool ExistsChild (char K);`
: Ga na of de huidige knoop het opgegeven kind K heeft.

`void GoChild (char K);`
: Verplaats het venster naar het aangegeven kind K van de huidige knoop.



Figuur 1: Trie (4-air), rechts als binaire boom (2 'bits').

`char FirstChild ();`
Bepaal het eerste kind van de huidige knoop; het nul-karakter '\0' als de knoop helemaal geen kinderen heeft.

`char NextChild (char K);`
Bepaal het kind dat volgt op het aangegeven kind K, gezien vanuit de huidige knoop; het nul-karakter als geen kind meer volgt.

`void AddChild (char K);`
Voeg het aangegeven kind K toe aan de huidige knoop; als de knoop al een kind K heeft, doe dan niets.

`void RemoveChild (char K);`
Verwijder het aangegeven kind van de huidige knoop; als dit kind geen blad van de trie blijkt te zijn, moet er niets verwijderd worden.

`void PutInfo (InfoTp I);`
Plaats de opgegeven informatie in de huidige knoop.

`InfoTp GetInfo ();`
Retourneer de informatie uit de huidige knoop.

`void GotoNode (TrieNode *Nd);`
Plaats het venster op de aangegeven knoop.

`TrieNode *GetNode ();`
Geef (het adres van) de huidige knoop.

Bij de functies `GoChild`, `NextChild`, en `RemoveChild` mag aangenomen worden dat het opgegeven kind daadwerkelijk bestaat. Bedenk verder dat een knoop in principe ook een kind '\0' kan hebben. Wanneer `FirstChild` of `NextChild` het nul-karakter oplevert, kan dat dus twee betekenissen hebben.

Opdracht 2A: implementatie van de trie

Wanneer de trie op de voor de hand liggende wijze geïmplementeerd wordt, is voor elke knoop een array van 256 pointers in gebruik, één voor elk mogelijk kind. Wanneer knopen slechts enkele kinderen hebben, wordt relatief veel geheugenruimte in beslag genomen voor weinig informatie.

Er bestaat een zuinigere implementatie, waarbij de 256-aire boom gereduceerd wordt tot een binaire boom. Het kind K van een trie-knoop kan gevonden worden door een pad van acht links/rechts takken te volgen, overeenkomend met de (acht) bits van K, te beginnen met het eerste (het meest significante) bit. Dit idee wordt uitgebeeld door de rechter boom in Figuur 1, alleen dan voor 2-bits karakters in plaats van 8-bits karakters.

Implementeer de trie zoals gesuggereerd, op de zuinige manier, in bestanden `trie.cc` en `trie.h`. Hoewel we bij deze opdracht alleen maar met informatievelden van type `int` werken, dient er toch rekening te worden gehouden met mogelijke andere types. Werk dus met `templates`.

Om de datastructuur te kunnen gebruiken (en testen) moet de syntax van de klasse `Trie` overeenkomen met het volgende fragment:

```
template <class InfoTp>
class  Trie
{ public:
    Trie ();
    ~Trie ();
    bool   AtLeaf ();
    void   GoRoot ();
    bool   ExistsChild (char K);
    void   GoChild (char K);
    char   FirstChild ();
    char   NextChild (char K);
    void   AddChild (char K);
    void   RemoveChild (char K);
    void   PutInfo (InfoTp I);
    InfoTp GetInfo ();
    void   GotoNode (TrieNode<InfoTp> *Nd);
    TrieNode<InfoTp> *GetNode ();
  private:
    TrieNode<InfoTp> *Window,  // het venster
                     *Root;    // de wortel
  // uw eigen methodes en velden, public of private
};
```

Een grove schets van de functie `GoChild` ziet er bijvoorbeeld als volgt uit:

```
template <class InfoTp>
void Trie<InfoTp>::GoChild (char K)
{
  for alle bits in K (te beginnen bij meest significante)
  do
    if  bit == 0
    then  ga naar links
    else  ga naar rechts
    fi
  od
}
```

N.B.: Via de webpagina van Datastructuren wordt een aantal hulpbestanden met toelichting beschikbaar gesteld, die het maken van de opdracht kunnen vergemakkelijken. Lees met name ook de `READ.ME` die daarbij zit.

Inleverdatum: uiterlijk vrijdag 26 oktober 2007. Comprimeer de bestanden `trie.cc`, `trie.h` en eventuele andere bestanden (bijvoorbeeld een `read.me` met daarin commentaar) tot een `tar.gz` bestand. Lever dit bestand in bij de corrector Sven van Haastregt via emailadres `svhaastr@liacs.nl`. Maak in het Subject duidelijk dat het om opdracht 2A gaat en vermeld de auteur(s) met de studentnummer(s). Stuur geen object bestanden of executables mee!!!

ZLW compressie

Het ZLW compressie algoritme codeert teksten door daarin strings van variabele lengte om te zetten naar een korte bit-code. Het algoritme werd in 1977 gepresenteerd door de wiskundigen J. Ziv en A. Lempel, en werd in 1984 aangevuld door hun collega T.A. Welch.[1] Het algoritme is zelf-lerend: de strings waarvoor bit-codes worden gebruikt, worden aan de hand van de tekst bepaald, op het moment dat de compressie plaatsvindt.

Gewoonlijk worden iets van 12 bits gebruikt voor de bit-codes. Er kunnen dan dus 4096 strings als 'enkele symbolen' gecodeerd worden. De code-tabel heeft een boomvorm, waarbij langs de takken invoer-letters staan, en bij de knopen de uiteindelijke bit-code, dus een getal. Zo'n boom is (inderdaad) een trie, met als informatieveld een integer.

De code-tabel (de trie dus) is dynamisch, en groeit tijdens het coderen. Bij de start van het coderen krijgen alle mogelijke enkele karakters een code. Dat betekent dat strings van lengte 1 een bit-code hebben. Het algoritme probeert steeds zo lang mogelijke woorden uit de tekst in de trie te volgen. Heeft het algoritme x gelezen, met a als volgende symbool, dan zoekt het naar xa in de trie. Als dit niet in de trie staat, wordt de code voor x naar de uitvoer geschreven, en wordt xa met een nieuwe code aan de trie toegevoegd. Het algoritme springt naar de wortel van de trie, en volgt daar a. De nieuw gecreëerde mogelijkheid voor xa wordt dus niet onmiddellijk gebruikt, maar pas bij de volgende keer dat dat woord gelezen wordt.

Als alle codes benut zijn, groeit de boom niet verder. De rest van de tekst wordt gecodeerd met de op dat moment beschikbare codes. Schematisch ziet het algoritme er dus als volgt uit:

```
initialiseer Trie voor ZLW codering
while not eof (SourceFile)
do
  input Letter
  if not (kind Letter bestaat in Trie)
  then
    output code uit Trie
    if not (maximaal aantal strings in Trie)
    then
      voeg nieuwe code via tak Letter aan Trie toe
    ga naar wortel Trie
  fi
  ga naar kind Letter in Trie
od
output code uit Trie
```

Voorbeeld. We gaan uit van een invoer-alfabet van vier letters, `a b c d`; dit alfabet is dus twee bits. We coderen naar vier bits. Daarmee kunnen maximaal zestien strings gecodeerd worden, waaronder de oorspronkelijke letters.

De invoer `abaabcababaa` geeft als uitvoer de codes `0 1 0 4 2 4 9 0`, volgens de onderverdeling `a.b.a.ab.c.ab.aba.a`. De woorden die achtereenvolgens aan de trie worden

[1] In de praktijk wordt vaak de afkorting 'LZW' gebruikt. Uit historisch oogpunt lijkt 'ZLW' echter correcter.

toegevoegd zijn, met hun codes, 4: `ab`, 5: `ba`, 6: `aa`, 7: `abc`, 8: `ca`, 9: `aba`, 10: `abaa`. De resulterende trie hebben we al gezien, namelijk in Figuur 1.

Merk op dat de uitvoer voor dit kleine voorbeeld *langer* is dan de invoer ($8 \cdot 4 = 32$ tegen $12 \cdot 2 = 24$ bits). Het gaat om het idee.

ZLW decompressie

Om de gecodeerde tekst weer om te zetten naar de oorspronkelijke, moet het decoderingsalgoritme (impliciet) dezelfde code-trie opbouwen als het coderingsalgoritme.

Als de decoder achtereenvolgens codes i en j binnenkrijgt, zoekt hij de bijbehorende strings, zeg x en y, op in zijn decodeer-tabel. Behalve dat deze strings naar de uitvoer geschreven worden, moet een nieuwe code voor xa aan de codeboom toegevoegd worden, waarbij a de eerste letter van y is. Dit is immers de actie die ook het coderingsalgoritme op dat moment ondernam.

Er is één klein probleem: het kan zijn dat j nog niet in de codeboom staat nadat i gelezen is. Dit gebeurt als j de code is van het net bij de codering nieuw toegevoegde woord xa. Het decoderingsalgoritme moet dan een stap vooruit werken om de nog onbekende letter a te bepalen. Omdat y (de string bij j) gelijk is aan xa (de nieuwe toevoeging), moeten de eerste letters van x en y gelijk zijn, dus a is óók de eerste letter van x. Daarmee is het probleem opgelost: de decoder voert x uit en voegt xa aan de trie toe (met code j).

Voorbeeld. Bij de code `0 1 0 4 2 4 9 0` is de werking van de decoder rechttoerechtaan tot het decoderen van de `9`. Achtereenvolgens wordt vertaald: `a.b.a.ab.c.ab`, en worden aan de trie toegevoegd, 4: `ab`, 5: `ba`, 6: `aa`, 7: `abc`, en 8: `ca`. Nu moet `9` gedecodeerd worden, terwijl de code 9 niet in de trie te vinden is. De betekenis van `9` moet wel de eerstvolgende toe te voegen string zijn; deze string begint met het laatste gedecodeerde woord, `ab`, en eindigt met zijn eigen eerste letter. De decoder schrijft dus `aba`, en voegt 9: `aba` aan de trie toe.

Om te kunnen decoderen moeten we bij een getal een string vinden. Het is voor de hand liggend om deze gegevens in een array bij te houden. De strings in de tabel kunnen echter relatief lang worden. Om de tabel compact te houden is het mogelijk om alleen de laatste letter bij elke code te zetten met een verwijzing naar de rest (het voorgaande stuk) van de code. De tabel geeft zo de takken van de trie weer. In Tabel 1 is dit idee uitgewerkt voor onze voorbeeldtekst en -codering.

Opdracht 2B: implementatie van ZLW (de-)compressie.

B1: Schrijf programma's voor ZLW compressie en decompressie, voor 8 bits invoer (karakters) en 12 bits codes. Deze 12 bits codes moeten in de vorm van karakters naar de uitvoer geschreven worden, dus twee codes opgeslagen in drie karakters.

Compressie vraagt een file-naam (zeg `xxx`), en zet `xxx` om in `xxx.cod`. Decompressie zet, wanneer `xxx` opgegeven wordt, `xxx.cod` om in `xxx.dec`: dit file moet natuurlijk identiek zijn aan `xxx`.

B2: Vergelijk je implementatie met die van een ander team voor (in ieder geval) de bestanden `testzlw.in` en `testzlw3.in` (beschikbaar via WWW) en een 'groot' tekstbestand. Elk van deze bestanden moet je (1) met de eigen implementatie comprimeren,

code	string	verkort	
0	a	a	-
1	b	b	-
2	c	c	-
3	d	d	-
4	ab	b	0
5	ba	a	1
6	aa	a	0
7	abc	c	4
8	ca	a	2
9	aba	a	4
10	abaa	a	9

Tabel 1: Decodeertabel (volledig en verkort).

waarna je het resultaat weer met de andere implementatie de-comprimeert, en (2) met de andere implementatie comprimeren, waarna je het resultaat weer met de eigen implementatie de-comprimeert. Vermeld het andere team bij je ingeleverde werk.

B3: Zoek (op internet of in een boek met fileformats) de beschrijving van `.gif`-files. De basis van de gif-compressie is het algoritme van ZLW. Hoe wijkt gif-compressie af van de hierboven beschreven methode? Vermeld duidelijk de bron waar je je informatie vandaan haalt.

Inleverdatum: uiterlijk vrijdag 23 november 2007. Alle bestanden die nodig zijn om het (de-)compressie programma te kunnen draaien moeten bij Sven van Haastregt ingeleverd worden, samen met een verslag van de vergelijking met het andere team (B2) en de literatuurstudie (B3).

Comprimeer alle benodigde bestanden tot een `tar.gz` bestand. Lever dit bestand in bij de corrector Sven van Haastregt via emailadres `svhaastr@liacs.nl`. Maak in het Subject duidelijk dat het om opdracht 2B gaat en vermeld de auteur(s) met de studentnummer(s). Stuur geen object bestanden, executables, of testinstanties mee!!!

Ten slotte: In principe hoef je je programma's niet op de Linux-pc's in de computerzalen te maken. De uiteindelijke versie moet echter gecompileerd kunnen worden (en werken) met g++ op zo'n Linux-pc!

Werk bij voorkeur in tweetallen. Werk gestructureerd. Voorzie je programma van voldoende commentaar. Indien je bij je implementatie bepaalde keuzes maakt, dien je die in het commentaar toe te lichten. Geef tevens een betrouwbare schatting van het aantal gewerkte uren.

Heb je nu nog vragen over de bedoeling van de opdracht, wend je dan tot Robert Brijder (kamer 156a, (intern) telefoonnummer 7143, rbrijder@liacs.nl). Heb je vragen over je eigen programma, wend je dan tot Sven van Haastregt.