Implementatie van een Priority Queue

Datastructuren najaar 2005, tweede programmeeropdracht

Fibonacci Queue

De *Fibonacci Queue* is een efficiënte implementatie van een *priority queue* met de gebruikelijke operaties: *IsEmpty* – test of de queue leeg is, *Minimum* – bepaal het kleinste element, *DeleteMin* – verwijder het kleinste element uit de queue, en *Insert* – voeg een element aan de queue toe, maar daarboven nog de extra operaties *Merge* – verenig de queue met een andere gegeven queue, en *Decrease* – verlaag de waarde van een aangewezen element tot de aangegeven waarde.

Deze laatste operatie is van belang bij het algoritme van Dijkstra om kortste paden in een graaf te vinden. Dit algoritme wordt in de loop van het college behandeld.

De complexiteit van de operaties is $\mathcal{O}(1)$ voor *IsEmpty*, *Minimum*, *Insert*, *Merge* en *Decrease*, en $\mathcal{O}(\log(n))$ voor *DeleteMin*, waarbij n het aantal elementen in de queue is. Deze complexiteit is niet zoals gewoonlijk een *worst-case* complexiteit (waar het aantal stappen geteld wordt dat de operatie in het slechtste geval kost, onafhankelijk van de eerder uitgevoerde operaties) maar een zogenaamde "*amortized*" complexiteit waar de kosten van een operatie uitgesmeerd worden over alle operaties die uitgevoerd werden.

Ter illustratie: Bekijken we het voorbeeld van een stapel in een verbonden lijst representatie, dan heeft de operatie *MakeEmpty* een *worst-case* complexiteit $\mathcal{O}(n)$ omdat alle stapelelementen één-voor-één weggehaald moeten worden. De *amortized* complexiteit is daarentegen $\mathcal{O}(1)$ omdat de kosten voor het ontstapelen omgeslagen kunnen worden over alle n stapel operaties die de huidige elementen op de stapel gezet hebben.

Basis-operaties

Een *MinBoom* is een boom waarin de waarde van elke knoop kleiner is dan of gelijk is aan die van zijn kinderen, maar anders dan bij de Heap is er geen bovengrens aan het aantal kinderen van elke knoop. De Fibonacci Queue (FiboQueue) wordt gerepresenteerd als een bos van MinBomen (van een speciale vorm). De wortel met de kleinste waarde is de minimale waarde aanwezig in de queue. Deze wortel wordt direct bijgehouden om snel beschikbaar te zijn.

Merge wordt zeer eenvoudig geïmplementeerd: voeg de twee bossen samen tot één bos. Een enkele test bepaalt welk minimum voor de samengevoegde queue gaat gelden.

De operatie *Insert* is net zo eenvoudig: Maak voor de nieuwe waarde een FiboQueue bestaande uit één boom met één knoop. Deze FiboQueue wordt met de operatie *Merge* aan de gegeven FiboQueue toegevoegd.

De operatie *DeleteMin* bestaat uit twee fasen. Allereerst wordt de minimale wortel verwijderd. Elk van de kinderen van deze wortel keert terug in de oorspronkelijke FiboQueue, maar nu als wortel van een MinBoom in het bos. Hierna wordt het bos compacter gemaakt: zolang er twee bomen zijn van gelijke graad (dwz. evenveel kinderen van de wortels) worden ze samengevoegd tot één boom van een hogere graad door de wortel van de ene boom op te nemen als kind van de wortel van de andere boom.

Hierbij kiezen we natuurlijk de kleinste van de twee wortels als nieuwe wortel, zodat we weer een MinBoom verkrijgen.

Na een *DeleteMin* operatie geldt dat van elke graad ten hoogste één boom in de FiboQueue aanwezig is. Wanneer we alleen van de genoemde drie operaties gebruik maken zal de vorm van deze bomen zeer strak vastliggen: wanneer een knoop graad n heeft, hebben de n kinderen de graden $0, 1, \ldots, n-1$.

In eerste instantie lijkt *Decrease* ook eenvoudig geïmplementeerd te kunnen worden. Verander de waarde van de aangewezen knoop. Om te zorgen dat we een bos van MinBomen houden wordt de knoop (indien zelf geen wortel) zonodig losgemaakt van zijn vader en, samen met heel zijn nageslacht, als boom aan het bos toegevoegd. Deze operatie verandert de strakke vorm van de bomen die we eerder steeds behielden. Helaas verliezen we daarbij ook de gunstige prestaties van de datastructuur.

De representatie wordt daarom zo aangepast dat we van elke knoop (behalve de wortels uit het bos) bijhouden of deze een kind heeft verloren bij een eerdere *Decrease* operatie. Een knoop die een kind verliest, wordt gemarkeerd. Verliest een knoop een tweede kind, dan wordt de knoop zelf ook van zijn vader gesnoeid. Dit proces kan zich voortzetten wanneer deze vader ook zelf eerder al een kind kwijtgeraakt is: in de literatuur spreekt men daarom van "*cascading-cut*". Wanneer een knoop wortel wordt, verliest deze zijn markering.

Implementatie details. Omdat de bomen geen beperkte graad hebben, worden de wortels van het bos, en de kinderen van elke knoop, telkens in een dubbel verbonden lijst gehouden. Elke knoop heeft vier velden, één wijzend naar de vader, twee wijzend naar linker-, respectievelijk rechter broer (de dubbelverbonden lijst), en één wijzend naar een van de kinderen. Het aantal kinderen wordt apart in een veld van de knoop bijgehouden.

Inderdaad, deze uitleg is te compact om alles helemaal duidelijk te maken. Raadpleeg daarom bijvoorbeeld het *Leesboek Datastructuren* of een recent Datastructuren of Algoritmen boek. Keus genoeg.

Literatuur

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest. Introduction to Algorithms. MIT Press, 1990. Chapter 21: Fibonacci Heaps.

[2] M.L. Fredman & R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* 34, pp. 596–615, 1987.

[3] E. Horowitz, S. Sahni, D. Mehta. Fundamentals of Data Structures in C++. Computer Science Press, 1995. Chapter 9: Heap Structures. 9.5: Fibonacci Heaps.

[4] M.A. Weiss. Data Structure and Algorithm Analysis in C++. Benjamin/Cummings, 1994. Chapter 11: Amortized Analysis. 11.4: Fibonacci Heaps.

[5] D. Wood. Data Structures, Algorithms, and Performance. Addison-Wesley, 1993. Chapter 11: Priority Searching. 11.2.4: Fibonacci Queues.

```
typedef int QValueTp;  // ipv. template

class FiboNode
{ friend class FiboQueue;
  private:
    QValueTp Value;
    int      Degree;   // het aantal kinderen van de knoop
    FiboNode *Father,
             *Left, *Right,
             *Child;
    bool     Mark;     // al een kind verloren of niet
  // uw eigen velden & methodes (public, private)
};

class FiboQueue
{ public:
    FiboQueue ();
    ~FiboQueue ();
    bool     IsEmpty ();
    void     Merge (FiboQueue *Fqptr);
    QValueTp Minimum ();
    QValueTp DeleteMin ();
    FiboNode *Insert (QValueTp NwValue);
    void     Decrease (FiboNode *Fnptr, QValueTp NwValue);
  private:
    FiboNode *Min;
  // uw eigen velden & methodes (public, private)
};
```

Opdracht 2a: een kale Fibonacci Queue

Werk de datastructuur Fibonacci Queue uit tot een klasse FiboQueue in C++. Om de ingeleverde opdracht te kunnen gebruiken (en testen) moet de syntax van de basistypes van de klasse FiboQueue overeenkomen met het fragment hierboven.

`bool IsEmpty ();`
: Test of de queue leeg is.

`void Merge (FiboQueue *Fqptr);`
: Verplaatst alle elementen van `*Fqptr` naar de huidige queue.

`QValueTp Minimum ();`
: Levert het kleinste element uit de queue op.

`QValueTp DeleteMin ();`
: Verwijdert een minimaal element uit de queue, en geeft deze terug.

`FiboNode *Insert (QValueTp NwValue);`
: Voegt een element met waarde `NwValue` toe. De plek van dit element in de queue wordt als waarde teruggegeven.

`void Decrease (FiboNode *Fnptr, QValueTp NwValue);`
: Verlaagt de waarde van het aangewezen element tot `NwValue`.

Via de WWW-pagina van Datastructuren zijn enkele testprogramma's en testinstanties voor je implementatie beschikbaar.

Inleveren: uiterlijk 28 oktober 2005. Lever `fiboqueue.cc` en `.h` per email in bij Sven van Haastregt (svhaastr@liacs.nl), bijvoorbeeld met het commando:

```
tar cvzf - fiboqueue.cc fiboqueue.h | uuencode opdracht2a.gz |
       elm -s 'opdracht2a' svhaastr@liacs.nl
```

Een print van de programma's is niet nodig.

Opdracht 2b: gebruik in Dijkstra's algoritme

Vervolg de opdracht: Implementeer het algoritme van Dijkstra voor het vinden van afstanden in een gerichte graaf met gewichten. Ga uit van een graaf in *adjacency lists* representatie, en maak ook op een verstandige wijze gebruik van deze representatie. Gebruik de door u ontwikkelde FiboQueue en in het bijzonder ook de operatie *Decrease*.
Invoer. Het programma vraagt om een naam `xxx`, en leest dan de graaf uit het bestand `xxx.grf` in. Dit bestand bevat als eerste getal het aantal knopen $N \geq 1$ dat de graaf heeft. Dan op aparte regels de takken van de graaf beschreven door drie integers: beginknoop, eindknoop en gewicht. De knopen zijn getallen liggend tussen 1 en N. Het gewicht is positief. Het bestand mag commentaar bevatten: dit zijn regels waarvan het eerste karakter '%' is. Een voorbeeldbestand staat op het net: `fibo.grf`.

Het programma vraagt om een beginknoop, en rekent alle afstanden uit vanuit de gegeven beginknoop naar de overige knopen in de graaf.
Uitvoer. Het programma creëert het bestand `xxx.dis` met daarin de afstanden: elke regel bevat telkens een knoopnummer, en de bijbehorende afstand, volgens oplopend knoopnummer.
Datastructuur. De FiboQueue die reeds geïmplementeerd is, kan integers bevatten. Voor het algoritme lijkt het nodig om met een waarde ook een knoopnummer op te slaan. Pas hiertoe de FiboQueue aan.
Inleveren: uiterlijk 25 november 2005. Alle bestanden die nodig zijn om het programma te kunnen draaien moeten ingeleverd worden. Zorg er voor dat je een directory hebt waarin de benodigde bestanden staan en ga in deze directory staan. Stel dat je alleen `.cc`, `.h` bestanden en een Makefile nodig hebt, dan kan je met de volgende opdracht alles naar de corrector toesturen:

```
tar cvzf - *.cc *.h Makefile | uuencode opdracht2b.gz |
       elm -s 'opdracht2b' svhaastr@liacs.nl
```

Als er nog andere bestanden zijn die je mee wil sturen, bijvoorbeeld een README met daarin commentaar, dan zet je deze net na Makefile in bovenstaande regel. De object bestanden en de executable dien je echter niet op te sturen, er wordt opnieuw gecompileerd. Ook testinstanties zijn niet nodig.

Ten slotte: In principe kun je je programma's op je favoriete C++-platform ontwikkelen. De uiteindelijke versie moet echter gecompileerd kunnen worden (en werken) met g++ op de 'beast' of op een Linux-pc in de computerzalen beneden.

Werk bij voorkeur in tweetallen. Werk gestructureerd. Geef voldoende commentaar. Voorzie je werkstuk van een betrouwbare schatting van het aantal gewerkte uren.

Heb je nog vragen over de bedoeling van de opdracht, wend je dan tot Robert Brijder (kamer 156a, telefoonnummer 071-527 7143, `rbrijder@liacs.nl`). Heb je vragen over je eigen programma, wend je dan tot Sven van Haastregt.