Optimale binaire zoekbomen

Project Algoritmen en Datastructuren II

Academiejaar 2013-2014

1 Inleiding

Dit project combineert twee thema's die in de cursus besproken worden, namelijk binaire zoekbomen en dynamisch programmeren. Jullie hebben reeds in DAI gezien dat binaire zoekbomen efficiënte datastructuren zijn om elementen in op te zoeken. De kost van een zoekbewerking hangt af van de lengte van het pad (het aantal vergelijkingen op dat pad) dat je doorloopt tot je kan beslissen of het element al dan niet in de boom zit. Zoekbomen zoals AVL-bomen en rood-zwart bomen herbalanceren de boom om de diepte van de bladeren klein te houden, waardoor de gemiddelde zoekkost goed is indien elk element in de boom met even grote kans wordt opgezocht. In de praktijk is dit echter niet altijd het geval! Denk bijvoorbeeld maar aan een zoekmachine of spellingscontrole waarbij sommige woorden veel vaker worden opgezocht dan andere. Hierdoor kunnen perfect gebalanceerde zoekbomen toch slecht presteren. Vooral voor heel grote bomen of toepassingen waarbij elementen in de boom vergelijken heel duur is, is het dus belangrijk om de boom te herorganiseren zodat veel opgezochte sleutels bovenaan zitten.

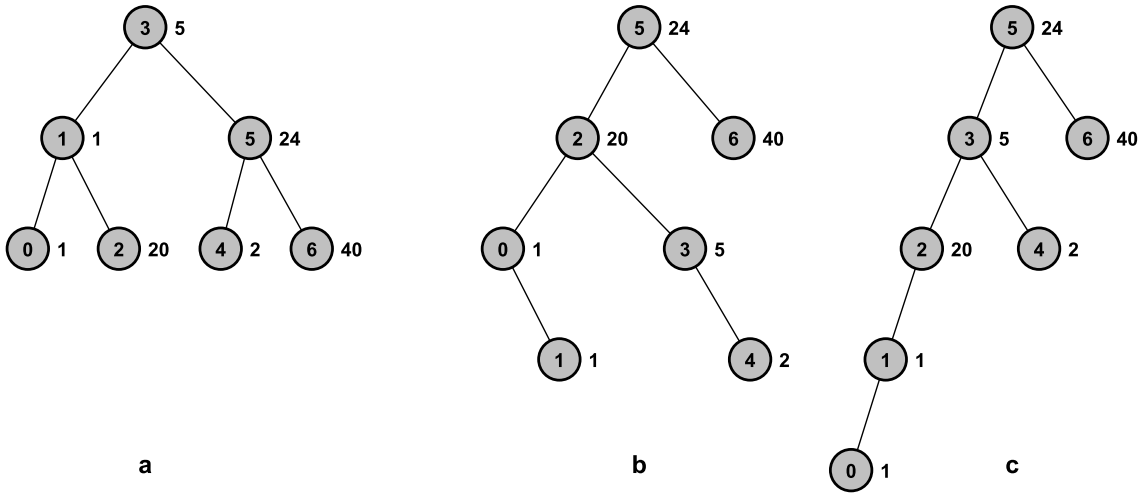
In het project wordt gevraagd om verschillende manieren te implementeren om zoekbomen te herorganiseren om de gemiddelde kost van zoekbewerkingen klein te houden, waarbij mogelijk de elementen niet met een even grootte kans worden opgezocht. Semi-splay bomen (besproken in de cursus) zijn binaire zoekbomen die aan dit criterium voldoen door na elke zoekbewerking het gezochte element dichter bij de wortel te plaatsen. Semi-splay bomen zouden dus in theorie zich moeten aanpassen aan de distributie van opzoekingen. Indien je echter de distributie op voorhand kent, kan je ook de zoekboom berekenen die de minimale kost heeft voor de gegeven distributie aan de hand van een dynamisch programmerenalgoritme. Alhoewel deze methode je een optimale boom oplevert, moet deze wel eerst worden berekend — wat tijd kost — en is de eigenlijke distributie niet altijd op voorhand gekend. De bedoeling van dit project is dus ook om deze methoden (en andere die je zou kunnen bedenken) onderling en met gewone binaire zoekbomen te vergelijken in verschillende situaties om zo de voor- en nadelen van de methoden te ontdekken.

In het project zal de boom op voorhand geen gekende distributie hebben die aangeeft wat de kans is dat een sleutel wordt opgezocht. Deze kans wordt berekend door bij te houden hoeveel keer een sleutel reeds werd opgezocht. Dit is het *gewicht* van de sleutel of top (doordat we spreken over binaire bomen, zullen we beide begrippen door elkaar gebruiken). Merk op dat **toevoegen** van een sleutel ook een soort zoekbewerking is en dus meetelt in het gewicht van de top. De kost van een top en de kost van een zoekboom kunnen aan de hand van het gewicht en de diepte van een top in de boom worden berekend.

Definitie 1 *Zij $v_0 < v_1 \ldots < v_n$ een verzameling sleutels. Een interval $[i, j]$ stelt de deelverzameling sleutels $v_i < v_{i+1} \ldots < v_j$ voor. Indien $i > j$ is de deelverzameling leeg. Zij $T[0, n]$ een binaire zoekboom voor de sleutels $v_0, \ldots, v_n$, waarvoor we de top met sleutel v_k ook als v_k noteren. De **diepte** van een top v_k in een binaire zoekboom wordt genoteerd als $d(v_k)$. Het **gewicht** $w(v_k)$ van sleutel v_k is het aantal keer dat een top tot nu toe werd opgezocht of toegevoegd. Het gewicht van een boom $T[i, j]$ is de som van de gewichten van de sleutels in de toppen: $w(T[i, j]) = \sum_{k=i}^{j} w(v_k)$ voor $i \leq j$ en $w(T[i, j]) = 0$ indien $i > j$.*
*De **kost** van een sleutel $c(v_k)$ is $c(v_k) = (d(v_k) + 1) \cdot w(v_k)$. Analoog volgt dat de kost van de binaire zoekboom gelijk is aan:*

$$c(T[i, j]) = \sum_{k=i}^{j} c(v_k) = \sum_{k=i}^{j} (d(v_k) + 1) \cdot w(v_k), i \leq j.$$

*Natuurlijk geldt $c(T[i, j]) = 0$ indien $i > j$. Een **optimale binaire zoekboom** voor de sleutels $v_0 < v_1 \ldots < v_n$ is een binaire zoekboom met de kleinst mogelijke kost voor gegeven gewichten.*



Figuur 1: Drie binaire zoekbomen met dezelfde sleutels en dezelfde gewichten van de sleutels (aangegeven naast de toppen). *a* is een perfect gebalanceerde binaire zoekboom waarvan de totale kost 244 bedraagt. *(b)* is een binaire zoekboom met minimale kost, namelijk 174. *(c)* is een semi-splay boom die werd bekomen voor de toegekende gewichten (een andere volgorde van opzoeken kan een andere boom opleveren). De kost van de semi-splay boom is 189.

Een voorbeeld van deze definities kan je zien in figuur 1. Op deze figuur staan verschillende binaire zoekbomen die dezelfde sleutels bevatten en waarvan de sleutels ook hetzelfde gewicht hebben. Sleutel 1 werd enkel toegevoegd en niet opgezocht en heeft dus gewicht 1. Daarentegen werd sleutel 6 reeds 40 keer opgezocht en is dus $w(6) = 40$. De meest linkse boom in figuur 1 is een perfect gebalanceerde boom. Het is echter meteen duidelijk dat dit geen optimale zoekboom is, aangezien toppen die veel worden opgezocht diep in de boom zitten en dus meer vergelijkingen moeten worden uitgevoerd om het element te vinden. De kost van de linkse boom is $1 \cdot 5 + 2 \cdot (24 + 1) + 3 \cdot (1 + 20 + 2 + 40) = 244$ (voor de eenvoud zijn de toppen met dezelfde diepte gegroepeerd in de som).

De middelste binaire zoekboom in figuur 1 is een optimale binaire zoekboom voor deze verdeling van gewichten. De kost van de boom is $1 \cdot 24 + 2 \cdot (40 + 20) + 3 \cdot (1 + 5) + 4 \cdot (1 + 2) = 174$. De minimale kost (en een boom die deze kost heeft) kan worden berekend aan de hand van een dynamisch programmerenalgoritme. Een aanzet hiervoor is een recursieve definitie die steunt op de ordening van de sleutels in de boom. De kost voor de gehele binaire zoekboom $T[0, n]$ is namelijk gelijk aan het gewicht $w(v_r)$ van de wortel v_r plus de kosten van de deelbomen $T[0, r-1]$ en $T[r+1, n]$, gerekend als afzonderlijk bomen, vermeerderd met de gewichten van alle toppen in die deelbomen (omdat de diepte met 1 verhoogd is voor alle toppen in de deelbomen). Dit geeft:

$$c(T[0, n]) = c(T[0, i-1]) + w(T[0, n]) + c(T[i+1, n]), \quad 0 \leq i \leq n.$$

De minimale kost van een binaire zoekboom is dus gelijk aan het minimum van bovenstaande som over alle mogelijke plaatsen van de wortel. Deze recursieve definitie kan dan als basis gebruikt worden van een dynamisch programmerenalgoritme.

De rechtse binaire zoekboom in figuur 1 is een semi-splayboom die dus na elke operatie de semi-splaybewerkingen uitvoert. Merk op dat de vorm van de semi-splayboom sterk afhangt van de volgorde waarin de sleutels werden opgezocht, en jullie dus een andere zoekboom kunnen bekomen indien de sleutels in een andere volgorde werden toegevoegd en/of gezocht. De kost van deze boom is 189. Dit is vrij goed, maar niet zo goed als de optimale binaire zoekboom. Om de voor- en nadelen van een semi-splayboom te vinden zal je natuurlijk wel moeten testen op grotere verzamelingen dan in dit klein voorbeeldje.

In dit project zullen de sleutels worden voorgesteld door *integers*. Om toch rekening te houden met toepassingen waarbij het vergelijken van twee elementen duurder is dan het vergelijken van twee getallen, kan naast de uitvoeringstijd van de methoden ook het **aantal navigaties** worden bijgehouden voor een zoekbewerking. Dit is gelijk aan de diepte van het pad waarop gezocht werd plus één. Op die manier kan berekend worden hoe duur een vergelijking moet zijn voor een bepaalde methode efficiënter zou worden dan een andere.

2 Implementatie en verslag

2.1 Algemeen

Binaire zoekboom

Als basis moet je een eenvoudige binaire zoekboom implementeren voor integer-getallen. Aan deze boom moet je elementen kunnen toevoegen en elementen opzoeken. De diepte van het pad tijdens het opzoeken moet worden weergegeven als resultaat van het opzoeken. Je moet ook de gewichten (aantal keer een element werd opgezocht) bijhouden en de totale kost van de binaire zoekboom kunnen teruggeven. Tot slot moet je een binaire zoekboom ook kunnen omvormen tot een gebalanceerde zoekboom. Een volledig overzicht van de te implementeren functies staat vermeld in sectie 2.3. De implementatie hoeft voor het bijhouden van de gewichten (en dus de kost van de boom) **geen** rekening te houden met elementen die niet in de boom zitten. De methoden moeten echter wel het juiste resultaat geven indien een element wordt opgezocht dat niet in de boom zit.

Semi-splay bomen

Implementeer semi-splaybomen (waarbij je enkel toevoegen en opzoeken van elementen moet implementeren).

Optimale binaire zoekbomen

Implementeer een dynamisch programmerenalgoritme dat een gegeven binaire zoekboom herorganiseert tot een optimale binaire zoekboom.
Beschrijf dit algoritme duidelijk in je **verslag** en bespreek de correctheid en tijds- en geheugencomplexiteit.

Vergelijking

Één van de belangrijkste onderdelen van het project is het onderzoeken van de voor- en nadelen van verschillende methoden om binaire zoekbomen te herorganiseren. Vergelijk heel goed de bovenstaande methoden en vergelijk ze ook met de eenvoudige methode waarbij je geen herordening uitvoert. Afhankelijk van je bevindingen kan je zelf **optimalisaties** of **eigen methoden** bedenken en deze ook vergelijken. In plaats van een dynamisch programmerenalgoritme zou je bijvoorbeeld een gretig algoritme kunnen toepassen, of semi-splay aanpassen zodat er soms niet wordt gesplayd... Vergelijk in het **verslag** voldoende aspecten en situaties waarin bepaalde methoden beter of slechter geschikt zijn. Bespreek zeker theoretische aspecten zoals tijd- en geheugencomplexiteit, maar ook resultaten van praktische tests (zie sectie 2.2).

2.2 Experimenten

Om de methoden beter te testen kunnen jullie gebruik maken van enkele programma's die jullie terugvinden op `minerva` in de map `project`. Deze programma's genereren getallen met specifieke distributies die weggeschreven kunnen worden naar een binair bestand en weer kunnen ingelezen worden met de klasse `NumberReader`. Meer informatie over deze bestanden is ook op `minerva` te vinden.

Een typische testsituatie begint met het inlezen van de sleutels die toegevoegd worden aan de zoekbomen. Vervolgens worden een hele reeks zoekbewerkingen uitgevoerd op sleutels die ingelezen worden uit de invoerbestanden. Op één of meerdere momenten zal op één van de varianten het algoritme worden opgeroepen om de boom te herorganiseren op basis van de gewichten die op dat moment gekend zijn. Van elke zoekbewerking wordt het aantal navigaties bijgehouden, zodat op het einde het totale aantal navigaties voor de hele dataset kan worden teruggegeven. Samen met de uitvoeringstijd van de methoden, kan dan worden nagegaan hoe duur of hoe goedkoop 1 vergelijking zou moeten zijn om een bepaalde methode voordeliger te maken.

Vermeld in je **verslag** hoe je de **correctheid** van alle onderdelen van je implementatie getest hebt en hoe je ervoor gezorgd hebt dat je **tijdsmetingen** accuraat waren. Vermeld bij je tijdsmetingen ook duidelijk welke soort input je gebruikt hebt: genereer **verschillende**

soorten testdata met bepaalde eigenschappen. Zorg er ook voor dat de data sets **voldoende groot** zijn. Bomen van enkele duizenden of tienduizenden toppen zijn normaal en het aantal opzoekbewerkingen moet hoog genoeg zijn om zowel goede tijdsmetingen te bekomen als om het effect van bepaalde distributies te voelen. Mogelijke situaties waarop kunnen worden getest zijn verschillende distributies, wanneer en hoeveel keer de boom wordt gereorganiseerd met dynamisch programmeren, ... Bedenk zelf nog enkele andere parameters die de vergelijking mogelijk beïnvloeden. De testcode zal ook ingediend moeten worden in een afzonderlijke map (zie sectie 3).

Voorzie duidelijke en correcte grafieken en tabellen, maar becommentarieer en verklaar je resultaten ook voldoende. Zorg ervoor dat je ook een duidelijk besluit vormt. Komen je testresultaten overeen met wat je verwacht had? Indien niet, **verklaar** verschillen.

2.3 Concrete implementatie

Al je implementaties van binaire zoekbomen dienen de interface `BST` te implementeren. Deze interface mag niet gewijzigd worden en de packagestructuur moet behouden blijven! De interface `BST` beschikt over volgende methoden:

`int contains(int value);`	geeft terug of de sleutel in de boom zit. De return waarde is de kost van het opzoeken (aantal navigaties). Als de sleutel niet in de boom zit, geef dan het aantal navigaties terug vermenigvuldigd met -1.
`int add(int value);`	voegt een sleutel toe aan de boom. De return waarde is dezelfde als voor de methode `contains`, maar nu niet-negatief als de sleutel nog niet in de boom zat en. negatief als de sleutel wel reeds in de boom zat.
`void balance();`	herorganiseert de binaire zoekboom tot een perfect gebalanceerde zoekboom.
`int cost();`	geeft de kost terug van de zoekboom, gebaseerd op de huidige gewichten van de sleutels.
`void optimize();`	herorganiseert de binaire zoekboom tot een optimale binaire zoekboom voor de huidige gewichten van de sleutels. Deze methode hoeft niet geïmplementeerd te worden voor zelforganizerende zoekbomen zoals semi-splay bomen.
`int size();`	geeft de grootte van de boom terug.

Belangrijk: voor de eenvoud hoef je geen rekening te houden met de kost van het opzoeken van sleutels die niet in de boom zitten. De `contains` methode moet echter wel steeds het correcte antwoord geven. De methode `optimize` moet niet worden geïmplementeerd voor zoekbomen die automatisch de boom herorganiseren.

Lees aandachtig de commentaar die zich in de code bevindt! De commentaar geeft aan in welke situaties welke uitvoer moet teruggegeven worden. Indien je toch nog twijfelt over het gedrag van een bepaalde methode vraag je dit aan de assistenten.

Opdat we je project op een efficiënte manier kunnen testen, vragen we je om alle klassen die de interface `BST` implementeren van een **default constructor** te voorzien. Klassen waarop de

methode `optimize` moet worden toegepast, geef je de naam `Obst1`, `Obst2`, ..., `Obst3`. Klassen die automatisch herorganiseren, zoals semi-splay bomen, geef je de naam `Sbst1`, `Sbst2`, ..., `Sbst3`.

Minstens 1 klasse moet een normale binaire zoekboom zijn waarop het dynamisch programmerenalgoritme kan worden uitgevoerd en minstens 1 klasse moet een implementatie van semi-splaybomen bevatten. Je code moet dus minstens `Obst1` en `Sbst1` bevatten. Vermeld in je **verslag** ook met welke algoritmes `Obst[1-N]` en `Sbst[1-N]` overeenkomen.

2.4 Theoretische vragen

Werk in je **verslag** ook volgende theoretische opgaven uit:

- *In het project werd geen rekening gehouden met sleutels die niet in de boom zitten. Deze opzoekingen vergen echter ook een aantal navigaties. Het is mogelijk om ook in deze situatie een zoekboom te bouwen zodat het* ***totaal aantal navigaties*** *van een reeks zoekopdrachten nog steeds minimaal is. Bespreek hoe een binaire zoekboom en de algoritmen moeten worden aangepast om dit te bewerkstelligen. Indien nodig geef je nieuwe definities voor nodige begrippen die in deze situatie niet meer gelden. Bespreek ook wat de invloed zou zijn op de efficiëntie van bepaalde methoden.*
- *De kost van een zoekbewerking in een binaire zoekboom werd in dit project gedefinieerd aan de hand van het aantal navigaties op het pad dat doorlopen wordt tijdens het zoeken. De echte kost van een navigatie hangt echter af van de vergelijkingen tussen sleutels. Doordat een sleutel kleiner, gelijk of groter kan zijn, moeten soms twee vergelijkingen uitgevoerd worden voor een navigatie en soms slechts één. Stel bijvoorbeeld dat eerst wordt getest of de gezochte sleutel kleiner is dan de sleutel in de huidige top (waardoor een navigatie naar het linkerkind goedkoper is dan een navigatie naar het rechterkind). Leg uit hoe de kost van een sleutel, de kost van een boom en het dynamisch programmeren algoritme moeten worden aangepast zodat gebruik kan gemaakt worden van deze nieuwe kost van een zoekbewerking. Denk je dat dit een groot effect heeft op de structuur van de boom en/of de snelheid van het zoeken in de praktijk? Bespreek.*

2.5 Overige

Presenteer je resultaten in overzichtelijke grafieken en bespreek ze ook grondig. Probeer al je resultaten te verklaren. Voorzie je code ook van commentaar, vermijd duplicatie van code, werk met overerving en licht al je ontwerpsbeslissingen toe in het verslag. Hou je verslag beknopt, maar zorg ervoor dat alles voldoende grondig besproken wordt.

3 Indienen

Er moet verplicht tweemaal worden ingediend. Een eerste versie dient tussentijds elektronisch ingediend te worden **vóór vrijdag 1 november 2013 om 17u00**. Deze versie dient ten minste een implementatie te bevatten van de binaire zoekboom met een implementatie van alle functies in de interface `BST`, inclusief het dynamisch programmerenalgoritme om een

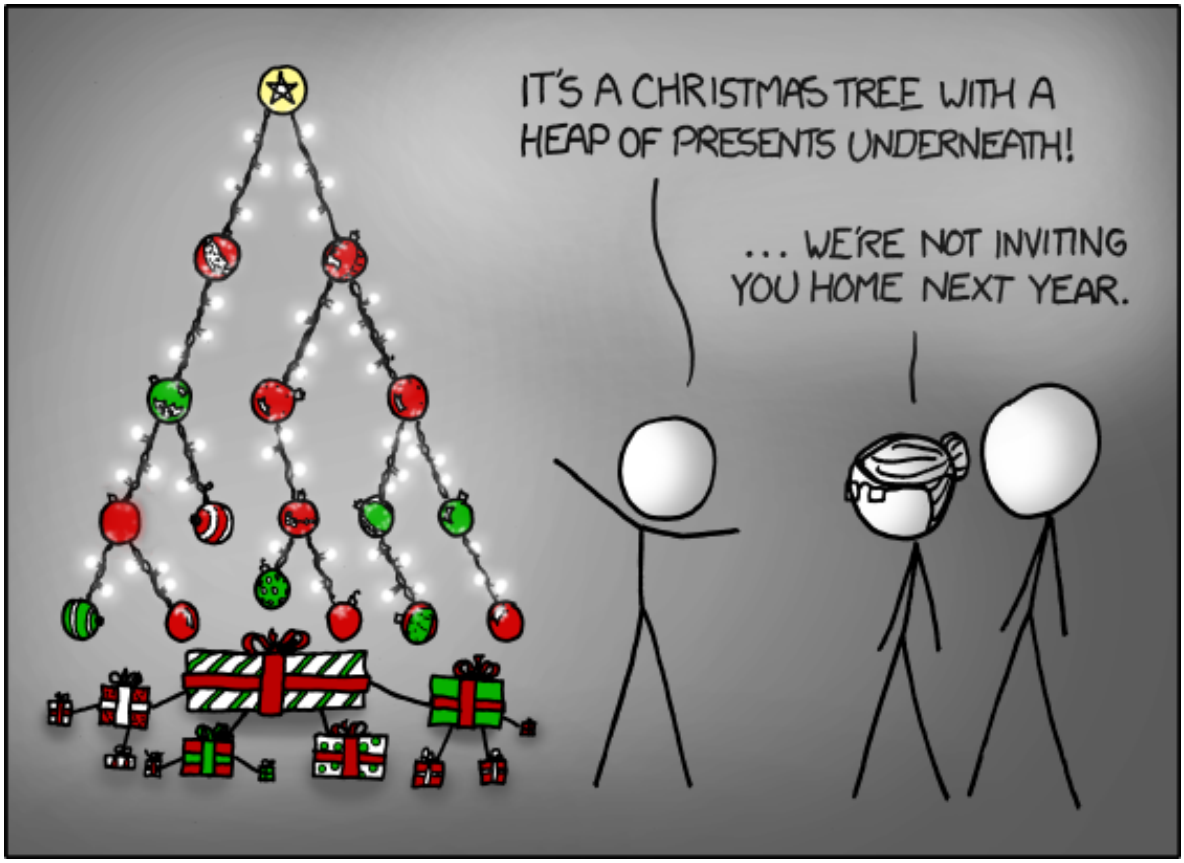
optimale binaire zoekboom op te bouwen. Je uiteindelijke project dien je in **vóór maandag 25 november 2013 om 17u00**. Code en verslag worden elektronisch ingediend. Van het verslag verwachten we ook een **papieren versie**. Nadien zal je mondeling je werk verdedigen; het tijdstip hiervoor wordt later advalvas bekendgemaakt.

Elektronisch indienen Voor de automatische verbetering is het belangrijk dat je bestanden correct ingediend worden. Pak de elektronische bestanden precies in zoals beschreven. Je dient één zipbestand in via `http://indianio.ugent.be` met de volgende inhoud:

- `src/` bevat alle broncode inclusief de ongewijzigde interfaces die gegeven werden. Behoud de package structuur en maak ook GEEN subpackages of subdirectories van `obst` aan!
- `tests/` alle testcode. De testcode mag wel in een ander package zitten.
- `extra/verslag.pdf` de elektronische versie van je verslag. In de map `extra` kan je ook extra bijlagen plaatsen.

Algemene richtlijnen

- Zorg ervoor dat je code volledig compileert met een Sun Java compiler v1.6 of hoger. Extra libraries zijn niet toegestaan (je kan echter voor alle duidelijkheid wel gebruik maken van de standaard JRE library). Voor de tests mag **wel** gebruik gemaakt worden van **JUnit**. Niet compileerbare code en projecten die niet correct verpakt werden **worden niet beoordeeld**.
- Schrijf efficiënte code maar ga niet overoptimaliseren: **geef voorrang aan elegante, goed leesbare code**. Kies zinvolle namen voor klassen, methoden, velden en variabelen (gebruik de correcte conventies) en voorzie voldoende documentatie.
- Het project wordt gequoteerd op **4** van de 20 te behalen punten voor dit vak, en deze punten worden ongewijzigd overgenomen naar de tweede examenperiode.
- Projecten die ons niet bereiken voor de deadline worden niet meer verbeterd: dit betekent het verlies van alle te behalen punten voor het project.
- Het is **strikt noodzakelijk** twee keer in te dienen: het niet indienen van de eerste tussentijdse versie betekent sowieso het verlies van alle te behalen punten voor het project.
- Dit is een individueel project en dient dus door jou persoonlijk gemaakt te worden. Het eerste, theoretische gedeelte bestaat volledig uit jouw ideeën en onderbouwde redeneringen, niet die van anderen. Voor de rest van het project is het toegestaan om andere studenten te helpen of om ideeën uit te wisselen, maar **het is ten strengste verboden code uit te wisselen of over te nemen van het internet**, op welke manier dan ook. Het overnemen van code beschouwen we als fraude (van **beide** betrokken partijen) en zal in overeenstemming met het examenreglement behandeld worden.
- Essentiële vragen worden **niet** meer beantwoord tijdens de laatste week voor de deadline.



Figuur 2: "Not only is that terrible in general, but you just KNOW Billy's going to open the root present first, and then everyone will have to wait while the heap is rebuilt."