Algoritmen en Datastructuren II
Project:
Variëren op Semi-splays

Academiejaar 2014–2015

1 Inleiding

Splaybomen hebben een aantal unieke eigenschappen in vergelijking met andere zelfbalancerende zoekbomen. Waar de meeste zelfbalancerende zoekbomen de structuur van de boom alleen wijzigen als er data wordt toegevoegd of verwijderd, wijzigen splaybomen ook de structuur als er data wordt opgezocht. Immers, van alle opgeslagen data wordt er vaak slechts een klein deel daadwerkelijk opgezocht. Daarbij wordt vaak reeds opgezochte data later vele malen nog eens opgezocht. Er is dus sprake van een sterke asymmetrie in het opzoekgedrag. Dit wordt in de informatica aangeduid als "lokaliteit van referentie", of ook als "principe van lokaliteit". Lokaliteit wordt onderscheiden naar tijd en naar ruimte:

Bij temporele lokaliteit geldt: Als op een zeker moment in de tijd een bepaalde locatie wordt gebruikt, dan is het waarschijnlijk dat in de nabije toekomst dezelfde locatie nog eens zal worden gebruikt.

Bij ruimtelijke lokaliteit geldt: Als een zekere locatie wordt gebruikt dan is het waarschijnlijk dat locaties in de nabijheid ook zullen worden gebruikt.

Men vindt het principe van de lokaliteit op allerlei plaatsen in de informatica terug. Het is de reden dat caching werkt. Bijvoorbeeld gebruikt men in de computerarchitectuur processorcaches om de traagheid van het RAM-geheugen te overkomen. Echter, dit werkt slechts dan als er sprake is van zowel temporele als ruimtelijke lokaliteit in het gedrag van een applicatie. Ontbreekt er één dan verliezen deze caches hun nut en zal de uitvoeringstijd aanzienlijk toenemen.

Bij splaybomen wordt met het lokaliteitsprincipe actief rekening gehouden door gebruikte paden in de boom te optimaliseren. Niet alleen opzoekgedrag is vaak asymmetrisch, maar ook toevoeggedrag en verwijdergedrag. Data die op een bepaald moment relevant is is vaak samengesteld of geclusterd. Dus ook voor verwijder- en toevoegoperaties is het lonend om de plaatsen van opslag dichter bij de wortel te brengen.

Dit alles is tot nu toe slechts informeel aangeduid. De cursus maakt de eigenschappen van splaybomen preciezer door middel van formalisaties. In dit project gaan wij semi-splaybomen, en varianten daarop, implementeren en met elkaar vergelijken.

2 Opdracht

Zoals al aangestipt is in de cursusnota's zijn semi-splaybomen een onderschatte variant op splaybomen. Semi-splays hebben onder veelvoorkomende omstandigheden een iets betere

performantie dan gewone splaybomen. Dit gegeven geeft aanleiding tot de vraag of er nog een andere variant op splaybomen te vinden is met een nog betere performantie. Dat is de wetenschappelijke onderzoeksvraag waarmee wij ons in dit project willen bezighouden.

Om te beginnen willen wij in dit project semi-splaybomen implementeren, inclusief de operaties om toppen toe te voegen, op te zoeken en te verwijderen. Naast deze implementatie willen wij ook **twee varianten** op semi-splay implementeren om deze daarmee te kunnen vergelijken door middel van experimenten op gegenereerde datasets. Wij onderscheiden de volgende splayboom varianten:

Semi-splay: Het splay-algoritme staat beschreven in Algoritme 2 Semi-splay langs een pad op pagina 14 van de cursusnota's. Deze moet altijd worden geïmplementeerd.

Splay: Het splay-algoritme voor de klassieke splayboom en de verschillen met semi-splay staan goed beschreven in de Nederlandstalige Wikipedia:
https://nl.wikipedia.org/wiki/Splayboom
Wij gebruiken steeds de *bottom-up* methode. Het belangrijkste idee achter gewone splay is dat de opgezochte top door het splayen als nieuwe wortel van de boom eindigt. Daarom wordt hier ook gesplayd als de deelboom uit twee toppen bestaat.

Independent semi-splay: Bij semi-splay nemen wij steeds de wortel van de deelboom van 3 toppen die onstaat uit één splay-stap als de nieuwe onderste top van de volgende deelboom van 3 toppen voor de volgende splay-stap. Daarentegen bij *independent semi-splay* laten wij deze reeds gesplayde wortel met rust en gaan wij eerst één stap verder omhoog alvorens de volgende splay-stap op 3 toppen te doen. Dit betekent dus dat bij één opzoek-, verwijder-, of toevoegoperatie een willekeurige top ten hoogste éénmaal deel uitmaakt van een splaybewerking. Dit in tegenstelling tot gewone semi-splay waar het mogelijk is dat één top meerdere keren deel kan uitmaken van een splaybewerking.

Extended semi-splay: Bij gewone semi-splay splayen wij bij elke splaystap steeds met 3 toppen tegelijk. De gesplayde deelboom wordt dan een gebalanceerde boom met een wortel met twee kinderen. Dit is een boom van $2^n - 1$ toppen met $n = 2$. Bij extended semi-splay stellen wij n op 3 en splayen wij op deelbomen van $2^3 - 1 = 7$ toppen.

Full semi-splay: Bij deze variant splayen wij alle toppen van het volledige pad ineens. Bij elke opzoek-, verwijder-, of toevoegoperatie is er daarom ten hoogste één splay-operatie. De enige eis, die wij steeds bij semi-splay hanteren, is dat het pad tenminste 3 toppen bevat.

Student semi-splay: De student is vrij om zelf een variant te bedenken, als gemotiveerd kan worden waarom deze interessantie performantie-eigenschappen zou kunnen hebben. Het is verplicht om deze motivatie even bij een assistent te **verifiëren**.

Concreet vragen we minstens implementaties van:

1. semi-splay,
2. splay **of** independent semi-splay,
3. extended semi-splay **of** full semi-splay **of** student semi-splay.

Wij zijn vooral geïnteresseerd in verschillen in performantie tussen de varianten. Om die goed aan te tonen moet er getest worden met grote verzamelingen sleutels. Dit wordt verder besproken in paragraaf 4 Experimenten.

3 Implementatie

Wij implementeren semi-splay en de twee varianten in Java, waarbij elk boomtype wordt gerealiseerd in een eigen klasse. Al deze klassen moeten voldoen aan de specificatie van de navolgende abstracte klasse `AbstractTree`. Uiteraard mag er wel gebruik gemaakt worden van overerving. Immers, de verschillen tussen de varianten worden bepaald door de splaytechniek. De rest van de implementatie zal daarom identiek kunnen zijn en is dus goed overerfbaar.

```
1
2  package semisplay;
3
4  import java.util.ArrayList;
5
6  /**
7   * Specification for an abstract tree class for binary search trees
8   * (specifically for semi-splay and related BSTs).
9   *
10  * This class includes the interface for Iterable, so as to be able to use it
11  * in for-each clauses. In this class and all subclasses we will be using
12  * the provided Top class (unmodified) to represent nodes in the tree.
13  *
14  * The goal of any implementation should be to reuse as much of the methods
15  * when implementing other methods. However, this only applies in sofar we
16  * can guarantee optimal performance. We want to minimize fiddling with
17  * references to children when possible.
18  *
19  * Recursion is a very powerful concept, but unfortunately recursion
20  * in Java can be rather slow and it brings the danger of stack overflows.
21  * If desired, we can always emulate recursion with a stack.
22  * To that end, we freely provide the array-based TopStack class.
23  *
24  * Use an editor setting of 80 columns maximum, replace all tabs with 8 spaces,
25  * indentation is always 4 spaces, and remove trailing spaces on file save.
26  *
27  * We suggest to implement the following methods approximately in the order
28  * they are presented here. This may help one to gradually become familiar
29  * with the structures and progressively tackle more subtle problems.
30  */
31 public abstract class AbstractTree implements TreeIterable<Key> {
```

```
32
33      /** @return A reference to the root node of the tree. */
34      public abstract Top getRoot();
35      /** @param root Use the given tree as the new tree unmodified. */
36      public abstract void setRoot(Top root);
37
38      /** Set a limitation on the number of splay operations
39       * which are done for each lookup/insert/remove operation.
40       * @param limit Limit splays if >= 0, unlimited splays if negative.
41       * By default the number of splay operations is not limited (-1). */
42      public abstract void setSplayLimit(int limit);
43
44      /** @return A new and complete deep-copy of the current tree.
45       * This method must be implemented first, in order to use Toppie! */
46      public abstract AbstractTree copy();
47
48      /** @return The depth of the tree. */
49      public abstract int getDepth();
50      /** @return The total number of top-nodes in the tree. */
51      public abstract int getSize();
52
53      /** Does the current tree conform to the formal specification of a BST? */
54      public abstract boolean isBinarySearchTree();
55
56      /** Check whether the current tree is a balanced tree. */
57      public abstract boolean isBalanced();
58      /** Rebalance the current tree. */
59      public abstract void rebalance();
60
61      /** Give all elements of the tree in natural order (in-order).
62       * For a definition see theorem 16.1.5 in the course notes of AD1.
63       * @return ArrayList with all keys. */
64      public abstract ArrayList<Key> toArrayList();
65
66      /** Return the smallest key in the tree.
67       * @return The smallest key if the tree was non-empty, else null. */
68      public abstract Key getSmallest();
69      /** Return the largest key in the tree.
70       * @return The largest key if the tree was non-empty, else null. */
71      public abstract Key getLargest();
72
73      /** Lookup the given Key in the current tree.
74       * @return True if and only if the key was found. */
75      public abstract boolean lookup(Key key);
76
77      /** Insert a new Key, but avoid duplicates.
78       * @return True if insertion succeeded (the key is not a duplicate). */
79      public abstract boolean insert(Key key);
80
81      /** Remove the given Key from the tree.
82       * @return True if and only if the key was found and deleted. */
83      public abstract boolean remove(Key key);
84
85      /** @return A new iterator over the current tree. */
86      public abstract TreeIterator<Key> iterator();
87
88      /** Maintain some counters for measurement purposes. */
89      private static int insertions, constructed, duplicates,
90                         deletions, notFounds, comparisons;
91      /** Reset all counters. */
```

```
92      public static void resetStatistics() {
93          insertions = constructed = duplicates =
94          deletions = notFounds = comparisons = 0;
95      }
96
97      /** Count how many keys have been inserted. */
98      public static void incrInsertions() { ++insertions; }
99      public static int getInsertions() { return insertions; }
100     /** Count how often a new Top object has been created. */
101     public static void incrConstructed() { ++constructed; }
102     public static int getConstructed() { return constructed; }
103     /** Count how many duplicate keys have not been inserted. */
104     public static void incrDuplicates() { ++duplicates; }
105     public static int getDuplicates() { return duplicates; }
106     /** Count how often a top has been deleted. */
107     public static void incrDeletions() { ++deletions; }
108     public static int getDeletions() { return deletions; }
109     /** Count how often a key lookup or deletion failed .*/
110     public static void incrNotFounds() { ++notFounds; }
111     public static int getNotFounds() { return notFounds; }
112     /** Count how many times we compared two keys .*/
113     public static void incrComparisons() { ++comparisons; }
114     public static int getComparisons() { return comparisons; }
115
116 }
```

Listing 1: AbstractTree.java specificeert de interface voor semi-splaybomen en varianten.

Zoals duidelijk uit deze listing blijkt voorziet deze klasse in een aantal statistieken. Deze worden gedeeld door alle subklassen. Steeds moet de implementatie van een boomvariant uitgerust worden met aanroepen naar de juiste `incr...` methoden om ervoor te zorgen dat de statistieken correct worden bijgehouden. Indien men naar eigen inzicht statistieken zou willen toevoegen, dan kan men daarvoor wellicht een abstracte subklasse `MyAbstractTree` definiëren.

Wij gebruiken als top een klasse met enkel velden voor een sleutel, een linkerkind en een rechterkind. In een top slaan wij dus *geen* informatie over ouders op. Na afloop zal wellicht blijken dat dit de implementatie versimpelt. In elk geval zorgt dit ervoor dat de top iets kleiner is. Tevens vermindert dit het aantal wijzigingen in kindtoppen tijdens het splayen. Zie voor de implementatie de klasse Top.java.

Wij stellen een sleutel in Java voor als een *long*, dat in Java een machinerepresentatie heeft van 64 bits. Echter, wij verbergen deze kennis in een klasse Key.java. Enkel het gegeven dat deze klasse de interface Comparable<Key> implementeert is voldoende voor de klassen `Top` en de implementaties van de semi-splaybomen. Ook klasse `Top` implementeert de interface `Comparable<Key>`. Een top kan dus vergeleken worden met sleutels. De methode `compareTop` in klasse `Top` vergelijkt een top met een andere. De uitkomst van een vergelijking is steeds een waarde `LT`, `EQ` of `GT` (met de respectieve betekenissen less than, equal to en greater than). Deze zijn gedefinieerd in de klasse `Key`.

Een semi-splayboom moet de meegeleverde interfaces TreeIterable.java en TreeIterator.java implementeren. Hierdoor kan de gebruiker van een boom itereren over de boom. Bijvoorbeeld met een Java for-each lus. Echter, hierdoor ontstaat de vraag in hoeverre dit invloed heeft op de gegarandeerde performantieëigenschappen van de datastructuur. Immers, een gebruiker kan de iteratie voortijdig afbreken, bijvoorbeeld met een `break`

opdracht. De student moet dit goed onderzoeken en de optimale eigenschappen van zijn oplossing formeel bewijzen.

Aangezien wij in kindtoppen geen informatie over oudertoppen bijhouden, zullen wij soms een pad door een boom moeten vastleggen. Wij bieden hiervoor de klasse TopStack aan, die echter niet verplicht is. Studenten mogen gerust andere oplossingen verzinnen, die handiger of beter zijn.

Alle Java broncodebestanden zijn steeds onderdeel van het Java package `semisplay`. Voor de boomvarianten schrijven wij de volgende verplichte namen voor: SemiSplay, Splay, Independent, Extended, Full en Student.

3.1 Visual Semi-splay Debugger

Om het programmeerwerk te vergemakkelijken stellen wij het programma Toppie beschikbaar. Zie figuur 1. Hiermee kan men interactief experimenteren met nieuwe implementaties van semi-splayboomvarianten. Wij leveren hiertoe de volgende broncodebestanden: Toppie.java, ToppieFXML.fxml, ToppieFXMLCompanion.java en AboutPopup.java. Men kan het programma compileren door in NetBeans een nieuw JavaFX-project aan te maken met de naam Toppie met als package `semisplay`. Vervolgens kopieert men genoemde bestanden naar de `semisplay` subdirectory. Dit is reeds behandeld tijdens Programmeren 2 in voorjaar 2014. Toppie *mag* naar believen aanpast worden. Dit hoeft niet ingeleverd te worden (maar mag wel). Men is volledig vrij om Toppie wel of niet te gebruiken.

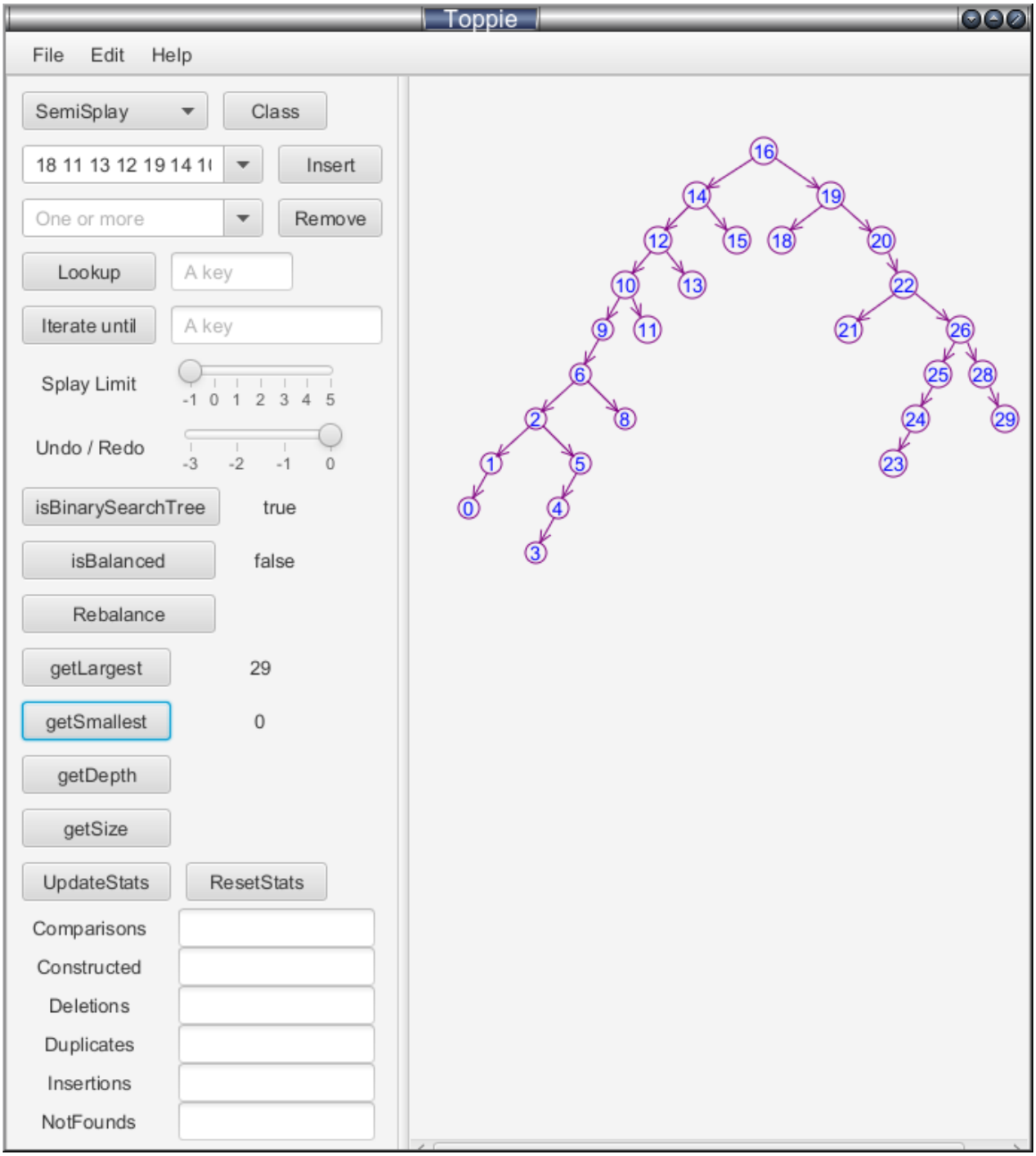
4 Experimenten

Om de eigenschappen van semi-splayboomvarianten goed te kunnen bepalen moet getest worden op enige omvangrijke datasets. Om dit te vergemakkelijken leveren wij twee programma's in `C` om datasets te genereren: makenumbers_random8020.c en makenumbers_random_zipf_vary.c. Meer informatie over deze bestanden is te vinden in de het bestand info_makenumbers.txt, maar ook in de bronbestanden zelf.

Deze programma's genereren getallen met specifieke distributies die weggeschreven kunnen worden naar een binair bestand. Wij stellen de klasse NumberReader ter beschikking om de data in te lezen, maar indien gewenst mag de student hiervoor ook zelf code schrijven.

Gebruik je experimenten om de gekozen varianten van semi-splay met elkaar te vergelijken op basis van de statistieken die beschreven staan in de klasse AbstractTree. Bedenk en experimenteer met situaties waarin bepaalde varianten een voordeel kunnen behalen ten opzichte andere varianten.

Het is noodzakelijk voor programmacode om getest te worden op correctheid. Omdat software regelmatig gewijzigd wordt, moeten deze correctheidstests geautomatiseerd herhaald kunnen worden. Voor Java software zijn JUnit testen hierbij een handig hulpmiddel. Studenten van AD1 hebben hiermee reeds kennis gemaakt. Implementeer voor de klasse SemiSplay een JUnit test die zo veel mogelijk relevante functies test. Implementeer voor



Figuur 1: Visueel semi-splaybomen debuggen met het programma Toppie

elk van de geïmplementeerde semi-splay varianten een JUnit test die de correctheid van de specifieke splaytechniek vaststelt.

5 Theoretische vragen

Werk in je verslag ook volgende theoretische opgaven uit:

- In het project wordt een zoekboom geïmplementeerd aan de hand van de klasse

Top. Een top bevat naast de sleutel ook een pointer naar zijn linker- en rechterkind. Een andere mogelijke implementatie ontstaat door gebruik te maken van een array. Hierbij wordt de wortel opgeslagen op positie 0. Het linker- en rechterkind van de wortel worden opgeslagen op respectievelijk posities 1 en 2. Algemeen worden de kinderen van de top op positie i opgeslagen op posities $2i + 1$ en $2i + 2$.

Beschrijf de voor- en nadelen van beide implementaties van semi-splay bomen. Heeft het type implementatie een effect op de tijd- en geheugencomplexiteit? Zo ja, beschrijf dan de verschillen in tijd- en geheugencomplexiteit.

- In de cursus zien jullie dat de geamortiseerde complexiteit van semi-splay gemiddeld $\mathcal{O}(\log n)$ is per bewerking (stelling 10). Hierbij zijn de bewerkingen het toevoegen, opzoeken en verwijderen van een sleutel. In het project moeten jullie echter ook een `iterator` implementeren.

  Bepaal de geamortiseerde complexiteit van semi-splay waarbij er naast de bewerkingen uit de cursus ook de iteratie over een semi-splayboom is toegelaten. Merk op dat elke stap in de iteratie één bewerking is en bij de laatste stap (einde iteratie door bijvoorbeeld het `break` commando op te roepen) wordt gesplayd.

6 Verslag

Schrijf een verslag waarin je de opzet en parameters van de experimenten uiteenzet. Beschrijf jouw implementaties, waaronder mogelijke optimalisaties die je hebt doorgevoerd. Vergelijk de varianten die je gekozen hebt en bespreek de mogelijke voor- en nadelen van deze varianten. Geef de meetresultaten in overzichtelijke grafieken weer en bespreek ze grondig. Wat kun je op basis van de experimenten concluderen? Komen je resultaten overeen met je verwachtingen. Probeer steeds om al je resultaten te verklaren.

Voorzie je code van commentaar en vermijd duplicatie van code. Werk met overerving en licht je ontwerpbeslissingen toe. Houd je verslag beknopt, maar zorg er wel voor dat alles voldoende grondig besproken wordt.

Het is belangrijk om steeds de in de cursus gehanteerde wiskundige notatie te gebruiken. Gebruik dit project als aanleiding om de inhoud van de cursus te oefenen. Dat is een goede voorbereiding op het examen.

7 Beoordeling

De beoordeling zal gebaseerd zijn op de volgende onderdelen:

- De kwaliteiten van de programmacode. Is de implementatie volledig? Is zij correct? Spreekt er uit dat de student zelf heeft nagedacht en met goede oplossingen is gekomen? Zijn de testen zinvol?
- De opzet en uitvoering van de experimenten. Tonen de experimenten de verschillen in performantie tussen de varianten goed aan? Zijn de parameters voor de datasets goed gekozen?

- De verslaglegging. Is zij toegankelijk, duidelijk, helder, ter zake, objectief en beknopt? Is er een juist gebruik van wiskundige notatie? Worden de theoretische vragen goed beantwoord?

8 Indienen

Voor dit project gelden twee indiendata. Een eerste versie moet worden ingediend uiterlijk op de avond van zondag 2 november 2014 om 23 uur 59. Deze bevat tenminste een volledige implementatie voor de klasse SemiSplay met JUnit testen **en** een gemotiveerde keuze van de gekozen varianten. Deze varianten moeten voor de eerste indiendatum echter nog niet geïmplementeerd zijn. Eerder indienen mag ook. Herhaaldelijk indienen van verbeterde versies mag eveneens. Meer werk inleveren mag ook, hetgeen is aan te raden.

Het tweede en finale indienmoment is zondagavond 30 november 2014 om 23 uur 59. 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.

8.1 Elektronisch indienen

Op https://indianio.ugent.be/ kan elektronisch ingediend worden. Maak daartoe een zipbestand en hanteer daarbij de volgende structuur:

- src/semisplay/ bevat alle Java broncode voor semi-splay bomen en varianten, inclusief de ongewijzigde interfaces die gegeven werden. Behoud de package structuur en maak GEEN subpackages of subdirectories aan.

  De code voor Toppie is *niet* nodig, maar indien er interessante wijzigingen zijn gemaakt *mag* dit worden ingeleverd. Dit is niet verplicht.

- tests/ alle testcode.

- extra/verslag.pdf de elektronische versie van je verslag bij het **tweede** indienmoment. In de map extra kan je ook extra bijlagen plaatsen, indien gewenst.

8.2 Algemene richtlijnen

- Zorg ervoor dat je code volledig compileert met een Oracle Java 8 compiler update 20 of hoger. Extra libraries zijn niet toegestaan (je kan uiteraard wel gebruik maken van de standaard JDK 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 en correcte 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 versie betekent 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 is niet toegestaan om code uit te wisselen of over te nemen van Internet. Wij gebruiken geavanceerde plagiaatdetectiesoftware om ongewenste samenwerking te detecteren. Zowel het gebruik van andermans werk, als het delen van werk met anderen, zal als examenfraude worden beschouwd.

A Lijst met te ontvangen bestanden

De volgende bestanden moeten onveranderd geïmplementeerd worden:
AbstractTree.java,
Key.java,
Top.java,
TreeIterable.java,
TreeIterator.java.

Deze mogen naar eigen inzicht gebruikt, gewijzigd, of genegeerd worden:
AboutPopup.java,
Toppie.java,
ToppieFXML.fxml,
ToppieFXMLCompanion.java,
TopStack.java,
info_makenumbers.txt,
makenumbers_random8020.c,
makenumbers_random_zipf_vary.c,
Makefile,
NumberReader.java.