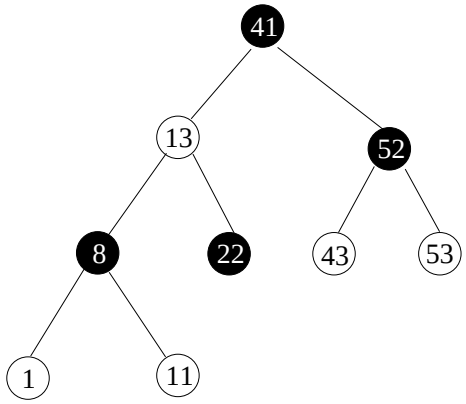
Examen Datastructuren en Algoritmen II

Naam : .....

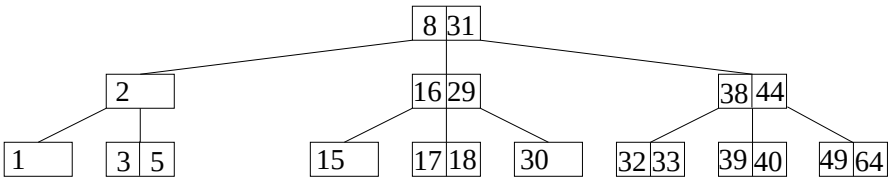
- Lees de hele oefening zorgvuldig voordat je begint ze op te lossen!
  Als je niet goed verstaat wat de vraag of taak is, vraag het aan de lesgever! Voor oefeningen die fout verstaan zijn, kunnen geen punten gegeven worden!
- Schrijf leesbaar. Oplossingen die niet leesbaar zijn, kunnen ook niet beoordeeld worden.
- Als technieken toegepast moeten worden, toon altijd voldoende tussenstappen om te kunnen zien wat er gebeurt en dat de technieken goed verstaan zijn.
- Stellingen uit de les mogen natuurlijk altijd gebruikt worden zonder dat het bewijs opnieuw gegeven moet worden (behalve in gevallen waar het expliciet anders staat)!
- Geef alleen dan een antwoord als je denkt dat je de oplossing kent. Verspil geen tijd met de poging gewoon lange teksten te schrijven waarin zekere sleutelwoorden opduiken – zoals dat vaak geprobeerd wordt. Dergelijke oplossingen halen nooit punten en voor bijzonder slechte oplossingen worden punten afgetrokken!

1. **Zoekbomen 2 pt**

- Voeg sleutel 54 toe aan de volgende rood-zwart boom. Kies zelf welke van de bewerkingen je kiest (de gewone of die, die het soms toelaten vroeger te stoppen).



- Voeg de sleutel 54 toe aan de volgende 2-3-boom.

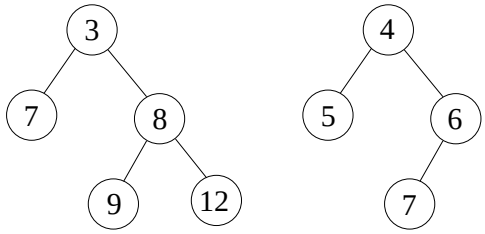


- In een binaire zoekboom T is een deelboom T' gegeven. Voor een top v in T geeft de functie `in_deelboom(v)` de waarde `true` terug als $v \in T'$ en anders `false`. Elke top heeft een pointer naar zijn kleinerdeelboom en naar zijn groterdeelboom. Het doel is nu een array `buitenbomen[]` in te vullen, waar alle pointers naar toppen die wortels van buitenbomen zijn (of NULL, als de buitenboom leeg is) in volgorde opgeslagen zijn.

  Geef de pseudocode van een recursieve functie die de array invult.

2. **Heaps 2.75 pt**

- Merge de volgende twee skew heaps door middel van de recursieve skew merge bewerking:



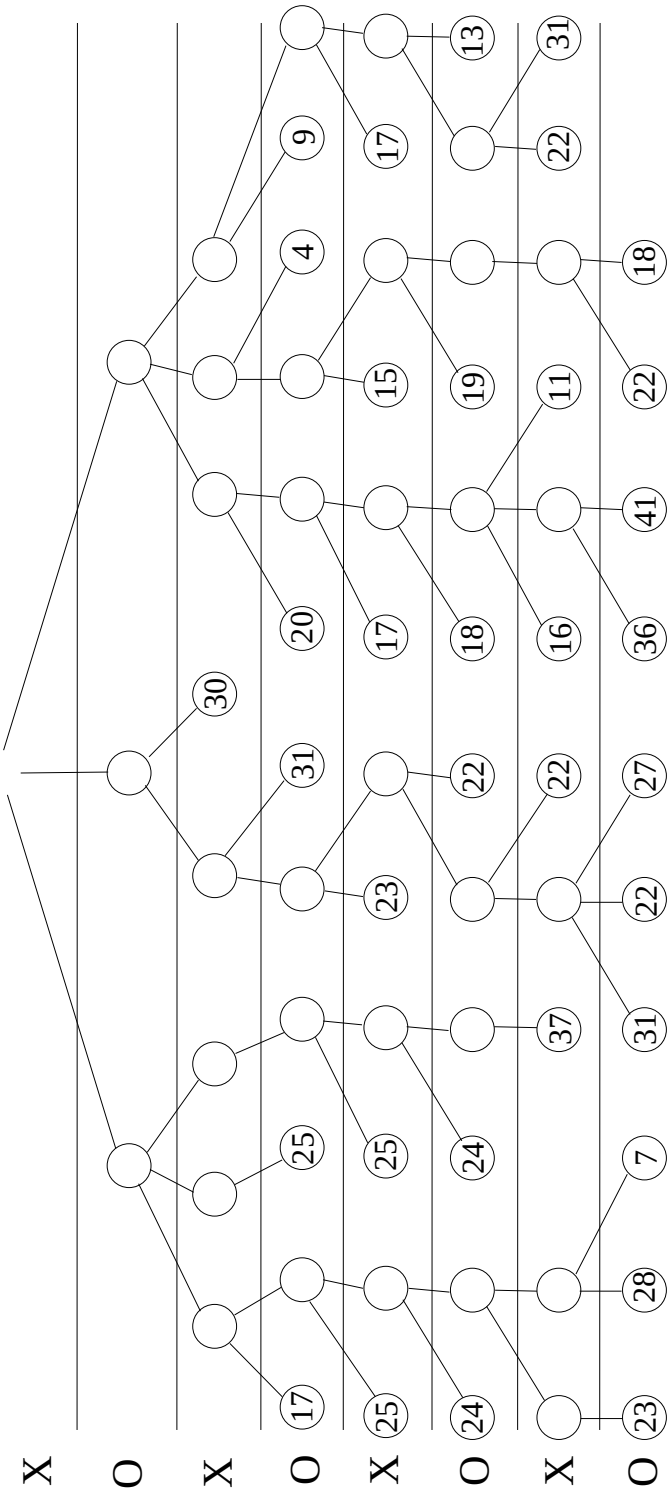
- Je hebt n bewerkingen op een initieel lege binomiale prioriteitswachtlijn. In elke bewerking wordt ofwel een element toegevoegd ofwel het kleinste element uit de wachtlijn verwijderd. Wat is de geamortiseerde kost van een bewerking in deze reeks van bewerkingen ($O()$-notatie)? Bewijs dat jouw antwoord juist is en dat jouw grens bestmogelijk is.

- Je hebt twee reeksen van sleutels $R_1 = a_1, a_2, \ldots, a_n$ en $R_2 = b_1, b_2, \ldots, b_n$. Stel dat t_1 de tijd voor het toevoegen van R_1 aan een initieel lege leftist heap is en t_2 de tijd voor het toevoegen van R_2 aan een initieel lege leftist heap.

  Wat is de maximale verhouding $\frac{t_1}{t_2}$ (op een constante na). Leg uit waarom jouw antwoord juist is. Geef bv. reeksen die bijzonder duur, resp. bijzonder goedkoop zijn en zeg waarom de bewerkingen niet goedkoper of duurder kunnen zijn.

3. **De waarde van een spel 1.5 pt**

- Pas α–β–snoeien op de volgende spelboom toe om de waarde van het spel te berekenen. Evalueer altijd eerst de linkertakken. Schrijf gebruikte grenzen **aan** de toppen en de teruggegeven waarden **in** de toppen. Plaats streepjes door de bogen die naar deelgrafen leiden die je niet evalueert omdat je snoeit.

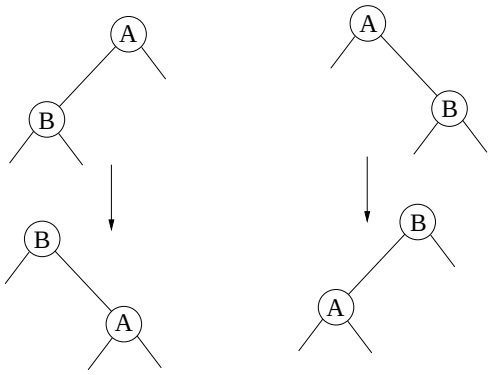


- Wij hebben in dit deel altijd geëist dat de winst van X gelijk is aan het verlies van O. Stel nu dat dat niet zo is, maar dat de bladeren gelabeld zijn met een 2-tal (x, o) waarbij x de winst van X is en o de winst van O. Dan zou een recursieve functie waar X op even diepten de tak met maximale x kiest en O op oneven diepten de tak met maximale o een 2-tal als waarde van het spel opleveren.

  Geef een redenering waarom iets gelijkaardigs aan α–β–snoeien niet zal werken.

4. **Geamortiseerde complexiteit 2.25 pt**

   Als de wortel van een vervangboom het kind van de wortel van de hele boom is dan stopt semi-splay. Stel nu, dat *alternatief semi-splay* dat niet doet. Het algoritme werkt net zoals semi-splay, behalve in het geval dat de wortel van de vervangboom een kind van de wortel van de boom is – dan wordt de deelboom bestaande uit de wortel van de vervangboom (B in de volgende afbeelding) en de wortel van de boom (A in de volgende afbeelding) gewisseld:



   - Wat is de maximale kost van een bewerking in een reeks van *n* bewerkingen op een alternatief semi-splay boom (O()-notatie? Bewijs dat jouw antwoord juist is.

- Wat is de geamortiseerde kost van een bewerking in een reeks van n bewerkingen op een alternatief semi-splay boom (O()-notatie?

  Bewijs dat jouw antwoord juist is. Alle stellingen uit de les en ook de deelresultaten 1,2,3 uit het bewijs van de geamortiseerde complexiteit van semi-splay waar het bewijs ook voor alternatief semi-splay geldig is, mogen gewoon geciteerd en gebruikt worden. Ook delen uit deelresultaat 4 (dat voor alternatief semi-splay niet geldig is) mogen gebruikt worden als nauwkeurig beschreven is wat je gebruikt.

5. **Dynamisch programmeren 1.5 pt**

   Gegeven is een Nederlandse tekst, die wel blanks bevat, maar geen linefeeds. Die moet nu geformateerd worden, zodat elke lijn ten hoogste m tekens bevat, waarbij wij m als een constante beschouwen. Het linefeed-teken wordt bij de m tekens meegerekend. Het mag verondersteld worden dat er geen deelstrings zonder blanks zijn die langer dan $m-1$ zijn. Daarbij is het (om het eenvoudig te houden) alleen toegelaten blanks door linefeeds te vervangen – scheidingstekens mogen dus niet gebruikt worden. Het doel is niet alleen weinig lijnen te hebben, maar de lijnen moeten ook zover mogelijk ongeveer gelijke lengte hebben (op de laatste lijn na). Om dat doel te benaderen definiëren wij de fout van een lijn als volgt:

   - de fout van de laatste lijn is 0.
   - als een lijn n tekens van de tekst bevat (het linefeed-teken meegerekend) en niet de laatste lijn is, dan is de fout $(m-n)^2$.

   De fout van de formatering is nu de som van de fouten van de lijnen.

   Geef een algoritme dat een formatering met minimale fout in tijd $O(n)$ berekent als n de lengte van de tekst is.

6. **Online algoritmen 1.75 pt**

- Wij willen een online algoritme voor het inpakprobleem ontwikkelen dat iets zwakkere eisen heeft dan de online algoritmen uit de les: je hoeft niet elk pakket onmiddellijk te plaatsen, maar je wacht altijd totdat je drie pakketten hebt en (behalve als er op het einde geen drie pakketten meer in de rij zijn) en pas dan beslis je hoe deze pakketten het best geplaatst worden. Doordat je meer informatie hebt op het moment dat je de pakketten moet plaatsen, kan je de pakketten dan soms beter plaatsen. Wij noemen een dergelijk algoritme hier een *zwak online inpakalgoritme*.

  Bewijs: Voor elk zwak online inpakalgoritme A en elke $k \in \mathbb{N}$ bestaat er een rij van gewichten die minimaal $m \geq k$ vrachtwagens vraagt en waarvoor A ten minste $\frac{4}{3}m$ vrachtwagens vraagt. Geef het bewijs – schets niet alleen de verschillen met het bewijs voor online inpakalgoritmen.

- Geef een voorbeeld waar de first-fit online heuristiek beter presteert dan de best-fit dalend offline heuristiek en een voorbeeld waar best-fit dalend beter presteert dan first-fit. Pas de algoritmen elke keer ook toe.

7. **Gretige algoritmen 1 pt**

In deze oefening gaat het om rechthoekige platen ijzer. Voor een plaat met randlengten x mm en y mm schrijven wij (x,y)-plaat en noemen wij (x,y) de dimensie van de plaat. Je kan een plaat doorzagen, maar dat moet je dan altijd helemaal doen en de snede zelf is 1 mm breed. Je kan dus bv een plaat van dimensie $(100,50)$ doorzagen om twee platen te krijgen met dimensies $(50,50)$ en $(49,50)$.

Je hebt een grote rechthoekige (x,y)-plaat en een lijst $(x_1,y_1),\ldots,(x_k,y_k)$ van dimensies. Voor elke dimensie in de lijst wil je één plaat hebben. Het doel is nu zo veel mogelijk van de grote plaat te gebruiken om de gewenste platen te krijgen. De winst is de som van de oppervlakten van de gesneden (nuttige) platen.

Geef een gretig algoritme voor dit probleem (pseudocode) en leg uit waarom het volgens jou *gretig* is. Je hoeft geen analyse te doen hoe goed de resultaten in vergelijking met optimale oplossingen zijn.

8. **Verzamelingen 2 pt**

Je werkt op een 64-bit computer met het universum $U = \{0, \ldots, 60\}$. Als je dan verzamelingen als bitvectoren voorstelt, is 1 computerwoord voldoende.
Veronderstel dat M, M' verzamelingen zijn en $0 \leq i \leq 60$. Het doel is nu elke keer **efficiënte** operaties te geven om een verzameling Y te krijgen die zekere eigenschappen heeft.

Voorbeeld: Y bevat precies element i als die ook in M zit en is anders leeg.
Oplossing: `Y= M & (1<<i)`

- Y bevat precies element i en alle elementen uit M.

- Y bevat precies de elementen uit M die groter zijn dan i

- Y bevat precies de elementen uit U die noch in M noch in M' zitten.

- Y bevat precies de elementen die in exact één van M, M' zitten.

9. **Gerandomiseerde algoritmen 1.25 pt**

Je zoekt in een graaf met 200 toppen $v_1, \ldots, v_{200}$ een toppenverzameling met 21 toppen, die een zekere eigenschap heeft (bv. dat de afstand tot toppen die niet in de verzameling zijn ten hoogste 2 is, of dat er een cykel door alle toppen gaat, of…). Uit structuureigenschappen van de graaf weet je al dat 5% van alle toppenverzamelingen met 21 toppen eraan voldoen. Het kost je 1 *ms* om te toetsen of een gegeven verzameling van 21 toppen deze eigenschap heeft.

Typisch voor dit soort problemen zijn branch and bound algoritmen, waarbij de bounding criteria vaak duur zijn en niet zeker kunnen zeggen of een deeloplossing uitgebreid kan worden. Maar je hebt geluk: hier heb je wel een heel goed bounding criterium: je kan ook in 1 *ms* voor elke verzameling van $k \leq 21$ toppen testen of die uitgebreid kan worden tot een verzameling met 21 toppen en de gezochte eigenschap. Deze test noemen wij `uitbreidtest()`. In het geval $k = 21$ toetst de test gewoon of de verzameling voldoet.

Veronderstel dat de tijd voor alle andere bewerkingen van jouw algoritme verwaarloosbaar is in vergelijking met de testen.

De volgende pseudocode zou een branch-and-bound algoritme zijn als je hem met $(\emptyset, 1)$ opstart:

```
find_opl(verzameling,next)
{

if (grootte(verzameling)=21) { output(verzameling); exit(); }

if (uitbreidtest(verzameling + v_(next)))
       find_opl(verzameling + v_(next), next+1);
else find_opl(verzameling, next+1);

  }
}
```

Door wat je al weet, weet je ook dat `uitbreidtest()` nooit voor `next` groter dan 200 wordt gebruikt – dat hoef je dus niet te toetsen.

- Geef de pseudocode voor een Las Vegas algoritme voor dit probleem.

- Analyseer welk algoritme vermoedelijk sneller een oplossing gevonden heeft – jouw algoritme of het branch and bound algoritme. (Tip: $0.95^{10} \approx 0.6$).

NOG NIET OMDRAAIEN !