Uitwerking tentamen Algoritmiek

9 juni 2015 14:00 – 17:00

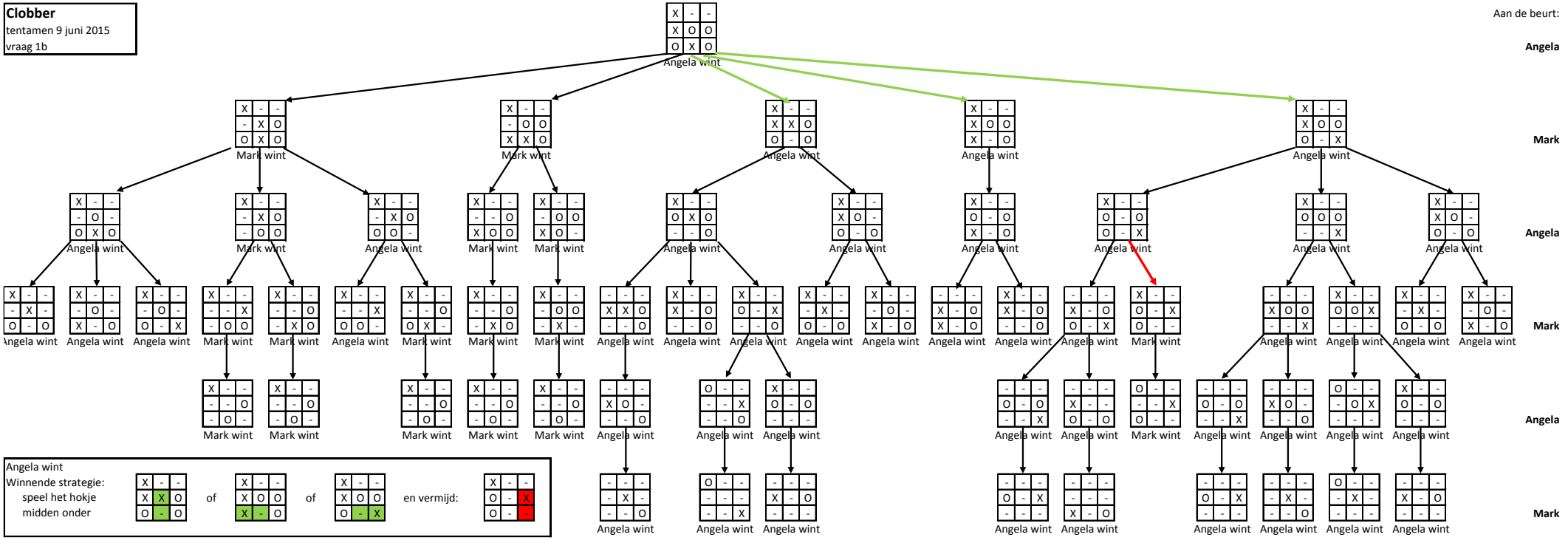
1. Clobber

a. Toestanden: *m* x *n* bord met in elk hokje een O, een X of een -. Hierbij is het aantal O gelijk aan het aantal X of er is hooguit één verschil. Aangegeven is per speelbord welke speler aan de beurt is. Dit kan evt. ook berekend worden uit het aantal lege hokjes. Is dit even dan is X aan de beurt, anders is O aan de beurt.

Acties: Kies een hokje dat een eigen steen bevat en waar in minimaal één horizontaal of verticaal aangrenzend hokje een steen van de tegenstander ligt. Sla één van deze stenen van de tegenstander door deze te vervangen door de eigen steen en het gekozen hokje leeg te maken (-).

b. Winnende strategie voor Angela:

- Angela kiest als eerste zet één van de zetten behorende bij de groene pijlen.
- In haar tweede beurt mag Angela de zet behorende bij de rode pijl niet zetten.



N.B. In deze toestand-actie-ruimte zijn de toestanden waarvan de laatste zetten vastliggen wel uitgewerkt. Dit mocht achterwege gelaten worden.

2. Red-Black Tree

a.
```
bool roodzwart( knoop* wortel )
{
  if ( wortel == NULL )
    return true;
  else
  {
    if ( wortel->kleur == 'r' )
      if ( ( wortel->links != NULL
             && wortel->links->kleur != 'z')
         || ( wortel->rechts != NULL
             && wortel->rechts->kleur != 'z') )
        return false;
    }
    return (  roodzwart( wortel->links )
           && roodzwart( wortel->rechts ) );
  }
}
```

// er is een knoop

// lege boom is red-black tree

// rode knoop
// check eigenschap 1:
// de kinderen van een rode knoop
// zijn zwart

// wanneer eigenschap 1 niet klopt: false

// check recursief of linker en rechter
// kind beide een red-black tree zijn.

b.
```
void black( knoop* wortel, int noir )
{
  if ( wortel != NULL )
  {
    if ( wortel->kleur == 'z' )
      noir++;
    wortel->zwart = noir;

    black( wortel->links, noir );
    black( wortel->rechts, noir );
}
```

// er is een knoop

// bij zwarte knoop noir verhogen

// noir opslaan in wortel

// recursief linker en rechter subboom
// bezoeken

c.
```
void herstel( knoop* wortel )
{
  bool naarlinks = TRUE;
  knoop* vorige = NULL;
  int z = 0;
  while (wortel != NULL )
  {
    vorige = wortel;
    if (wortel->kleur == 'z')
      z++;
    if ( naarlinks )
      wortel = wortel->links;
    else
      wortel = wortel->rechts;
    naarlinks = ! naarlinks;
  }
  if ( vorige != NULL )
    vorige->zwart = z;
}
```

// toggle voor het zigzaggen
// laatst bezochtte knoop
// zwart teller
// loop naar de onderkant van de boom
// (als wortel == NULL => vorige is het blad
// laatst bezochtte knoop onthouden
// tel zwartte knopen

// zig of zag naar kind

// verander toggle

// herstel zwartwaarde

3. Stijgende deelrij

a.
```
int stijgend( int A[], int n, int k  )
{
  int tel = 1;                                  // teller  voor het aantal stijgende elementen
  for ( int i = 0; i < n - 1; i++ )             // loop alle elementen langs. Let op i.v.m. vergelijken
  {
    if ( A[i] < A[i+1] )                        // vergelijk 2 naast elkaar liggende elementen
    {
      tel ++;                                   // als oplopend
      if ( tel >= k )                           // tellen
        return i – tel;                         // als ik k oplopende heb
    }                                           // deelrij gevonden
    else
      tel = 1;                                  // niet oplopend
  }                                             // start opnieuw met tellen
  return -1;
}                                               // geen deelrij gevonden
```

b.
```
int oplopend( int  A[], int k, int i )          // recursie van achteraf van het array
{
  if ( i < k-1 )                                // als er minder dan k elementen zijn t/m i
    return -1;                                  // geen deelrij van k elementen

  if ( A[i] < A[i-1] )                          // deze if kan weg gelaten worden, want wordt
    return oplopend( A, k, i - 1 );             // automatisch goed afgehandeld door de loop die nu
  else                                          // bij else staat.
  {                                             // het is echter wel de decrease-by-one stap
    int tel = k;
    while ( ( A[i] > A[i-1] )                   // kijk of er een deelrij is van k elementen
              && ( tel > 1 )
              && ( i > 0 ) )
    {
      tel --;
      i --;
    }

    if ( tel == 1 )                             // als als tel == 1 dan is er een deelrij van
      return i;                                 // k elementen gevonden vanaf 1 en verder
    else                                        // als niet gevonden
      return oplopend( A, k, i - 1 );           // check recursief vanaf i-1 terug.
  }                                             // want dan was A[i] < A[i-1]
}
```

```
int deelrij( int A[], int k, int links, int rechts )
    {                                                   // deelrij van 1 element
      if ( k <= 1 )                                     // is altijd oplopend
        return links;

                                                        // tebekijken deelrij
      if ( rechts - links + 1 < k )                     // bevat minder dan k elementen
        return -1;

                                                        // het midden bepalen
      int m = ( links + rechts ) / 2;

                                                        // onderzoek de linker deelrij
      int pos1 = deelrij( A, k, links, m );             // deelrij gevonden => klaar
      if ( pos1 >= 0 )
        return pos1;

                                                        // onderzoek de rechter deelrij
      int pos2 = deelrij( A, k, m + 1, rechts );        // deelrij gevonden => klaar
      if ( pos2 >= 0 )
        return pos2;

                                                        // Als A[m] > A[m+1], dan is er
      if ( A[m] > A[m+1] )                              // geen stijgende deelrij die
        return -1;                                      // A[m] én A[m+1] bevat.
                                                        // bepaal het deel van A van links t/m
      if ( m – k + 2 > links )                          // rechts waar een deelrij van
        links = m – k + 2;                              // k elementen kan zitten omdat
      if (m + k – 1 < rechts)                           // de deelrij A[m] én A[m+1] bevat
        rechts = m + k – 1;

                                                        // gebruik het algoritme van vraag a
      int tel = 1;                                      // om te kijken of er van links t/m rechts
      for ( int i = links; i < rechts; i++ )            // een stijgende deelrij van k elementen
      {                                                 // bevat
        if ( A[i] < A[i+1] )
        {
          tel ++;
          if ( tel >= k )                               // deelrij gevonden
            return i – tel;
        }
        else                                            // geen deelrij
          tel = 1;                                      // start opnieuw met tellen
      }
      return -1;                                        // geen deelrij gevonden

    }
```

Bonus

worst case algoritme uit a: n-1.
Als er een stijgende deelrij is en deze bevindt zich in A[n-k] t/m A[n-1]. Of als er een stijgende deelrij van k-1 elementen is in A[n-k] t/m A[n-2], maar A[n-1] is kleiner dan A[n-1].

best case algoritme uit a: k-1.
Als er een stijgende deelrij is en deze bevindt zich in A[0] t/m A[k-1]. Na k-1 vergelijkingen is tel gelijk aan k en stopt het algoritme.

4. Rondreis (Hamiltonkring)

a. Bij best-fit-first branch and bound bouwen we oplossingen stap voor stap op.

Bij iedere deeloplossing die we genereren, berekenen we onmiddelijk een ondergrens (het is een minimalisatie probleem) voor de waarde die een complete oplossing kan hebben die uit deze deeloplossing voortkomt (**bound**).

Tijdens het algoritme pakken we steeds de deeloplossing met de hoogste ondergrens (**best-fit-first**) en werken die één stap verder uit. Dat wil zeggen: we breiden de deeloplossing op alle mogelijke manieren één stap uit naar grotere deeloplossingen (**branch**).

Bij alle nieuwe deeloplossingen berekenen we natuurlijk weer een ondergrens.

We breiden een deeloplossing niet verder uit:

- Als we zien dat er geen geldige complete oplossing uit voort kan komen (de deeloplossing is **infeasable**, **snoeien**).
- Als er nog precies één complete oplossing uit voort kan komen. In dat geval construeren we deze complete oplossing, berekenen haar echte waarde (geen ondergrens) en als deze waarde hoger is dan de hoogste tot nu toe gevonden echte waarde, onthouden we die.
- Als de ondergrens van de deeloplossing tot nu toe ≤ hoogste tot nu toe gevonden echte waarde. Dan kan de deeloplosing geen hogere waarde meer opleveren (**snoeien**).

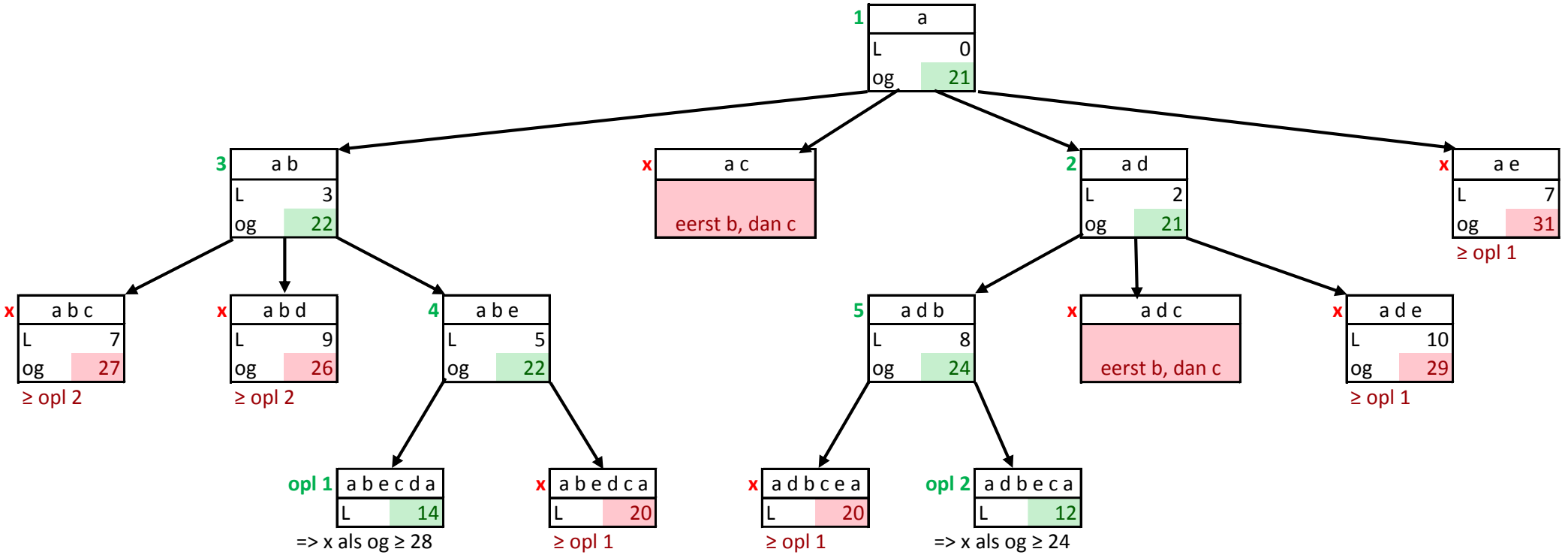
b. Ondergrens:

De som van 2 takken per knoop. Hiervoor worden per knoop de al gekozen takken genomen, aangevuld met de kleinste uitgaande takken. Dit is de ondergrens voor de dubbele lengte omdat er dubbel zoveel takken worden meergerekend dan nodig zijn in de totale oplossing. Elke knoop levert wel twee takken, maar van de uiteindelijke oplossing behoort elke gekozen tak toe aan twee knopen.

Algoritme:

- Start in knoop a met het pad bestaande uit alleen a en lengte 0.
- Bepaal de ondergrans door per knoop de kleinste 2 gewichten van uitgaande takken op te tellen.
- Itereer zolang er nog geschikte deeloplossingen zijn:
  - Kies de deeloplossing met de laagste ondergrens.
  - Creëer nieuwe deeloplossingen door aan de huidige deeloplossing vanuit de laatst toegevoegde knoop een nieuwe knoop toe te voegen die bereikbaar is vanuit deze knoop en die nog niet eerder is gekozen. Oplossingen waarbij b niet meer voor c kan worden gekozen worden afgekeurd. Deze oplossingen zijn het spiegelbeeld van een andere oplossing waarbij b wel voor c gekozen kan worden. Dit levert een versnelling op waarbij slechts de helft van alle mogelijke deeloplossingen wordt bekeken.
  - Bepaal de lengte van het pad van de deeloplossing.
  - Bepaal de ondergrens zoals boven beschreven is.
  - Als een oplossing is gevonden en dit is de eerste oplossing, of deze oplossing is beter dan de beste oplossing tot nu toe:
    - Dit is nu de beste oplossing tot nu toe.
    - Verwijder alle deeloplossingen waar van de ondergrens groter of gelijk is aan 2x de lengte van de gevonden oplossing. (Deze kunnen niet leiden tot een betere oplossing).

c. State space tree



Optimale Hamiltonkring: a d b e c a met lengte 12

5. Heapify

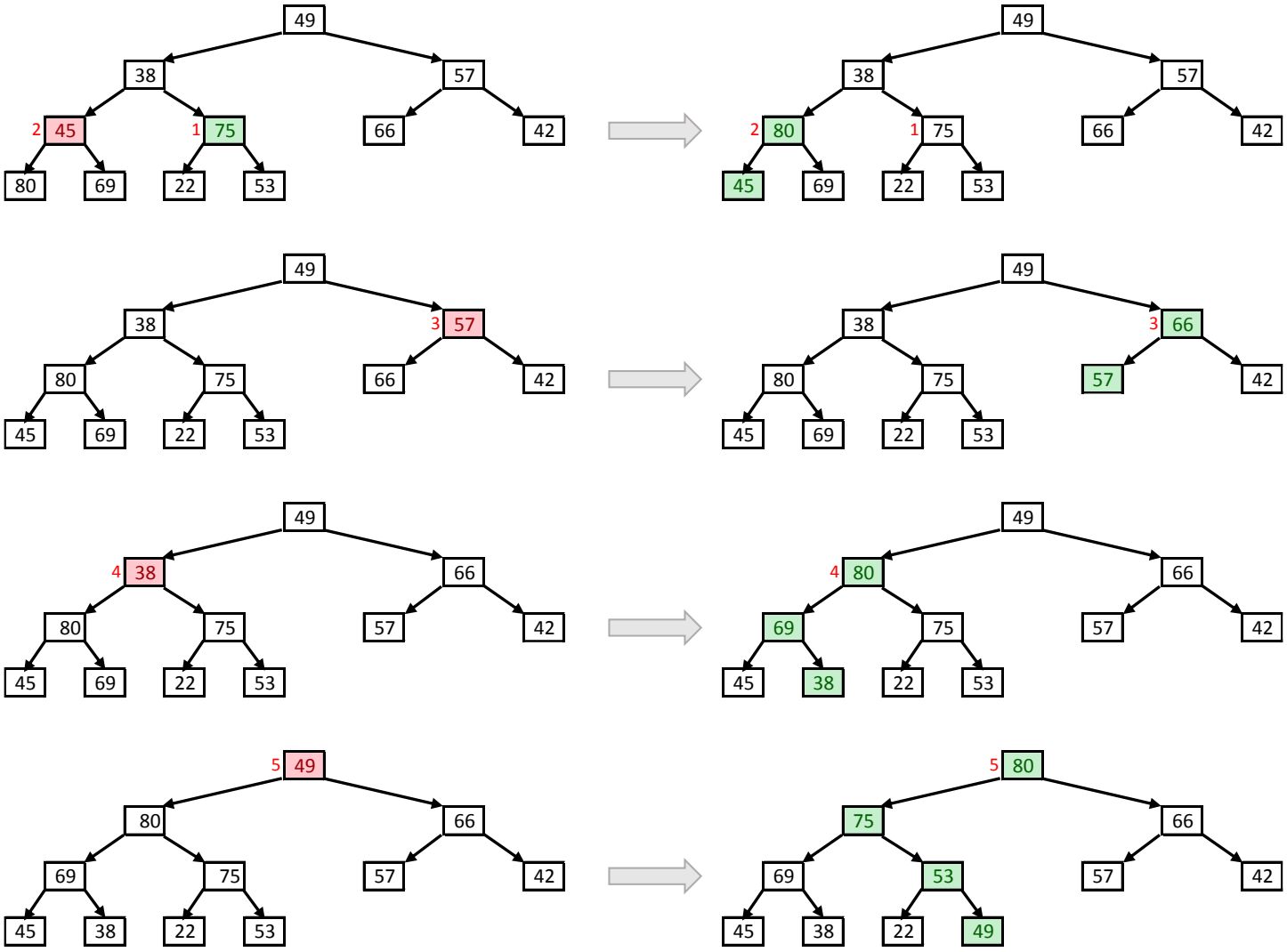
a. Een binaire boom is een heap als (1) de binaire boom compleet is en (2) als voor elke knoop geldt dat zijn waarde groter of gelijk is aan die van zijn kinderen.

Een complete binaire boom is binaire boom waarvan alle nivo's geheel gevuld zijn, behalve eventueel het onderste. Op het onderste nivo mogen alleen de meest rechter knopen missen.

b. De grootste waarde bevindt zich in de wortel van de boom.
De kleinste waarde bevindt zich in één van de bladeren.

c. Heapify

49	38	57	45	75	66	42	80	69	22	53



80	75	66	69	53	57	42	45	38	22	49

1 volgorde waarin de knopen worden bezocht
45 knoop die niet aan de heapeigenschap voldoet
45 knoop na correctie heapeigenschap