UNIVERSITEIT VAN AMSTERDAM

Toets Tentamen datastructuren 2020-2021

Vraag 1 − ADTs − 212697.2.0

Selecteer hieronder alle abstracte datatypes.

- **A** List
- **B** Queue
- **C** Binary tree
- **D** Binary heap
- **E** Priority queue
- **F** Map
- **G** Hash table
- **H** Suffix tree
- **I** Stack

Vraag 2 − Big O − 165363.2.0

Welke van de volgende functies zijn in $O(\log n)$?

- **A** $\log\left(n^2\right)$
- **B** $\sqrt{n}$
- **C** 1
- **D** $(\log n)^2$
- **E** $n^{0.0001}$
- **F** $(5 + 31/n)^2 \log(n)$

Vraag 3 − Big Omega − 165365.2.0

Welke van de volgende functies zijn in $\Omega(n)$?

A n^2

B $n/1000$

C $n - \sqrt{n}$

D $\frac{n}{\log(n)}$

E $\frac{n^{1.0001}}{\log(n)}$

F 2^n

Vraag 4 − Bijection − 212787.2.0

We introduceren een nieuw abstract datatype: "Bijection". Een Bijection kan paren van keys opslaan, en gegeven de eerste key de bijbehorende tweede vinden, en vice versa. De operaties die horen bij dit ADT zijn:

```
// in: a pair (key1, key2) of keys
// action: stores the keys in the bijection
void insert(Key1Type key1, Key2Type key2);

// in:  the first key
// out: a pointer to the corresponding second key as stored in the Bijection
//      (or NULL if the first key is not found)
Key2Type* lookup_1to2 (Key1Type key1);

// in:  the second key
// out: a pointer to the corresponding first key as stored in the Bijection
//      (or NULL if the second key is not found)
Key1Type* lookup_2to1 (Key2Type key2);
```

Vraag: hoe zou je dit ADT efficiënt kunnen implementeren, en wat wordt dan de worst case tijdcomplexiteit van de drie gegeven operaties (in grote O notatie)? Gebruik natuurlijke taal of pseudocode en houd het zo kort als je kan.

Beoordelingsvoorschrift

Criterion 1 (Aantal punten: 1)

Er wordt een werkende Bijection beschreven (als duidelijk is dat de student een werkend idee heeft wordt het punt toegekend, ook bij kleine foutjes in de beschrijving)

Criterion 2 (Aantal punten: 1)

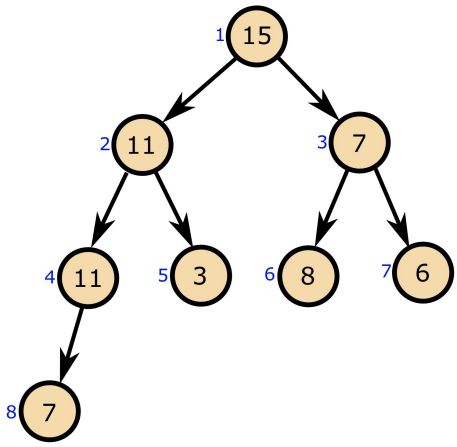
De implementatie is efficiënt: alle operaties zijn O(1)

Criterion 3 (Aantal punten: 1)

De student geeft de juiste tijdcomplexiteit voor zijn/haar operaties. Dit punt wordt ook toegekend als de operaties inefficiënt zijn maar de complexiteiten correct.

Vraag 5 − Binary heap − 165502.3.0

De afbeelding hieronder toont een binaire boom die we willen gebruiken als een max-heap, maar hij voldoet niet helemaal aan de heap eigenschap. De prioriteiten van de keys zijn weergegeven in de knopen; de indices van de knopen zijn in blauw aangeduid.



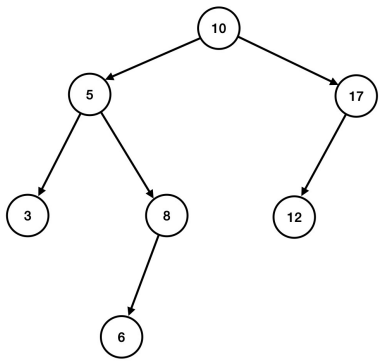
1. Wat is de index van de root van de kleinste subboom die niet aan de heap eigenschap voldoet? 3 .
2. Het is mogelijk om de heap eigenschap te herstellen met een enkele aanroep van `percolate_down`. Met welke index moet je `percolate_down` aanroepen? 3
3. Het is mogelijk om de heap eigenschap te herstellen met een enkele aanroep van `percolate_up`. Met welke index moet je `percolate_up` aanroepen? 6
4. Nadat we de heap eigenschap hebben hersteld, voegen we de waarde 13 in door die op index 9 van de boom (dus als rechter kind van index 4) toe te voegen, en daarna `percolate_up` aan te roepen op index 9. Wat is daarna de prioriteit van de key op index 9? 11
5. Vergeet nu even deze specifieke heap en denk aan een hele grote denkbeeldige heap. Wat zou de index zijn van het rechterkind van de knoop met index 1235? 2471 En wat zou de index zijn van de parent van 1235? 617

[Numeriek]	[Numeriek]	[Numeriek]	[Numeriek]	[Numeriek]
3	**3**	**6**	**11**	**2471**

[Numeriek]
617

Vraag 6 − Boom − 212120.3.0

Wat kun je zeggen over onderstaande boom?



A Het is een binaire boom

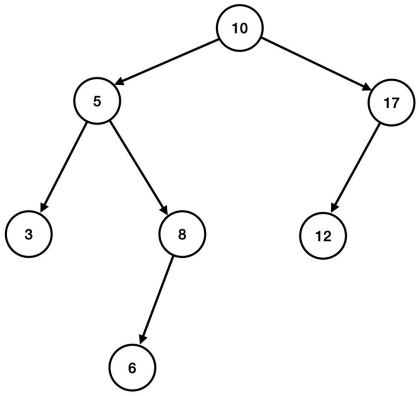
B Het is een binaire zoekboom die voldoet aan de AVL-eigenschap

C Het is een binaire heap

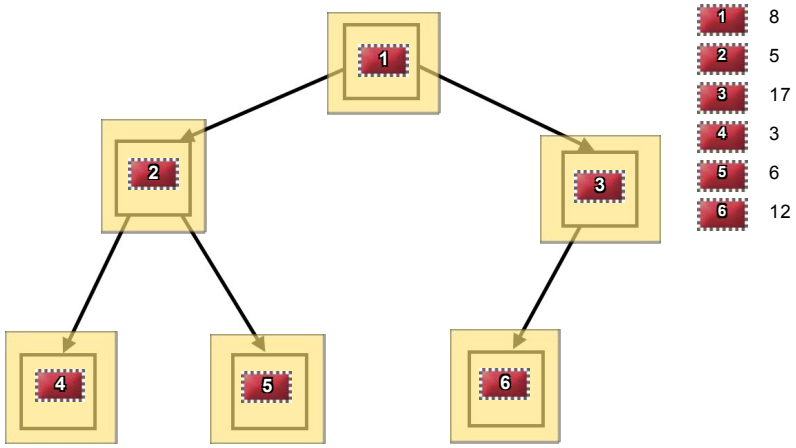
D De boom is perfect gebalanceerd

Vraag 7 − BST delete − 212675.5.0

In de hieronder getekende binaire zoekboom wordt de root verwijderd. In deze implementatie gebeurt verwijdering van een knoop door hem te vervangen door de meest rechter knoop in de linker subboom.



Geef aan welke keys de knopen bevatten na verwijdering van de root, door de juiste key naar de juiste knoop te slepen.



Vraag 8 − Circulair gelinkte lijst − 213487.3.0

Een circulaire gelinkte lijst lijkt op een enkel gelinkte lijst:

```
struct ListNode {
  KeyType          key;
  struct ListNode* next;
};
```

alleen wijst de next pointer van het laatste element terug naar het allereerste element van de lijst.

Bij het onderstaande mag je ervanuit gaan dat de circulair gelinkte lijst niet leeg is. Hint: het is bij deze vraag misschien een goed idee om plaatjes te tekenen van de situatie.

Welke uitspraken zijn waar?

A Circulaire gelinkte lijsten worden vaak gebruikt om een Queue te implementeren. Je benadert dan de circulair gelinkte lijst altijd via een pointer naar het *oudste* (langst geleden toegevoegde) element.

Als de next pointer van de oudste ListNode wijst naar de nieuwste, en de next pointers van alle andere list nodes wijzen naar een oudere ListNode, dan kunnen zowel enqueue als dequeue in O(1) worden geïmplementeerd.

B Circulaire gelinkte lijsten worden vaak gebruikt om een Queue te implementeren. Je benadert dan de circulair gelinkte lijst altijd via een pointer naar het *nieuwste* (meest recentelijk toegevoegde) element.

Als de next pointer van de nieuwste ListNode wijst naar de oudste, en de next pointers van alle andere list nodes wijzen naar een nieuwere ListNode, dan kunnen zowel enqueue als dequeue in O(1) worden geïmplementeerd.

C Je kunt met deze datastructuur, gegeven enkel een pointer naar het laatste element van de lijst, zowel elementen aan het begin als aan het eind van de lijst toevoegen in O(1).

D Je kunt met deze datastructuur, gegeven enkel een pointer naar het laatste element van de lijst, zowel elementen aan het begin als aan het eind van de lijst verwijderen in O(1).

Vraag 9 − Doubly linked list verification − 165567.2.0

De code hieronder voegt een nieuwe knoop toe aan een dubbel gelinkte lijst. Is de code correct?

```
struct ListNode {
  KeyType          item;
  struct ListNode* next;
  struct ListNode* prev;
};

void dll_insert_after(struct ListNode* node, KeyType item) {
  struct ListNode* new_node = dll_allocate_new_node();
  new_node->item = item;

  // attach new node to existing nodes
  new_node->next = node->next;
  new_node->prev = node->prev;

  // attach existing nodes to the new node
  if (node->next != NULL) { node->next->prev = new_node; }
  if (node->prev != NULL) { node->prev->next = new_node; }
}
```

A De implementatie is correct.

B Deze implementatie verwijdert `node` uit the lijst, en veroorzaakt daarmee een memory leak.

C Deze implementatie voegt `new_node` toe *voor* `node` in plaats van erna.

D Deze functie is voor een enkel gelinkte lijst, niet voor een dubbel gelinkte lijst.

Vraag 10 − Hash code functie − 212759.2.1

We willen een Set maken met als keys linked lists (met integer waardes erin). De keys van de Set zijn dus zelf weer lijsten!

We doen dit met een hash table implementatie van het Set ADT. Dit betekent dat we dus een hash functie moeten maken voor linked lists. Hieronder zijn een aantal mogelijke implementaties. Orden de implementaties van goed (in de zin dat willekeurige keys goed worden gespreid over de buckets) naar slecht.

1

```
unsigned hash(ListNode* N) {
  if (N==NULL) { return 0; }
  return 213741 * hash(N->next) + N->data;
}
```

2

```
unsigned hash(ListNode* N) {
  unsigned res = 0u;
  for (int i=0; i<10; i++) {
    if (N==NULL) { break; }
    res = 213741 * res + N->data;
    N = N->next;
  }
  return res;
}
```

3

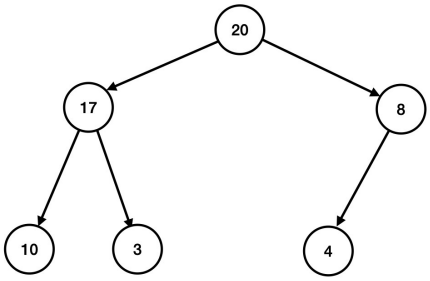
```
unsigned hash(ListNode* N) {
  return N==NULL ? 0 : N->data;
}
```

4

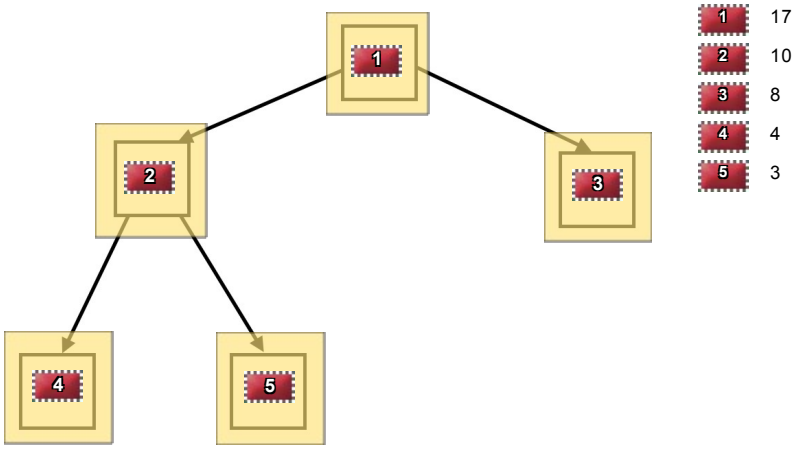
```
unsigned hash(ListNode *N) {
  return 213741;
}
```

Vraag 11 − Heap dequeue − 212682.3.0

In de onderstaande heap wordt een dequeue operatie uitgevoerd.



Geef hier onder de nieuwe toestand van de heap aan door de priorities van de keys naar de juiste vakjes te slepen.



1 17
2 10
3 8
4 4
5 3

Vraag 12 − Hoogte van boom − 213488.2.0

Wat kun je zeggen over de structuur van een binaire zoekboom met 12 knopen?

De boom heeft maximaal hoogte [11]

De boom heeft minimaal hoogte [3]

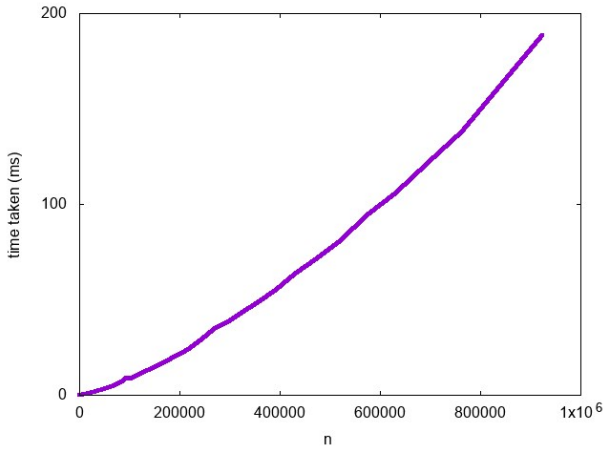
Als je weet dat de boom aan de AVL eigenschap voldoet, dan heeft hij maximaal hoogte [4]

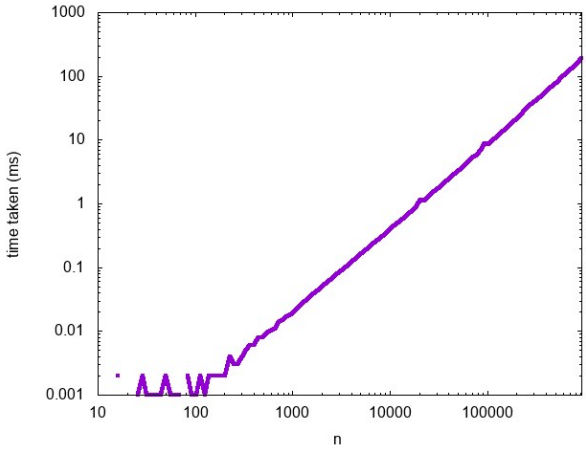
[Alfanumeriek]	[Alfanumeriek]	[Alfanumeriek]
11	**3**	**4**

Vraag 13 − Measuring run time − 165589.2.0

Hieronder zie je metingen aan een algoritme dat de eerste n priemgetallen genereert. De linkergrafiek toont een normale plot, de rechtergrafiek toont dezelfde resultaten op een dubbel logaritmische schaal.

Voor n=1000 is de gebruikte tijd 0.023ms. Voor n=1000000 is de gebruikte tijd 207ms. Geef aan welke aannames consistent zijn met deze gegevens.





A De gebruikte tijd is een exponentiële functie van n.

B De gebruikte tijd is een polynomiale functie van n.

C De gebruikte tijd is in $\Theta(n^2)$.

D De gebruikte tijd is $O(n^2)$.

Vraag 14 − Miscellaneous questions − 165590.2.1

Geef aan welke van de volgende uitspraken waar zijn.

A Een goed geimplementeerde binaire heap is een perfect gebalanceerde binaire boom. (Perfect gebalanceerd in de zin dat de diepte van elke twee bladeren ten hoogste 1 uit elkaar ligt.)

B Als een binaire boom minstens drie knopen heeft, en de keys in de knopen zijn allemaal verschillend, dan kan de boom een binaire zoekboom zijn, of een binaire heap, maar onmogelijk allebei tegelijk.

C Stel we slaan een verzameling coordinaten op in een hashtable, waarbij de hash code wordt gedefinieerd door de volgende functie: `unsigned hashcode(int x, int y) { return (unsigned)(x+y); }`.

Dan zullen reeksen van coordinaten die diagonale lijnen vormen zoals `{ (5,2), (6,1), (7,0), (8,-1), ... }` bijzonder efficiënt in de hashtable worden opgeslagen.

D $\Theta(f) = \Omega(f) \cap O(f)$.

Vraag 15 − Open addressing hash − 165477.6.0

We gebruiken een open addressing hashtable om gehele getallen op te slaan. De hashtable heeft 11 buckets (met indices van 0 tot en met 10), en we gebruiken quadratic probing. De hashcode is gelijk aan de integer die wordt ingevoegd zelf: bijvoorbeeld, als we het getal 23 toevoegen, dan is de eerste bucket die wordt uitgeprobeerd op index hashcode(23) mod 11, en dat is dus gelijk aan 23 mod 11, met andere woorden, 1.

Daarna wordt quadratic probing gebruikt, dat wil zeggen, de buckets worden geprobeerd in de volgorde $(1 + 1^2) \bmod 11$, $(1 + 2^2) \bmod 11$, $(1 + 3^2) \bmod 11$, enzovoort.

We beginnen met een lege hashtable, en we voegen de getallen die hieronder worden genoemd in volgorde toe. Geef voor elk getal de index van de bucket waar het getal zal worden opgeslagen.

1. Het getal 22 wordt toegevoegd, en eindigt in bucket [0].
2. Het getal 34 wordt toegevoegd, en eindigt in bucket [1].
3. Het getal 33 wordt toegevoegd, en eindigt in bucket [4].
4. Het getal 37 wordt toegevoegd, en eindigt in bucket [5].
5. Het getal 2 wordt toegevoegd, en eindigt in bucket [2].
6. Het getal 12 wordt toegevoegd, en eindigt in bucket [10].

[Numeriek]	[Numeriek]	[Numeriek]	[Numeriek]	[Numeriek]
0	**1**	**4**	**5**	**2**

[Numeriek]
10

Vraag 16 − Strange queue − 165378.3.0

De volgende code is onderdeel van een correcte (maar enigszins ongebruikelijke) implementatie van het Queue ADT.

```
struct Queue {
  struct Stack *in;
  struct Stack *out;
};

void enqueue(struct Queue* Q, Item x) {
  Q->in->push(x);
}

Item dequeue(struct Queue* Q) {
  if (Q->out->empty()) {
    if (Q->in->empty()) { fprintf(stderr, "Queue empty!"); exit(1); }
    while (!Q->in->empty()) {
      Q->out->push(Q->in->pop());
    }
  }
  return Q->out->pop();
}
```

Soms is dequeue traag. We willen daarom een amortized analyse doen. We kijken daarbij naar drie stukjes van de code:

- De functie enqueue. Ga ervan uit dat dit in realiteit 1 eenheid tijd kost.
- De functie dequeue, het gedeelte in de while-lus. Ga ervan uit dat dit in realiteit size(Q->in) eenheden tijd kost, waarbij size(Q->in) staat voor het aantal elementen dat op dat moment in de "in" stack zit.
- De rest van de functie dequeue, dus alles behalve de while-lus. Ga ervan uit dat dit 1 unit tijd kost.

Beantwoord nu kort en in je eigen woorden de onderstaande vragen.

1. Eerst zonder amortizatie. Stel dat de queue n keys bevat. Geef een upper bound op de worst case tijd complexiteit van dequeue, met grote O notatie.
2. Nu willen we amortizatie toepassen. Hoeveel wil je sparen / opnemen van de tijd bank in elke van de bovengenoemde 3 onderdelen van de code?
3. Hoe weet je dat je, als je dat doet, nooit op een negatief saldo uitkomt?
4. Wat is, met jouw amortizatiestrategie, de amortized worst case time complexity van dequeue?

Beoordelingsvoorschrift

Criterion 1 (Aantal punten: 1)

Deel 1 correct: upper bound is O(n)

Criterion 2 (Aantal punten: 1)

Deel 2 correct: spaar 1 unit bij enqueue, neem size(Q->in) op bij dequeue, en doe niks voor de rest van dequeue (het is niet erg als studenten hier ook 1 unit tijd sparen, hoewel dit niet hoeft).

Criterion 3 (Aantal punten: 1)

Deel 3 correct: het saldo is altijd precies gelijk aan size(Q->in): beide nemen toe met 1 tijdens enqueue, en af met size(Q->in) als het saldo wordt opgenomen.

Criterion 4 (Aantal punten: 1)

Deel 4 correct: amortized time complexity van dequeue is O(1).