DMA — Ugeopgave 7g

Helga Rykov Ibsen `<mcv462>`

8. december 2021

Del 1

(a)

`Heapsort` algoritmen sorterer en array ved at bygge et binært træ, som er sorteret så længe man vedligholder heap-egenskabet. Dette heap-egenskab er vedligholdt af `Max-Heapify` fra CLRS.

Vi kan sætte en boolian variabel til at returnere `false` som default og ellers 1 for hver gang `Max-Heapify` finder at forældre-knuden har samme vægt som et af sine børn.

På den måde vil algoritmen `SameWeight` returnere 1 hvis to knuder vejer det samme og ellers 0. Dette vil tage $O(n \lg n)$, fordi `Max-Heapify` bruger $O(\lg n)$ på at sammenligne forældre-knuder med børnene og fordi man kører koden så mange gange der er knuder i træet, n gange.

(b)

Der er 10 lette knuder i træet (a): letteKnuder A [1, 2, 4, 2, 0, 0, 6, 8, 3, 0].

Der er 6 lette knuder i træet (b): letterKnuder B [2, 1, 2, 6, 7, 3].

(c)

Vi skal vi at der mindst er halvdelen af knuderne, bortset fra roden, som er lette. Hvis en knude er lettere end søskende knude, så er den også let — så må mindst den ene søsende være let.

Den nedre grænse for antallet af lette knuder i et binært træ kan derfor sættes til $\frac{n-1}{2}$.

Vi skal vise at højst alle knuder bortset fra roden er lette. Ifølge præmissen kun søskende knuder kan være defineret som lette. Med andre ord er roden aldrig let.

Den øvre grænse for antallet af lette knuder i et binært træ kan derfor sættes til $n - 1$.

Jeg er meget i tvivl om hvordan spørgsmålet om tætte grænser skal forstås. Men mit bud er at der altid vil være et lige antal lette knuder i et binært træ, unaset om det er et træ med den nedre eller den øvre grænse. Og i og med at der enten kan være $n - 1$ eller $\frac{n-1}{2}$ lette knuder, men ikke begge dele på en og samme tid, så kan man sige at grænserne er tætte.

(d)

Algorithm 1 helper(left, right)

```
1: if left == NIL then
2:     return 0
3: else if left ≤ 2· right then
4:     return 1
5: else if right ≤ 2· left then
6:     return 1
7: else
8:     return 0
```

Algorithm 2 CountLightNodes(r)

```
1: if r == NIL then
2:     return 0
3: else
4:     helper(r.left, r.right) + helper(r.right,r.left) +
5:     CountLightNodes(r.left) + CountLightNodes(r.right)
```

Køretiden her er den samme som i delopgave (a), altså $O(n)$ fordi alle skridt i funktionen er konstant og man kører koden så mange gange der er knuder i træet.

Del 2

(a)

Der er seks kædeknuderne i træet: [B, G, J, F, I, K].

(b)

Vi får altså at $T(n) = \mathcal{O}(1)$ hvilket også giver god mening da vi maksimalt udfører hver operation én gang.

Algorithm 3 ChainNode(x)

```
1: if x.left = NIL xor x.right = NIL then        ▷ One of them NIL? - 1
2:     return true                                ▷ - 1
3: else                                           ▷ - 1
4:     return false                               ▷ none are NIL - 1
```

Algorithm 4 CountChainNodes(r)

```
1: counter = 0                                    ▷ No.of chain nodes set to 0 - 1
2: if ChainNode(x) then                           ▷ ChainNode return true - 1
3:     counter = counter + 1                      ▷ - 1
4: else if r.left <> NIL then                     ▷ Left node is not NIL - 1
5:     counter = counter + ChainNode(r.left)      ▷ (n-1)/2
6: else if r.right <> NIL then                    ▷ Right node is not NIL - 1
7:     counter = counter + ChainNode(r.right)     ▷ (n-1)/2
8: else                                           ▷ ChainNode return false - 1
9:     return counter                             ▷ No.of light nodes - 1
```

(c)

Køretiden for CountChainNodes(r) er den samme som for CountLightNodes(r), altså $T(n) = O(n)$ fordi alle skridt i algoritmen tager konstant tid og man kører koden så mange gange der er knuder i træet.

Del 3

(a)

Hvis jeg forstår spørgsmålet korrekt, så bliver der spurgt her hvornår vi er sikker på at der sker en kollision. Man kan derfor sige at længden n af sekvensen kan maximalt være den samme som antallet af mulige kombinationer af ordene (dvs. sandsynligheden for at der sker en kollision højst er $n^{\frac{2}{m}}$).

Forudsat signaturen er w-bit bred og hashværdien er f. eks 8 bits, så er der kun 256 (2^8) forskellige ord før der med sikkerhed sker en kollision. For at undgå dette skal vi altså have at $2^w >> n^2$, hvilket kræver at $w > 2 \cdot \lg n$.

Længden af sekvensen af signaturer kan altså højst være 2^w.

(b)

We begynder med at initiere en ordbog. I vores tilfælde er biblioteket et open address hash tabel som bruger lineær probing for at løse kollisioner. Et givet ords signatur — som er "maskeret" i form af et heltal og som også afhænger af størrelsen på inputtet (antallet af ord) — fungerer som et index.

Denne maskering ændrer sig, hvis ordbogen er næsten fyldt op, hvilket trigger

en "re-hashing" af den, som tager $O(n)$ tid. Ordbogen gemmer så det givne ords signatur, samt antallet af gange det kommer op.

Så itererer vi igennem de givne ord og sætter dem ind i ordbogen enten ved at indsætte deres signatur samt at sætte tælleren til 1 eller ved at inkrementere tælleren, hvis vi møder den samme signatur.

Til sidst løber vi igennem alle værdier i ordbogen og inkrementerer antallet af unikke tal for hver tæller = 1.

(c)

Hvis vi antager at både strenge og heltal kræver samme plads, så vil ordbogen bruge ca. $2n$ plads (dvs. ordets signatur og tælleren). Forudsat alle orderne er ens, vil en normal implementation af en ordbog stadig bruge $O(n)$ plads.