2021

IN2010

OBLIG 1

SVEIN-GISLE SÆTRE

Contents

Oppgave 1: Teque

a)

Jeg velger å implementere en Teque ved å kombinere to Deque. Da vil jeg fort kunne finne midten ved å enten se etter det siste/først elementet i enten den første eller siste dequen.

```
Deque<Integer> first = new ArrayDeque<Integer>();
Deque<Integer> last = new ArrayDeque<Integer>();
```

Input: Int x

```
1.  Procedure push_front(x)
2.   | if first.size > last.size do
3.   | | last.addFirst(first.removeLast())
4.   | | first.addFirst(x)
5.   | else
6.   | | first.addFirst(x)
```

```
1.  Procedure push_back(x)
2.   | if first.size < last.size do
3.   | | first.addLast(last.removeFirst())
4.   | | last.addLast(x)
5.   | else
6.   | | last.addLast(x)
```

```
1.  Procedure push_middle(x)
2.   | if first.size < last.size do
3.   | | first.addLast(last.removeFirst())
4.   | | last.addFirst(x)
5.   | else
6.   | | last.addFirst(x)
```

```
1.  Procedure get(x)
2.   | if x < first.size() do
3.   | | Iterator itr = first.iterator
4.   | | for i <- 0 to x do
5.   | | | itr.next
6.   | | print(itr.next)
7.   | else
8.   | | Iterator itr = last.iterator
9.   | | for I <- 0 to x – first.size do
10.  | | | itr.next
11.  | | print(itr.next)
```

b)

Java-kode i vedlagt fil. I forhold til pseudokoden ryddet jeg litt opp i den faktiske koden slik at det ble færre linjer. For get så ser jeg at det er lite effektivt å bruke en iterator, så jeg hadde noen forsøk med å lage min egen versjon av deque som bruker arrays. Jeg kom ikke helt i mål med det, så siden min versjon fungerer beholdt jeg den som den er. Jeg synes også det var litt vanskelig å finne

dokumentasjon på hvor effektive de forskjellige metodene er. Jeg antar at en array sin indeksering ikke er O(n), men var usikker på hvor jeg skulle få bekreftet dette.

c)

```
public void push_front(int x){
        if (first.size() > last.size()){
            last.addFirst(first.removeLast());
        }
        first.addFirst(x);
    }
```

Prosedyren har 6 primitive steg:

1. Metodekall: first.size()
2. Metodekall: last.size()
3. Sammenligning: first.size() > last.size()
4. Metodekall: last.addFirst()
5. Metodekall: first.removeLast()
6. Metodekall: first.addFirst()

Push_front er O(1) siden den ikke vokser når x blir større.

```
public void push_back(int x){
        if (first.size() < last.size()){
            first.addLast(last.removeFirst());
        }
        last.addLast(x);
    }
```

Prosedyren har 6 primitive steg:

1. Metodekall: first.size()
2. Metodekall: last.size()
3. Sammenligning: first.size() < last.size()
4. Metodekall: first.addLast()
5. Metodekall: last.removeFirst()
6. Metodekall: last.addLast()

Push_back er O(1) siden den ikke vokser når x blir større.

```
public void push_middle(int x){
        if (first.size() < last.size()){
            first.addLast(last.removeFirst());
        }
        last.addFirst(x);
    }
```

Prosedyren har 6 primitive steg:

1. Metodekall: first.size()
2. Metodekall: last.size()
3. Sammenligning: first.size() < last.size()
4. Metodekall: first.addLast()
5. Metodekall: last.removeFirst()

6. Metodekall: last.addLast()

Push_middle er O(1) siden den ikke vokser når x blir større.

```
public void get(int x){
        if (x < first.size()){
            Iterator<Integer> itr = first.iterator();
            for (int i = 0; i < x; i++){
                itr.next();
            }
            System.out.print(itr.next() + "\n");
        } else{
            x = x - first.size();
            Iterator<Integer> itr = last.iterator();
            for (int i = 0; i < x; i++){
                itr.next();
            }
            System.out.print(itr.next() + "\n");
        }
    }
```

Om vi antar at x alltid er et tall som ikke er høyere enn størrelsen på Teque'et så vil koden kjøre x/2 ganger. Om x er mindre enn den første Deque'en kjører den x ganger, men den første Deque'en kan maksimalt være halvparten så stort som hele Teque'et. Om x er større enn den første Deque'en vil løkken kjøre x minus størrelsen på den første Deque'en, som igjen blir halvparten av hele Teque'et sin størrelse totalt.

get() er $O\left(\frac{n}{2}\right) \rightarrow O(n)$

d)

Om N er begrenset vil det ikke påvirke kompleksiteten i O-notasjon for push_front, push_back eller push_middle,

For get sitt eksempel er det verste caset at N-1 push blir kallet før get(N-1) blir kallet. Get må da gjøre N-1 operasjoner og er $O(1)$. Det vil da alltid finnes en konstant c (mest åpenbart N) vi kan sette foran en funksjon g(n) som gjør at get() oppfyller definisjonen på O-notasjon.

For hele Teque'et har jeg også sett på hva som blir worst case basert på hvilke kall som blir kjørt:

1. 1 push blir kallet før get blir kallet N-1 ganger. Get vil da alltid treffe på første forsøk og gjøre totalt N-1 operasjoner som gjør at algoritmen er $O(n)$
2. $\frac{n}{2}$ push blir kallet før get blir kallet $\frac{n}{2}$ ganger. Get må da gjøre $\frac{n}{2}$ operasjoner $\frac{n}{2}$ ganger som blir $\frac{n^2}{4}$ som gjør at algoritmen er $O(n^2)$
3. N-1 push blir kallet før get blir kallet 1 ganger. Get vil da i verste fall gjøre N-1 operasjoner som gjør at algoritmen er $O(n)$.

Det ser ut som det verste tilfellet er at like mange push-kall blir kjørt som get-kall, hvor algoritmen i så tilfelle er $O(n^2)$

```
for (int i = 0; i < N; i++){
            String[] line = br.readLine().split(" ");
            String cmd = line[0];
            int x = Integer.parseInt(line[1]);
            try{
                Method method = myTeque.getClass().getMethod(cmd, int.class);
                method.invoke(myTeque, x);
            } catch(NoSuchMethodException|IllegalAccessException|InvocationTar
getException e){
                System.out.println(e.toString());
            }
        }
```

Oppgave 2: Binærsøk

Siden dette er en lenket liste antar jeg at get-funksjonen vil kjøre gjennom alle elementene i listen via next-kall. Det vil si at get(n) betyr n metodekall.

I verste fall er tallet man leter etter det siste, og man vil da halvere listen "oppover". Første gang vil get() funksjonen kjøre 1/2|A|, deretter 3/4|A| → 7/8|A| → … → |A|.

Vi vet at denne halveringen er O(log n) og siden det må itereres gjennom hvert punkt får vi at det kjøres i O(n log n)

Hadde det vært en dobbeltlenket liste kunne pekeren ha startet fra slutten om det man leter er i andre halvdel av listen, og da må man maksimalt kjøre n ganger og er O(n). Dette gitt at halveringer går mot 1. $\sum_{n=1}^{\infty} \frac{1}{2^n} = 1$. Det ser derimot ikke ut som om algoritmen har fasilitert for denne effektiviseringen og blir da også O(n log n).

Ved et vanlig array, der man ikke trenger å kjøre gjennom alle pekerene hver gang, halveres problemet hver gang og funksjonen blir da O(log n).

Oppgave 3: Kattunge

a)

Pseudokode:

```
1.  Procedure findExit()
2.   | int intToLookFor;
3.   | int firstInt;
4.   | while firstInt is not -1
5.   | | line = br.readLine().split(" ")
6.   | | firstInt = first int of line
7.   | | for i <- 1 to length of line do
8.   | | | if intToLookFor == int in line
9.   | | | | intToLookFor = int in line
10.  | | | | reset while loop
```

Planen er å huke tak i katten sin posisjon og så søke gjennom hver linje til jeg finner igjen det tallet i en linje der det ikke er det første tallet. Når jeg finner tallet vet jeg at det første tallet i den linjen er forelderen til posisjonen katten er i og begynner så samme søket etter det tallet. På den måten leter jeg etter utveien via foreldrene til noden katten er i.

b)

Java-kode vedlagt.

```
InputStreamReader fileIn = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(fileIn);

        // Les første linje for å hente ut katten sin posisjon
        String[] line = br.readLine().split(" ");
        int intToLookFor = Integer.parseInt(line[0]);
        System.out.print(intToLookFor);
        int firstInt = 0;

        // Marker starten av filen for å kunne resete readeren
        br.mark(50);

        while (!(firstInt == -1)){
            line = br.readLine().split(" ");
            firstInt = Integer.parseInt(line[0]);
            for (int i = 1; i < line.length; i++){
                if (intToLookFor == Integer.parseInt(line[i])){
                    intToLookFor = firstInt;
                    System.out.print(" " + intToLookFor);
                    br.reset();
                }
            }
        }
```

Algoritmen min ser ved første øyekast ut som den er $O(n^2)$ siden den bruker to løkker, en while- og en for-løkke. Den indre for-løkken vokser derimot ikke når n vokser, der n er lik antall linjer i filen vi leser inn. For-løkken har en øvre grense på maks antall barn en node i treet har, da den bare leter gjennom barnene til nodene på treet. While-løkken, som leter først etter forelderen til kattens posisjon, og så forelderen til den posisjonen etc til den når toppen (eller roten), vil i verste fall kjøre $n + (n-1) + (n-2) + \cdots + 1$ ganger til den har funnet utveien, noe som gjør algoritmen til O(n).

Koden er en del endret siden pseudokoden, selv om den gjør hovedsakelig det samme. Måten min å løse oppgaven på var å iterere gjennom linjene flere ganger frem til alle foreldrene hadde blitt funnet, og dermed også en utvei. Jeg slet litt med å finne en måte å resette BufferedReader da jeg ikke har jobbet med InputStreamReader eller BufferedReader før, men det ser ut som det fungerer. Til å begynne med forsøkte jeg å bygge opp treet samtidig som jeg leste gjennom linjene, i en tanke om at man bare kan trenge å lese gjennom linjene en gang. Jeg så derimot ikke en løsning på hvordan jeg skulle holde orden på pekerene for å sette alt sammen til et helt tre på slutten.

Oppgave 4: Bygge balanserte søketrær

a)

Siden treet skal være både balansert og et binærtre må det nødvendigvis være slik at rotnoden må være det miderste tallet. Da kommer halvparten av tallene som er mindre på venstre side og andre halvpart som er høyere på høyre side. Slik blir det også for alle subtrær som kommer nedover, så eksempelvis den første noden til høyre må være det miderste tallet av den halvparten som var høyere enn det midterste tallet i den originale listen. Vi må dermed gjøre samme operasjon for hvert subtre nedover.

Tanken er dermed å bruke en funksjon som tar det miderste tallet og printer det, og dermed kjører samme funksjon på de to sub-listene på hver sin side av det midterste tallet.

Input: Liste A

```
1. Procedure balancer(list listToCheck)
2. | int low = 0
3. | int high = listToCheck.length – 1
4. | int mid = (low+high)/2
5. | print(mid)
6. | balancer(listToCheck, mid+1, high)
7. | balancer(listToCheck, low, mid-1)
```

```
1. Procedure balancer(list listToCheck, int low, int high)
2. | If (low > high)
3. | | return
4. | Int mid = (low+high)/2
5. | print(mid)
6. | balancer(listToCheck, mid+1, high)
7. | balancer(listToCheck, low, mid-1)
```

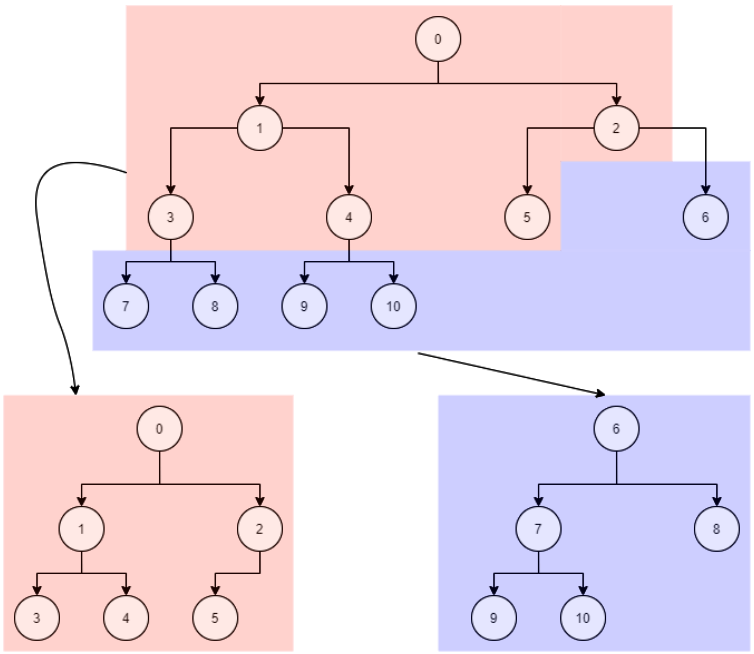
Ferdig kode ble ganske lik pseudokoden, selv om jeg gjorde noen rare matematiske krumspring for å forsikre meg om at jeg fikk det korrekte midterste tallet. Den endelige koden endte også opp med å bruke ArrayList istedenfor Arrays, da jeg blanket på hvordan man skulle finne ut av antall linjer det er i innlesingen. Dette hadde jeg trengt ved initialiseringen av et array.

b)

Som i forrige oppgave må man finne det midterste elementet først, og siden oppgaven hinter til at man kan bruke mange heaps mistenker jeg at en måte å gjøre dette på er å lage et nytt heap av halvparten av det originale heapet. Om man fjerner halvparten og legger de i et nytt heap ender man dermed opp med å ha det midterste tallet i rotnoden.

Som i forrige oppgave bør man så kunne gjøre dette videre for hver sub-heap som blir laget, slik at tallene blir sortert.

Illustrasjonen til høyre viser hvordan jeg ser for meg at dette vil skje om vi bruker eksempel-inputen fra 0-10. Det røde området er ny heap vi lager, mens blått område er det som gjenstår av original heap – som nå har midterste tall som rotnode



Pseudokode:

```
1. Procedure sortHeap(PriorityQueue pq)
2. | int halfSize = pq.size()/2
3. | PriorityQueue pqHalf = new PriorityQueue()
4. | for (i ← 0 to halfSize do
5. | | pqHalf.offer(pq.poll())
6. | System.out.println(pqHalf.peek())
7. | sortHeap(pqHalf)
8. | sortHeap(pq)
```