Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Filip Strömbäck och Magnus Nielsen

26 oktober 2023

Datastrukturer och algoritmer Lösningsförslag till tentamen 2023-10-26

1. (a) Nej, det är inte ett binärt sökträd. Trädet är ett binärt träd. Det är dock inte ett sökträd i och med att villkoret att för alla delträd ska alla vänsternoder vara mindre än roten och alla högerträd vara större än roten. Det uppfylls exempelvis inte i och med att 7 är större än 1.

   (b) Nej. Det är inte komplett i och med att det finns ”hål” när vi gör en level-order traversering av trädet (8 saknar högerbarn).

   (c) Nej, ett krav för att det ska vara en heap är att trädet är komplett.

   (d) 1 - 7 - 8 - 15 - 20 - 12 - 42 - 21 - 35 - 42 - 22 - 51

2. Rad 1: a
   Rad 2: b
   Rad 3: d
   Rad 4: c
   Rad 5: finns ej i listan

   Den sista algoritmen är heapsort. Det kan man exempelvis se genom att de sista 3 elementen är på rätt plats, och att resten av arrayen är i form av en heap (70 är det stösta kvarvarande elementet, det är först).

3. (a) För alternativ 1:

   Vi sätter in studenten på första lediga plats ($\mathcal{O}(1)$), förutsatt att vi håller reda på hur många vi har lagrat). Vi kör sedan en iteration av insertion sort ($\mathcal{O}(n)$). Totalt: $\mathcal{O}(n)$

   För alternativ 2:

   Först slår vi upp namnet i hashtabellen. Detta tar $\mathcal{O}(1)$ (amorterad) tid, givet en bra hashfunktion. Finns inte studenten lägger vi sedan in elementet i heapen, vilket tar $\mathcal{O}(\log n)$ tid att ”bubbla upp/swim” elementet. Totalt: $\mathcal{O}(\log n)$

   (b) För alternativ 1:

   Elementet finns på sista upptagna index. Vi kan ta ut det i $\mathcal{O}(1)$ tid förutsatt att vi har en räknare med antalet studenter i listan.

   För alternativ 2:

   Det minsta elementet finns i toppen av min-heapen. Vi kan därmed plocka ut det direkt i $\mathcal{O}(1)$ tid.

   (c) För alternativ 1:

   Likt i uppslagning hämtar vi bara sista studenten i listan och sätter platsen till null samt räknar ner hur många studenter vi lagrar med 1. $\mathcal{O}(1)$.

   För alternativ 2:

   Vi plockar bort elementet från min-heapen genom att byta det med det sista elementet. Sedan återställer vi heap-egenskapen genom att ”bubbla ner” toppelementet. Detta tar $\mathcal{O}(\log n)$ tid. Vi tar också bort namnet från hashtabellen, vilket tar $\mathcal{O}(1)$ tid givet en bra hashfunktion. Totalt: $\mathcal{O}(\log n + 1) = \mathcal{O}(\log n)$

   (d) Alternativ 2 är bäst. Sätter vi in n element i alternativ 1 så tar det $\mathcal{O}(n \cdot n) = \mathcal{O}(n^2)$ tid, medan i alternativ 2 tar det bara $\mathcal{O}(n \log n)$ tid.

   Utöver detta är de båda alternativen lika bra på att hitta nästa student, men alternativ 2 är lite mer krävande vid borttagning. Skillnaden i insättning är betydligt större än vid borttagning. Alternativ 2 har dock den lilla nackdelen att det använder lite mer minne (inte asymptotiskt, dock) i och med hashtabellen.

(e) Vi kan lösa detta genom att modifiera hashtabellen så att den i stället för att lagra bara namn, så lagrar den *namn* → *positioniheap*. Detta gör så att vi vid insättning kan slå upp studenten i hashtabellen och hitta var den ligger i heapen i $\mathcal{O}(1)$ tid. Vi kan sedan bara "bubbla ner"' elementet i heapen från dess nuvarande position för att återställa heapegenskapen. Detta tar $\mathcal{O}(\log n)$ tid totalt.

Vi måste då också modifiera våra heapoperationer så att de uppdaterar hashtabellen. Detta går att göra: heapen lagrar redan namn, så att vi kan med hjälp av namnet där enkelt hitta och uppdatera rätt plats i hashtabellen.

4. (a) En linjär loop. Den körs n gånger. Bara konstanta operationer utanför och inuti.
Detta ger: $\mathcal{O}(n+1) = \mathcal{O}(n)$

(b) Loopen går från 1 till n, och i dubbleras varje gång. Det ger $\mathcal{O}(\log n)$. Sedan anropas $f(n)$, vilket ger $\mathcal{O}(n)$ enligt ovan.
Totalt: $\mathcal{O}(\log n + n) = \mathcal{O}(n)$

(c) Vi har en loop från 1 till n. Detta kör $f(m)$ n gånger.
Alltså: $\mathcal{O}(n \cdot m)$

(d) I bästa fall:

Vi hittar elementet tidigt (ex. det första elementet) och returnerar direkt: $\mathcal{O}(1)$.

I värsta fall:

Vi hittar inte elementet och kör sedan `add`. Detta tar $\mathcal{O}(n)$ för att söka, och $\mathcal{O}(n)$ för `add` (värsta fallet). Totalt: $\mathcal{O}(n+n) = \mathcal{O}(n)$

5. (a) Exempelvis kan vi använda ett hashset för att ta bort dubbletter från resultatet för varje klipp:

```
int[] computeLaughs(ArrayList<ArrayList<Integer>> input) {
     // Skapa en array till resultatet
     int[] output = new int[input.size()];

     // För varje klipp: (O(n) iterationer)
     for (int i = 0; i < input.size(); i++) {
         ArrayList<Integer> clip = input.get(i);

         // Kopiera skratten till ett hashset: O(s) tid, totalt
         HashSet<Integer> laughs = new HashSet<>(clip);

         // Vi sätter antalet skratt för denna video
         // till storleken på vårt hashset: O(1) tid.
         output[i] = laughs.size();
     }

     return output;
}
```

(b) Givet att vi har en bra hashfunktion tar operationerna på vårt hashset $\mathcal{O}(1)$ tid (dvs. $\mathcal{O}(s)$ tid att sätta in s element). Totalt kommer vi att sätta in s element i hashtabellen. För att `spara antalet` räcker det med att kontrollera storleken på vår hashtabell. Vi gör konstant arbete för varje skratt och varje klipp. Detta ger oss totalt: $\mathcal{O}(n+s)$.

(c) Det enda extra minnet vi behöver är vårt hashset (den innehåller maximalt s element (ofta mycket mindre)) samt arrayen som lagrar resultatet (exakt n element). Alltså $\mathcal{O}(n+s)$.

6. (a) Uppgiften beskriver en topologisk sortering av tabellen. Vi behöver bara tänka på kolumnerna *ID* och *Måste ätas efter*.

Vi antar att grafen lagras i följande struktur:

```
class Node {
  // Namn, för utskrift senare.
  String name;
  // Restaurang, för att hitta rätt senare.
  String restaurant;

  // Beroenden.
  ArrayList<Integer> dependencies;

  // För algoritmen, initieras till false.
  boolean visited = false;
  // Relevanta getters och setters
}
```

Här kan vi alltså se att vi representerar varje rätt som en nod i en graf. Bågarna representerar beroenden. Hela grafen lagras som en array av `Node`. Till skillnad från ”klassiska” topologiska sorteringar så går bågarna i omvänd riktning. Vi vill alltså hitta en sortering där a kommer före b om det finns en båge från b till a. Detta gör bara lösningen enklare dock.

Vi kan lösa problemet med en DFS ungefär som nedan:

```
ArrayList<Node> find_order(ArrayList<Node> graph) {
  ArrayList<Node> result;

  for (int i = 0; i < graph.size(); i++) {
    find_recursive(result, graph, i);
  }

  return result;
}

// Rekursiv hjälpfunktion.
void find_recursive(ArrayList<Node> result, ArrayList<Node> graph, int node) {
  if (graph.get(node).getVisited()) {
    return;
  }
  graph.get(node).setVisited(true);

  // Traversera beroenden, lägg till dem först.
  for (int dep : graph.get(node).getDependencies()) {
    find_recursive(result, graph, dep);
  }

  // Lägg till noden själv.
  result.add(graph.get(node));
}
```

(b) Detta tar $\mathcal{O}(n+m)$ tid: vi traverserar alla bågar och alla noder ett konstant antal gånger.

(c) Uppgiften beskriver ett kortaste-vägen-problem. Vi kan därmed lösa det med en BFS. Vi använder samma grafrepresentation som i uppgiften innan. I och med att vi inte vet var vi ska, så börjar vi i noden k och slutar så snart vi hittar en nod utan beroenden.

```
ArrayList<Node> findOrder(ArrayList<Node> graph, int k) {
    Queue<Integer> toVisit = new LinkedList<>();
    HashMap<Integer, Integer> previous = new HashMap<>();

    previous.put(k, -1);
    toVisit.add(k);

    while (!toVisit.isEmpty()) {
        int currentIndex = toVisit.peek();
        Node current = graph.get(currentIndex);
        toVisit.remove(); // tänk dequeue

        // Om noden är tom, är detta den närmsta rätten utan beroenden.
        if (current.getDependencies().isEmpty()) {
            ArrayList<Node> result = new ArrayList<>();
            for (int i = currentIndex; i > 0; i = previous.get(i)) {
                result.add(graph.get(i));
            }
            return result;
        }

        for (int child : current.getDependencies()) {
            if (previous.containsKey(child)) {
                continue;
            }
            previous.put(child, currentIndex);
            toVisit.add(child);
        }
    }

    // No solution.
    return new ArrayList<Node>();
}
```

(d) Tidskomplexiteten blir $\mathcal{O}(n + m)$, vi behandlar varje nod och varje båge ett konstant antal gånger.