Teoriuppgifter Labb 1

Adeel Hussain & Rakin Ali
CINTE 3

1. **Vilka är rollerna vid parprogrammering och vilka uppgifter har varje roll?**

Föraren → Skriver kod och löser aktuella problem.

Navigatören → Granskar varje kodrad när den matas in. Ska även komma på ideer till förbättringar och funderar över problem som kommer att behövas lösas senare.

2. **Indexinformationen för ett ord (det vill säga i vilka tecken positioner ordet förekommer i den stora texten) kan bli mycket stor. Hur bör positionerna lagras för att det ska bli effektivt, som text eller binärt (data streams i Java)? Bör indexinformationen lagras tillsammans med själva ordet eller på ett separat ställe?**

Indexinformationen bör lagras tillsammans med själva ordet. Indexet innehåller byte-positionerna till vardera ord i texten. Detta eftersom att lagra informationen separat kan leda till mindre effektivitet då flera filacesser behövs göras för varje ord.

Hur positionerna lagras, antingen som text eller binärt, spelar inte roll då det är samma informationen som tar lika mycket plats, däremot läser datorn av binärt lite mer effektivt då en ASCII konversion ej behövs vilket är varför det är mer effektivare att lagra som binärt.

3. **I denna labb ska datastrukturen för konkordansen huvudsakligen ligga på fil, vilket betyder att sökningar görs i filen istället för som vanligt i internminnet. Det påverkar till exempel hur man representerar pekare (lämpligen som bytenummer i filen). Diskutera för- och nackdelar med olika implementationer av konkordansen med avseende på följande egenskaper:**
    - snabbhet (antal filläsningar och filpositioneringar per sökning),

- minneskomplexitet för fillagringen (bara konstant mycket internminne ska användas)
- enkelhet att konstruera och lagra på fil.

Ta åtminstone upp följande datastrukturer:

- binärt sökträd

Med binärt sökträd kan man hitta ordet inom O(log(n)) tid förutsatt att trädet är balanserad. Att balansera ett obalanserad träd tar O(n) tid. Denna datastruktur är inte enkel att konstruera då mycket tid måste läggas på att balansera trädet efter varje insatt nod och noden ska innehålla pekare till filen. Utan en balanserad träd förlorar binärt sökträd sin finess. Minneskomplexitet blir *n.*

- sorterad array,

Förutsatt att vi vet det exakta det indexet blir minnes komplexiteten mycket kort. Vid element-sökning kan det ta O(log(n)) vid binärsökning. Detta är enkel att konstruera men kan snabbt bli svårt och väldig tidskrävande beroende på hur vi insätter element för att hålla det sorterad. Ifall storleken för texten ej är känd så kommer det bli ineffektivt både tidsmässigt och minnesmässigt då det kommer behövas skapa nya större arrayer och flytta över data för att kunna fortsätta lagra ny information.

- hashtabell,

Minneskomplexitet blir *n*. En perfekt hashtabell ger konstant söktid däremot kan olika ord ha samma hashvärde vilket kan leda till en krock, och detta är mer troligt i större filer. Detta försämrar effektiviteten. Att konstruera hashtabell kan bli svårt och mindre effektivt då minneskomplexiteten blir stor. Hashtabellen kommer att innehålla flera tomma oanvända platser som datastrukturen kommer att allokera oavsett.

- trie (träd där varje nivå motsvarar en bokstav i ordet),

Kräver massa look ups där tidskomplexiteten kan bli mindre effektiv då det krävs iterationer genom flera noder. Minneskomplexiteten blir mindre jämfört med de andra datastrukturerna eftersom att man ej behöver lagra alla ord utan endast de bokstäver som behövs, djupet kommer endast vara lika långt som det längsta ordet. Däremot kan minneskomplexiteten ifall man ska lagra datastrukturen till en fil då man behöver lagra pekare till varje bokstav från varje nod. Denna datastruktur är svårast att konstruera.

- latmanshashning

Latmanshashning är lämpligt för sökning med få disk accesser i stor text. Tidskomplexiteten blir effektiv då man endast behöver hasha de 3 första bokstäverna i sökordet, däremot kan det ske flera krockar då det är mycket större chans att olika ord kommer att ha samma hashvärde. Minneskomplexiteten kommer även att bli lite större då man kommer behöva lagra alla möjliga trebokstavskombinationer och dess hashvärde samt en fil med alla orden och dess positioner sorterade.

Redovisa för- och nackdelarna i en tabell.

Algoritmer	Fördelar	Nackdelar
Binär Sökträd	● Ett balanserat träd kan hitta ord inom O(log(N)) tid.	● Tidsmässigt ineffektivt att konstruera balanserad träd vid insättning
Sorterad Array	● Snabbt uppslagsbart om vi vet exakta index positionen ● Snabb uppslagsbart om vi har ett effektivt sökning algorithm.	● Tidskrävande att sortera varje element vid insättning och kräver genomtanke.
Hashtabell	● Snabbaste sättet att hitta ett ord	● Kollisioner kan bli svårt att hantera . ● Desto större fil, desto större sannolikhet att kollisioner sker ● Inte effektivt lagringssätt då hashtabellen kommer innehålla tomma platser
Trie	● Djupet kommer endast vara lika långt som det längsta ordet.	● Svårast att konstruera ● Ineffektivt att lagra som fil

	• Mindre minneskomplexitet	
Latmans Hashning	• Kan snabbt slå upp ord • Lätt att konstruera	• Hashtabellen som skapats kan ha kollisioner och vara ineffektivt • Kan bli svårt att hantera bokstäver

4. **Ge exempel på minst 7 indata (dvs ord) som är lämpliga testfall i labb 1 och motivera varför.**

1. **ö _ _ (sjätte ordet i textet förutsatt att mellanslag räknas ej)**

   → Special case i svenska alfabetet och ett ord med ett bokstav. Vi vet att det ligger i det sjätte ordet i texten. Också det passar inte prefixed.

2. **amager (det allra första ordet i korpus)**

   → Första ordet i texten

3. **_ ö _**

   → Special case i filen. Kontrollerar om det kan läsa in ordet med ett mellanslag innan?

4. **ánthropos**

   → Special case som innehåller accent. Sker i position 80828 i texten.

5. **Samarbetet**

   → Crash case. Försöker kolla om ordet “samarbetet” och “Samarbetet” urskiljer sig eller inte. Tokenizer gör att de INTE urskiljer sig (första ligger på position 702 292 → stort S)

6. **samarbetet**

   → Se 5ans resonnemang, (första ligger på position 992 586, litet S)

7. **Mult**

→ Sista ordet i texten.