Auto complete 6 poeng

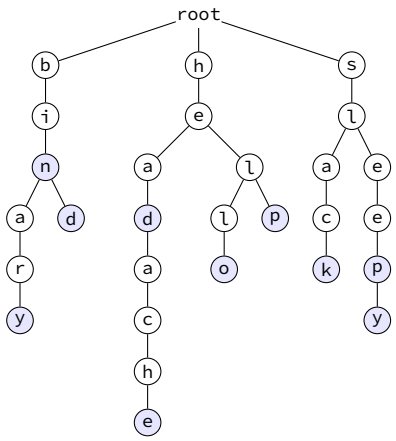
De fleste teksteditorer har funksjonalitet for «auto completion», altså der det dukker opp en liten boks med mulige måter å fullføre et påbegynt ord. Nedenfor gir vi to mulige datastrukturer som kan brukes for å implementere en enkel mekanisme for å fullføre et påbegynt ord.

Vi viser eksempler på hvordan datastrukturene ser ut dersom de lagrer ordene: `bin`, `binary`, `bind`, `head`, `headache`, `hello`, `help`, `slack`, `sleep`, `sleepy`.

Strategi 1

Den første strategien er å bruke et *prefiks-tre* som holder på strenger. Et prefiks-tre har en rot. Hvis en streng s er lagt inn i treet, har rotnoden et barn med s sin første bokstav. Den noden har igjen en peker til s sin andre bokstav, og så videre. Den siste bokstaven i s er markert som en *terminal* node (terminalnodene er fargelagt blå i eksempelet). Det vil si at hvis du følger en sti fra rotnoden til en terminalnode, så har du stavet et ord som er lagt inn i treet.

Fra et prefiks-tre er det enkelt å finne alle måter å fullføre et påbegynt ord. Det gjøres ved å returnere alle stier fra rotnoden som begynner med bokstavene fra et gitt ord og ender i en terminalnode.



Strategi 2

Den andre strategien er å bruke et hashmap, der vi lar hver prefiks av et ord mappe til en liste av alle mulige måter å fullføre ordet. Hvis `H` er et hashmap, og s er en streng som skal legges til, ser vi på hver prefiks p av s og legger s til i listen `H[`p`]`. I eksempelet ved siden av ser dere hvordan hashmapet kan se ut dersom alle eksempelordene er lagt (merk at noen av listene er kortet ned, indikert ved «...»).

Fra et slikt hashmap er det svært enkelt å finne alle måter å fullføre et påbegynt ord. Det gjøres ved å slå opp det påbegynte ordet i hashmapet og returnere listen.

Nøkkel	Verdi
`""`	`"bin", ..., "sleepy"`
`"b"`	`"bin", "binary", "bind"`
`"bi"`	`"bin", "binary", "bind"`
`"bin"`	`"bin", "binary", "bind"`
`"bina"`	`"binary"`
`"binar"`	`"binary"`
`"binary"`	`"binary"`
`"bind"`	`"bind"`
`"h"`	`"head", ... "help"`
`"he"`	`"head", ... "help"`
`"hea"`	`"head", "headache"`
`"head"`	`"head", "headache"`
`"headа"`	`"headache"`
`"headac"`	`"headache"`
`"headach"`	`"headache"`
`"headache"`	`"headache"`
`"hel"`	`"hello", "help"`
`"hell"`	`"hello"`
`"hello"`	`"hello"`
`"help"`	`"help"`
`"s"`	`"slack", "sleep", "sleepy"`
`"sl"`	`"slack", "sleep", "sleepy"`
`"sla"`	`"slack"`
`"slac"`	`"slack"`
`"slack"`	`"slack"`
`"sle"`	`"sleep", "sleepy"`
`"slee"`	`"sleep", "sleepy"`
`"sleep"`	`"sleep", "sleepy"`
`"sleepy"`	`"sleepy"`

Drøft fordeler og ulemper ved de to ulike strategiene både med tanke på kjøretid og minnebruk

Dette er en oppgave som går ut på å gjøre rimelige antagelser og anvende dem til å si noe fornuftig om hvilken fremgangsmåte man bør bruke. Dette er den oppgaven fra eksamenssettet hvor sensoren er nødt til å anvende mest skjønn i poengsettingen. Tanken bak oppgaven er å få studenten til å reflektere over hvilken fremgangsmåte man bør velge i lys av kjøretidskarakteristikkene til de to strategiene. Det bør legges vekt på at teksten er godt skrevet (dette betyr ikke at det skal trekkes poeng for enkle skrivefeil, men at teksten må være tydelig og enkel å forstå; en fin tommelfingerregel er at en setning man ikke forstår etter to gjennomlesninger ignoreres). Under er et eksempel på et svar som inneholder mange refleksjoner vi ønsker å se.

Strategi 1 bruker et prefiks-tre, som er en minneeffektiv representasjon av ordlisten. Det er viktig å merke seg at denne representasjonen kan være mer plassbesparende enn å kun lagre ordlisten som en liste av strenger. Desto flere ord som deler prefiks, desto mer plassbesparende er representasjonen i forhold til en ordliste.

Å legge til et nytt ord i prefiks-treet er veldig effektivt, med tidsbruk proporsjonalt med antall bokstaver i det nye ordet.

I tillegg gir prefiks-treet opphav til en veldig naturlig algoritme for å finne måter å fullføre et påbegynt ord. Vi kan følge stien som er diktert av det påbegynte ordet, og fra subtreet til noden kan ordene samles. Dette tar lineær tid med hensyn til størrelsen av subtreet. Det vil si at dersom det på begynte ordet er den tomme strengen, så må hele ordlisten genereres, som tar lineær tid med hensyn til hele treet. Dersom vi antar at treet har høy forgrening (for eksempel 26, en for hver bokstav i det engelske alfabetet), og at størrelsen på hvert subtre er ca. like stort (som ikke er en helt rimelig antagelse), så vil hver bokstav som legges til i det påbegynte ordet korte ned antall måter å fullføre det på med 26. Det vil altså si at antall måter å fullføre ordet på minker drastisk for hver bokstav som legges til, og som gjør at det etter all sannsynlighet vil være veldig effektivt dersom vi kun lar «auto-completion»-mekanismen til å slå til dersom vi har skrevet noen få bokstaver.

Strategi 2 bruker et hashmap fra prefikser til ordlister. Dette er en veldig lite minneeffektiv representasjon. Den tomme strengen vil mappe til hele ordlisten, mens lenger ord vil mappe til kortere lister. Plassbruken vil være proporsjonalt med summen av antall bokstaver av alle ord i ordlisten.

Å legge til et nytt ord i hashmapet er veldig effektivt, med tidsbruk proporsjonalt med antall bokstaver i det nye ordet. Dette er under antagelsen om at rekkefølgen på ordlistene ikke spiller noen rolle (ofte vil man kanskje ha de sortert eller prioritert etter frekvens).

Å hente ut alle måter å fullføre et påbegynt ord er forventet konstant tid, ettersom vi kan returnere en liste som allerede er lagret. Dette er altså mer effektivt enn prefiks-trevarianten. Det sagt vil forslagene vises frem, og det krever lineær tid med hensyn til antall «completions».

Til slutt ender dette med å være hovedsakelig en avveining mellom tidsbruk og minnebruk. Hvis vi tar det å vise ordene med i betraktningen, så virker det som at det er lite tid å spare på det raske oppslaget i hashmapet.