T9

Hiba Qutbuddin Habib

Oktober 2022

Introdukion

T9 har varit en teknik för textinmatning för mobiler telephoner, då mobiltelephoner hade nio knappar och varje knapp skulle stå för tre bokstäver från alfabetet. T9 tekniken underlättade då inmatning av text i dem icke smarta mobilen, då T9 kunde hålla koll på samtliga möjliga ord som kunde encodes redan i början av sekvens med nycklar. I denna uppgiften kommer vi att använda T9 struktur för att implemetera en nästa samma T9 teknik, då vi använder oss av en lista av de vanligaste svenska ord.

'Trie', en ovanlig träd

Trie datastrukturen har sin grund i träd datastrukturen. Fördelen med trie datastrukturen är att den kan användas för att lagra en del stränger och kunna utföra effektiva sökoperationer på de. Trie kan användes om man behöver sortera ett antal strängar alfabetisk samt för att kunna komma åt dessa stränger med en viss prefix.

För att kunna implementera T9 så är trie en perfekt datastruktur för den, då om de två strängar har en gemensam prefix så bör de ha samma förälder nod i trie.

I denna implementering kommer vi att bortse ifrån de två bokstäverna 'q' samt 'w' för att vi ska kunna få med bokstäverna 'ä', 'ö' samt 'å'. Detta resulterar i att vi kommer i slutet att få 27 tecken vilket mappas bra i nio tangenter om varje tangent innehåller tre tecken.

Node

Node klassen kommer att innehåller en trie datastruktur. Denna datastrukturen kommer att bestå av en root som innehåller 27 grenar samt en "Boolean" värde, denna värdet kommer att fungera som en flaga alltså om man har ett löv som innehåller en validerat ord så sätts denna flagen till sann, annars är det falskt. Grund till klassen T9 kan se ut enligt:

```
public class T9 {
    Node root;
```

```
public class Node {
     public Node[] next;
     public boolean word;
public Node() {
     next = new Node[27];
      word = false; }}
```

Kod, Index samt Nycklar

I denna delen kommer vi att implementera ett antal olika användbara metoder i trie klassen. Den första metoden som man behöver är en metod som för varje karaktär returnerar den en kod, exampelvis för karaktären 'a' så bör metoden returnera "0", jag implementerade metoden enligt:

```
public int giv_character_get_code(char character) {
//om ingen karäktar är givna eller den givna karäktar
//är inte med i cases nedan, så får man värde -1
  int code = -1;
   switch (character) {
        case 'a':
        code = 0;
            break;
        case 'b':
        code = 1;
            break;
        case 'c':
            code = 2;
             break;
        case 'd':
            code = 3;
            break;
//cases fortsätter fram till karäktaren 'ö' vilket returnerar kod 26 som sista
        .......
    return code; }
```

Den omvända metoden implementeras också enligt:

```
public char give_code_get_character(int code) {
    char character = '_';
    switch (code) {
        case 0:
         character = 'a';
            break;
        case 1:
        character = 'b';
             break;
```

```
        .....
    return character ;}
```

Ett annat användabar metod som man kan implemtera är metoden som tar emot en nyckel och returnerar en index, dessa indexer används sen i vektor, metoden implemteras enligt:

```
  public int giv_key_get_index(char key){
    int index = -1;
    switch (key){
        case '1':
        index = 0;
            break;
        case '2':
        index = 3;
             break;
         case '3':
        index = 6;
              break;
        .......
```

Eftersom varje tanget innehåller tre tecken så den första nyckel kan vara en av de tre tecken och den andra nyckel startar direkt från den tredje tecken.

Lägga till ord

I denna delen implementeras en metod för att addera ord till trie, då man startar från node, om noden är tom som skapas det en nod, annars läggs till ord, genom att varje tecken får en index och dessa läggs till i trädet genom att man arbetar sig neråt i trädet.

```
  public void add(String word) {
    Node nod = root;
    for (int i = 0; i < word.length(); i++) {
    //varje tecken i ordet omvandlas till int,
    //sedan användes dessa som indexer i trie.
        char cha = word.charAt(i);
        int index = giv_character_get_code(cha);
        System.out.println(index);
        if (nod.next[index] == null) {
            Node newnode = new Node();
            nod.next[index] = newnode;
            nod = newnode;
        } else {
            nod = nod.next[index];} }
//när vi har adderat ett gilltig ord så sätts flaga till true
```

```
        nod.word = true;
```

Söka efter ett ord med en sekvens

I denna delen implementerars en metod för att söka efter ord i trie. Metoden i vårt fall kommer att returnera all möjliga ord för en given sekvens. Metoden kallas för "Search" och implementeras rekursivt. Då man startar från rooten och all möjliga alternativ för den första nyckel söks, då varje nyckel generera tre tecken, när man når slutet av sekvensen så bör vi ha ett antal giltiga ord, detta kan man kontrollera genom att kolla värdet på flagan, då varje gång ett giltig ord läggs till så blir värdet av flagan "true". Metoden kommer också att returnera en sekvens "path", denna skevensen kommer att representera vägen för ordet, alltså bokstäverna som man passerade för att få en given ord.

Vanligaste orden i Svenskan

För att kontrollera om vårt trie implementering fungerar så kommer vi att använda en lista med de vanligaste orden i Svenska spårket, denna listan kallas för "kelly-listan"[1]. Listan är en txt dokument, för kunna lägga dessa ord i trie så kommer vi behöva läsa av filen för att sedan dessa ord adderas till trie datastruktur, då man startar från rooten. För att läsa av filen använder man javas "Buffer" och "file reader" enligt:

```
try (BufferedReader br = new BufferedReader(new FileReader
("/Users/hibaqutbuddinhabib/IdeaProjects/
T9/src/kelly.txt"))){
   String string = "";
       while((string = br.readLine()) != null){
        Node current = root;
```

Sammanfattning

Trie datastruktur är en användbar datastruktur om man behöver lagra ett antal strängar, då med trie datastrukturen blir sök operationen mycket effektivare, detta eftersom om fler strängar har samma nyckel så kommer de att ha gemensamma förfäder vilket gör sökning mer effektivare jämfört om man skulle ha använt en "HashTabel" då man skulle ha behövt applicera en hash funktion vilket gör att implementering blir inte lik enkelt jämfört med trie datastruktur.

[1] https://canvas.kth.se/courses/34958/assignments/213088