Pràctica 3: Índex alfabètic d'un text

Christian Callau, Dani Diaz

Curs 2016 - 2017

TADs Escollits i implementacions triades

La taula de dispersió triada es un hash set, en que el valor del element es la clau, en aquest cas la string que representa la paraula es al mateix temps el valor i la clau a partir de la qual es calcularà la posició a la taula de hash. Es un set en el sentit de que no accepta valors repetits.

Pel que fa al TAD arbre hem triat un arbre tipus trie ja que es l'estructura idònia per guardar strings, el temps de consulta es el mes ràpid possible, l'únic inconvenient es que depèn de la implementació pot ocupar molt d'espai extra a memòria.

A l'hora d'implementar el hash s'ha utilitzat una taula de memòria estàtica de referencies a Nodes que contenen la informació de la clau i una referencia al següent node per guardar les col·lisions. El hash set s'ha implementat de forma genèrica.

En la implementació del trie s'ha utilitzat taules estàtiques de 26 posicions, cadascuna fa referencia a cada lletra del abecedari, i cada posició fa referencia a un altre arbre trie, d'aquesta manera, seguint recursivament les referencies de les taules, i segons la posició d'aquestes, podem guardar i consultar strings. Hem triat una implementació de memòria estàtica per millorar al màxim el temps d'execució a costa del cost espaial. Un petit inconvenient es que aquesta implementació no dona lloc a ser genèrica, tant sols pot guardar strings de caràcters de la 'a' a la 'z'.

Per construir una Interface comuna per a les dos implementacions també s'ha agut de sacrificar l'aspecte genèric, per això en el cas del hash set s'ha creat una classe extra que implementa la interface i l'únic que fa es construir un hash set que emmagatzemi strings.

Anàlisi i comparació del cost temporal i espaial

Anàlisi de la taula de dispersió:

Afegir(): O(1) -> O(n) si es produeix un increment de mida automàtic.

Esborrar(): O(1) -> O(n) si es produeix un decrement de mida automàtic.

Consultar(): O(1)

Per realitzar qualsevol de les operacions anteriors es realitza el mateix procediment que consisteix en calcular la posició a partir de la clau i la funció de hashing (temps constant) i a partir de la posició identificar el node / posició de entre tots els nodes que hi han col·lisions. Teòricament el temps d'aquestes operacions en el pitjor cas seria O(n) si tots els elements col·lisionen en una sola posició de la taula. No obstant la probabilitat de que es doni depèn de la funció de hash i la mida de la taula, i si la implementació es fa correctament aquesta probabilitat es despreciable.

Pel que fa al cost espaial s'han implementat funcions de increment i decrement de mida que s'executen al afegir o esborrar un element respectivament. Aquestes funcions s'asseguren que la mida (m) de la taula es, en el pitjor cas $m = 4n$ (factor de carrega = 0.25) i en el millor cas $m = \frac{4}{3}n$ (factor de carrega = 0.75). Cada cop que es crida aquesta funció s'ha de fer un *rehashing* i el cost d'aquesta operació es O(n).

Anàlisi de l'arbre trie:

Afegir(): O(1)

Consultar(): O(1)

Esborrar(): O(1)

De la mateixa manera que en la taula de dispersió, les operacions anteriors tenen un procediment molt similar. El cost real de les operacions depèn directament de la llargada de la string, aquest valor es considera constant per tant el cost temporal es O(1).

El principal problema amb la implementació triada es l'espai; utilitzant taules estàtiques podem veure que per guardar una sola paraula de 10 caràcters estem ocupant: 10 taules * 26 posicions * mida de referencia (64 bits) = 2080 bytes o 2 KB. L'espai emprat en memòria es podria millorar si en comptes d'utilitzar taules estàtiques s'utilitzessin taules de dispersió per guardar les referencies. No podríem utilitzar llistes dinàmiques ja que el propòsit inicial es tenir un random access de cost constant.