Progetto #3

Per quanto riguarda l'interfaccia grafica del progetto, abbiamo assunto che questa venga usato su un sistema embedded, allo stadio, in modo da permettere agli ospiti di fare analisi su determinati campionati.

Al fine di stampare l'elenco delle squadre dato il campionato, abbiamo optato nell'usare una tabella hash con separate chaining.

Un'istanza di una tabella hash ha principalmente due parametri che ne influenzano le prestazioni: la capacità iniziale e il load factor.

La capacità è il numero di bucket nella tabella hash, e la capacità iniziale è semplicemente lo stesso, ma appena creata la tabella hash.

Il fattore di carico è una misura di quanto è consentito che una tabella hash si riempia prima che la sua capacità è aumentata automaticamente. Quando il numero di inserimenti nella tabella hash supera il prodotto del load factor e la capacità corrente, la tabella hash viene rifatta, similmente alla struttura dati array ne viene raddoppiata la size, e la funzione hash adattata a questa nuova size.

I campionati contenuti nel file excel sono 22, quelli che dovevamo considerare ai fini del progetto sono 11, noi abbiamo supposto di dimensionare la nostra tabella hash per supportare tutti i campionati contenuti nel file excel.

Nel caso di separate chaining sappiamo da studi sperimentali che dobbiamo garantire almeno $\lambda \leq 0{,}9$ mentre per linear probing $\lambda \leq 0{,}5$, poiché il load factor è definito come $\lambda = \frac{n}{N}$, dove n è il numero di elementi della tabella hash, mentre N è il numero di bucket in separate chaining e in linear probing, N è il numero di elementi della tabella.

Facendo i calcoli sia per separate chaining che per linear probing noi otteniamo: $\frac{n}{0{,}9} \leq N_{sc} => \frac{22}{0{,}9} = 24{,}44 \leq N_{sc}$, dunque per separate chaining la tabella hash deve avere capacità di 25 bucket almeno, ovviamente scegliendo N maggiore di 25 si avrebbero prestazioni ancora

migliori, ma questo consumerebbe anche spazio aggiuntivo, non necessario per la nostra applicazione.

Facciamo lo stesso discorso per linear probing: $\frac{n}{0,5} \leq N_{lp} => \frac{22}{0,5} = 44 \leq N_{lp}$, dobbiamo avere una tabella hash con capacità 44.

Ci chiediamo inoltre, dopo quanto sarà ridimensionata la tabella hash? Nel caso di separate chaining: $\lambda_{sc} * N_{sc} = 23$, mentre nel caso di linear probing: $\lambda_{lp} * N_{lp} = 22$.

In entrambi i casi abbiamo vantaggi e svantaggi, nel caso del linear probing, teoricamente consumiamo meno spazio in quanto non abbiamo bisogno di spazio aggiuntivo per i bucket, ma possiamo inserire meno elementi prima di dover ridimensionare la tabella al fine di poter garantire basse collisioni.

Tipicamente le squadre di calcio sono già note alla lettura del file, poichè si ricavano dalla seguente formula:

$$N_{SQUADRE}\left(N_{SQUADRE} - 1\right) = N_{RIGHE} - 1$$

Questa equazione di secondo grado, chiaramente, risolta ci da due soluzione è necessario scegliere quella fisicamente accettabile, la seguente:

$$N_{SQUADRE} = \frac{1 + \sqrt{1 + 4 * N_{RIGHE}}}{2}$$

Dunque alla lettura del file abbiamo già note il numero di squadre per campionato, ciò ci ha fatto optare per usare una tabella hash con separate chaining.

E' vero che consumiamo più spazio per le strutture dati aggiuntive, ma è pur vero che rispetto a linear probing istanzieremo un numero di elementi della tabella hash minore.

Inoltre avendo scelto 0.9 come fattore di carico, la probabilità di avere collisioni è abastanza bassa.

Facciamo un esempio: per il campionato italiano il quale è costituito da 20 squadre, è necessario istanziare la tabella hash con capacità iniziale di 23 elementi, mentre per linear probing ben 40 elementi.

Abbiamo visto che sperimentalmente con separate chaining si verificano su questo caso massimo 3 collisioni, allungando il tempo di accesso di pochissimo.

Abbiamo dunque optato per questa soluzione, in ragione del fatto che il numero di squadre è noto ed è fisso.

Soluzione diversa abbiamo preso in considerazione per la struttura dati che rappresenta i campionati, per questa abbiamo usato una tabella hash con linear probing.

L'assunzione fatta è stata: è vero che con separate chaining consumo meno spazio, ma in linea teorica in un applicazione pratica potrei voler analizzare tutti i campionati del mondo, tipicamente le squadre per campionato sono poche e il tempo che si perde per accedervi con una tabella hash che utilizza separate chaining è ragionevole.

Nel caso dei campionati proprio perchè si può voler aggiungere un numero non noto a priori di campionati, lo spazio aggiuntivo e il tempo di accesso dovuto alle strutture dati linkate non è ragionevole.

Dunque, è vero che la struttura dati verrà raddoppiata prima, ma i vantaggi in termini di tempo di accesso, sono evidenti rispetto allo spazio consumato.

Poiché conosciamo il numero di giornate, e vi vogliamo accedere attraverso un indice, che è il numero di giornata, per le giornate è stata scelta la struttura dati lista di python, la quale ha complessità $O(1)$ sia per inserimenti che cancellazioni.

Al fine di risolvere il problema proposto dal progetto sono state necessarie delle assunzioni:

Chi calcola il calendario per un girone all'italiana, usa l'algoritmo di Berger, la prima assunzione che è stata fatta è che non è possibile rinviare più di una partita per giornata, questo perché neanche un operatore umano se guardasse i dati disponibili sarebbe in grado di capire quali sono le partite effettivamente rinviate e quali no.

Facciamo un esempio, se alla prima giornata venissero rinviate ben due partite, nelle giornate successive possiamo ritrovare le 4 squadre a giocare una delle seguenti partite, date dalle disposizioni di 4 su 2:

$$N_{partite} = \frac{4!}{4! * (4-2)!} = 12$$

Ben 12 partite, di cui nessuno senza avere il calendario originario potrebbe mai capire quali sono state rimandate e quali no.

La seconda assunzione fatta è stata: visto che alcuni campionati calcolano in modo diverso le giornate di un campionato, a seconda del campionato modifichiamo la formula di calcolo, per esempio per quello Scozzese.

Per implementare la seconda parte del progetto, essendo che find_kmp restituisce l'indice dell'inizio del primo pattern trovato in una data stringa, è bastato riapplicare find_kmp nuovamente nella stringa a partire dall'indice dopo il pattern trovato.

È stato necessario find_kmp al fine di poter migliorare l'efficienza temporale del metodo count_kmp.

In count_kmp calcoliamo i fail una sola volta, che passiamo a find_kmp.

Applicando k volte, find_kmp si riescono a contare tutte le occorenze del pattern in una data stringa, visto che Knuth-Morris-Pratt ha complessità $O(m+n)$, il nostro metodo per contare il numero di occorrenze avrà complessità $O(m+n)$.