Algoritmi e Principi dell'Informatica
Seconda Prova in Itinere

3 Febbraio 2016

Avvisi importanti
Il tempo a disposizione è di **1 ora e 45 minuti.**

Esercizio 1 (punti 6/15-esimi)

Si consideri il linguaggio $L = \{c^m a^{l_1} b\, a^{l_2} b \ldots a^{l_m} b \ldots a^{l_z} b\, d^n \mid n, m, l_i, z > 0, l_m = n\}$.
Per esempio ccaaabaaaababababdddd $\in$ L.
Si descrivano una macchina di Turing e una macchina RAM che accettano L, calcolandone le complessità spaziali e temporali, sia a criterio di costo costante che logaritmico.

Esercizio 2 (punti 11/15-esimi)

Si consideri la gestione delle tabelle hash mediante indirizzamento aperto. E' noto che in tal caso per eliminare un elemento della sequenza di ispezione associata a una chiave k non basta assegnargli il valore NIL, perché altrimenti si "romperebbe la catena". Un rimedio normalmente utilizzato in questi casi consiste nel marcare l'elemento come *Deleted*, in modo tale che la ricerca di altri elementi lungo la sequenza non si debba interrompere.
In questo modo però, a lungo andare si genera una forma di "*garbage*" che ovviamente diminuisce la memoria disponibile e peggiora anche le prestazioni temporali.
Si progetti un algoritmo di *Compattamento* il quale, per una data tabella T (per semplicità la si consideri una variabile globale), ricevendo come argomento una chiave k, elimini dalla *sequenza associata alla chiave k* tutti gli elementi marcati come *Deleted* ricompattando così la sequenza e recuperando la memoria inutilizzata.
NB: se l'algoritmo elimina anche altri elementi *Deleted* oltre a quelli indicati, il risulato è ugualmente accettabile (anzi, è addirittura preferibile); *purché non interrompa alcuna sequenza.*
Si rammenta che la notazione T[p], data una posizione p della tabella, restituisce NIL, *Deleted*, o *k*, ossia la chiave contenuta nella posizione, se questa non è né *Deleted* né vuota. Si assuma inoltre che la funzione *h(k, i)* sia del tipo *h(k, i) = (f(k) + g(i)) mod m* con *g(0) = 0* (non si tratta quindi di doppio hashing) con *f* e *g* funzioni calcolabili in tempo costante.
Si valuti la complessità asintotica dell'algoritmo.

Parte opzionale (ulteriori 2 punti)
L'algoritmo prodotto è applicabile anche al caso in cui la funzione *h* sia una funzione di tipo doppio hashing? Giustificare brevemente la risposta.

Tracce di soluzioni

Esercizio 1

Entrambe le macchine devono contare i *c* in ingresso, la MT in unario e la RAM in una cella es. M[1].
Alla prima *a*, si contano i gruppi di *a* separati da *b*, fino all'arrivo dell'm-esimo.
Si può usare M[1] o il nastro della MT a questo punto per contare il numero di *a*.
Si ignorano i successivi *a* e *b*, controllando solo che la sequenza finisca con *b*; fino all'arrivo dei *d*, che devono essere pari al numero memorizzato in M[1] o nel nastro.

Complessità:
RAM (costo costante): $T(n) = \Theta(n)$, $S(n) = \Theta(1)$
RAM (costo logaritmico): $T(n) = \Theta(n \log n)$, $S(n) = \Theta(1og\ n)$
MT: $T(n) = \Theta(n)$, $S(n) = \Theta(n)$

Esercizio 2

Descrizione informale dell'algoritmo.
L'approccio naturale al problema consiste nello scandire completamente la sequenza associata a una chiave k, ossia fino alla marca NIL; quando si incontra una posizione marcata Deleted si mette al suo posto la posizione corrente marcando questa a sua volta come Deleted; quando la posizione corrente raggiunge il NIL le eventuali posizioni intercorrenti tra l'ultima posizione aggiornata,ormai solo Deleted, e il NIL vengono tutte messe a NIL.
In questo modo però si rischia di imbattersi in posizioni occupate da altre sequenze: in tal caso sarebbe un errore spostarne il contenuto; quindi conviene accontonare questi elementi in una lista temporanea e marcarli come Deleted, per poi reinserirli ex-novo a compattamento ultimato. Siccome però la funzione di hash costruisce la stessa sequenza per ogni chiave con ugual valore di f(k), è possibile e opportuno eliminare tutti gli elementi delle sequenze che hanno la stessa origine.

Pseudocodice

```
compact(k)
/*compattamento della lista associata alla chiave k; T denota la tabella
(variabile globale) cui l'algoritmo viene applicato.
Sia h(k,i) la funzione di hash; T[p] indica il contenuto di T nella posizione p,
ossia NIL, Deleted, oppure la chiave presente in p.
In realtà il presente algoritmo "ripulisce" tutte le posizioni Deleted di tutte
le sequenze che hanno la stessa origine in h(k,0) mediante un semplice test;
poiché tali sequenze coincidono non si rischia di interrompere impropriamente
altre sequenze.
Per semplicità si assume che la sequenza termini comunque in un elemento NIL,
assunzione garantita, ad esempio, segnalando un overflow se in fase di
inserimento si giunge a un punto in cui h(k, i) = h(k, 0).*/
{
i = 0; PosCorrente = h(k,i);
while (T[PosCorrente] != NIL && T[PosCorrente] != Deleted)
      {i++; PosCorrente = h(k,i); }
if (T[PosCorrente] == NIL) return; //non ci sono elementi Deleted
else // T[PosCorrente] è Deleted
    {j = i; ProxDeleted = PosCorrente;
      while (T[PosCorrente] != NIL)
      {
            while (T[PosCorrente] != NIL)
                  if (T[PosCorrente] == Deleted) {j++; PosCorrente = h(k,j)};
                  else
                     if h(T[PosCorrente],0)!= h(k,0)
                        {inserisci T[PosCorrente] in una lista temporanea TempL;
                        T[PosCorrente] = Deleted;
                        /*in questo modo gli elementi che non fanno parte di una
                        delle sequenze che hanno origine in f(k) vengono
                        accontonati
                        j++; PosCorrente = h(k,j)
                        };

            /* si giunge a fine sequenza o al primo elemento non NIL e
            non Deleted di una sequenza che ha origine in h(k,0)*/
            if (T[PosCorrente] != NIL)
                  /*cioè PosCorrente appartiene a una sequenza che
                  ha origine in f(k)*/
                  {T[ProxDeleted] = T[PosCorrente];
                  T[PosCorrente] = Deleted;
                  i++; ProxDeleted = h(k, i)}
            else //l'elemento Deleted era l'ultimo della sequenza
                  {T[ProxDeleted] = NIL;}
      }
      /* a questo punto tutti gli elementi tra i e j sono deleted e si
      possono porre a NIL*/
      while (i != j) {T[h(k, i)] = NIL; i++;}

    }
/* a questo punto non resta che reinserire in tabella gli elementi
accantonati che appartenevano a sequenze non originanti in f(k)*/
Forall elements k in TempL Insert(T, k)
}
```

Si noti che entrambi gli indici, i, e j utilizzati nel programma percorrono tutta la sequenza che parte da h(k, 0), includendo anche gli elementi che non sono stati inserti a partire dalla chiave k ma che si trovano in posizione h(k, i) per qualche i. Quindi la complessità della prima parte dell'algoritmo, fino al reinserimento in T degli elementi di TempL, è O(n), n essendo la lunghezza complessiva della sequenza. La complessità di quest'ultima parte è O(p.m), dove p è la lunghezza di TempL e m

la lunghezza dell'intera tabella, poiché ogni inserimento, nel caso pessimo costa O(r), con 1 <= r <= m. E' però probabile che nel caso medio anche questa parte dell'algoritmo abbia una complessità più vicina a O(p) che a O(p.m) perché i nuovi inserimenti avverranno probabilmente in posizioni rese disponibili dalla passata precedente.
Quindi la complessità totale dell'algoritmo è O(n+ p.m). Come sempre nel caso delle tabelle hash nel caso pessimo "teorico" $p \approx n \approx m$ e quindi si ha una complessità $O(m^2)$, caso comunque solitamente lontano dal caso medio: normalmente si può ipotizzare $p < n << m$.

Parte opzionale
L'algoritmo sfrutta l'ipotesi che, se $h(k_1, 0) = h(k_2, 0)$ per qualche chiave k_1, k_2, la sequenza generata a partire da $h(k_1, 0)$ coincide con quella generata a partire da $h(k_2, 0)$; quindi confrontando semplicemente l'origine della sequenza originata da f(k) con quella originata da f(k') dove k' è la chiave contenuta nella posizione corrente, è possibile decidere se eliminare o meno l'elemento contenuto; si noti che in tal modo si eliminano in un colpo solo tutti gli elementi Deleted dell'unica sequenza originata in f(k) per tutte le chiavi con lo stesso valore di f(k), nel caso di clustering secondario.
Al contario, se si adotta un doppio hashing è possibile che esistano due chiavi k_1, k_2, e due indici i_1, i_2, tali che $h(k_1, i_1) = h(k_2, i_2)$, con $h(k_1, 0) = h(k_2, 0)$ ma appartenenti a sequenze diverse e quindi tali che si debba eliminare l'elemento appartenente all'una ma non all'altra.

Osservazione
Un algoritmo leggermente meno efficiente potrebbe più semplicemente accantonare in TempL tutti gli elementi non Deleted e poi reinserirli. In questo caso il p della formula precedente sarebbe maggiore e la passata di reinserimento in tabella rischierebbe di avvicinarsi maggiormente a una complessità quadratica. Al solito, però si tatta di considerazioni di tipo "euristico".

Soluzione alternativa
Lo pseudocodice seguente è sostanzialmente equivalente a quello precedente anche in termini di complessità, ma fa uso di tre liste separate, ossia gli indici della sequenza, le chiavi delle sequenze con origine in h(k,0), le chiavi delle sequenze non originanti in h(k,0).

```
compact_p(T, k):
    i = 0
    pos = hash(k, i, max)

    while T[pos] != NIL && i < m
        list_insert(indexes, pos)
        if T[pos] != Deleted
            currk = T[pos]
            if hash(currk, 0) == hash(k, 0)
                list_insert(keys, currk)
            else
                list_insert(others, currk)
        T[pos] = NIL
        i++
        pos = hash(k, i, max)

    curr = last(indexes)
    i = 0
    for x in keys
        T[curr.value] = x
        curr = curr.prev
    for x in others
        hash_insert(T, x)
```