Appunti Algoritmi e Strutture Dati

Luca Seggiani

30 Aprile 2024

1 Dizionari

Un dizionario è una struttura che si discosta dal concetto (adottato finora) di indirizzamento diretto. Conviene usare dizionari quando la dimensione del dominio degli input è di molto maggiore alla dimensione di archiviazinoe che abbiamo a disposizione. Il dizionario può essere implementato attraverso il meccanismo appena visto dell'hashing, e quindi della scrittura in memoria di coppie chiave-valore: questa struttura dati prende nello specifico il nome di *hash table* (tabella hash).

Hash table semplice

Sviluppiamo una tabella hash che gestisce interi maggiori di 0, usa chiavi coincidenti coi valori, e la funzione modulo come funzione hash. Per convenzione, prendiamo 0 come valore vuoto. Una possibile classe che implementa la tabella è:

```
class HashTable {
  int* _table;
  int _size;

public:
  HashTable(int size);
  bool insert(int key);
  void print();
  int hash(int key);
};
```

dove ci siamo dotati di un'array (in memoria dinamica) di interi, un intero rappresentante la dimensione dell'array, e le funzioni membro HashTable(), costruttore, insert(), che inserisce coppie chiave-valore (qua coincidenti) nella hash table, print() che stampa i contenuti della tabella, e hash, che è la funzione hashing stessa.

- **Costruttore**
  Il costruttore avrà la forma:

```
HashTable::HashTable(int size) {
  _table = new int[size];
  _size = size;
  memset(_table, 0, suze * sizeof(int));
}
```

Notiamo l'uso della funzione di libreria memset(address, value, size), che serve ad inizializzare più velocemente la nostra array. La memset prende come argomenti un indirizzo, un valore, e una dimensione che determina il blocco di memoria contigua a partire dall'indirizzo da riempire col valore fornito (qui 0, che segnala l'elemento vuoto).

- **Hash**
  La funzione di hashing, come già detto, avrà la forma:

```
  int hash(int key) {
    return key % _size;
  }

```

- **Insert**
  La funzione dovrà, in ordine:
  - Trovare l'indice possibile tramite la funzione hash;
  - Controllare se la posizione è già occupata, e adoperare in tal caso misure preventive;
  - In caso contrario, inserire l'elemente assegnando la chiave all'indirizzo trovato.

  Una possibile implementazione di questo algoritmo sarà:

```
bool HashTable::insert(int key) {
  int index = hash(key);
  if(_table[index] != 0)
    return false;
  _table[index] = key;
  return true;
}
```

Si palesa il problema di gestire eventuali collisioni. L'approccio appena utilizzato (quello di evitare completamente l'inserimento nel caso non sia possibile) non è infatti ottimale. Un'approccio possibile è quello di trasformare ogni elemento del vettore in una lista, dove gli oggetti in collisione verrano memorizzati successivamente uno dopo l'altro. Questo significherà trasformare il nostro vettore di interi in un vettore di

puntatori ad intero, che farà da radice per una lista (volendo anche bidirezionale, agevola le eliminazione) di interi. Chiamiamo questa lista *lista di trabocco*. Mettiamo il tutto in uno struct:

```
struct Elem {
  int key;
  Elem* next;
  Elem* prev;
  Elem() : next(NULL), prev(NULL) {}
};
```

e a questo punto ridefiniamo la classe HashTable:

```
class HashTable {
  Elem** _table;
  int _size;

public:
  HashTable(int size);
  bool insert(int key);
  void print();
  void printOccupancy();
  int hash(int key);
  bool find(int key);
};
```

dove abbiamo aggiunto le funzioni printOccupancy(), che restituisce l'occupazione di ogni chiave del vettore; e la find, che va a cercare una chiave (eseguendo lo scorrimento linare della lista corrispondente alla sua chiave). Questo ci permette di riscrivere la insert come:

```
bool HashTable::insert(int key) {
  int index = hash(key);
  Elem* n = new Elem();
  n->key = key;
  n->next = _table[index];
  if(n->next != NULL)
    n->next->prev = n;
  _table[index] = n;
  return true;
}
```

e di scrivere la find come:

```
bool HashTable::find(int key) {
  int index = hash(key);
  Elem* pun = _table[index];
  while(pun != nullptr) {
    if(pun->key == key)
      return true;
```

```
    pun = pun->next;
  }
  return false;
}
```

Possiamo adesso fare una considerazione: potrebbe essere utile usare, invece di una lista, una struttura dati più avanzata che permetta di velocizzare gli scorrimenti dei singoli elementi del vettore, oppure semplicemente di mantenere l'ordinamento della lista in fase di inserimento. Notiamo però che, come sempre, un operazione di questo tipo richiede un tradeoff: possiamo velocizzare le ricerche, ma a costo di rallentare gli inserimenti.

Hashing di stringhe

Vediamo come realizzare questo tipo di struttura dati sulle stringhe. Per la funzione di hashing, potremmo pensare di usare la prima lettera:

```
int hash(string key) {
  return key[0] % _size;
}
```

Notiamo però che quest'approccio è assolutamente terribile per l'occupazione del vettore: molto probabilmente andremo a riempire esageratamente solo alcuni indici (quante frasi iniziano con la lettera x?). Un'approccio migliore potrebbe essere quello di sommare i valori di ogni carattere:

```
int hash(string key) {
  int index = 0;
  for(int i = 0; i < key.length(); i++) {
    index += key[i];
  }
  return index % _size;
}
```

Che ci assicurebbe di avere una distribuzione più uniforme di chiavi.

Considerazioni sulla funzione di hash

Dalla funzione di hash si possono richiedere diverse caratteristiche per diverse applicazioni. In generale, possiamo parlare di:

- **Uniformità**, ovvero la copertura (o non) che ha la funzione di hashing per tutti i valori del dominio (nell'esempio precedente abbiamo visto come non è conveniente concentrare più chiavi sugli stessi indici).
- **Non reversibiilità**, ovvero la difficoltà di ricondurre un indirizzo alla sua chiave. Questo è particolarmente importante nell'ambito della sicurezza informatica.

In generale, si può anche dire che la maggior parte delle funzioni di hash lavorano sulla rappresentazione binaria delle chiavi.

La versione STL

La STL ci fornisce un'implementazione della funzione di hash: la cosiddetta stl::map:

```
stl::map <key_T, obj_T > table;
```

crea una tabella di oggetti di tipo obj_T con chiavi di tipo key_T. Potremo a questo punto fare le operazioni:

```
//inserimento
table['uno'] = "Valore uno";
//ricerca
table.find('uno');
```