Map & Dictionary

Corso: Strutture Dati

Docente: Annalisa De Bonis

Definizione informale

- Il TDA Map memorizza coppie formate da una chiave k e da un valore v
  - La coppia è chiamata entry
- Ogni chiave deve essere unica
  - La chiave associata ad un oggetto può essere vista come una sorta di indirizzo unico dell'oggetto
- Operazioni principali
  - Ricerca
  - Inserimento
  - Cancellazione

Applicazioni delle mappe

- Memorizzare i dati relativi agli studenti di un'università
  - Chiave: numero di matricola
  - Valore: un oggetto contenente le informazioni relative allo studente, quali i dati anagrafici, gli esami superati, il piano di studi

I metodi del TDA Map

- size()
- isEmpty()
- get(k)
  - Se esiste un'entry con chiave k, restituisce il valore associato a k, altrimenti restituisce null
- put(k, v)
  - Se la mappa non ha un entry con chiave uguale a k, allora aggiunge l'entry (k,v) alla mappa e restituisce **null**; altrimenti rimpiazza con v il valore esistente associato alla chiave k e restituisce il vecchio valore

I metodi del TDA Map

- remove(k)
  - Cancella dalla mappa l'entry con chiave uguale a k e restituisce il valore associato a k; se la mappa non ha tale entry, viene restituito il valore null
- keys()
  - Restituisce una collezione iterabile contenente tutte le chiavi della mappa
- values()
  - Restituisce una collezione iterabile contenente tutti i valori associati alle chiavi della mappa
- entries()
  - Restituisce una collezione iterabile contenente tutti le entrate della mappa

Interfaccia Map

```
public interface Map <K,V>{
// Restituisce il numero degli elementi nella mappa
public int size();

// Restituisce true se la mappa è vuota
  public boolean isEmpty();

// Inserisce una coppia (chiave,valore) nella mappa,
// rimpiazzando il valore precedente se la chiave è già nella mappa
public V put(K key, V value)  throws InvalidKeyException;
```

Interfaccia Map

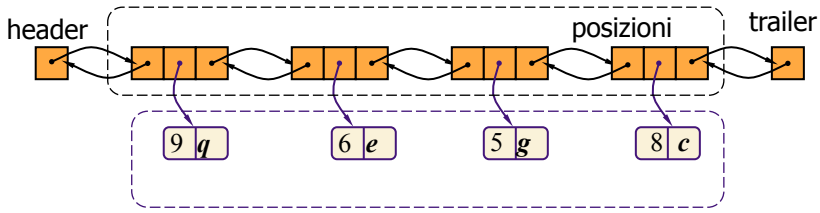
```
//Restituisce il valore associato alla chiave key
  public V get(Kkey) throws InvalidKeyException;

// Rimuove la coppia (chiave,valore) specificata da key
  public V remove(K key)
                    throws InvalidKeyException;

// Restituisce una collezione iterabile di tutte le chiavi della mappa
  public Iterable<K> keys();

// Restituisce una collezione iterabile di tutti i valori della mappa
  public Iterable<K> values();

// Restituisce una collezione iterabile di tutte le entrate della mappa
  public Iterable <Entry<K,V>>entries();

}
```


Confronto con java.util.Map

TDA Map	java.util.Map
size()	size()
isEmpty()	isEmpty()
get(*k*)	get(*k*)
put(*k,v*)	put(*k,v*)
remove(*k*)	remove(*k*)
keys()	keySet()
values()	values()
entries()	entrySet()

Una semplice implementazione basata su lista

- Possiamo implementare efficientemente il TDA Map usando una lista non ordinata
  - Memorizziamo le voci della mappa in una lista S (basata su una lista doppiamente linkata), in un ordine arbitrario



La classe MyEntry

- Le entrate sono istanze della classe MyEntry

```
protected static class MyEntry <K,V>implements
  Entry<K,V> {
protected K key;
protected V value;
public MyEntry(K k, V e) {
  key = k; value = e;
}
public K getKey() { return key; }
public V getValue() { return value; }
}
```

La classe ListMap

```
public class ListMap<K,V> implements Map<K,V> {

private PositionList<Entry<K,V>> L;

public ListMap(){
   L =new NodePositionList<Entry<K,V>>();
}
…
```


La classe ListMap: il metodo get

```
public V get(K key) throws InvalidKeyException{
 checkKey(key);
 for(Position<Entry<K,V>> p: L.positions()){
   Entry<K,V> e= p.element();
   if(e.getKey().equals(key))  return e.getValue();
   }
  return null;
}
```

Complessità O(size)

La classe ListMap: il metodo put

```
public V put(K key, V value) throws InvalidKeyException {
checkKey(key);
 for(Position<Entry<K,V>> p: L.positions()){
   Entry<K,V> e= p.element();
   if(e.getKey().equals(key)){
            V v = p.element().value()
            L.set(p, new MyEntry<K,V>(key,value));
            return v;
}   }
  L.addLast(new MyEntry<K,V>(key,value));
  return null;
}
```

Complessità O(size)

La classe ListMap: il metodo remove

```
public V remove(K key) throws InvalidKeyException {
checkKey(key);
 for(Position<Entry<K,V>> p: L.positions()){
   Entry<K,V> e= p.element();
   if(e.getKey().equals(key)){
            V v = p.element().value()
            L.remove(p);
            return v;
}}
 return null;
}
```

Complessità O(size)

Il TDA dizionario

- Il TDA dizionario è una collezione di entrate del tipo (chiave, elemento)
- Operazioni principali
  - Ricerca
  - Inserimento
  - Cancellazione
- **Si possono avere più elementi con la stessa chiave**
- Applicazioni:
  - Dizionari linguistici: collezione di coppie (parola,definizione)
  - Sistema DNS: collezione di coppie (nome dominio,indirizzo IP),
    - Es.: (datastructures.net, 128.148.35.101)

I metodi del TDA dizionario

- Metodi di accesso
  - find(k)
    - Restituisce un'entrata del dizionario D con chiave k. Se non esiste alcuna entrata con chiave k restituisce null.
  - findAll(k)
    - Restituisce una collezione iterabile di tutte le entrate che hanno chiave k.
  - entries( )
    - Restituisce una collezione iterabile di tutte le entrate del dizionario

I metodi del TDA dizionario

- Metodi di aggiornamento
  - insert(k, o)
    - Inserisce e restituisce in output l'entrata (k, o)
  - remove (e)
    - Rimuove e restituisce in output l'entrata e. Se e non appartiene al dizionario restituisce null
- Metodi generici
  - size(),
  - isEmpty()

- Se ai metodi find, findAll e insert viene passata in input una chiave non valida allora si verifica un errore
- Se al metodo remove viene passata un'entrata non valida si ha un errore

L'interfaccia Dictionary

```
public interface Dictionary<K,V> {
public int size();
public boolean isEmpty();
public Entry<K,V> find(K key)  throws InvalidKeyException;
public Iterable<Entry<K,V>> findAll(K key)
        throws InvalidKeyException;
public Entry<K,V> insert(K key, V value)
        throws InvalidKeyException;
public Entry<K,V> remove(Entry<K,V> e)
        throws InvalidEntryException;
public Iterable<Entry<K,V>> entries();
}
```

Tipi di dizionari

- Due tipi di dizionari
  - Dizionari non ordinati
    - Non è definita alcuna relazione di ordinamento sulle chiavi
    - Si può stabilire sono se due chiavi sono uguali o meno
  - Dizionari ordinati
    - Sulle chiavi è definita una relazione di ordine totale
    - Al costruttore del dizionario viene passato un comparatore come argomento
    - Si possono definire metodi addizionali che fanno riferimento all'ordinamento delle chiavi

Log file

- Un log file è un dizionario non ordinato implementato con una lista (non ordinata)
  - Le entrate sono memorizzate nella lista in un ordine arbitrario
- insert richiede tempo $O(1)$ se implementata con addFirst o addLast
- find and remove richiedono tempo $O(n)$ in quanto nel caso pessimo, che si verifica quando la chiave non è presente nel dizionario, si scandisce l'intera sequenza per trovare l'entrata con la chiave desiderata
- E' indicato implementare un dizionario come log file solo se si tratta di un dizionario di piccola dimensione o nel caso in cui si effettuano molti inserimenti e poche operazioni di cancellazione e di ricerca.
  - Esempio: I log file mantenuti dai sistemi operativi

Ordered Search table

- Una ordered search table è un dizionario ordinato implementato con un array list (ordinato)
  - Le entrate vengono imagazzinate in un array list nel quale sono ordinate in base al valore delle chiavi
  - Si usa un comparatore esterno per le chiavi
- find richiede tempo O(log n) con la ricerca binaria
- insert richiede tempo O(n) perchè occorre shiftare a destra tutte le entrate con chiave maggiore della chiave della nuova entrata
- remove richiede O(n) perchè occore shiftare a sinistra tutte le entrate che seguono l'entrata che deve essere rimossa
- E' indicato implementare un dizionario come ordered search table solo se si tratta di un dizionario piccolo o se si effettuano molte operazioni di ricerca e poche operazioni di inserimento e cancellazione

Tabella Hash

- Modo di implementare il TDA Map e il TDA Dictionary mediante un array
- Problemi dell'implementazione semplice con un array A:
  - Un'entrata con chiave k viene memorizzata nella cella A[k]
    - Le chiavi devono essere **di tipo intero** e **a due a due distinte** (questa seconda condizione è verificata solo nel caso di Map)
    - **L'array A deve avere dimensione pari alla chiave più grande del dizionario**

Tabella Hash

- Una tabella hash per un dato tipo di chiavi consiste di
  - Una funzione hash h
  - Un array (tabella) di dimensione chiamato bucket array

Funzioni hash

- Una funzione hash h mappa un insieme chiavi di un certo tipo in un intervallo prefissato di interi [0, N–1]
- Esempio: h(x) = x mod N
  - h(x) in questo caso è definita per chiavi di tipo intero
- h(x) è chiamato valore hash di x
- Lo scopo di una funzione hash è di distribuire le chiavi uniformemente nell'intervallo [0, N-1]

Funzioni hash

- Si usa il metodo hashValue per computare i valori hash delle chiavi
- La funzione hashValue computa il valore da assegnare ad una certa chiave k in due fasi:
  - fase 1: la chiave k viene "mappata" in un intero chiamato hash code di k
  - fase 2: l'hash code di k viene "mappato" in intero nel range di interi [0, N-1] (N = capacità bucket array) mediante una funzione di compressione

Hash code

- Non deve necessariamente essere un intero nel range [0,N-1]
- Deve essere computato in modo da evitare quanto più possibile le collisioni
- Due chiavi uguali devono avere lo stesso hash code

Hash code

- La classe Object di Java fornisce il metodo hashCode che restituisce un int (32 bit) che rappresenta l'oggetto
  - Bisogna stare attenti quando si usa il metodo hashCode di Object in quanto in molte implementazioni di Java restituisce un intero ottenuto manipolando l'indirizzo in memoria dell'oggetto

    ➔ oggetti uguali memorizzati in locazioni diverse hanno hash code diversi
  - In molti casi è meglio riscrivere il metodo hashCode nelle classi usate come tipi per le chiavi

Modi di computare l'hash code

- Cast ad intero: Se le chiavi sono di un tipo la cui rappresentazione binaria richiede al più 32 bit allora si può semplicemente reinterpretare la rappresentazione binaria della chiave come un intero
- Esempio: se le chiavi sono di tipo byte, short, o char si può fare il cast ad int di questi tipi. Se le chiavi sono di tipo float si può usare il metodo **Float.floatToIntBits**(k) per trasformare k in un intero

Modi di computare l'hash code

- Se la rappresentazione binaria delle chiavi è più lunga di 32 bit allora il cast ad int farebbe perdere parte dell'informazione
- Esempio:
  - Se le chiavi sono di tipo long (64 bit) allora con il cast ad int vengono scartati i 32 bit più significativi

  ➔ due chiavi che differiscono solo nei 32 bit più significativi avrebbero lo stesso hash code

Modi di computare l'hash code

- Somma delle parti: la rappresentazione binaria di una chiave k viene suddivisa in un certo numero numero m di segmenti che vengono visti come m interi $k_0,..,k_{m-1}$ L'hash code è ottenuto sommando questi m interi.
- Esempio:
  - Se le chiavi sono di tipo long (64 bit) allora la chiave può essere suddivisa in 2 segmenti di 32 bit, ciascuno dei quali può essere interpretato come un int
  - l'hash code è ottenuto sommando l'intero corrispondente ai primi 32 bit all'intero corrispondente agli ultimi 32 bit.

```
int hashCode(long i) {return (int)((i >> 32) + (int) i);}
```

Modi di computare l'hash code

- Nel metodo della somma delle parti i bit di una chiave k vengono suddivisi in un certo numero numero m di segmenti e l'hash code è ottenuto sommando gli m interi $k_0,..,k_{m-1}$ corrispondenti a questi m segmenti
- Problema: due chiavi diverse k e k' potrebbero avere lo stesso hash code in quanto le due m-uple $(k_0,..,k_{m-1})$ e $(k'_0,..,k'_{m-1})$ potrebbero essere uguali a meno di una permutazione delle componenti.
  - Esempio: le chiavi sono stringhe e il valore hash di una chiave è ottenuto sommando i valori ASCII dei caratteri della chiave ➔ le stringhe "abcd" e "cbad" hanno lo stesso hash code

Modi di computare l'hash code

- Hash code polinomiale: si sceglie una costante a diversa da 0 e da 1 e si computa il polinomio

$$p(a) = k_0a^{m-1}+k_1a^{m-2}+ \ldots k_{m-2}a+k_{m-1}$$

- In questo modo ogni componente della m-upla $(k_0,..,k_{m-1})$ dà un contributo che dipende non solo dal suo valore ma anche dalla sua posizione nella m-upla
- Il polinomio $p(a)$ può essere valutato in tempo $O(n)$ usando la regola di Horner:
  - I seguenti polinomi sono computati successivamente: ciascuno viene computato a partire dal precedente in tempo $O(1)$
  - $p_0(a) = k_0$
  - $p_i(a) = k_i + ap_{i-1}(a)$ per i = 1, 2, …, m -1
  - Si ha $p(a) = p_{m-1}(a)$

Funzione di compressione

- i = hash code di k; N = capacità bucket array
- Una buona funzione di compressione dovrebbe garantire che la probabilità che due chiavi vengano "mappate" nello stesso bucket è 1/N.
- Metodo della divisione
  - |i| mod N , collisioni meno probabili se N è un primo
    - Genera molte collisioni se molte chiavi hanno un hash code della forma
      pN+q per diversi valori di p
- Metodo MAD (multiply, add and divide)
  - |ai+b| mod N, dove a e b sono costanti intere scelte in modo casuale tali che a>0, b ≥ 0, a mod N ≠0
  - N è primo

Dizionari implementati con tavole hash

- l'entrata (k, o) è memorizzata nell'entrata di indice i = h(k)
- Si verifica una collisione quando due chiavi del dizionario hanno lo stesso valore hash

  Schemi di risoluzione di una collisione:
  - Separate Chaining: le entrate che hanno generato la collisione vengono memorizzate in una sequenza
  - Open addressing: una delle entrate che hanno generato la collisione viene sistemata in un'altra cella della tabella

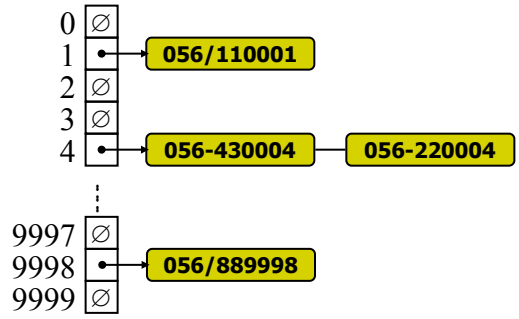
Separate Chaining

- Ciascuna entrata dell'array è un riferimento ad una sequenza
- A[i] → S_i
  - S_i memorizza tutte le entrate le cui chiavi hanno valore hash uguale ad i
  - S_i può essere visto come un piccolo dizionario implementato con una lista (log file)

Esempio

- tabella hash per un dizionario che memorizza entrare della forma (matricola, nome studente)
  - Array di dimensione N=10000
  - Valore hash di h(x)= ultime 4 cifre di x
  - Risoluzione delle collisioni con chaining

L'entry con matricola 056-430004 e quello con matricola 056-220004 collidono



Open Addressing

- Il metodo dell' *open addressing* risolve le collisioni sistemando l'entrata che provoca una collisione in un'altra locazione dell'array
- Questo metodo è utile quando non si ha molto spazio a disposizione
  - Non si utilizzano strutture dati ausiliare a differenza di quanto avviene nel separate chaining

Open Addressing con Linear Probing

- Il metodo del *linear probing* risolve le collisioni sistemando l'entry che provoca una collisione nella prossima cella disponibile della tabella (vista come struttura circolare)

  $h(k,i)=(h'(k)+i) \bmod N,\ i=0,\ldots,N-1$

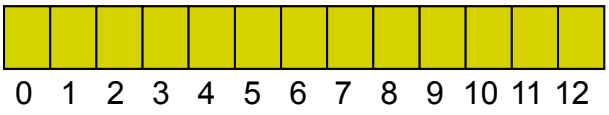
- Ogni volta che viene ispezionata una cella per vedere se è libera si dice che si effettua "probe"
- Problema:
  - Primary clustering: le entrate tendono ad essere disposte in lunghi blocchi di celle consecutive aumentando così il numero di probe necessari per cercare un'entrata o inserire una nuova entrata

Metodi alternativi al linear probing

- *quadratic probing:* $h(k,i)=(h'(k)+c_1 i+c_2 i^2) \bmod N$, c_1, $c_2 \neq 0$ costanti
  - N, c_1, c_2 devono essere scelti in modo appropriato altrimenti potrebbero esserci locazioni inutilizzate
  - Problema: come nel linear probing se $h'(k_1) = h'(k_2)$ allora $h(k_1,i)= h(k_2,i)$ per ogni i ➔ le sequenze dei probe effettuati per k_1 e per k_2 sono le stesse ➔ secondary clustering
- *double hashing:* $h(k,i)=(h_1(k)+ih_2(k)) \bmod N$, $h_2(k) \neq 0$ per ogni k

Esempio

- h(x) = x mod 13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine



La ricerca di un'entrata con Linear Probing

- Si comincia facendo il probe della cella h(k)
- Si fa il probe di celle consecutive fino a che
  - Viene trovata un'entrata con chiave k

  oppure
  - Viene trovata una cella vuota
  - E' stato effettuato il probe di tutte le celle senza aver trovato un'entrata con chiave k

```
Algorithm
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
        else if c.key () = k
            return c //return c.value per Map
        else
            i ← (i + 1) mod N
            p ← p + 1
    until     p = N
    return null
```


Cancellazioni con Linear Probing

- Per gestire le cancellazioni, isi usa un'entrata speciale chiamata AVAILABLE che rimpiazza gli elementi cancellati
- remove (e) è realizzata in 3 passi
  - Si cerca l' entry e
  - Se e è stata trovata, la si rimpiazza con AVAILABLE e la restituisce in output
  - Altrimenti restituisce null
- AVAILABLE indica all'algoritmo di inserimento che la cella è libera
- Si usa AVAILABLE al posto di ∅ per evitare che l'algoritmo di ricerca si arresti quando fa il probe di una cella in cui è avvenuta una cancellazione

Inserimenti con linear probing

- insert(k, o) è realizzata in 3 passi
  - Si comincia con la cella h(k)
  - Si effettua il probe di celle consecutive fino a che
    - Si trova una cella **i** che è vuota o contiene l'entrata speciale **AVAILABLE**
  - Si inserisce (k,o) nella cella **i**

  oppure
    - È stato fatto il probe di tutte le celle senza trovarne una disponibile per cui si lancia un'eccezione di tabella piena oppure si trasferiscono le entrate in una tabella più grande

Load factor

- N = dimensione bucket array
- n = numero entrate nella tabella
- Load factor: X = n/N
  - Bisogna fare in modo che X< 1
  - Analisi sperimentali suggeriscono
    - X < 0.5 per open addressing
    - X < 0.9 per separate chaining
      - java.util.HashMap mantiene X ≤ 0.75
  - Per mantenere X al di sotto di una certa soglia si effettua il ***rehashing*** delle entrate in una tabella più grande
    - Conviene che la dimensione della nuova tabella sia almeno 2 volte quella della vecchia tabella (si veda analisi ammoritizzata nella lezione sul TDA Arra List)

Esercizi

- Implementare il TDA Map tramite
  - Un'istanza di PositionList
  - Una tabella hash che usa il metodo dell'open addressing con linear probing per risolevere le collisioni
- Scrivere un programma per testare tutti i metodi delle implementazioni
- Assumendo che un utente non inserisca mai entrate con la stessa chiave, illustrare come si possa usare una mappa per implementare il TDA Dictionary

Esercizi

- Implementare una tabella hash usando il metodo del chaining per risolvere le collisioni
  - Ciascuna locazione del bucket array è un riferimento ad un log-file
- Implementare una tabella hash usando il metodo dell'open addressing con linear probing per risolvere le collisioni