Priority Queue

Corso: Strutture Dati

Docente: Annalisa De Bonis

Definizione informale

- Una coda a priorità è un contenitore di elementi a ciascuno dei quali è assegnata una chiave
  - La chiave viene assegnata nel momento in cui l'elemento è inserito nella coda
  - Le chiavi determinano la priorità degli elementi, ovvero l'ordine in cui vengono rimossi dalla coda
- Applicazioni
  - Viaggiatori in standby
  - Processi in attesa di usare una risorsa condivisa
  - Struttura ausiliaria di algoritmi

Priority scheduling

- Ad ogni processo è assegnata una priorità
- I processi in attesa di essere eseguiti sono inseriti in una coda priorità
- Viene estratto dalla coda ed eseguito il processo con priorità più grande
- Problema
  - Starvation: i processi con priorità più bassa non vengono mai eseguiti
- Soluzione
  - Aging: le priorità dei processi in coda vengono gradualmente aumentate

Esempi dell'uso della coda a priorità cose struttura dati ausiliaria in un algoritmo

- Prim e Dijkstra
- Algoritmi di ottimizzazione *greedy*
- Scelta greedy basata sul valore delle chiavi assegnate ai vertici
- I vertici del grafo vengono inseriti in una coda a priorità
  - Ad ogni passo viene estratto dalla coda il vertice con priorità più alta (chiave più piccola)

Relazione di ordine totale (≤)

- Proprietà
  - Riflessiva:
    $$x \leq x$$
  - Antisimmetrica:
    $$x \leq y \wedge y \leq x \Rightarrow x = y$$
  - Transitiva:
    $$x \leq y \wedge y \leq z \Rightarrow x \leq z$$
- Su qualsiasi insieme finito sono sempre definiti sia il max che il min

Il TDA PriorityQueue

- Contenitore di oggetti
- Ogni oggetto è una coppia (key, element)
- Metodi principali
  - insert (k, o)
    inserisce l'entrata (k, o) restituendola in output; se k non è una chiave valida si verifica un errore
  - removeMin()
    rimuove e restituisce l'entrata con chiave (key) più piccola; se la coda è vuota si verifica un errore

Il TDA PriorityQueue

- Metodi aggiuntivi
  - min(P)
    restituisce, senza rimuoverla, l'entrata con chiave più piccola della coda a priorità; se la coda è vuota si verifica un errore
  - size(P), isEmpty(P)

Design Pattern

- Come si fa ad associare le chiavi ai rispettivi elementi?
  - Si usa il design pattern composizione
- Come si confrontano le chiavi in modo da individuare la chiave più piccola?
  - Si usa il design pattern comparatore

Il design pattern Composizione

- Definisce un singolo oggetto come composizione di due o più oggetti
  - E' possibile definire composizioni di oggetti costituite da altre composizioni di oggetti in modo da formare una struttura gerarchica basata sulla composizione
  - Implementazione: occorre definire una classe che memorizza ciascun oggetto in una sua variabile di istanza e fornisce metodi per accedere e modificare queste variabili

Il pattern Composition : l'interfaccia entry

```
public interface Entry<K,V> {
  public K getKey();
  public V getValue();
}
```

L'interfaccia PriorityQueue

```
public interface PriorityQueue<K,V> {
public int size();
public boolean isEmpty();
public Entry<K,V> min() throws
   EmptyPriorityQueueException;
public Entry<K,V> insert(K key, V value) throws
   InvalidKeyException;
public Entry<K,V> removeMin() throws
   EmptyPriorityQueueException;
}
```


Design Pattern Comparatore

- Fornisce le regole in base alle quali effettuare il confronto delle chiavi
- Ci sono diversi modi di confrontare le chiavi
  - A seconda del tipo
    - 5<13 se 5 e 13 sono oggetti di tipo integer
    - 5>13 se 5 e 13 sono stringhe confrontate lessicograficamente
  - A seconda dell'uso
    - Chiavi costituite da punti del piano possono essere ordinate in base a una qualsiasi delle due coordinate.
      - Da sinistra a destra (in base alla prima coord.) (4,5) < (8,3)
      - Dal basso in alto (in base alla seconda coord.) (8,3) < (4,5)

Design Pattern Comparatore

- Alternative:
  - Implementare una classe priority queue per ciascun tipo di chiave che si vuole usare e per ciascun modo di confrontare ciascun tipo di chiave
    - Problema: non è un metodo generale e richiede di riscrivere molto codice
  - Usare chiavi che sono istanze di classi che contengono metodi di confronto
    - Problema: il metodo di confronto dipende dal contesto e potrebbe non essere stato previsto al momento della creazione della classe chiave

Design Pattern Comparatore

- Un oggetto comparatore è esterno alle chiavi
- Una generica coda a priorità usa un comparatore ausiliario per confrontare due chiavi
  - Quando viene creata una coda a priorità, viene passato in input al costruttore il comparatore.
  - Se necessario, il comparatore usato da una coda a priorità può essere rimpiazzato da un nuovo comparatore.

Il TDA Comparator

- Metodi
  - int compare(E x, E y)

    restituisce:
    - i<0 se x<y
    - i=0 se x=y
    - i>0 se x>y

    Provoca un errore se x e y non sono confrontabili

Java fornisce l'interfaccia java.util.Comparator

- Include anche il metodo equals per confrontare un comparator con un altro comparator

La classe Point2D

```
public class Point2D {
 protected int xc, yc;

public Point2D(int x, int y) {
   xc = x;
   yc = y;
}

 public int getX() { return xc; }
 public int getY() { return yc; }
}
```

Ordinamento lessicografico di punti del piano

```
public class Lexicographic<E extends Point2D> implements
    java.util.Comparator<E> {
    public int compare(E a, E b) {
    if (a.getX() != b.getX())
      return a.getX() - b.getX();
    else return a.getY() - b.getY();
 }
}
```


Il comparatore di default

```
public class DefaultComparator<E> implements
    Comparator<E> {
public int compare(E a, E b) throws
    ClassCastException {
    return ((Comparable<E>) a).compareTo(b); }
}
```

- Tra le classi che implementano l'interfaccia java.lang.Comparable, ricordiamo: String, Integer, Long, Double, Short, Byte, ecc.

Implementazione PriorityQueue con una lista non ordinata

- Memorizza gli elementi della coda in una lista in un ordine qualsiasi
- Complessità:
  - insert richiede tempo $O(1)$ in quanto possiamo semplicemente inserire l'elemento alla fine della lista eseguendo una insertLast()
  - removeMin, min richiedono tempo $O(n)$ in quanto bisogna scorrere tutta la lista per determinare l'elemento con chiave minima

Implementazione di PriorityQueue con lista non ordinata

```
public class UnsortedListPriorityQueue<K,V> implements
    PriorityQueue<K,V> {
  protected PositionList<Entry<K,V>> entries;  //lista contenente le entrate
  protected Comparator<K> c;

...
```

Implementazione PriorityQueue con una lista ordinata

- Memorizza gli elementi della coda in una lista per valore di chiave in ordine non decrescente
- Complessità:
  - Insert richiede tempo $O(n)$ in quanto occorre trovare il posto dove inserire l'oggetto in modo da mantenere l'ordine non decrescente delle chiavi
  - removeMin, min richiedono tempo $O(1)$ in quanto la chiave minima è all'inizio della lista

L'implementazione di Priority Queue con lista ordinata

```
public class SortedListPriorityQueue<K,V> implements
    PriorityQueue<K,V> {
   //lista contenente le entrate
    protected PositionList<Entry<K,V>> entries;
    protected Comparator<K> c;
```

Continua nella prossima slide

Implementazione di Priority Queue con lista ordinata

```
//classe interna che definisce il tipo delle entrate
  protected static class MyEntry<K,V> implements Entry<K,V> {
   protected K k; // chiave
   protected V v; // valore
   public MyEntry(K key, V value) { k = key; v = value; }
  // metodi dell'interfaccia
  public K getKey() { return k; }
  public V getValue() { return v; }
 }
```

Continua nella prossima slide

Implementazione di Priority Queue con lista ordinata

```
/** crea una coda a priorità con comparatore di default. */
public SortedListPriorityQueue () {
entries = new NodePositionList<Entry<K,V>>();
c = new DefaultComparator<K>();
}
 /** crea una coda a priorità con il comparatore passato come
    argomento. */
public SortedListPriorityQueue (Comparator<K> comp) {
  entries = new NodePositionList<Entry<K,V>>();
  c = comp; }
```

Continua nella prossima slide

Implementazione di Priority Queue con lista ordinata

```
public Entry<K,V> min () throws EmptyPriorityQueueException {
  if (entries.isEmpty())
     throw new EmptyPriorityQueueException(" la coda a priorità è vuota");
   else
     return entries.first().element();
}
public Entry<K,V> insert (K k, V v) throws InvalidKeyException {
   checkKey(k);
   Entry<K,V> entry = new MyEntry<K,V>(k, v);
   insertEntry(entry);
   return entry;
}
```

Continua nella prossima slide

Implementazione di Priority Queue con lista ordinata

```
protected void insertEntry(Entry<K,V> e) {
  if (entries.isEmpty()) { entries.addFirst(e); }
  else if (c.compare(e.getKey(), entries.last().element().getKey()) > 0)
    entries.addLast(e);
  else {
       Position<Entry<K,V>> curr = entries.first();
        while (c.compare(e.getKey(), curr.element().getKey())> 0)
          curr = entries.next(curr);
        entries.addBefore(curr, e);
  }
}
```

Continua nella prossima slide

Implementazione di Priority Queue con lista ordinata

```
public Entry<K,V> removeMin() throws EmptyPriorityQueueException {
  if (entries.isEmpty())
    throw new EmptyPriorityQueueException("la coda a priorità e vuota");
  else return entries.remove(entries.first());

}
```


Implementazione di Priority Queue con lista ordinata

```
protected boolean checkKey(k key){
boolean result;
  try {
    result = (c.compare(key,key)==0);
  } catch (ClassCastException e)
    {     throw new InvalidKeyException("chiave non confrontabile"); }
  return result;
 }
```

Esercizi

- Implementare PriorityQueue con:
  - una lista non ordinata (UnsortedListPriorityQueue)
  - una lista ordinata (SortedListPriorityQueue)
- Testare i metodi nelle due implementazioni

Ordinamento con PriorityQueue

- Si può usare una coda a priorità per ordinare un insieme di elementi confrontabili
  1. inserisci un elemento alla volta con insert
  2. rimuovi gli elementi uno alla volta con removeMin operations
- L'analisi della complessità di tempo di questo algoritmo dipende da come è implementata PriorityQueue

Algorithm *PQ-Sort*(*S*, *C*)
Input sequenza *S*, comparatore *C* per gli elementi di *S*
Output sequenza *S* ordinata per valori crescenti rispetto a *C*
P ← coda a priorità con comparatore *C*
while !(*S.isEmpty* ())
e ← *S.remove* (*S. first* ())
P.insert(*e*,*e*)
while !(*P.isEmpty*())
e ← *P.removeMin*().*getKey*()
S.addLast(*e*)

Selection-Sort

- Selection-sort è una variante di PQ-sort dove la coda a priorità è implementata con una lista non ordinata
- Tempo di esecuzione di Selection-sort:
  1. Inserire gli elementi nella coda richiede n chiamate a insert, e quindi tempo $O(n)$
  2. Rimuovere gli elementi dalla coda in ordine richiede n chiamate a removeMin, e quindi tempo

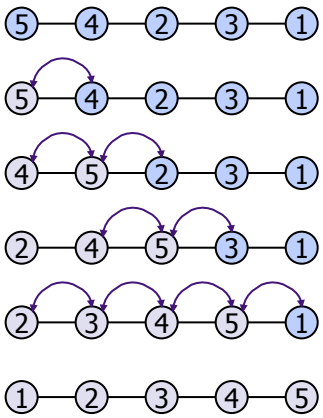
     $O(1 + 2 + \ldots + n-1) = O(n(n-1)/2) = O(n^2)$
- Selection-sort richiede tempo $O(n^2)$ e spazio aggiuntivo $O(n)$

Insertion-Sort

- Insertion-sort è una variante di PQ-sort dove la coda a priorità è implementata con una lista ordinata
- Tempo di esecuzione di Insertion-sort:
  1. Inserire gli elementi nella coda in ordine richiede n chiamate a insert, e quindi tempo $O(1 + 2 + \ldots + n-1) = O(n(n-1)/2) = O(n^2)$
  2. Rimuovere gli elementi dalla coda in ordine richiede n chiamate a removeMin, e quindi tempo $O(n)$
- Insertion-sort richiede tempo $O(n^2)$ e spazio aggiuntivo $O(n)$

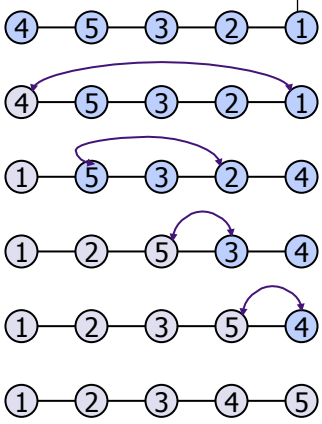
Insertion-sort sul posto

- Invece di usare una struttura di appoggio, sia Insertion-sort che Selection-sort si possono implementare sul posto
- Usiamo swapElements per spostare gli elementi invece di modificare la struttura della lista
- Una porzione della sequenza in input serve come coda a priorità
- **Insertion-sort sul posto**
  - Manteniamo ordinata la prima parte della sequenza
  - Ad ogni passo prendiamo un elemento della seconda parte della sequenza e lo **inseriamo** nella parte già ordinata della sequenza



Selection-sort sul posto

- **Selection-sort sul posto**
  - Manteniamo ordinata la prima parte della sequenza
  - Al passo i-esimo **selezioniamo** l'elemento minimo tra quelli non ancora ordinati e lo scambiamo con l'i-esimo elemento della sequenza



Esercizi

- Implementare ***PQ-Sort*** con:
  - UnsortedListPriorityQueue (Selection Sort)
  - SortedListPriorityQueue (Insertion Sort)
- Testa il corretto funzionamento di entrambe le implementazioni.
- Implementare Insertion Sort sul posto (cioè senza usare una struttura di appoggio come in PQ-Sort). Discutere complessità computazionale.