Set & Partition

Corso: Strutture Dati

Docente: Annalisa De Bonis

Il TDA Set

- Il TDA Set è un contenitore di oggetti
  - Gli oggetti sono a due a due distinti
  - In generale non è definita una relazione di ordine sugli elementi

Le operazioni del TDA Set (nella forma di metodi java)

- union(B)
  - Invocato sull'insieme A, rimpiazza A con l'unione di A e B
- intersect(B)
  - Invocato sull'insieme A, rimpiazza A con l'intersezione di A e B
- subtract(B)
  - Invocato sull'insieme A, rimpiazza A con la differenza di A e B

Interfaccia Set

```
public interface Set<E> {
   // Restituisce il numero degli elementi nell'insieme
   public int size();
   // Restituisce true se l'insieme è vuoto
   public boolean isEmpty();
   // Rimpiazza this con l'unione di this e B
   public Set<E> union(Set<E> B)
   // Rimpiazza this con l'intersezione di this e B
   public Set<E> intersect(Set<E> B)
   // Rimpiazza this con la differenza di this e B
   public Set <E>subtract(Set <E> B)
   }
```

Una semplice implementazione di Set

- Il modo più semplice per implementare il TDA insieme è quello di memorizzare gli elementi dell'insieme in una sequenza (PositionList, IndexList, Sequence)

Implementazione di Set con una sequenza ordinata

- Per rendere più efficienti le operazioni insiemistiche è utile definire una relazione d'ordine totale sull'insieme
  - gli elementi dell'insieme vengono memorizzati in una sequenza ordinata
  - vantaggio:È possibile usare lo schema della procedura merge (usata nel merge sort) per implementare i metodi: union, intersect e subtract
  - La classe che implementa Set avrà
    - una variabile di istanza del tipo usato per rappresentare la sequenza (IndexList, PositionList, o Sequence)
    - una variabile di istanza di tipo Comparator usata per confrontare gli elementi dell'insieme

Implementazione di Set con una sequenza ordinata

- Esempio: A = { 1 2 4 6 } B = { 3 4 5 7 10 }
  - Union: C = { 1 2 3 4 5 6 7 10 }

Implementazione di Set con una sequenza ordinata

- Esempio: A = { 1 2 4 6 } B = { 3 4 5 6 10 }
  - Intersect: C = { 4 6 }

Implementazione di Set con una sequenza ordinata

- Esempio: A = { 1 2 4 6 } B = { 3 4 5 7 10 }
  - Subtract: C = { 1 2 6 }

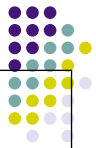
Bozza della classe che implementa Set con una lista ordinata

```
public class OrderedListSet <E> implements Set<E> {
   Comparator c;
   PositionList <E> L;
   int size;  //si può  anche omettere

   OrderedListSet() {
        L = new NodePositionList<E>();
        c = new DefaultComparator();
         size=0;
   }
  ...
}
```

Implementazione di Set con una sequenza ordinata

- possiamo utilizzare un algoritmo generico (template method pattern) di merge per effettuare le tre operazioni insiemistiche
- A.union(B), A.intersect(B) e A.subtract(B), implementate con uno schema generico di merge, richiedono tempo O(n+m)
- Quale sarebbe la complessità di A.union(B), A.intersect(B) e A.subtract(B), se gli insiemi non fossero ordinati ?

Algoritmo Generico di Merge di due liste



- Template method genericMerge
- Metodi ausiliari
  - aIsLess
  - bIsLess
  - bothAreEqual
- Tempo $O(n_A + n_B)$ se i metodi ausiliari sono eseguiti in tempo $O(1)$

```
Algorithm genericMerge(A, B)
    S ← empty sequence
    while !(A.isEmpty()) ∧ !(B.isEmpty())
        a ← A.first().element();
        b ← B.first().element()
        if a < b
            aIsLess(a, S); A.remove(A.first())
        else if b < a
            bIsLess(b, S); B.remove(B.first())
        else { b = a }
            bothAreEqual(a, b, S)
            A.remove(A.first()); B.remove(B.first())
    while !(A.isEmpty())
        aIsLess(a, S); A.remove(A.first())
    while !(B.isEmpty())
        bIsLess(b, S); B.remove(B.first())
    return S
```

Uso di Generic Merge per le operazioni insiemistiche

- intersection: copia solo gli elementi che appaiono in entrambe le liste
  - Occorre riscrivere il metodo bothAreEqual
- union: copia ogni elemento e butta via duplicati
  - Occorre riscrivere tutti e tre i metodi ausiliari
- subtract: copia solo gli elementi che appaiono in A e non appaiono in B
  - Occorre riscrivere aIsLess

Il TDA Partition

- Una partizione è una collezione di insiemi S_1, …,S_k a due a due disgiunti
  - $S_i \cap S_j = \varnothing$ per ogni $i \neq j$

Le operazioni del TDA Partition

Il TDA Partition supporta i seguenti metodi:

- makeSet(x)
  - crea l'insieme contenente il solo elemento x e lo aggiunge alla partizione
- union(A, B)
  - Aggiunge alla partizione l'insieme unione di A e B distruggendo gli insiemi A e B
- find(x)
  - restituisce l'insieme che contiene l'elemento x

L'interfaccia Partition

```
public interface Partition <E> {
// Restituisce il numero degli insiemi nella
//partizione
public int size();
// Restituisce true se la partizione è vuota
public boolean isEmpty();
// Restituisce l'insieme contenente il solo elemento x
public Set<E> makeSet(E x)
```

L'interfaccia Partition

```
// Restituisce l'unione di A e B, distruggendo
// i vecchi insiemi A e B

public Set <E>union(set<E> A, set<E> B)

// restituisce l'insieme che contiene l'elemento x
public Set <E> find(E x)
}
```

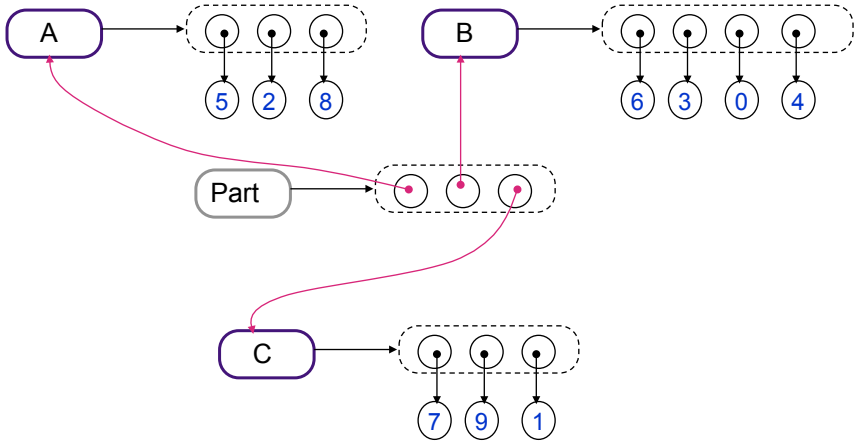

Implementazione di Partition

- Una semplice implementazione di Partition consiste nel memorizzare gli insiemi della partizione all'interno di una sequenza (IndexList, PositionList, Sequence)
  - La sequenza contiene oggetti di tipo Set (Set è a sua volta implementato mediante una sequenza)

La classe che implementa Partition con una lista

```
public class ListPartition <E>
        implements Partition <E>{
PositionList<Set <E>> partizione; // lista di insiemi
…
}
```


Implementazione di Partition

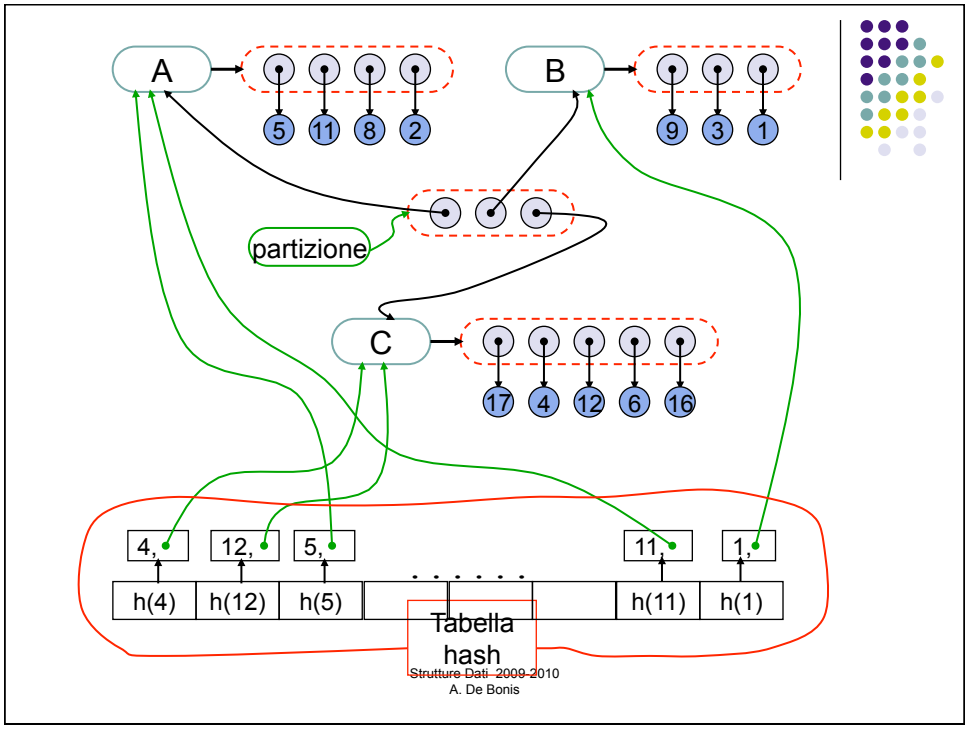


Implementazione efficiente

- Ad ogni elemento possiamo associare l'insieme a cui appartiene
- Serve per implementare find in O(1)
- Due tecniche:
  - Inseriamo nella partizione una mappa che ha come voci le coppie (elemento, insieme)
    - Rappresentiamo la mappa con una tabella hash
    - Find ha tempo medio di esecuzione O(1) se si usa una buona funzione hash e si tiene il load factor sotto controllo
  - Aggiungiamo alla classe che implementa Position la variabile di instanza Set
    - Set è un riferimento all'insieme in cui è contenuto l'elemento

La classe che usa una mappa per associare gli elementi agli insiemi

```
public class ListPartition <E>
                          implements Partition <E>{
  Map <E,Set<E>> elementi;
  PositionList <Set<E>> partizione;
…
}
```



Implementazione efficiente di Partition

- Dato che in una partizione gli insiemi sono due a due disgiunti, siamo sicuri che un elemento apparterrà solo ad un insieme. Di conseguenza possiamo aggiungere all'implementazione di Set i metodi
  - public Set<E> fastUnion(Set B)
  - public E fastInsert(E x)

Implementazione efficiente di Partition

- public Set <E>fastInsert(E x)
  - Inserisce x nell'insieme su cui è invocato senza verificare se x appartiene ad A
  - Complessità O(1)
- public Set <E>fastUnion(Set<E> B)
  - Inserisce gli elementi di B nell'insieme su cui è invocato senza verificare se appartengono già ad A
  - Complessità O(|B|)

Implementazione efficiente di Partition

- Quando si esegue un'unione, spostiamo sempre gli elementi dall'insieme più piccolo a quello più grande (unione pesata)
  - Ogni volta che spostiamo un elemento, esso va a finire in un insieme che è almeno due volte più grande dell'insieme da cui proviene
  - In questo modo, un elemento può essere mosso al più O(log n) volte
- Per ogni elemento spostato aggiorniamo la voce corrispondente nella mappa oppure la variabile Set della classe che implementa Position

Implementazione efficiente di Partition

- union(A, B)
  - Se |A|>|B|
    - aggiungiamo gli elementi di B ad A
    - rimuoviamo B dalla partizione
  - Altrimenti
    - aggiungiamo gli elementi di A a B
    - rimuoviamo A dalla partizione

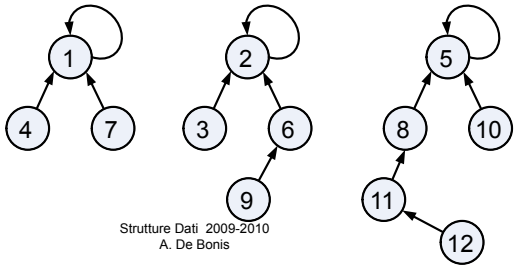
- Invece di cercare in partizione la position **p** relativa all'insieme da rimuovere inseriamo nella classe che implementa Set una variabile di tipo Position che tiene traccia della posizione in cui si trova l'insieme.

Il metodo Union di Partition

```
public Set<E> union(Set <E>A, Set<E> B) {
    MySet<E> toAdd=null;
    MySet <E>AUB=null;
    if(A.size()>B.size()) toAdd=B; AUB=A;
    else toAdd=A; AUB=B;

   Position<Set<E>> toRemove;
    toRemove=toAdd.location(); //metodo che restituisce la posizione del set

     while(!toAdd.isEmpty()) {
       E el= toAdd.remove();  //metodo che cancella un elemento del set
       AUB.fastInsert(el);

    /*inserire l'istruzione per aggiornare la mappa se  questa è presente
    nell'implementazione */
     }
     partizione.remove(toRemove); //rimuove l'insieme toAdd dalla partizione
}
```

Implementazione basata sugli alberi

- Ogni insieme nella partizione è visto come un albero
- Ogni elemento di un insieme è memorizzato in una posizione (nodo dell'albero)
- Ogni insieme è rappresentato dall'elemento del nodo radice
- Esempio: gli insiemi "1", "2" e "5"

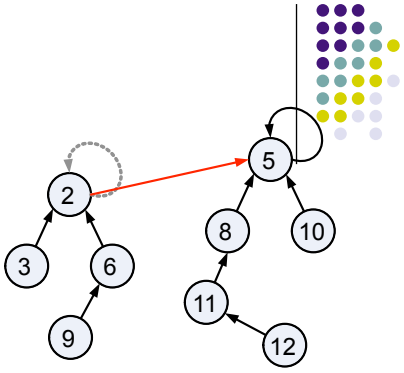


Implementazione basata sugli alberi

- Ogni posizione è implementata con un nodo contenente
  - element che contiene il riferimento all'elemento dell'insieme
  - parent che contiene il riferimento al nodo padre (a se stesso per la radice)
- Gli alberi usati sono specifici per Partition non costituiscono un'implementazione del TDA Tree

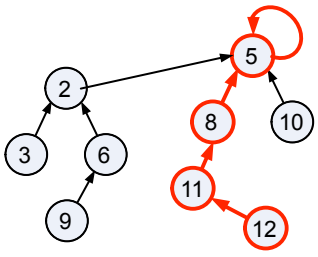
Union

- Per eseguire una union, si fa puntare semplicemente la radice di un albero alla radice dell'altro



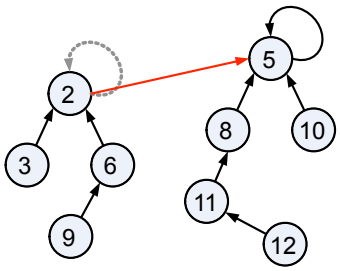
Find

- Per eseguire una find, si segue il percorso dal nodo che contiene l'elemento passato in input fino alla radice (nodo il cui campo parent punta a se stesso)



Euristica per migliorare l'efficienza

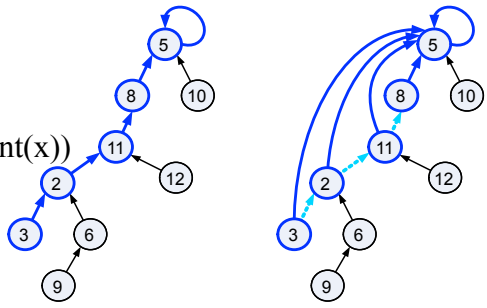
- **Union-by-size**
  - nelle operazioni di unione l'albero più piccolo diventa sottoalbero della radice dell'altro
  - find richiede tempo O(log n)
    - Ogni volta che attraversiamo un puntatore al genitore, passiamo in un sottoalbero (quello radicato nel genitore) che contiene almeno il doppio degli elementi contenuti nel sottoalbero in cui eravamo
    - quindi in totale si attraversano al più O(log n) puntatori
  - n operazioni di union, find e makeSet richiedono tempo O(nlog n)



Un'altra euristica per l'efficienza

- **Path Compression**
  - Dopo aver eseguito un'operazione di find, tutti i nodi attraversati nella ricerca avranno come nodo genitore la radice

```
Algorithm find(x)
 if x ≠parent(x)
  then x.parent ← find(parent(x))
return parent(x)
```



Union-by-size e path-compression

- Se si utilizzano le euristiche union-by-size e path-compression allora una sequenza di m operazioni di union e find effettuate su una partizione con n elementi richiede tempo $O(m\ \alpha\ (m,n))$
- **$\alpha(m,n))$ è l'inversa della funzione di Ackermann $\alpha(m,n))<4$ per tutti i valori pratici di m ed n**