Esercitazione 4
Parsing di Alberi e Coda di priorità

Corso di Fondamenti di Informatica II

BIAR2 (Ing. Informatica e Automatica) e BSIR2 (Ing. dei Sistemi)

A.A. 2013/2014

8 Novembre 2013

Sommario

Scopo della esercitazione è quello di effettuare il parsing di un albero e di realizzare una struttura dati per gestire uno heap.

1 Parsing di un Albero n-ario

Si vuole ricostruire una struttura dati di tipo albero n-ario a partire dalla sua rappresentazione parentetica. La rappresentazione è definita dalla seguente grammatica:

```
Tree → ( <key> Children )

Children → Tree Children | ϵ
```

dove ϵ rappresenta la stringa vuota e `<key>` una chiave intera. Ad esempio, `( 3 ( 5 ) ( 2 ) )` rappresenta un albero con chiave 3 nella radice, chiave 5 nel primo figlio della radice e chiave 2 nel secondo figlio.

Data la rappresentazione parentetica, si vuole ricostruire l'albero n-ario corrispondente e stamparlo.

Specifiche. Scrivere una classe Java `TreeParser` contenente i seguenti metodi

```java
public static Tree<Integer> readTree() throws IOException;
```

Il metodo `readTree()` legge da `System.in` la rappresentazione parentetica e costruisce l'albero n-ario associato. La lettura avviene sfruttando la classe `Tree` realizzata nelle esercitazioni precedenti (arricchita di un metodo `addSubtree`, vedi oltre). Il metodo `readTree()` legge un'intera riga da `System.in`, crea un nuovo oggetto di tipo `StringTokenizer`, ne estrae il primo token e passa sia lo `StringTokenizer` che il primo token (`nextToken`) al metodo ausiliario `readTree(StringTokenizer,String)`.

```
private static Tree<Integer> readTree(StringTokenizer st,
        String nextToken) throws IOException;
```

Il metodo ricorsivo `readTree(StringTokenizer st, String nextToken)` legge tramite lo `StringTokenizer` un generico albero. Si utilizzi il metodo `nextToken()` dello `StringTokenizer` per aggiornare la variabile `nextToken` al token successivo di cui effettuare il parsing. La variabile `nextToken` viene usata per decidere se si sta per effettuare il parsing di un nuovo sottoalbero (parentesi aperta) o se si sta invece chiudendo l'albero corrente (parentesi chiusa). Non si richiede di gestire rappresentazioni non corrette.

Si proceda a testare `TreeParser` attraverso il driver `Test.java` fornito nella cartella dell'esercitazione e le classi `Tree<E>` e `TreeUtil`. `Tree` è stata arricchita di un nuovo metodo `addSubtree(Tree<E> s)` che permette di aggiungere un intero sottoalbero alla radice dell'albero corrente. `TreeUtil` contiene un metodo di stampa generico (`print`) per `Tree<E>`.

Nota. In caso la rappresentazione parentetica sia molto lunga, anziché scrivere su `System.in` da tastiera, si consiglia di utilizzare il comando

```
java Test < albero.txt
```

dove albero.txt contiene la rappresentazione parentetica che si vuole scrivere su `System.in`.

2 Rappresentazione di un heap

Una *coda di priorità* è un tipo di dato astratto che contiene un insieme di dati su cui sono definite delle priorità (quindi coppie ⟨valore, chiave⟩, dove la chiave rappresenta la priorità), che supporta sia inserimento di un elemento arbitrario sia cancellazione di elementi in ordine di priorità.

Una efficiente realizzazione di una coda di priorità è quella che si ottiene mediante l'*heap* e si basa sull'idea di memorizzare i dati in un albero binario completo. In un heap, per ciascun nodo v, esclusa la radice, la chiave memorizzata in v è maggiore o uguale della chiave memorizzata nel padre di v. Di conseguenza le chiavi dell'albero sono sempre incontrate in ordine non decrescente e la chiave più piccola si trova nella radice dell'albero.

Per quanto riguarda l'albero binario, utilizzeremo una rappresentazione basata su `ArrayList` (vedi Esercitazione 3, basata su array), che risulta particolarmente appropriata per implementare alberi completi. Si ricorda che in questa rappresentazione il generico nodo v dell'albero T è memorizzato in un `ArrayList` in posizione $p(v)$ ed in particolare:

- se v è la radice, allora $p(v) = 1$;
- se v è figlio sinistro di u, allora $p(v) = 2 * p(u)$;
- se v è figlio destro di v, allora $p(v) = 2 * p(u) + 1$;

Specifiche. Scrivere una classe Java parametrica `MinHeap<V>` con la seguente interfaccia. Viene fornita, nella cartella dell'esercitazione, la classe `HeapEntry<V>`, per memorizzare all'interno della struttura dati `MinHeap<V>` i valori e le relative priorità. La classe serve a memorizzare, oltre alla coppia ⟨valore, chiave⟩, una ulteriore informazione: l'*indice* nel quale l'elemento è memorizzato. Questo campo, contenendo esplicitamente l'indice, serve a velocizzare le operazioni di accesso ai dati. Si noti che tale informazione deve essere mantenuta consistente quando un metodo richiede di riposizionare gli elementi all'interno dell'heap.

```
public MinHeap()
```

Costruisce un (min) heap vuoto.

```
public HeapEntry<V> getMin()
```

Restituisce il minimo dell'heap senza toglierlo dall'heap.

```
public HeapEntry<V> insert(int p, V value)
```

Inserisce un nuovo valore (e priorità) nell'heap. Internamente i valori sono memorizzati mediante la classe `HeapEntry<V>`.

```
public boolean isEmpty()
```

Determina se l'heap è vuoto.

```
public HeapEntry<V> removeMin()
```

Restituisce il minimo dell'heap e lo rimuove dall'heap.

```
public int size()
```

Restituisce il numero di elementi nell'heap.

```
public static void heapsort(int[] v)
```

Utilizza un heap per realizzare un algoritmo di ordinamento per vettori con valori interi (algoritmo Heap Sort).

Si proceda a testare `MinHeap<V>` attraverso il driver `Test.java` fornito nella cartella dell'esercitazione e la classe `HeapEntry<V>`.

3 Heap a priorità adattabile (per casa)

Si vogliono aggiungere alla classe `MinHeap` i metodi per la modifica dinamica delle priorità degli elementi `HeapEntry<V>`. Le priorità adattabili permettono di implementare in maniera più efficiente molti algoritmi fondamentali, come l'algoritmo di Dijkstra per il calcolo del cammino minimo in un grafo.

Programma Java. Si vogliono aggiungere in particolare i metodi:

```
public HeapEntry<V> replaceValue(HeapEntry<V> e, V value)
```

Sostituisce il valore di `e` con `value`. Restituisce la entry aggiornata.

```
public HeapEntry<V> replaceKey(HeapEntry<V> e, int key)
```

Sostituisce la chiave di `e` con `key` e aggiorna l'heap in modo da riposizionare correttamente l'entry con la nuova chiave. Restituisce la entry aggiornata.

```
public HeapEntry<V> remove(HeapEntry<V> e)
```

Rimuove e restituisce la entry `e`.

Riferimenti bibliografici

[1] M. T. Goodrich and R. Tamassia. *Strutture dati e algoritmi in Java*. Zanichelli, 2007.