Fondamenti di Informatica II

Esercitazione del 24 Marzo 2020 - Prof. Becchetti

Scopo dell' esercitazione è quello di realizzare una struttura dati per gestire un heap.

Attenzione: L'esercitazione può essere svolta sia in linguaggio C che in Java. Entrambe le cartelle sono fornite dello scheletro delle classi da completare e di un programma di prova (`driver.c` e `Driver.java`).

Non occorre creare nuovi file, ma soltanto completare le classi Java/moduli C forniti. Si consiglia agli studenti di implementare le funzioni C/metodi Java nell'ordine in cui sono proposti.

Task 1. Rappresentazione di un heap

Un heap è una struttura dati basata sugli alberi che soddisfa la proprieta' di heap: se a è un genitore di b, allora la chiave di a è ordinata rispetto alla chiave di b. In particolare, in un max heap, le chiavi di ciascun nodo sono sempre maggiori o uguali di quelle dei figli, e la chiave dal valore massimo appartiene alla radice. In un min heap, le chiavi di ciascun nodo sono sempre minori o uguali di quelle dei figli, e la chiave dal valore minimo appartiene alla radice. La Figura 1 fornisce un esempio delle due tipologie di heap. In questa esercitazione è richiesto di implementare un heap basato su alberi binari (semi-) completi. In particolare, si richiede di sviluppare una rappresentazione di heap basata su array. Si ricorda che in questa rappresentazione il generico nodo v dell'albero T è memorizzato in un array in posizione $p(v)$ e in particolare, assumendo che gli indici dell'array inizino a 0:

- Se v è la radice: $p(v) = 0$
- Se v il figlio sinistro di u: $p(v) = 2p(u) + 1$
- Se v è il figlio destro di u: $p(v) = p(u) + 2$

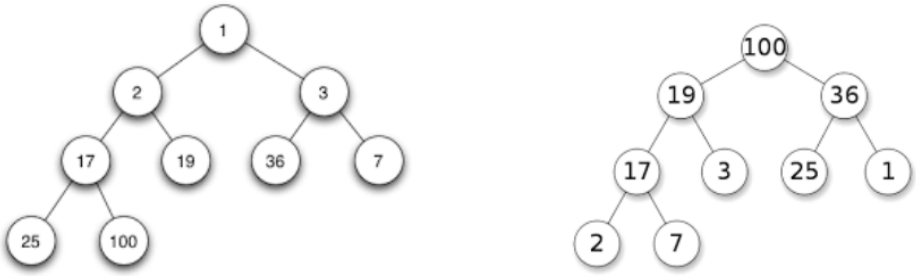

Figura 1: Esempio di un *min heap* (sinistra) e di un *max heap* (destra).

Specifiche. Tutte le operazioni definite su heap in questa esercitazione devono rispettare i costi asintotici visti a lezione. Completare il modulo C con intestazione `heap.h` (rispettivamente la classe Java `Heap.java`) contenente le seguenti funzioni (metodi se Java):

- Una funzione /metodo che inizializza un heap vuoto:

```
heap * heap_new ( HEAP_TYPE type , int initialCapacity );
public Heap ( HEAP_TYPE type , int initialCapacity );
```

di tipo `HEAP_TYPE` (`MIN_HEAP` o `MAX_HEAP`) con un array interno di dimensione iniziale pari a `initialCapacity`.

- Una funzione/metodo che restituisce il tipo di heap:

```
HEAP_TYPE heap_type ( heap * h ); // C

public HEAP_TYPE getType (); // Java
```

- Una funzione/metodo che restituisce il numero di chiavi presenti nell'heap:

```
int heap_size ( heap * h ); // C

public int getSize (); // Java
```

Tale operazione deve avere costo costante.

- Una funzione/metodo che restituisce la chiave della radice:

```
int heap_peek ( heap * h ); // C

public int peek (); // Java
```

- Una funzione/metodo che aggiunge una chiave all'heap e restituisce un puntatore (riferimento se Java) al nodo contenente la chiave:

```
heap_entry * heap_add ( heap * h , int key ); // C

public HeapEntry add ( int key ); // Java
```

- una funzione/metodo che dato un puntatore (riferimento) ad un heap entry (HeapEntry) restituisce la chiave memorizzato in esso:

```
int get_key_entry ( heap_entry * e ); // C

public int getEntryKey ( HeapEntry e ); // Java
```

- una funzione/metodo che rimuove la chiave nella radice del heap, aggiorna opportunatamente l'heap, e restituisce la chiave rimossa:

```
int heap_poll ( heap * h ); // C

public int poll (); // Java
```

- (solo in C) una funzione che elimina l'heap:

```
void heap_delete ( heap * h );
```

- Una funzione/metodo che stampa in stdout una rappresentazione del heap:

```
void heap_print ( heap * h ); // C

public void print (); // Java
```

> Tale funzione deve stampare le chiavi, separate da uno spazio e sempre sulla stessa riga di testo, in base alla loro posizione: < *chiave in posizione p(0)* > <*chiave posizione p(1)* > … < *chiave in posizione p(n)*

Si proceda a testare il codice sviluppato utilizzando `driver.c` (rispettivamente `Driver.java`) passando come argomento `min_heap`. Il driver testerà la struttura dati passando come argomento `MIN _HEAP` come tipo `HEAP_TYPE`. Procedere con il prossimo esercizio per eseguire dei test con un heap di tipo `MAX_HEAP`.

Task 2. Costruzione di heap da array - array2heap/heapify

Si vuole implementare una funzione che, dato in input un array contentente delle chiavi di tipo intero, costruisca un heap in tempo lineare. Al fine di ottenere tale bound sul tempo di esecuzione è necessario sfruttare opportunatamente (e ripetutamente) l'operazione *downHeap*, perseguendo un approccio bottom-up. Si faccia riferemento alle slide del corso per maggiori dettagli.

Specifiche. Estendere il modulo C (o classe Java) sviluppato nel precedente esercizio con la seguente funzione C (metodo statico se Java):

```
heap * array2heap ( int * array , int size , HEAP_TYPE type ); // C

public static Heap array2heap ( int [] array , HEAP_TYPE type ); // Java
```

che ritorna un heap di tipo `HEAP_TYPE` costruito usando le chiavi presenti nell'array di input. Si proceda a testare il codice sviluppato utilizzando `driver.c` (rispettivamente `Driver.java`) passando come argomento `array`.

Task 3. Heapsort

Si vuole implementare l'algoritmo di ordinamento *heap sort*.

Specifiche. Estendere il modulo C (o classe Java) sviluppato nel primo esercizio con la seguente funzione C (metodo statico se Java):

```
void heap_sort ( int * array , int size ); // C

public static void heapSort ( int [] array ); // Java
```

che ordina (side-effect) in modo crescente l'array di input sfruttando un heap. Si proceda a testare il codice sviluppato utilizzando `driver.c` (rispettivamente `Driver.java`) passando come argomento sort.

Task 4. Aggiornamento chiave

Si vuole implementare una funzione che, dato in input un heap entry (`HeapEntry`), aggiorni la chiave memorizzata nel nodo dell'albero binario.

Specifiche. Estendere il modulo C (o classe Java) sviluppato nel primo esercizio con la seguente funzione C (metodo se Java):

```
void heap_update_key ( heap * h , heap_entry * e , int key ); // C

public static void updateEntryKey ( HeapEntry e , int key ); // Java
```

La chiave del nodo e viene aggiornata con `key`. Si proceda a testare il codice sviluppato utilizzando `driver.c` (rispettivamente `Driver.java`) passando come argomento `change`.

Task 5. Mantenimento k-esimo elemento (es. per casa)

Si vuole implementare una struttura dati per mantenere in modo efficiente il k-esimo elemento di un insieme ordinato di chiavi. In particolare, si vuole che l'accesso al k-esimo elemento più piccolo avvenga in tempo $O(1)$. Si noti che k è una costante positiva nota a priori. Tale struttura dati deve essere implementata sfruttando una combinazione di *min heap* e *max heap*.

Specifiche. Scrivere un modulo C `kth.c` con intestazione `kth.h` (o classe Java `Kth.java`) con le seguenti funzione C (metodo di instanze se Java):

- Una funzione che costruisce la struttura dati (inizialmente vuota):

  ```
  kth * kth_new ( int k ); // C

  public Kth ( int k ); // Java
  ```

- (Solo C) una funzione che dealloca la struttura dati:

  ```
  void kth_delete ( kth * d );
  ```

- Una funzione che restituisce il k-esimo elemento dell'insieme ordinato:

  ```
  int kth_get ( kth * d ); // C

  public int get (); // Java
  ```

  Si noti che tale operazione ritorna un risultato significativo solo se sono stati inseriti almeno k elementi.

- Una funzione che elimina il k-esimo elemento:

  ```
  void remove ( kth * d ); // C

  public void remove ( int k ); // Java
  ```

  Si proceda a testare il codice sviluppato utilizzando `driver.c` (rispettivamente `Driver.java`) passando come argomento `k`.