Lab 9: Heap Binari di tipo Min

```
namespace priorityQueue {

        // struct "standard" per la rappresentazione del tdd lista mediante array dinamico
        struct dynamicArray {
            Elem* data;
            int size;
            int maxsize;
        };
typedef dynamicArray PriorityQueue;
```

createEmpty

```
priorityQueue::PriorityQueue priorityQueue::createEmptyPQ(int dim) {
  PriorityQueue pq;                 // creo una nuova variabile priorityqueue, che rappresenterà la coda
                                       di priorità che verrà restituita dalla funzione.
  pq.size = 0;                      // si inizializza il campo size di pq a 0. questo campo rappresenta il
                                       numero corrente di elementi nella coda, e all'inizio è vuota, quindi
                                       size è zero.
  pq.maxsize = dim;                 // si assegna il valore del parametro dim campo maxsize di pq. questo
                                       definisce la dimensione massima della coda di priorità, ovvero il
                                       numero massimo di elementi che può contenere.
  pq.data = new Elem[pq.maxsize];   // viene allocato dinamicamente un array di tipo elem. l'array ha una
                                       lunghezza pari a pq.maxsize. questo array sarà utilizzato per
                                       memorizzare gli elementi della coda.

  return pq;
}
```

insert

```
bool priorityQueue::insert(PriorityQueue &pq, const Elem &elem) {
  if (pq.size == pq.maxsize) {     // nel caso in cui la priorityqueue sia piena (size = maxsize)
    return false;                     ritorna false
  }

  ++pq.size;                      // incrementa la dimensione di pq.size. un nuovo elemento sarà aggiunto
                                     alla coda, quindi il numero di elementi aumenta di uno.
  pq.data[pq.size - 1] = elem;    // inserisce l'elemento elem nell'array pq.data della coda,
                                     posizionandolo all'indice pq.size - 1, ossia l'ultima posizione libera
                                     nell'array
  moveUp(pq, pq.size);            // chiama la funzione ausiliaria moveup, che ha il compito di
ripristinare
                                     l'ordine della coda di priorità. il parametro pq è la coda di priorità
                                     in cui si sta operando, e pq.size è la posizione dell'elemento appena
                                     inserito.
  return true;                    // l'inserimento è andato a buon fine e ritorno true
}
```

Funzione ausiliaria moveUp

```
void moveUp(priorityQueue::PriorityQueue &pq, int index) {
// Questo ciclo while continua a iterare finché index è maggiore di 0, il che significa che l'elemento corrente
non si trova alla radice della coda (la radice è sempre all'indice 0). L'idea è di far "risalire" l'elemento
confrontandolo con il suo genitore finché non raggiunge una posizione corretta o diventa la radice
  while (index > 0) {
    int parentIndex = (index - 1) / 2;       // Si calcola l'indice del genitore dell'elemento corrente.
                                                In un heap binario, l'indice del genitore di un nodo con
                                                indice index è dato dalla formula (index - 1) / 2.
    if (pq.data[index] >= pq.data[parentIndex]) { // Si verifica se l'elemento corrente è maggiore o
                                                     uguale al genitore
      break;                                    // Se l'elemento corrente è maggiore o uguale, la
                                                   proprietà dell'heap è soddisfatta, quindi si esce
                                                   dal ciclo usando break.


    }
    swap(pq, index, parentIndex);      // Se l'elemento corrente è minore del suo genitore, allora i due
                                          elementi vengono scambiati di posizione chiamando la funzione
                                          swap. Questo scambio serve a mantenere la proprietà dell'heap,
                                          che in un min-heap richiede che ogni nodo genitore sia minore
                                          o uguale ai suoi figli.
    index = parentIndex;               // L'indice corrente index viene aggiornato per essere l'indice del
                                          genitore, parentIndex. Questo permette al ciclo while di
                                          continuare, spostando l'elemento verso l'alto finché non si
                                          trova in una posizione dove la proprietà dell'heap è rispettata
                                          o raggiunge la radice.
  }
}
```

DeleteMin

```
bool priorityQueue::deleteMin(PriorityQueue &pq) {
       if (isEmpty(pq)) {              // Se la coda è vuota, non c'è nulla da eliminare,
    return false;                         quindi la funzione restituisce false e termina.
  }
  pq.data[0] = pq.data[pq.size - 1];  // Qui l'elemento alla radice della coda (che è il minimo, poiché in
                                         un heap min la radice è sempre l'elemento più piccolo) viene
                                         sostituito con l'ultimo elemento nell'heap. pq.size - 1 è l'indice
                                         dell'ultimo elemento nell'array, quindi si sposta l'ultimo
                                         elemento in cima alla coda, sovrascrivendo l'elemento minimo.
--pq.size;                            // elimino l'ultimo elemento decrementando la size
  moveDown(pq, 0);          // Si chiama la funzione moveDown, passando pq e 0 (la posizione della radice).
                               La funzione moveDown riorganizza l'heap, spostando l'elemento che è stato
                               spostato in cima verso il basso, fino a quando trova la posizione corretta per
                               mantenere la proprietà dell'heap: l'elemento deve scendere finché non è
                               minore di entrambi i suoi figli, ripristinando l'ordine della coda.
  return true;              // La cancellazione è andata a buon fine e ritorno true
}
```

Funzione ausiliaria moveDown

```
void moveDown(priorityQueue::PriorityQueue &pq, int index) {
  int leftChild = 2 * index + 1;       // Si calcola l'indice del figlio sinistro del nodo corrente (index).
                                          In un heap binario, l'indice del figlio sinistro di un nodo con
                                          indice index è dato dalla formula 2 * index + 1.
  int rightChild = 2 * index + 2;      // Si calcola l'indice del figlio destro del nodo corrente. In un
                                          heap binario, l'indice del figlio destro di un nodo con indice
                                          index è dato dalla formula 2 * index + 2.
  int smallest = index;                // Si assume inizialmente che il nodo corrente (all'indice index)
                                          sia il più piccolo (in un min-heap). La variabile smallest
                                          conterrà l'indice dell'elemento più piccolo tra il nodo corrente
                                          e i suoi figli.

// Si controlla se l'indice leftChild è valido, ossia se è all'interno dei limiti dell'heap (ovvero leftChild < pq.size).
Se l'elemento nel figlio sinistro è minore di quello dell'elemento smallest, significa che il figlio sinistro è più
piccolo del nodo corrente.
  if (leftChild < pq.size && pq.data[leftChild] < pq.data[smallest]) {
    smallest = leftChild;                 // Si aggiorna smallest per indicare che il figlio sinistro è il
                                             nuovo candidato più piccolo.
  }

// Si esegue lo stesso controllo per il figlio destro. Se l'indice rightChild è valido (ovvero rightChild < pq.size) e
l'elemento nel figlio destro è minore dell'elemento attualmente considerato il più piccolo (pq.data[smallest]). . .
  if (rightChild < pq.size && pq.data[rightChild] < pq.data[smallest]) {
    smallest = rightChild;                // . . . si aggiorna smallest per farlo puntare al figlio destro.
  }

  if (smallest != index) {              // Se smallest non è più uguale a index, significa che uno dei
                                           figli è più piccolo del nodo corrente.
    swap (pq, index, smallest);         // Si scambia il nodo corrente con il nodo più piccolo (usando la
                                           funzione swap), spostando così il nodo corrente più in basso
                                           nell'heap.
    moveDown(pq, smallest);             // Dopo lo scambio, si chiama ricorsivamente moveDown sul
                                           nuovo indice smallest (che ora contiene l'indice del nodo
                                           spostato). Questo processo continua finché il nodo trova la
                                           sua posizione corretta nell'heap
  }
}
```

Funzione ausiliaria swap

```
void swap(priorityQueue::PriorityQueue &pq, int pos1, int pos2) {
  priorityQueue::Elem aux = pq.data[pos1];
  pq.data[pos1] = pq.data[pos2];
  pq.data[pos2] = aux;
}
```