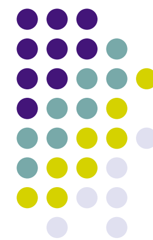
Heap

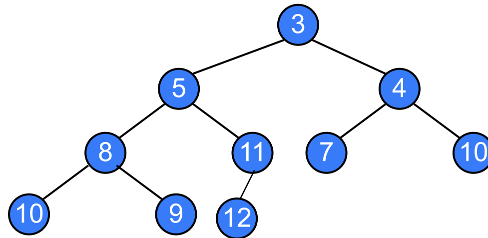
Corso: Strutture Dati
Docente: Annalisa De Bonis



Definizione

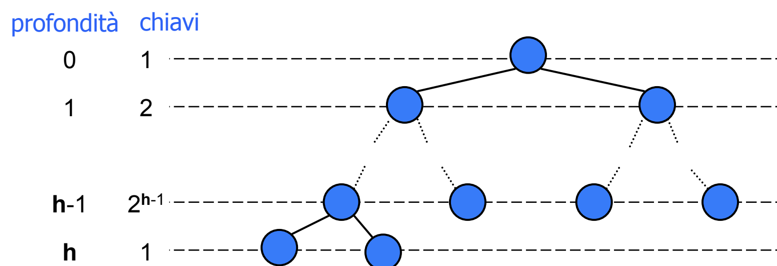
- Un heap è un albero binario che contiene entrate della forma (key, value) nei suoi nodi e soddisfa le seguenti proprietà:
  - Heap-Order: per ogni nodo v ≠ radice
    - key(v) ≥ key(parent(v))
  - Albero binario completo: dato un heap di altezza **h**
    - per $i = 0, \dots, h - 1$, ci sono 2^i nodi di profondità i (tutti i livelli, salvo al più l'ultimo, sono pieni)
    - L'ultimo livello è riempito da sinistra verso destra

Esempio



Altezza di un heap

- Un heap che memorizza n chiavi ha altezza $\lfloor \log n \rfloor$
  Dimostrazione: Sia h l'altezza dell'albero
  - Ci sono 2^i chiavi a profondità $i = 0, \ldots, h - 1$ ed almeno una chiave a profondità **h** → $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1 = 2^h$
  - quindi $h \leq \log n$



Altezza di un heap

- D'altra parte sappiamo che il numero max di nodi di un albero binario di altezza h è
  - $n \leq 1 + 2 + 4 + \ldots + 2^h = 2^{h+1}-1$
- ➔ $2^h \leq n \leq 2^{h+1}-1$ ➔ $\log(n+1)-1 \leq h \leq \log n$

  ➔ $\log(n)-1 < h \leq \log n$

  $h = \lfloor \log n \rfloor$

Strutture Dati 2009-2010
A. De Bonis

Il TDA CompleteBinaryTree

- Specializza il TDA BinaryTree
- Supporta i metodi addizionali
  - Position<E> add(E o):
    - Inserisce una foglia che contiene l'elemento o
      - La nuova foglia ha come padre il primo nodo dell'albero che ha meno di 2 figli
    - Restituisce la position della nuova foglia
  - E remove()
    - rimuove l'ultimo nodo z dell'albero
    - restituisce l'elemento di z

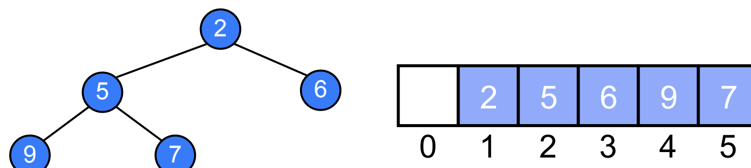
Strutture Dati 2009-2010
A. De Bonis

L'interfaccia CompleteBinaryTree

```
public interface CompleteBinaryTree <E> extends BinaryTree <E>
  {
 public Position <E> add(E elem);
 public E remove();
}
```


Implementazione con i vettori

- Per un albero con n nodi si usa un vettore di dimensione n+1
  - entrata di rango 0 vuota
- Per un nodo di indice i
  - Il figlio sinistro ha indice 2i
  - Il figlio destro ha indice 2i+1



Implementazione con i vettori

- Si usa un **Array list** T

  IndexList<BTPos<E>> T
- L'entrata di indice 0 si pone uguale a null
  - Il numero di elementi nell'heap sarà pari a T.size() -1
- Si usa la classe BTPos (implementa Position) per rappresentare gli elementi nei nodi
  - 2 variabili di istanza:
    - element
    - index (indice della posizione nel vettore)

ArrayListCompleteBinaryTree

```
public class
 ArrayListCompleteBinaryTree<E>
 implements CompleteBinaryTree<E> {
  IndexList<BTPos<E>> T;
```

La classe innestata BTPos

```
protected static class BTPos<E> implements Position<E> {
    E element; int index;
    public BTPos(E elt, int i) {
     element = elt;
     index = i; }
    public E element() { return element; }
    public int index() { return index; }
   public E setElement(E elt) {
    E temp = element;
    element = elt;
    return temp; }
}
```


Il metodo checkPosition

```
protected BTPos<E>checkPosition(Position<E>
   v)
   throws InvalidPositionException
 {
   if (v == null || !(v instanceof BTPos))
     throw new InvalidPositionException(“La
   posizione non è valida");
   return (BTPos<E>) v;
 }
```

I metodi remove e add

```
public E remove() throws EmptyTreeException {
   if(isEmpty()) throw new EmptyTreeException("L'albero è
   vuoto");
   return T.remove(size()).element();   //size()=T.size()-1
 }
public Position<E> add(E e) {
   int i = size() + 1;   // size() + 1=T.size()
   BTPos<E> p = new BTPos<E>(e,i);
   T.add(i, p);
   return p;
 }
```


Il metodo hasLeft

```
public boolean hasLeft(Position<E> v) throws
   InvalidPositionException {
   BTPos<E> vv = checkPosition(v);
   return (2*vv.index() <= size());
 }
```

Il metodo positions

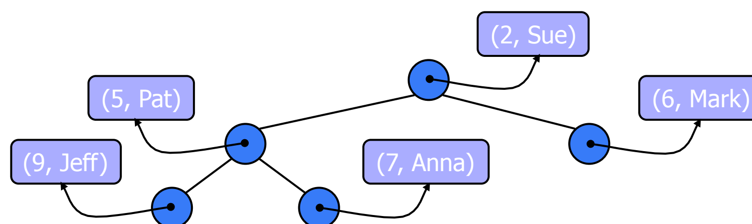
```
public Iterable<Position<E>> positions() {
NodePositionList<Position<E>> P = new
   NodePositionList<Position<E>>();
for(int i=1;i<T.size();i++)
  P.addLast(T.get(i));
return P;
}
```

Il metodo iterator

```
public Iterator<E> iterator() {
NodePositionList<E> list = new
   NodePositionList<E>();
for(int i=1;i<T.size();i++)
  list.addLast(T.get(i).element());
return list.iterator();
}
```

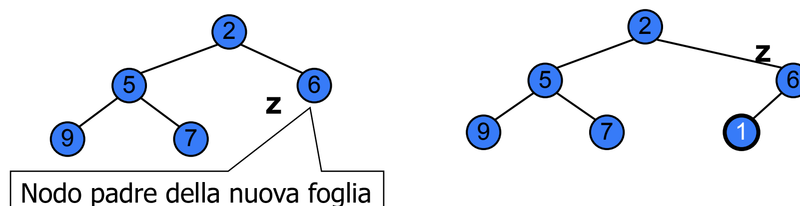
PriorityQueue implementata con heap

- Immagazziniamo un entrata (key, element) in ciascun nodo
- Un comparatore comp definisce la relazione di ordine totale tra le chiavi



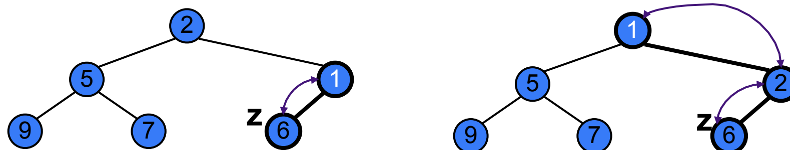
insert

- Il metodo insert del TDA PriorityQueue corrisponde all'inserimento di un'entrata (k,v) nell'heap
- Si svolge in 3 passi
  - Immagazzina (k,v) in un nuovo nodo e lo aggiunge all'heap mediante il metodo add()
  - Ristabilisce l'heap-order

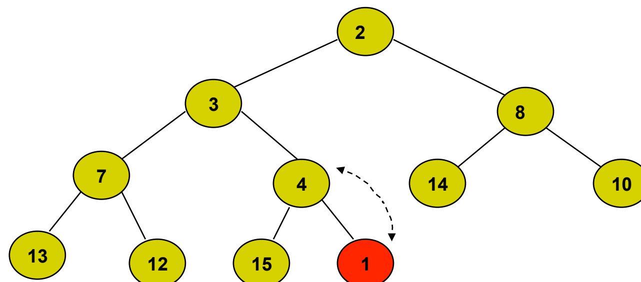


Ripristino dell'heap-order

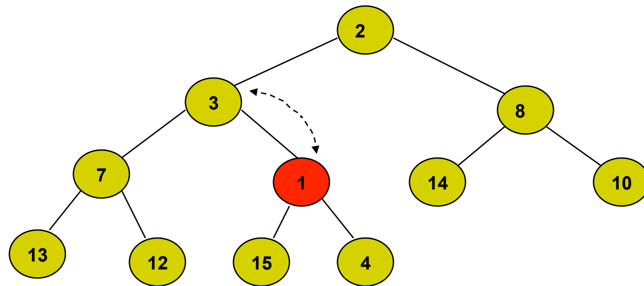
- L'algoritmo upheap ripristina l'heap-order scambiando (k,v) con le entrate dei suoi antenati fino a che (k,v) raggiunge la radice o si incontra un antenato con chiave minore di **k**
- Siccome un heap ha altezza **O**(log **n**), l'algoritmo upheap ha tempo di esecuzione in **O**(log **n**) time



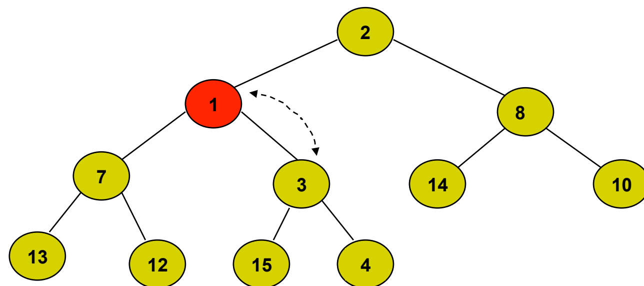
Inserimento della chiave 1



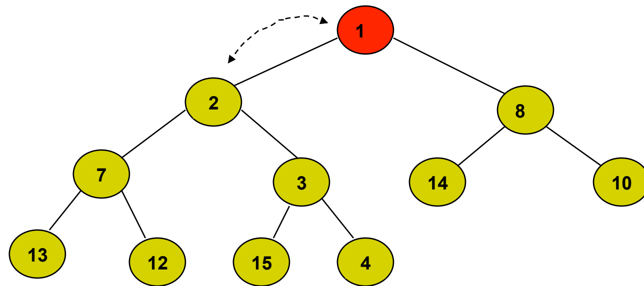
Inserimento della chiave 1



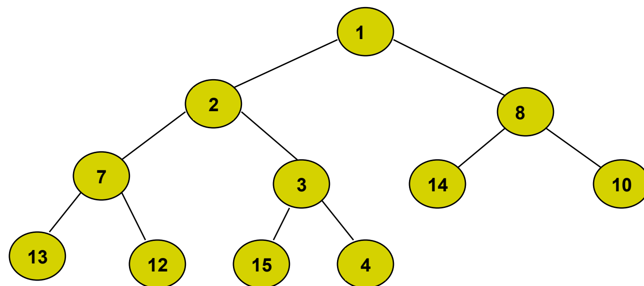
Inserimento della chiave 1



Inserimento della chiave 1

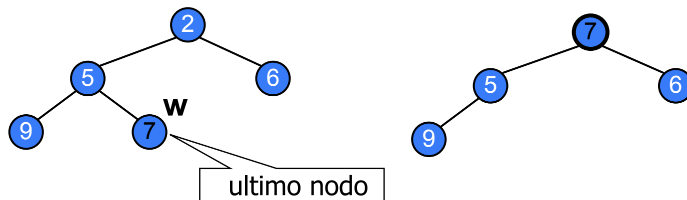


Inserimento della chiave 1



removeMin

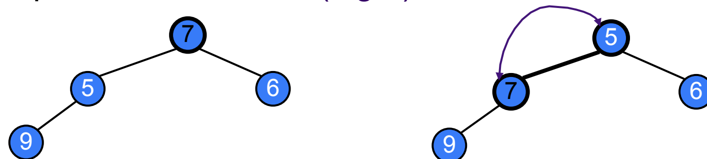
- Il metodo removeMin del TDA PriorityQueue è implementato rimuovendo l'entrata nella radice dell'heap
- L'algoritmo di rimozione consiste di 3 passi:
  - Sostituisci l'entrata della radice con l'entrata dell'ultimo nodo w
  - Rimuovi w con remove()
  - Ripristina l'heap-order che potrebbe essere stato violato dalla sostituzione dell'entrata della radice



Strutture Dati 2009-2010
A. De Bonis

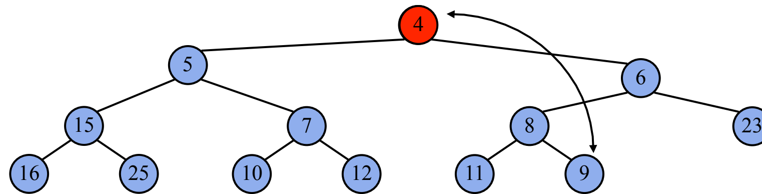
Downheap

- L'algoritmo downheap ripristina l'heap-order scambiando ad ogni passo l'entrata (**k**,**v**) con l'entrata del figlio che ha chiave più piccola
- L'algoritmo downheap termina quando (**k**,**v**) raggiunge un nodo **z** tale che **z** è una foglia o le chiavi dei figli di z sono maggiori o uguali di **k**
- Siccome l'altezza dell'heap è **O**(log n), downheap ha tempo di esecuzione **O**(log n)

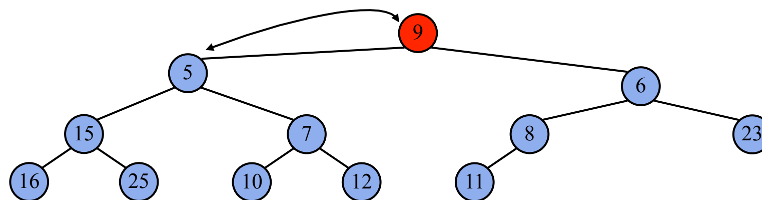


Strutture Dati 2009-2010
A. De Bonis

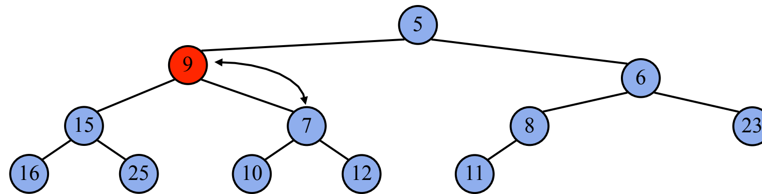
Cancellazione del minimo



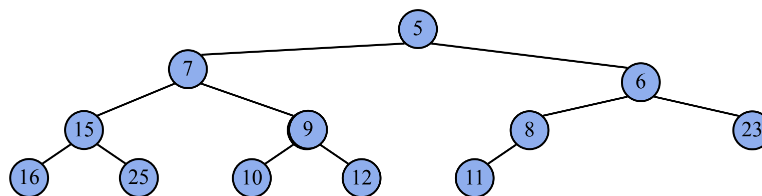
Cancellazione del minimo



Cancellazione del minimo



Cancellazione del minimo



La classe HeapPriorityQueue

```
public class HeapPriorityQueue<K,V> implements
   PriorityQueue<K,V> {
  protected CompleteBinaryTree<Entry<K,V>> heap;
  protected Comparator<K> comp;
```


La classe innestata MyEntry

```
protected static class MyEntry<K,V> implements
   Entry<K,V> {
  protected K key; protected V value;
  public MyEntry(K k, V v) { key = k; value = v; }
   public K getKey() { return key; }
   public V getValue() { return value; }
   public String toString() { return "(" + key + "," +
   value + ")"; }
}
```

La classe HeapPriorityQueue

```
public HeapPriorityQueue() {
     heap = new ArrayListCompleteBinaryTree<Entry<K,V>>();
     comp = new DefaultComparator<K>();
 }
public HeapPriorityQueue(Comparator<K> c) {
  heap = new ArrayListCompleteBinaryTree<Entry<K,V>>();
  comp = c;
}
public int size() { return heap.size(); }
public boolean isEmpty() { return heap.size() == 0; }
```


La classe HeapPriorityQueue

```
protected void checkKey(K key) throws
 InvalidKeyException {
  try { comp.compare(key,key); }
  catch(Exception e) { throw new
 InvalidKeyException(“chiave non valida"); }
```

La classe HeapPriorityQueue

```
public Entry<K,V> insert(K k, V x) throws
   InvalidKeyException {
   checkKey(k);
   Entry<K,V> entry = new MyEntry<K,V>(k,x);
   upHeap(heap.add(entry));
   return entry; }
```


La classe HeapPriorityQueue

```
protected void upHeap(Position<Entry<K,V>> v)
  { Position<Entry<K,V>> u;
 while (!heap.isRoot(v)) {
   u = heap.parent(v);
   if (comp.compare(u.element().getKey(),
                      v.element().getKey())  <= 0)
      break;
  swap(u, v);
  v = u;
 }
}
```

La classe HeapPriorityQueue

```
public Entry<K,V> removeMin()
   throws EmptyPriorityQueueException {
  if (isEmpty())
    throw new EmptyPriorityQueueException("Coda a priorità
   vuota");
  Entry<K,V> min = heap.root().element();
  if (size() == 1) heap.remove();
  else {
   heap.replace(heap.root(), heap.remove());
   downHeap(heap.root()); }
  return min; }
```


La classe HeapPriorityQueue

```
protected void downHeap(Position<Entry<K,V>> r) {
 while (heap.isInternal(r)) {
   Position<Entry<K,V>> s;
   if (!heap.hasRight(r))     s = heap.left(r);
   else if (comp.compare(heap.left(r).element().getKey(),
                         heap.right(r).element().getKey()) <=0)
     s = heap.left(r);
  else s = heap.right(r);
  if (comp.compare(s.element().getKey(), r.element().getKey())<0)
 {
  swap(r, s);
   r = s; }
 else break; } //fine while
}
```

La classe HeapPriorityQueue

```
protected void swap(Position<Entry<K,V>> x,
   Position<Entry<K,V>> y) {
   Entry<K,V> temp = x.element();
   heap.replace(x, y.element());
   heap.replace(y, temp); }
```


Esercizi

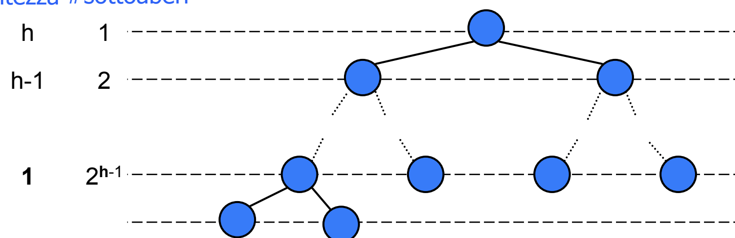
- Implementare un algoritmo di ordinamento (HeapSort) che usa come struttura ausiliaria una PriorityQueue implementata con heap. Discuterne la complessità computazionale.
- Scrivere una classe PQStack che implementa Stack ed ha solo due variabili di istanza una delle quali è di tipo PriorityQueue

Costruzione di un heap con n entrate in tempo lineare

- Se conosciamo in anticipo gli elementi che costituiscono la coda a priorità allora possiamo costruire l'heap che implementa la coda a priorità in tempo lineare
  - Non invochiamo n volte insert() sulla coda come nel codice nella prossima slide

Costruzione bottom-up di un heap

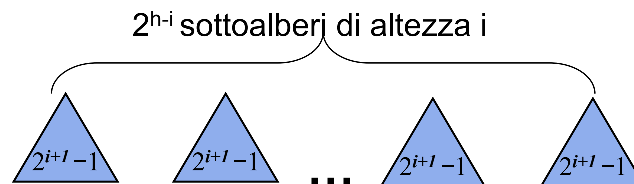
- Inseriamo tutti le entrate nell'albero
- Ripristiniamo heap-order dal basso verso l'alto
- Per ogni $j=1,\dots,h$, invochiamo down-heap su tutti i nodi di altezza j



Costruzione bottom-up di un heap

- Possiamo costruire un heap contenente n chiavi in $h = O(\log n)$ fasi
- Dopo la fase i, tutti i sottoalberi di altezza i soddisfano l'heap-order. Ogni fase richiede tempo $O(i2^{h-i})$
- In totale il tempo di esecuzione è $O(\sum_{i=1}^{h-1} i\, 2^i) = O(2^h) = O(n)$

2^{h-i} sottoalberi di altezza i



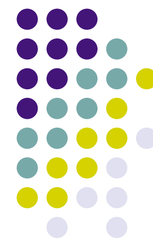
Esercizio

- Aggiungere alla classe HeapPriorityQueue il costruttore

  HeapPriorityQueue(V v[] , K k[])

  che costruisce la coda a priorità in tempo lineare

Adaptable Priority Queue



Il TDA Adaptable Priority Queue

- Il TDA Adaptable Priority Queue estende il TDA Priority Queue con le seguenti operazioni:
  - Remove(e) : rimuove e restituisce l'entrata **e** dalla coda a priorità
  - replaceValue(e,v) : rimpiazza con **v** il valore dell'entrata **e** restituendo in output il vecchio valore
  - replaceKey(e,k) : rimpiazza con **k** la chiave dell'entrata **e** restituendo in output la vecchia chiave

L'intefaccia AdaptablePriorityQueue

```
public interface AdaptablePriorityQueue<K,V> extends
PriorityQueue<K,V> {

  public Entry<K,V> remove(Entry<K,V> e);

  public K replaceKey(Entry<K,V> e, K key) throws
InvalidKeyException;

public V replaceValue(Entry<K,V> e, V value);
}
```


Motivazioni

- Alcuni algoritmi richiedono di cancellare un'entrata qualsiasi o di aggiornare l'elemento o la chiave di un'entrata qualsiasi
- Esempio:
  - Prim e Dijkstra effettuano O(E) operazioni di aggiornamento di key (decreaseKey)

Implementazione

- I metodi remove, replaceElement e replaceKey richiedono di conoscere il posto in cui l'entrata si trova all'interno dell'heap
- Esempio:
  - Durante l'esecuzione di Prim e Dijkstra, per effettuare l'aggiornamento della chiave di un vertice v occorre conoscere il posto dell'entrata associata a v nella coda a priorità
- Soluzione inefficiente: il metodo che implementa decreaseKey effettua la ricerca dell'entrata associata a v
- Soluzione efficiente: indichiamo al metodo che implementa decreaseKey il posto dell'entrata associata a v

Locator

- Il TDA Coda a Priorità non supporta la nozione di Position sebbene possa essere implementato mediante un contenitore posizionale
- Abbiamo bisogno di un meccanismo per accedere direttamente ad un'entrata della coda

Locator

- Possiamo aggiungere alla classe che implementa Entry un campo che tiene traccia del posto (location) dove si trova l'entrata nella struttura dati usata per implementare la coda
- Se la struttura dati usata per implementare la coda supporta la nozione di Position allora location sarà di tipo Position
  - Implementazione con una lista: location contiene il riferimento alla Position della lista in cui è contenuta l'entrata
  - Implementazione con Heap: location contiene il riferimento al nodo dell'heap contenente l'entrata

Locator

- Quando un'entrata viene inserita nella coda la sua location viene inizializzata con il riferimento alla sua Position nella struttura dati
- La location viene aggiornata ogni volta che l'entrata cambia posizione nella struttura dati
- Esercizio: Analizzare la complessità dei metodi di AdaptablePriorityQueue nelle implementazioni mediante lista ordinata, mediante lista non ordinata e mediante heap. Si assuma che ciascuna entrata contenga la sua location.

HeapAdaptablePriorityQueue

```
public class HeapAdaptablePriorityQueue<K,V>
  extends HeapPriorityQueue<K,V>
  implements AdaptablePriorityQueue<K,V> {

  public HeapAdaptablePriorityQueue() {
    super();
  }

 public HeapAdaptablePriorityQueue(Comparator comp) {
    super(comp);
  }
```


La classe innestata LocationAwareEntry

```
protected static class LocationAwareEntry<K,V>
    extends MyEntry<K,V> implements Entry<K,V> {

     protected Position <Entry<K,V>> loc;

    public LocationAwareEntry(K k, V v) {
      super(k, v);
    }
    public LocationAwareEntry(K k, V v, Position pos) {
      super(k, v);
      loc = pos;
    }
```

La classe innestata LocationAwareEntry

```
protected Position < Entry<K,V>> location() {
    return loc;
  }
  protected Position < Entry<K,V> >setLocation(Position< Entry<K,V>> pos) {
    Position Entry<<K,V>> oldPosition = location();
    loc = pos;
    return oldPosition;
  }
  protected K setKey(K k) {
    K oldKey = getKey();
    key = k;
    return oldKey;
  }
  protected V setValue(V v) {
    V oldValue = getValue();
    value = v;
    return oldValue;
  }
}
```


Il metodo insert

```
public Entry<K,V> insert (K k, V v) throws InvalidKeyException {
   checkKey(k);
   LocationAwareEntry<K,V> entry = new
   LocationAwareEntry<K,V>(k,v);
   Position <Entry<K,V>> z = heap.add(entry);
   entry.setLocation(z);
   upHeap(z);
   return entry;
 }
```

Il metodo remove

```
public Entry<K,V> remove(Entry<K,V> entry) throws
   InvalidEntryException {
   LocationAwareEntry<K,V> ee = checkEntry(entry);
   Position < Entry<K,V>>p = ee.location();
   if(size() == 1)
     return (Entry<K,V>) heap.remove();
   replaceEntry(p,(LocationAwareEntry<K,V>)heap.remove());
   upHeap(p);
   downHeap(p);
   ee.setLocation(null);
   return ee;
 }
```

Il metodo replaceKey

```
public K replaceKey(Entry<K,V> entry, K k)
   throws InvalidEntryException
 {
   checkKey(k);
   LocationAwareEntry<K,V> ee = checkEntry(entry);
   K oldKey = ee.setKey(k);
   upHeap(ee.location());
   downHeap(ee.location());
   return oldKey;
 }
```

Il metodo replaceValue

```
public V replaceValue(Entry<K,V> e, V value)
    throws InvalidEntryException
  {
    LocationAwareEntry<K,V> ee = checkEntry(e);
    return ee.setValue(value);
  }
```

Il nuovo metodo ausiliario swap

```
protected void swap(Position<Entry<K,V>> u,
   Position<Entry<K,V>> v) {
    super.swap(u,v);
    getEntry(u).setLocation(u);
    getEntry(v).setLocation(v);
 }
```

Il metodo ausiliario replaceEntry

```
protected Position< Entry<K,V>> replaceEntry
   (Position<Entry<K,V>> v,
   LocationAwareEntry<K,V> e) {
     heap.replace(v,e);
     return e.setLocation(v);
}
```

Il metodo ausiliario getEntry

```
protected LocationAwareEntry<K,V> getEntry
 (Position < Entry<K,V> >p) {
  return (LocationAwareEntry<K,V>) p.element();
}
```

Il metodo checkEntry

```
protected LocationAwareEntry<K,V> checkEntry
  (Entry<K,V> e)
   throws InvalidEntryException
 {
  if(e == null || !(ent instanceof LocationAwareEntry))
    throw new InvalidEntryException("entrata non valida");
  return (LocationAwareEntry) e;
 }
```


Esercizio

- Scrivere l'implementazione della classe HeapAdaptablePriorityQueue (deve essere inclusa nel progetto)
- Testare tutti i metodi della classe

Esercizi

- Sviluppare la classe SortedListAdaptablePriorityQueue che implementa AdaptablePriorityQueue estendendo la classe SortedListPriorityQueue
- Sviluppare la classe UnsortedListAdaptablePriorityQueue che implementa AdaptablePriorityQueue estendendo UnsortedListPriorityQueue