lezione 9

Heap

e

Alberi posizionali generali

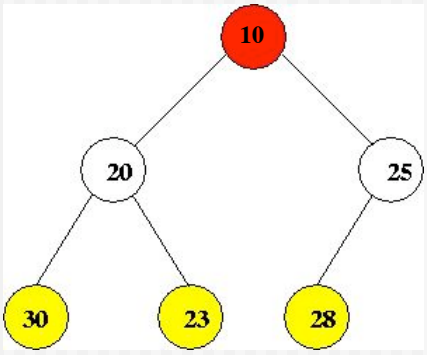
min-heap binario

Un min-heap è un albero binario quasi completo in cui ogni nodo i diverso dalla radice soddisfa la seguente proprietà:

> "il valore memorizzato in parent(i) è minore o uguale quello memorizzato nel nodo i"

Equivalentemente, si può dire che:

- la radice ha valore minimo
- il valore dei nodi cresce muovendo dalla radice verso le foglie lungo tutti i cammini

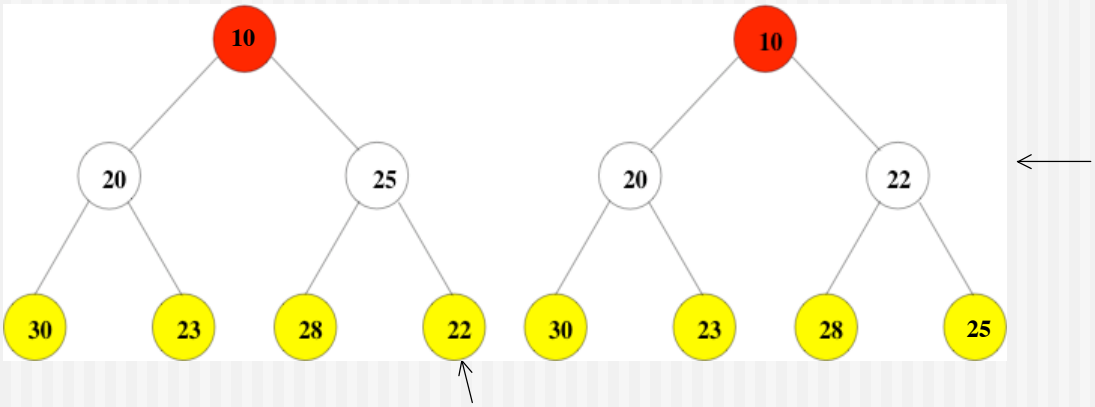


Un max-heap può essere definito analogamente.

min-heap: inserimento

Per mantenere l'albero quasi completo si inserisce un nuovo elemento come foglia posta "a destra dell'ultima foglia", cioè l'albero viene completato per livelli.

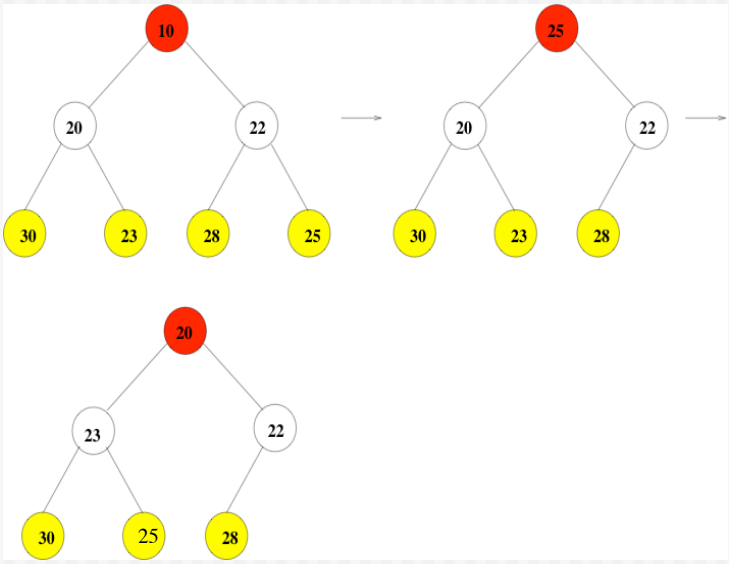
La nuova foglia viene poi sistemata nella posizione corretta del cammino che porta dalla sua posizione alla root.



min-heap: rimozione minimo

L'estrazione del minimo consiste nella rimozione della radice dello heap. Come si realizza questa operazione?

- eliminando l'ultima foglia inserita e salvandone chiave e valore in radice
- ripristinando la proprietà di essere min-heap (usando la procedura heapify)



Heap binari con array

Lo heap binario si rappresenta in modo efficiente usando un array che, letto da sinistra a destra, rappresenta la visita in ampiezza dell'albero.

Ogni nuovo elemento si inserisce dopo l'ultima foglia più a destra (cioè nella prima posizione libera dell'array) e quindi l'albero viene riempito per livelli. Il fatto che lo heap sia un albero quasi completo permette di stabilire delle relazioni tra i figli e il padre di ogni nodo e l'indice in cui compare nell'array.

Precisamente, supponendo che gli indici dell'array partano da 1 si ha:

- parent(i) = i/2

- left(i) = 2i

- right(i) = 2i+1

La complessità delle operazioni di inserimento e rimozione è O(lg n) dove n è il numero di elementi dello heap.

Tipo di dato Coda a Priorità

Il tipo di dato coda a priorità permette di gestire collezioni di elementi a cui sono associate chiavi prese da un dominio totalmente ordinato.
I dati ammessi sono quindi un insieme di coppie (*elem, key*), dove la chiave *key* esprime la priorità associata all'elemento *elem* nella coda.
Si tratta cioè di una coda in cui la disciplina FIFO viene modificata dalla priorità associata a ciascun elemento:

- nelle code a max-priorità, maggiore è la priorità e più velocemente l'elemento uscirà dalla coda;
- nelle code a min-priorità, minore è la priorità e più velocemente l'elemento uscirà dalla coda.

Tipo di dato Coda a Priorità

Le operazioni definite per una coda a max-priorità sono:

- insert(Q, elem,p): inserimento di elem con priorità p
- extract-max(Q): elimina e restituisce l'elemento di priorità massima
- maximum(Q): ritorna l'elemento con priorità massima
- increase-key(Q,x,k): aumenta il valore della chiave dell'elemento x al nuovo valore di k, che si suppone sia almeno pari al valore corrente della chiave associata ad x

Le operazioni definite per una coda a min-priorità sono analoghe.

Realizzazione di code a priorità

Le code priorità possono essere realizzate in vari modi (ad esempio: con array, liste concatenate, alberi).

In particolare, esse possono essere realizzate in modo efficiente con heap binari.

Se una coda a max-priorità viene realizzata con un max-heap binario, la complessità delle operazioni definite è:

- inserimento → O(lg n)
- estrazione del massimo → O(lg n)
- interrogazione del massimo → O(1)
- incremento chiave di un elemento → O(lg n)

dove n è il numero di elementi in coda

Heap ternari

Gli heap ternari sono una generalizzazione degli heap binari in cui gli alberi non sono più binari ma ternari (cioè sono alberi in cui un nodo può avere tre figli).

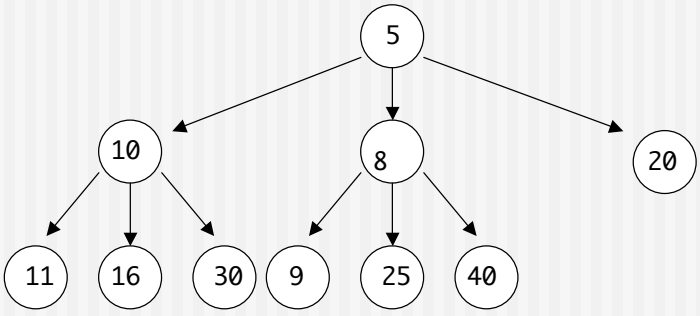
La definizione di un max-heap ternario è la seguente:

Un max-heap ternario è un albero ternario quasi completo in cui ogni nodo i diverso dalla radice soddisfa la seguente proprietà:

"il valore memorizzato in parent(i) è maggiore o uguale quello memorizzato nel nodo i"

Un min-heap ternario è definito in modo analogo.

min-heap ternario: esempio



Le operazioni di inserimento e rimozione del minimo vengono realizzate in modo analogo al min-heap binario.

Heap ternari con array

Anche uno heap ternario si rappresenta in modo effiiente con un array.

Cosa cambia rispetto allo heap binario? …quasi nulla!

Anche in questo caso l'array, letto da sinistra a destra,, rappresenta la visita in ampiezza dell'albero.

Ogni nuovo elemento si inserisce dopo l'ultima foglia più a destra (cioè nella prima posizione libera dell'array) e quindi l'albero viene riempito per livelli.

Heap ternari con array

Anche nel caso ternario, il fatto che lo heap sia un albero quasi completo permette di stabilire delle relazioni tra i figli e il padre di ogni nodo e l'indice in cui compare nell'array.

Precisamente, supponendo che gli indici dell'array partano da 1 si ha:

- parent(i) = i/3
- first(i) = 3i - 1
- second(i) = 3i
- third(i) = 3i + 1

La complessità delle operazioni di inserimento e rimozione è $O(\lg_3 n)$ dove n è il numero di elementi dello heap.

Code a priorità con heap ternari

Abbiamo visto che le code a priorità vengono realizzate in modo efficiente con gli heap binari.

Cosa cambia con gli heap ternari?

Se la coda viene realizzata con un max-heap ternario, la complessità delle operazioni definite è:

- inserimento → $O(\lg_3 n)$
- estrazione del massimo → $O(\lg_3 n)$
- interrogazione del massimo → $O(1)$
- incremento chiave di un elemento → $O(\lg_3 n)$

dove n è il numero di elementi in coda

Heap k-ari

Le definizioni che abbiamo visto per gli heap si possono generalizzare al caso di alberi k-ari.

Un albero è k-ario se tutti i suoi nodi hanno al più k figli.

Un albero k-ario è completo se tutti i suoi nodi interni hanno esattamente k figli e tutte le foglie si trovano allo stesso livello.

Quindi:

Un min-heap k-ario è un albero k-ario quasi completo in cui ogni nodo i diverso dalla radice soddisfa la seguente proprietà:

"il valore memorizzato in parent(i) è minore o uguale quello memorizzato nel nodo i"

Si possono analogamente estendere

- la rappresentazione degli heap k-ari con array e
- la realizzazione di code a priorità con heap k-ari

Alberi posizionali generali

Definizione ricorsiva: Un albero posizionale T è un insieme di nodi tale che:

- o è vuoto
- oppure contiene un nodo radice x, e un insieme di alberi posizionali T1, T2,…,Tn detti primo, secondo, … n-esimo sottoalbero di T e tali che, per ogni i:
  - Ti è figlio di x
  - se Ti non è vuoto allora anche T1, … Ti-1 non sono vuoti.

Analogamente agli alberi binari, sono definiti anche per gli alberi generali degli algoritmi di visita in ampiezza e profondità (già presentati dalla prof. Bossi).

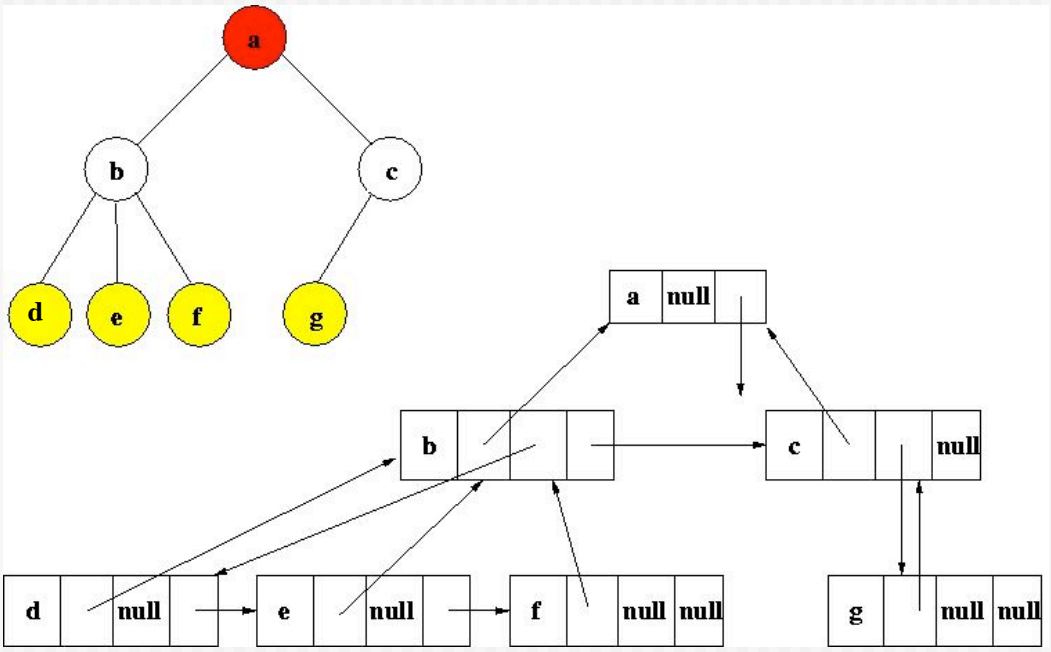
Alberi: rappresentazione figlio-fratello

Usiamo una struttura collegata per rappresentare un albero generale.
Allora un albero generale T è individuato dalla sua radice e ciascun nodo dell'albero contiene le seguenti informazioni:

- il dato memorizzato nel nodo (key)
- il riferimento al primo figlio (child)
- il riferimento al fratello successivo (sibling)
- il riferimento al padre (parent) *opzionale*

NOTA:
-le foglie non hanno figli
-la radice non ha fratelli

Esempio



Alberi: assunzioni per l'implementazione

Operazioni di spostamento: utilizziamo un cursore per spostarci all'interno dell'albero

- reset: riporta il cursore sulla radice
- moveDown(i) si sposta sull'i-esimo figlio, se possibile
- moveUp si sposta sul padre, se possibile

Inserimento:

- il nuovo nodo viene inserito come figlio del nodo corrente aggiungendolo come fratello dell'ultimo figlio (sempre possibile!)

Cancellazione:

- il nodo corrente viene cancellato solo se è una foglia
- dopo la cancellazione il nodo corrente diventa il padre del nodo cancellato (se esiste)

Struttura del package Trees

Il package comprende i seguenti file:

- Tree.java: interfaccia per gli alberi generali
- TreeNode.java: classe che permette di memorizzare un nodo
- GenTree.java: classe che realizza le operazioni definite nell'interfaccia Tree.java

classe TreeNode

```
package Trees;
class TreeNode {
    Object key;           // valore associato al nodo
    TreeNode parent;      // padre del nodo
    TreeNode child;       // primo figlio del nodo
    TreeNode sibling;     // fratello destro del nodo

    // post: ritorna un albero di un solo nodo, con valore value e
    //       sottoalberi vuoti
    TreeNode(Object ob) {
        key = ob;
        parent = child = sibling = null;
    }
    // post: ritorna un albero contenente value e i sottoalberi
    //       specificati
    TreeNode(Object ob, TreeNode parent, child, sibling) {
        key = ob;
        this.parent = parent;
        this.child = child;
        this.sibling = sibling;
    }
}
```

Interfaccia Tree

```
package Trees;
public interface Tree {

    // post: ritorna il numero di elementi dell'albero
    public int size();

    // post: ritorna true sse l'albero e' vuoto
    public boolean isEmpty();

    // post: svuota l'albero
    public void clear();

    // pre:  ob non nullo
    // post: se l'albero e' vuoto, inserisce ob come root.
    //       Altrimenti inserisce ob come figlio del nodo corrente.
    //       Ritorna true.
    public boolean insert(Object ob);

    // pre:  albero non vuoto
    // post: se l'elemento corrente e' una foglia, la rimuove e ne ritorna
    //       il valore; il nuovo elemento corrente e' il padre del nodo
    //       rimosso (se esiste).
    //       Altrimenti ritorna null
    public Object remove ();
```

Interfaccia Tree

```
    // pre: albero non vuoto!
    // post: ritorna il valore del nodo puntato da cursor
    public Object getValue();

    // pre: albero non vuoto e ob diverso da null
    // post: imposta il valore del nodo puntato da cursor
    public void setValue(Object ob);

    // post: cursor si sposta sulla radice
    public void reset();

    // post: se possibile cursor si sposta sul figlio in posizione
    //       childIndex e ritorna true; altrimenti rimane dov'e' e
    //       ritorna false
    public boolean moveDown (int childIndex);

    // post: se possibile sposta cursor sul nodo padre e ritorna true;
    //       altrimenti ritorna false;
    public boolean moveUp ();

    // post: ritorna una stringa che rappresenta l'albero, secondo la
    //       visita in profondita'
    public String preorderVisit();}
```

classe GenTree (1)

```
package Trees;
public class GenTree {
    private TreeNode root;      // radice dell'albero
    private int count;          // numero di nodi dell'albero
    private TreeNode cursor;    // riferimento al nodo corrente

    // post: costruisce un albero vuoto
    public GenTree() { clear(); }

    // post: svuota l'albero
    public void clear() { root = cursor = null; count = 0; }

    // post: ritorna il numero di nodi dell'albero
    public int size() { return count; }

    // post: ritorna true se l'albero e' vuoto; false altrimenti
    public boolean isEmpty() { return count == 0; }

    // pre: albero non vuoto!
    // post: ritorna il valore del nodo puntato da cursor
    public Object getValue() { return cursor.key; }

    // pre: albero non vuoto e ob diverso da null
    // post: imposta il valore del nodo puntato da cursor
    public void setValue(Object ob) { cursor.key = ob; }
```

classe GenTree (2)

```
// pre:  ob non nullo
    // post: l'oggetto e' inserito come figlio del nodo corrente,
     //       o come radice se l'albero e' vuoto. Ritorna true.
    public boolean insert (Object ob) {
        if (isEmpty())
            cursor = root = new TreeNode(ob);
        else {
            if (cursor.child == null)
               cursor.child = new TreeNode(ob, cursor, null,null);
            else {
               TreeNode index = cursor.child;
               while (index.sibling != null)
                  index = index.sibling;
               index.sibling = new TreeNode(ob,cursor, null,null);
            }
        }
        count++;
        return true;
    }
```

classe GenTree (3)

```
// pre:  albero non vuoto
// post: se cursor e' una foglia, la rimuove e ne ritorna il valore;
//       cursor viene assegnato al padre del nodo rimosso (se esiste)
//       Altrimenti ritorna null
public Object remove () {
    if (cursor.child != null)
       return null;

    Object temp = cursor.key;
    if (cursor == root)
       cursor = root = null;
    else {
        if (cursor.parent.child == cursor) // cursor e' il primo figlio
           cursor.parent.child = cursor.sibling;
        else {
           TreeNode index = cursor.parent.child;
           while (index.sibling != cursor)
              index = index.sibling;
           index.sibling = cursor.sibling;
        }
        cursor = cursor.parent;
    }
    count --;
    return temp;
}
```

classe GenTree (4)

```
// post: cursor si sposta sulla radice
public void reset() { cursor = root; }

// post: se possibile cursor si sposta sul figlio in posizione childIndex
//       e ritorna true; altrimenti rimane dov'e' e ritorna false
public boolean moveDown (int childIndex) {
    if (isEmpty() || cursor.child == null || childIndex < 1)
       return false;
    TreeNode index = cursor.child;
    int i;
    for (i = 1; i<childIndex && index.sibling != null; i++)
        index = index.sibling;
    if (i == childIndex) {
       cursor = index;
       return true;
    }
    else
       return false;
}
```

classe GenTree (5)

```
    // post: se possibile sposta cursor sul nodo padre e ritorna true;
    //       altrimenti ritorna false;
    public boolean moveUp () {
        if (isEmpty() || cursor.parent == null)
           return false;
        else {
           cursor=cursor.parent;
           return true;
        }
    }

 // post: ritorna una stringa che rappresenta l'albero
    public String DFvisit() {
        StringBuffer sb = new StringBuffer();
        sb.append("Tree: <");
        if (root != null)
            DFformat(root,sb);
        sb.append(">");
        return sb.toString();
    }
```

classe GenTree (6)

```
    // pre: n diverso da null
    // post: appende a sb una stringa che rappresenta il sottoalbero con
    //       radice n
    private void DFformat(TreeNode n, StringBuffer sb) {
        sb.append(" " + n.key.toString() + " ");
        if (n.child != null) {
            sb.append("<");
            DFformat(n.child,sb);
            sb.append("> ");
        }

        if (n.sibling != null)
           DFformat(n.sibling,sb);
    }

}
```