

# Laboratorio di algoritmi e strutture dati

Docente: Violetta Lonati

giovedì 15 dicembre 2016

L'esercitazione di oggi prevede di utilizzare gli alberi binari e in particolare gli alberi binari di ricerca.

## 1 Esercizio: costruzione e stampa di alberi binari

Scrivete un programma che:

1. generi a caso una sequenza di interi (di lunghezza massima fissata con una opportuna macro) e la memorizzi in un array;
2. costruisca un albero binario a partire dall'array (come descritto in seguito);
3. stampi l'albero nella rappresentazione *a sommario* (come descritta in seguito);
4. stampi i nodi dell'albero negli ordini determinati rispettivamente dalle visite in preordine, inordine e postordine.

### 1.1 Funzioni preliminari per elaborare alberi binari

Innanzitutto, definite i tipi

---

```
struct bit_node {
    Item item;
    struct bit_node *l, *r;
}

typedef struct bit_node *Bit_node;
```

---

e scrivete (e testate!) le funzioni

---

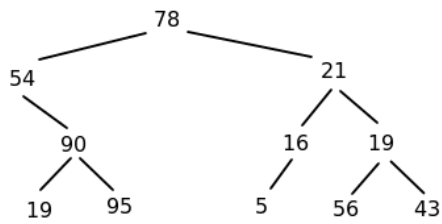
```
Bit_node bit_new( Item );
void bit_preorder( Bit_node );
void bit_inorder( Bit_node );
void bit_postorder( Bit_node );
```

---

oltre ad eventuali altre funzioni utili sugli alberi binari, come dichiarate in `bistree.h`. Ricordate che le funzioni riguardanti gli alberi sono denotate dal prefisso `bit` (**b**inary **t**ree), mentre quelle specifiche per gli alberi di ricerca sono denotate dal prefisso `bist` (**b**inary **s**earch **t**ree).

### 1.2 Stampa di alberi *a sommario*

Scrivete quindi una funzione che stampi un albero binario nella rappresentazione usata nei sommari dei libri, oppure in un file browser, come nel seguente esempio:



```

*78
 *54
  *
 *90
  *19
  *95
*21
 *16
  *5
  *
 *19
  *56
  *43
  
```

Notate che, nel caso in cui un nodo abbia un solo figlio, per poter distinguere tra figlio destro e figlio sinistro, è necessario evidenziare con una riga vuota l'assenza del figlio mancante.

Per realizzare questa funzione, potete usare una visita in preorder ricorsiva, in cui la funzione di visita è data dalla stampa, correttamente indentata, del contenuto del nodo. Il numero di spazi può essere dato come parametro della funzione, che quindi avrà prototipo

---

```
void bit_printassummary( Bit_node p, int spaces );
```

---

e andrà chiamata nel modo seguente:

---

```
bit_printassummary( root, 0 );
```

---

### 1.3 Dal vettore all'albero

Scrivete una funzione

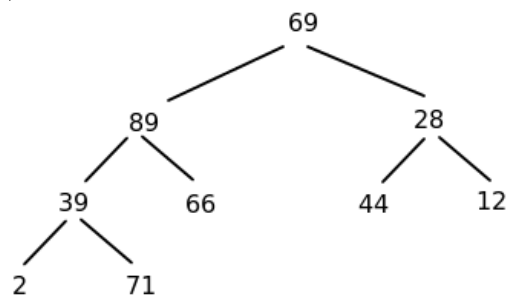
---

```
Bitnode bit_arr2tree( Item a[], int size, int i)
```

---

che, dato un array  $a$  di lunghezza  $size$  ed un indice  $i$ , costruisca ricorsivamente l'albero binario (completo) con radice contenente l'Item  $a[0]$  e tale che valga la seguente proprietà: se un nodo è etichettato con  $a[i]$ , allora il suo figlio sinistro è etichettato con  $a[2*i+1]$  e il suo figlio destro è etichettato con  $a[2*i+2]$ .

Ad esempio, dato  $a = \{69, 89, 28, 39, 66, 44, 12, 2, 71\}$ , la funzione deve costruire l'albero



## 2 Esercizio: tabella delle occorrenze delle parole in un testo

L'obiettivo dell'esercizio è scrivere un programma che sappia fare alcune semplici operazioni riguardanti le occorrenze di parole in un testo. Formalmente, una *occorrenza* è definita come una coppia  $\langle word, n \rangle$ , dove *word* è una parola di lettere minuscole e *n* un intero positivo che rappresenta il numero di occorrenze di *word*. Sull'insieme delle occorrenze consideriamo l'ordine indotto dall'ordinamento alfabetico sulle parole.

L'input del programma è dato da un testo *t*, terminato dalla parola STOP, seguito da una sequenza di parole  $w_1, w_2, \dots, w_n$ , terminata ancora dalla parola STOP. Si assume che tutte le parole abbiano al massimo WORD caratteri, dove il valore di WORD è definito nel programma con un'opportuna macro. La punteggiatura va ignorata e non si deve fare distinzione tra lettere maiuscole e minuscole.

L'output deve essere formato da tre parti:

- l'elenco, in ordine alfabetico, di tutte le occorrenze del testo;
- lo stesso elenco, ma nell'ordine inverso;
- il numero di occorrenze della parola  $w_i$ , per ogni *i* da 1 a *n*.

### Esempio di funzionamento

INPUT	OUTPUT
A cavallo, a cavallo, il re del Portogallo! Il re delle paperette suona suona le trombette. STOP il re gatto STOP	OCCORRENZE DI PAROLE NEL TESTO IN ORDINE ALFABETICO: ***** a 2 cavallo 2 del 1 delle 1 il 2 le 1 paperette 1 portogallo 1 re 2 suona 2 trombette 1  OCCORRENZE DI PAROLE NEL TESTO IN ORDINE INVERSO: ***** trombette 1 suona 2 re 2 portogallo 1 paperette 1 le 1 il 2 delle 1 del 1 cavallo 2 a 2  RICERCA DI PAROLE: ***** il 2 re 2 gatto 0

## 2.1 Struttura dati

Occorre rappresentare un insieme di occorrenze su cui siano definite le usuali operazioni sugli insiemi dinamici ordinati (inserimento, ricerca, ecc.). Poichè tutte le operazioni (a parte le operazioni di stampa) richiedono la ricerca di una parola, occorre usare una struttura dati in cui la ricerca per nome sia efficiente.

E' quindi utile usare un albero binario di ricerca, in cui gli Item siano puntatori ad occorrenze, le quali a loro volta possono essere definite come strutture del tipo:

---

```

struct occorrenza{
    char* word;
    int n;
};

```

---

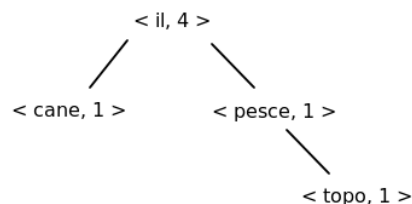
La ricerca avviene in base alle parole, pertanto la chiave di un'occorrenza  $\langle word, n \rangle$  sarà data dalla sua prima componente *word*.

La relazione di ordine tra le occorrenze è quella indotta dall'ordinamento lessicografico tra parole; a tale relazione dovrà quindi fare riferimento al funzione `cmp` dichiarata in `itemkey.h` (potete usare `strcmp` dal file header `string.h`).

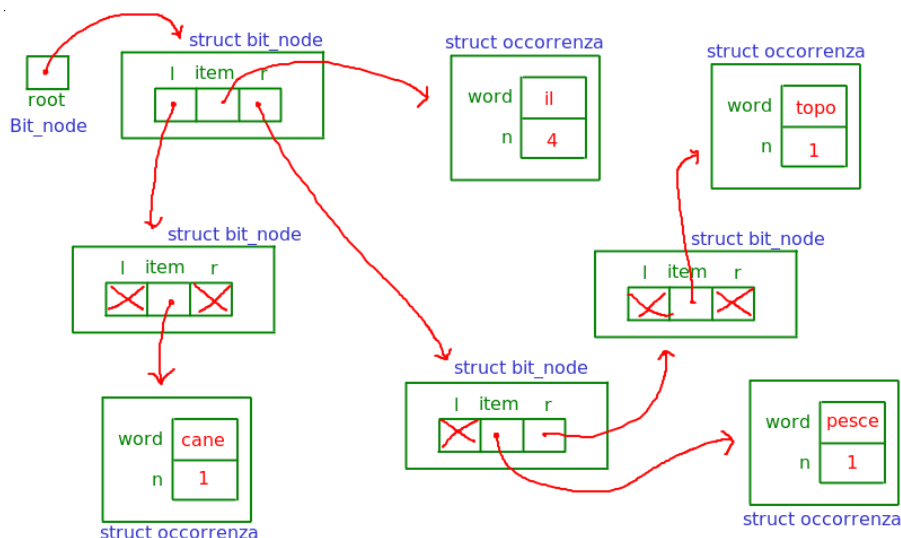
Ad esempio, supponiamo che il testo *t* dato in input sia:

il cane il pesce il topo il STOP

L'albero binario di ricerca ottenuto sarà allora:



Ed ecco la rappresentazione in memoria dell'albero della figura precedente (in verde le allocazioni di memoria, in rosso i valori contenuti, in blu i tipi; il puntatore NULL è indicato con una croce):



**Nota.** Se si cambia l'ordine delle parole nel testo *t*, i nodi potrebbero essere disposti in modo diverso, tuttavia l'insieme rappresentato rimarrebbe invariato.

## 2.2 Struttura del codice

Il programma sarà composto da vari moduli:

**bist:** interfaccia (`bistree.h`) della struttura dati *alberi di ricerca*, in cui il contenuto dei nodi è un tipo astratto `Item`, e relativa implementazione (`bistree.c`).

In particolare, serviranno le due funzioni

---

```
void bist_orderprint( Bit_node p );  
void bist_invorderprint( Bit_node p );
```

---

che stampano il contenuto dell'albero di ricerca in ordine (dal nodo con chiave minima al nodo con chiave massima) e nell'ordine inverso.

Per la stampa in ordine, basta fare una visita inorder (se l'albero non è vuoto, si stampa il sottoalbero sinistro, poi la radice, poi l'albero destro). Per stampare la tabella in ordine alfabetico inverso, occorre visitare i sottoalberi in ordine inverso rispetto al caso precedente.

**itemkey:** interfaccia (`itemkey.h`) per il tipo astratto `Item` e relativa implementazione.

I tipi `Item` e `Key` possono essere definiti modificando direttamente le `typedef` nel file `itemkey.h`, ad esempio nel modo seguente:

---

```
typedef struct occorrenza *Item;  
typedef char *Key;
```

---

L'implementazione delle funzioni di `itemkey.h` può essere inserita direttamente nel file client `occorrenze.c`. Oppure fatta in un apposito file `itemkey_occ.c`.

La funzione `print_item( Item i )` dovrà stampare l'occorrenza puntata da `i` come specificato dal formato di output.

All'interfaccia bisogna inoltre aggiungere una funzione

---

```
Item newItem( Key k );
```

---

che crea e restituisce un nuovo `Item` con chiave `k`.

**occorrenze:** modulo principale del programma (`occorrenze.c`) contenente in particolare il `main`.

Il `main` conterrà innanzitutto la dichiarazione della variabile `Bit_node root` che rappresenta alla radice dell'albero di ricerca.

Il `main` dovrà quindi essere composto da tre parti.

1. Lettura del testo  $t$  e costruzione/aggiornamento dell'albero delle occorrenze; per individuare le parole del testo è opportuno scrivere una funzione che legga la prossima parola, ignorando i caratteri non alfabetici iniziali e interrompendo la lettura al primo carattere non alfabetico (vedi funzione "readline" di una esercitazione precedente).
2. Stampa della tabella delle occorrenze in ordine alfabetico e nell'ordine inverso.
3. Ciclo che, per ogni iterata, legge la prossima parola  $w_i$  da standard input, la cerca nella tabella e stampa il numero delle sue occorrenze nel testo  $t$ . Notate che non è necessario memorizzare tutte le parole  $w_i$ , ma è possibile cercarle nell'albero man mano che vengono lette.