Laboratorio di Algoritmi e Strutture Dati

Docente: Camillo Fiorentini

18 dicembre 2007

Esercizio 1: rappresentazione di una tabella di occorrenze

L'obiettivo è quello di rappresentare in modo efficiente una *tabella di occorrenze*, ossia una tabella che permette di contare le parole che compaiono in un testo.

Formalmente, una *occorrenza* è una coppia $\langle word, num \rangle$, dove *word* è una parola e *num* un intero positivo che rappresenta il numero di occorrenze di *word*.

Operazioni

Si richiede di implementare una struttura dati efficiente che permette di eseguire le seguenti operazioni.

- **addWord(***word***)**

  Se la parola *word* non è nella tabella, aggiunge alla tabella l'occorrenza $\langle word, 1 \rangle$. Altrimenti incrementa di uno il numero di occorrenze di *word*.

- **addWordFromFile(***fileName***)**

  Se non esiste alcun file di nome $fileName$ stampa un messaggio di errore. Altrimenti, per ogni parola *word* nel file $fileName$ compie l'operazione **addWord(***word***)**.

- **delWord(***word***)**

  Se la parola *word* è nella tabella, decrementa di uno il numero di occorrenze di *word*. Se il numero di occorrenze diventa 0, l'occorrenza contenente *word* è tolta dalla tabella (è inutile mantenere nella tabella le coppie $\langle word, 0 \rangle$).

- **printOcc(***word***)**

  Stampa il numero di occorrenze della parola *word*.

- **printTab()**

  Stampa il contenuto della tabella in ordine alfabetico (come nell'esempio più avanti). Se la tabella è vuota, stampa un messaggio opportuno.

- **printTabInv()**

  Stampa il contenuto della tabella in ordine alfabetico inverso (come nell'esempio più avanti). Se la tabella è vuota, stampa un messaggio opportuno.

Formato per l'input e l'output

Il programma deve leggere da standard input una sequenza di linee, ciascuna delle quali corrisponde a una linea della prima colonna della tabella qui sotto, dove *word* è una parola e $fileName$ è il nome di un file. Si assume che tutte le parole abbiano al massimo `WORD` caratteri, dove il valore di `WORD` è definito nel programma.

Quando una linea è letta, viene eseguita l'operazione associata. Le operazioni di stampa sono effettuate sullo standard output e ogni operazione deve iniziare su una nuova linea.

Istruzione in input	**Operazione**
`+` *word*	**addWord**(*word*)
`r` *fileName*	**addWordFromFile**(*fileName*)
`-` *word*	**delWord**(*word*)
`?` *word*	**printOcc**(*word*)
`p`	**printTab**()
`i`	**printTabInv**()
`f`	termina l'esecuzione

Si assume che l'input sia secondo il formato della tabella e che il file *fileName* contenga delle parole aventi al massimo `WORD` caratteri separate da uno o più caratteri di spaziatura (spazi, a capo, ecc.).

Struttura dati

Occorre rappresentare un insieme di occorrenze su cui siano definite le usuali operazioni sugli insiemi dinamici (inserimento, cancellazione, ricerca, ecc.).

Per rappresentare una occorrenza $\langle word, num \rangle$ usare la struttura

```
struct occorrenza{
  char*  word;
  int num;
};

typedef struct occorrenza occorrenza;
```

Poiché 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. Utilizziamo un *albero binario di ricerca* dove:

- L'insieme V dei nodi è l'insieme delle occorrenze $\langle word, num \rangle$.
- La relazione $<$ sull'insieme V è definita come segue

$$\langle word_1, num_1 \rangle \ < \ \langle word_2, num_2 \rangle \qquad \Longleftrightarrow \qquad w_1 <_l w_2$$

  dove $w_1 <_l w_2$ se e solo se w_1 è minore di w_2 rispetto all'ordinamento lessicografico sulle parole. È facile verificare che $<$ è un ordinamento lineare.

Per rappresentare un nodo dell'albero usare la struttura:

```
struct nodo {
    occorrenza* key; //  indirizzo occorrenza rappresentata dal nodo
    struct nodo* up;
    struct nodo* left;
    struct nodo* right;
};

typedef struct nodo nodo;
```

Il campo `key` di un nodo dell'albero di ricerca è l'indirizzo dell'occorrenza contenuta nel nodo; i campi `up`, `left` e `right` hanno il solito significato (indirizzo del nodi padre, figlio sinistro e figlio destro).

Esempio

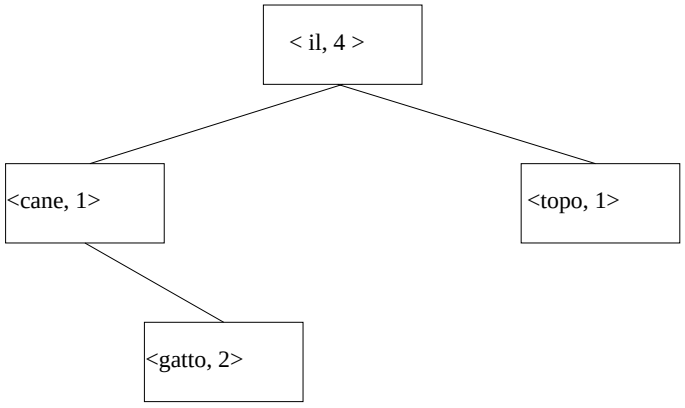
Supponiamo che l'input sia

```
+ il
+ cane
+ il
+ gatto
+ il
+ topo
+ il
+ gatto
```

Dopo queste operazioni, viene costruito l'insieme di occorrenze $\mathscr{I}_{occ}$ così definito:

$$\mathscr{I}_{occ} \;=\; \{\, \langle \text{il}, 4\rangle, \quad \langle \text{cane}, 1\rangle, \quad \langle \text{gatto}, 2\rangle, \quad \langle \text{topo}, 1\rangle \,\}$$

L'albero binario di ricerca ottenuto è:

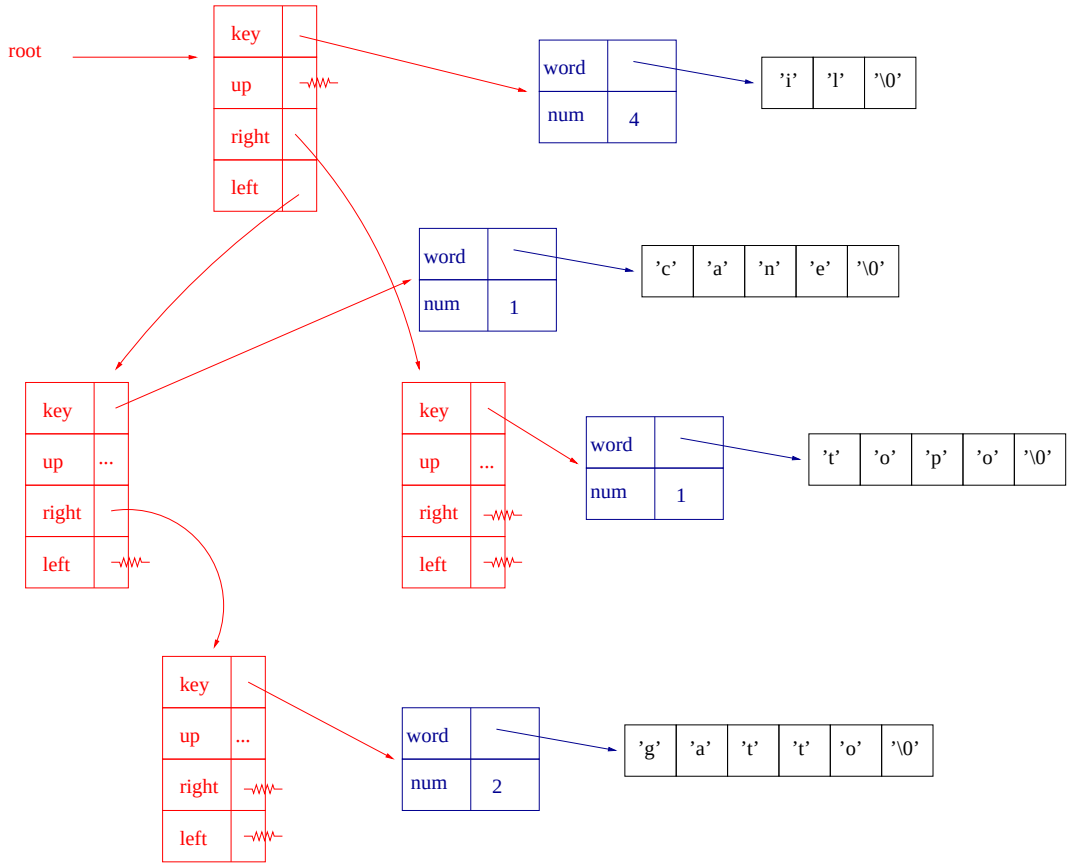


Nota

Se si cambia l'*ordine* delle istruzioni di input, i nodi potrebbero essere disposti in modo diverso, tuttavia l'insieme rappresentato è lo stesso $\mathscr{I}_{occ}$ definito sopra.

Per rappresentare l'albero delle occorrenze, in `main()` va definita una variabile `root` di tipo `nodo*` che rappresenta la *radice* dell'albero. Questo significa che `root` vale `NULL` se l'albero è vuoto, altrimenti `root` è l'indirizzo del nodo radice dell'albero.

La rappresentazione in memoria dell'albero della figura precedente è (per chiarezza le frecce corrispondenti ai campi `up` non sono segnate):



L'output del comando `p` è:

```
OCCORRENZE:
cane 1
gatto 2
il 4
topo 1
```

L'output del comando `i` è:

```
OCCORRENZE:
topo 1
il 4
gatto 2
cane 1
```

Struttura del codice

Occorre completare il programma `occ.c`.

Per gestire le operazioni sull'albero si possono usare le funzioni per gli alberi binari di ricerca nel file `tree.c`. Tuttavia in C, a differenza di quanto avviene in Java (o più in generale nei linguaggi object oriented), per poter riutilizzare del codice occorre spesso intervenire per riscriverne delle parti.

Le funzioni di `tree.c` riguardano alberi binari di ricerca di interi, infatti il campo `next` della struttura `struct node` (definita in `tree.h`) ha tipo `int`. Quindi vanno *riscritte* le parti in cui viene utilizzata l'informazione del campo `key` di un nodo.

Scrivere il codice delle operazioni seguendo l'ordine qui indicato.

addWord(*word*)

Occorre definire le seguenti funzioni.

(1) La funzione

```
occorrenza* newOccorrenza(char *w)
```

crea l'occorrenza $\langle w, 1\rangle$ e restituisce un puntatore alla nuova occorrenza.

L'occorrenza deve contenere una nuova copia di `w` (usare la funzione `newWord()`).

(2) La funzione

```
node* newNode(char* w)
```

crea un nodo contenente l'occorrenza $\langle w, 1\rangle$ e restituisce un puntatore al nuovo nodo.

Copiare la funzione `newNode()` di `tree.c` inizializzando correttamente il campo `key` del nuovo nodo (l'occorrenza va creata con la funzione `newOccorrenza()`).

(3) La funzione

```
node *treeInsert(char *w, node *r)
```

inserisce un nodo contenente l'occorrenza $\langle w, 1\rangle$ nell'albero di radice `r` e restituisce l'indirizzo del nuovo nodo.

Copiare le funzione `treeInsert()` di `tree.c` facendo le dovute modifiche. In particolare, vanno modificate istruzioni

```
n < q->key                n < pq ->key
```

che confrontano la chiave da inserire con la chiave di un nodo dell'albero. Per il confronto, occorre definite una funzione `strMinore()` per il confronto di parole.

(4) La funzione

```
node *findWord(char *w, node *r)
```

cerca un nodo contenente l'occorrenza di `w` nell'albero r (ossia, il campo `key` del nodo è l'indirizzo a una occorrenza contenente la parola `w`). Se il nodo esiste restituisce il suo indirizzo, altrimenti restituisce `NULL`.

Copiare la funzione `treeFind()` di `tree.c` riscrivendo opportunamente il codice per il confronto. In particolare, per controllare l'uguaglianza definire una funzione `strUguale()`.

(5) La funzione

```
void incrementaOcc(node *p)
```

incrementa di 1 il valore di `num` dell'occorrenza in `p`.

Si può ora scrivere il codice della funzione

```
void addWord(char *w, node **proot)
```

che implementa l'operazione **addWord(***word***)**, dove `proot` è un puntatore alla radice dell'albero delle occorrenze (passaggio *per indirizzo*). Questo significa che, all'interno della funzione, per denotare la radice dell'albero si può usare l'espressione `*proot`.
Si noti che per implementare l'operazione **addWord(***word***)** occorre un parametro in più rispetto a quelli nominati nella sua definizione astratta.

printOcc(*word*)

Occorre anzitutto scrivere il codice della funzione

```
void printNode(node  *p)
```

che stampa l'occorrenza nel nodo `p`.
Scrivere quindi il codice della funzione

```
void printOcc(char *word, node* r)
```

che implementa l'operazione **printOcc(***word***)**, dove `r` è la radice dell'albero delle occorrenze.
Nell'esempio precedente, le istruzioni

```
? il
? bue
```

devono stampare

```
il 4
bue 0
```

printTab()

Per stampare la tabella in ordine alfabetico, occorre stampare l'albero delle occorrenza facendo una visita *inorder* (se l'albero non è vuoto, si stampa il sottoalbero sinistro, poi la radice, poi l'albero destro).
Scrivere il codice della funzione

```
void printInorder(node *r)
```

che stampa l'albero delle occorrenze `r` nel modo descritto.
Scrivere quindi il codice della funzione

```
void printTab(node *r)
```

che implementa l'operaziome **printTab()**, dove `r` è la radice dell'albero delle occorrenze.

printTabInv()

Per stampare la tabella in ordine alfabetico inverso, occorre visitare i sottoalberi in ordine inverso rispetto al caso precedente.
Procedere come nel punto precedente scrivendo il codice delle funzioni

```
void printInorderInv(node *r)
```

e

```
void printTabInv(node *r)
```

addFromFile($fileName$)

Scrivere il codice della funzione

```
void addFromFile(char* fileName, node **proot)
```

che implementa l'operazione **addFromFile(**$fileName$**)**, dove `proot` è un puntatore alla radice dell'albero delle occorrenze (passaggio *per indirizzo*).
Occorre:

- Aprire in lettura il file di nome `fileName` con l'istruzione

  ```
  fp = fopen(fileName,"r");
  ```

  dove la variabile `fp` deve avere tipo `FILE*` (*file pointer*). Se il file non esiste, `fopen()` restituisce `NULL`.

- Leggere una parola per volta dal file `fp` chiamando

  ```
  fscanf(fp, "%s", word)
  ```

  dove `word` è un array di `char` (definire la dimensione in modo corretto!). La funzione `fscanf()` si comporta come `scanf()` (il primo parametro deve essere il file pointer associato al file da cui leggere), quindi salta automaticamente gli spazi fra le parole e resituisce 1 se è stata letta una stringa, `EOF` se il file è terminato.

  Per aggiungere la parola letta nell'albero delle occorrenze usare `addWord()`.

- Al termine della lettura, il file `fp` va chiuso:

  ```
  fclose(fp);
  ```

delWord(*word*)

La funzione `treeDelete()` di `tree.c` non richiede alcuna manipolazione.
Occorre definire le seguenti funzioni.

- La funzione

  ```
  void freeNode(node *p)
  ```

  cancella la memoria occupata dal nodo `p` (va cancellato *tutto* lo spazio di memoria utilizzato per rappresentare il nodo).

- La funzione

  ```
  void decrementaOcc(node *p)
  ```

  decrementa di 1 il valore di `num` dell'occorrenza in `p`.

Si può ora scrivere il codice di

```
void delWord(char *word,  node **proot)
```

che implementa l'operazione **delWord(*word*)**, dove `proot` è un puntatore alla radice dell'albero delle occorrenze (passaggio *per indirizzo*).

Ulteriori operazioni

1. Aggiungere una operazione per cancellare tutta la tabella.

2. Aggiungere una operazione per stampare la tabella in ordine di numero di occorrenze. Nell'esempio precedente deve essere stampato:

```
OCCORRENZE:
il 4
gatto 2
cane 1
topo 1
```

In questo caso la struttura dati non permette di usare una strategia efficiente (non sempre esiste una struttura dati che risulti efficiente per tutte le operazioni richieste!). Occorre copiare le occorrenze in un array e ordinarlo.

3. Assumere che le parole abbiano lunghezza arbitraria (occorre scrivere una opportuna funzione `strRead()` che legge una parola la cui lunghezza non è nota a priori e restituisce l'array contenente la parola letta).

1 Esercizio 2: alberi generati casualmente

Lo scopo è quello di scrivere un programma che genera casualmente un albero binario di ricerca contenente 2^k nodi, dove k è un valore inserito dall'utente, in modo da valutare sperimentalmente quanto l'altezza dell'albero ottenuto si discosti dal valore minimo k e dal valore massimo $2^k - 1$. Per semplicità, consideriamo alberi di interi.

1. Scrivere in un file `tRand.c` il codice della funzione

```
node *treeRand(int k)
```

che genera un albero binario di ricerca contenente k nodi distinti generati casualmente e restituisce la radice dell'albero ottenuto. Usare le funzioni in `tree.c` (in `tRand.c` occorre includere `tree.h`).

Suggerimento. Anzitutto occorre creare l'albero vuoto. Quindi, si genera con `rand()` un intero casuale e, se non è già contenuto nell'albero (verificarlo con `treeFind()`), lo si inserisce (con `treeInsert()`). L'operazione va ripetuta fino a quando sono stati inseriti k interi, quindi si restituisce la radice dell'albero.

In realtà la funzione non è molto efficiente nel caso k assuma valori molto alti in quanto il generatore potrebbe aver difficoltà a generare numeri "nuovi" (non ancora inseriti nell'albero).

2. Per verificare il comportamento di `treeRand()`, scrivere in `tRand.c` una funzione `main()` che legge un intero k da standard input, genera un albero casuale di k elementi e lo stampa. Il programma termina quando viene inserito 0.

Per compilarlo:

```
gcc   tRand.c  tree.c
```

Ricordarsi che il generatore di numeri casuali va inizializzato ponendo all'inizio di `main()` l'istruzione

```
srand(time(NULL)); // occorre includere   <time.h>
```

3. Scrivere un programma che legge da standard input un intero k, genera un albero casuale di 2^k nodi e stampa il valore dell'altezza dell'albero ottenuto. Il programma termina quando viene inserito 0.

Risultati ottenuti con $k = 10$ (1024 nodi):

```
Altezza: 18
Altezza: 22
Altezza: 20
```

L'altezza è vicina al valore minimo 10 (come ci si aspetta dal teorema visto a lezione).

4. Scrivere un programma che legge da standard input due interi k e h, quindi genera degli alberi casuali di 2^k nodi terminando quando viene generato un albero di altezza maggiore o uguale a h (si assume $k \leq h \leq 2^k - 1$). Al termine dell'esecuzione, il programma stampa il numero di alberi generati.

Esercizio 3: costruzione di un albero binario

Lo scopo dell'esercizio è quello di scrivere delle funzioni per costruire un alberi binario di interi a partire dalla sua definizione.

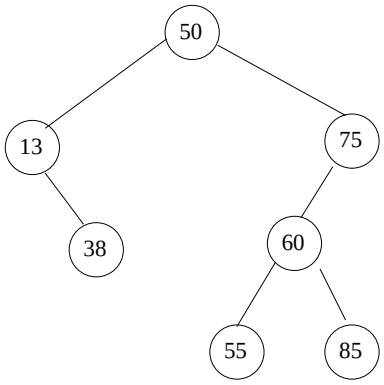
Per l'input usiamo la seguente convenzione.

- L'albero vuoto è denotato dal carattere `b`.
- L'albero $\langle n, T_1, T_2 \rangle$ (dove n è un intero positivo) è denotato da

  ```
  <  n  L  R >
  ```

  dove `L` e `R` denotano rispettivamente T_1 e T_2 .

La descrizione di un albero è letta da standard input e deve terminare con "end-of file". I vari simboli (token) della descrizione *possono* essere separati da uno o più caratteri di spaziatura (spazio, a capo, tabulazione, ecc.). L'albero



può essere descritto da

```
<  50
 < 13 b  <38 b b>  >
 < 75
   < 60   < 55 b b>  <85 b b> >
   b
 >
>
```

oppure

```
<50   <13 b <38bb>>   < 75 <60 <55bb><85bb>> b>   >
```

La descrizione di un albero viene codificata nel programma come una successione di interi (che sarà memorizzata in un array). Per codificare i caratteri '`<`', '`>`' e '`b`' in modo che non vengano confusi con i valori di un nodo usare le seguenti definizioni:

```
#define BEGIN_TREE  -1  // inizio albero
#define END_TREE  -2   // fine albero
#define EMPTY_TREE  -3  // albero vuoto
```

Ad esempio, `< 10 b <20bb> >` è memorizzato come:

-1	10	-3	-1	20	-3	-3	-2	-2

a. Scrivere il codice della funzione

```
int nextSymbol()
```

che legge il prossimo simbolo da standard input e restituisce il codice corrispondente al simbolo letto. Se viene letto "end-of-file", restituisce `STOP` definito come

```
#define STOP  -4
```

Si noti che in questo caso è pericoloso restituire `EOF` perché tale valore potrebbe coincidere con uno dei valori usati nelle definizioni precedenti.

Suggerimento. Conviene anzitutto cercare di leggere da standard input un intero con la chiamata

```
scanf("%d", &n)
```

Se questa ha successo si restituisce l'intero letto. In caso contrario, occorre leggere il prossimo carattere da standard input e restituire il valore opportuno (se il carattere letto è `b` va restituito `EMPTY_TREE`, se è `<` va restituito `BEGIN_TREE`, ecc.).

b. Scrivere il codice della funzione

```
int endTree(int tree[], int begin)
```

che, dato un array `tree[]` di interi contenente la codifica di un albero T e un indice `begin` di `tree[]` tale che in `tree[begin]` inizia la codifica un sottoalbero T' di T, restituisce l'indice dell'array `tree[]` in cui termina la codifica di T'.

Suggerimento. Per ipotesi, in `tree[begin]` si trova il valore `EMPTY_TREE` oppure `BEGIN_TREE`. Nel primo caso occorre restituire `begin`. Nel secondo caso bisogna cercare il simbolo `END_TREE` che chiude il `BEGIN_TREE` in `tree[begin]`. È sufficiente scorrere l'array fino a quando il numero di `END_TREE` è uguale a quello di `BEGIN_TREE`.

c. Scrivere il codice della funzione

```
node *buildTree(int tree[], int begin)
```

che, dato un array `tree[]` e un indice `begin` di `tree[]` tale che in `tree[begin]` inizia la codifica di un albero T, costruisce T e restituisce l'indirizzo della radice di T.

Suggerimento. Se l'albero non è vuoto, fare due chiamate ricorsive per costruire i sottoalberi (per sapere dove inizia il codice del sottoalbero destro usare la funzione `endTree()` del punto precedente).

Se i punti precedenti sono stati svolti correttamente, il codice seguente legge da standard input la descrizione di un albero, costruisce l'albero e stampa i nodo in preorder (prima stampa la radice, poi ricorsivamente il sottoalbero sinistro e il sottoalbero destro).

Si assume che la codifica dell'albero contenga al massimo `MAX` simboli.

```
# define MAX 10000 // massimo numero simboli di una codifica
.....

/* Stampa in preorder l'albero di radice r */
void print(node* r){
  if(r! = NULL){
    printf("%d\n", r->key); // stampa radice
    print(r->left);   // stampa sottoalbero sinistro
    print(r->right);  // stampa sottoalbero destro
  }
}

int main(){
  int k, s;
  node *root; // radice dell'albero
  int treeDescriptor[MAX];  // memorizza  codifica dell'albero
  for(k=0 ; (s=nextSymbol())!= STOP ; k++) // lettura  input
    treeDescriptor[k] = s;
  root = buildTree(treeDescriptor,0); // costruisce albero
  print(root); // stampa albero
  return 0;
}
```

Supponiamo ora di voler costruire alberi di interi in cui i nodi possono assumere un qualunque valore intero. Il programma dell'esercizio precedente non funziona in quanto, ad esempio, un nodo di valore -1 verrebbe confuso con `BEGIN_TREE`. Per risolvere il problema senza dover riscrivere tutto il codice, per rappresentare un nodo di valore n usiamo la seguente codifica

$$Code(n) = \begin{cases} n & \text{se } n \geq 0 \\ n-4 & \text{se } n < 0 \end{cases}$$

In questo modo, l'albero

```
< -1 b  <-2 bb> >
```

è codificato come

-1	-5	-3	-1	-6	-3	-3	-2	-2

e ogni intero della codifica è univocamente interpretabile.

c. Scrivere il codice della funzione

```
int code (int n)
```

che, dato un intero n, restituisce la sua codifica $Code(n)$. Scrivere anche la funzione inversa `decode()` che, dato un valore k tale che $k \geq 0$ oppure $k \leq -5$, restituisce l'intero n codificato da k (ossia, tale che $Code(n) = k$).

d. Modificare la funzione `nextSymbol()` dell'esercizio precedente facendo in modo che, nel caso venga letto un intero n, restituisca $Code(n)$.

Per poter utilizzare il programma di prima, occorre ancora modificare la funzione per la stampa, in quanto il valore di un nodo, prima di essere stampato, va decodificato.

```
void print(node * r){
  if(r != NULL){
    printf("%d\n", decode(r->key)); // stampa nodo r
    print(r->left);
    print(r->right);
  }
}
```