Laboratorio di algoritmi e strutture dati

Docente: Violetta Lonati

1 Esercizi su array frastagliati

1.1 Indice della parola più piccola

Scrivete una funzione con prototipo `int smallest_word_index( char *s[], int n )` che, dato un array `s` lungo `n` di stringhe, restituisca l'indice della parola più piccola (secondo l'ordine alfabetico) contenuta nell'array. Per effettuare confronti tra stringhe, potete usare la funzione `strcmp` dal file di intestazione `string.h`.

Inizializzare un array frastagliato da standard input può essere *doloroso*; consiglio quindi di testare la vostra funzione `smallest_word_index` usando un `main` così strutturato:

```
int main( void ) {
  char *dict[] = { "ciao", "mondo", "come", "funziona", "bene", "questo", "programma" };
  int lun = 7, pos;
  ... min;
  ... max;

  ...

  pos = smallest_word_index( dict, lun );
  printf( "La parola minima si trova in posizione %d.\n", pos );
  return 0;
}
```

Modificate le inizializzazioni di `dict` e `lun` in modo da testare la funzione con altri argomenti.

1.2 La parola minima e la parola massima

Scrivete una funzione con prototipo

```
void smallest_largest( char *s[], int n, char **smallest, char **largest )
```

che, dato un array `s` lungo `n` di stringhe, trovi gli elementi minimo e massimo nell'array (secondo l'ordine alfabetico). Per effettuare confronti tra stringhe, potete usare la funzione `strcmp` dal file di intestazione `string.h`.

2 Esercizi su argomenti da linea di comando

2.1 Alfabeto farfallino

Quando la vostra docente di laboratorio di algoritmi era bambina, usava a volte, per comunicare con le sue amiche, uno speciale alfabeto, detto *alfabeto farfallino*. L'alfabeto farfallino consiste nel sostituire, a ciascuna vocale, una sequenza di tre lettere della forma vocale-f-vocale. Per esempio, alla lettera *a* viene sostituita la sequenza *afa*, alla lettera *e* la sequenza *efe* e così via.

Dovete scrivere un programma, di nome `farf` che, ricevendo come argomento (sulla riga di comando) una parola, ne stampi la traduzione in alfabeto farfallino. Potete assumere che la stringa in input non contenga lettere maiuscole.

Provate a modificare il programma in modo che accetti più parole sulla riga di comando.

Esempio di funzionamento

```
$./farf mamma
mafammafa
$./farf aiuola
afaifiufuofolafa
$./farf farfalla
fafarfafallafa
```

2.2 La parola minima e la parola massima – rivisitato

Provate a testare la funzione scritta per l'esercizio 1.2 scrivendo un programma che legga da linea di comando una serie di parole e poi invochi la funzione `smallest_largest` passando come argomenti `argv` e `argc` (o meglio, delle espressioni che coinvolgono `argv` e `argc`: ricordate che `argv[0]` contiene il nome del programma, che non va passato alla funzione `smallest_largest`!).

2.3 La strana sillabazione

Il professor Precisini, sostenendo che le regole di sillabazione della lingua italiana sono troppo complesse e piene di eccezioni, propone un nuovo e originale metodo di sillabazione. Il metodo consiste in questo: una sillaba è una sequenza massimale di caratteri consecutivi che rispettano l'ordine alfabetico. Per esempio, la parola *ambire* viene sillabata come *am-bir-e*: infatti la lettera *a* precede la lettera *m*, e le lettere *b*, *i* e *r* rispettano anch'esse l'ordine. Analogamente, la parola *sotterfugio* viene sillabata come *s-ott-er-fu-gio*.

Dovete scrivere un programma, di nome `sillaba` che, ricevendo come argomento (sulla riga di comando) una parola, la sillabi. Potete assumere che la stringa in input sia costituita solo da lettere minuscole.

Esempio di funzionamento

```
$./sillaba amore
amor-e
$./sillaba scafroglia
s-c-afr-o-gl-i-a
```

2.4 Palindrome (con argomenti da linea di comando)

Scrivete una funzione che stabilisca se il suo argomento è una parola palindroma oppure no, usando due puntatori per scorrere la parola partendo dall'inizio e dalla fine. Quindi scrivete un programma che stabilisca, per ciascun argomento fornito da linea di comando, se si tratta di una parola palindroma oppure no.

3 Esercizi su allocazione dinamica della memoria

3.1 La vostra malloc

- Scrivete una funzione `my_malloc` che allochi memoria usando la funzione `malloc` della libreria standard (file di intestazione `stdlib.h`) e verifichi il buon esito dell'allocazione. In caso di esito positivo, la funzione deve

restituire l'indirizzo dello spazio allocato; in caso contrario la funzione deve stampare un messaggio di errore e provocare la terminazione del programma attraverso una chiamata della funzione `exit`).

- Scrivete un'analoga funzione `my_realloc`.

3.2 Rovescia

Scrivete tre programmi che leggano una sequenza di interi e la stampino al contrario, allocando la memoria necessaria in modo dinamico attraverso l'uso della funzione `malloc`.

1. L'input è dato da un intero n e da una sequenza di n numeri; basta una sola chiamata di `malloc` per allocare un vettore di dimensione n.
2. L'input è dato da una sequenza di numeri terminata da 0; non potendo prevedere quanti numeri verranno inseriti, il vettore andrà ridimensionato man mano: partite da una dimensione fissata piccola (es: 2) e raddoppiate la lunghezza del vettore ogni volta che questo si riempie.
3. L'input è dato da una sequenza di numeri terminata da 0, come nel caso precedente; di nuovo, il vettore andrà ridimensionato man mano: partite da una dimensione fissata (es: 15) e allungate il vettore di una lunghezza fissa (es: 10) ogni volta che questo si riempie.

3.3 Lettura di stringhe con allocazione di memoria

Scrivete due funzioni che leggano da standard input una sequenza di caratteri e la memorizzino in una stringa di dimensione opportuna allocata dinamicamente (scegliete la strategia che preferite, ad esempio una di quelle proposte nell'esercizio precedente):

1. `char *read_line( void )` deve leggere una riga terminata da `\n`;
2. `char *read_word( void )` deve leggere una parola di caratteri alfanumerici (nota: se il primo carattere letto non è alfanumerico, la stringa restituita sarà la stringa vuota).

Entrambe le funzioni devono restituire l'indirizzo del primo carattere della stringa memorizzata o il puntatore `NULL` in caso di errore.

3.4 Rettangoli

Aggiungete al programma dell'esercizio 1.6 del 9 novembre una funzione che crei un nuovo rettangolo e lo inizializzi con dati inseriti dall'utente, allocando la memoria necessaria attrverso l'uso della funzione `malloc`.

4 Un esercizio un po' più elaborato: Registro di prenotazione

L'obiettivo dell'esercizio è scrivere un programma per gestire un registro di prenotazione di posti numerati da 0 a $n-1$. Il valore di n (numero dei posti prenotabili) è inserito dall'utente all'atto della creazione del registro. Un cliente è identificato da una stringa.

Funzionalità da implementare

Il programma deve implementare varie funzionalità. E' opportuno strutturare il programma in funzioni e commentare ciascuna funzione indicando chiaramente cosa fa e quali parametri usa.

- **newBook** (n)

  Crea un nuovo registro che permetta la prenotazione di n posti, da 0 a $n-1$. Se esiste già un registro di prenotazione, quest'ultimo deve essere cancellato.

- **book**$(k, name)$

  Se il posto k è libero, prenota il posto k per il cliente identificato da *name*. Altrimenti, stampa un messaggio di errore.

- **cancel**(k)

  Se il posto k è occupato, cancella la prenotazione di k. Altrimenti, stampa un messaggio di errore.

- **move**$(from, to)$

  Sposta il cliente dal posto $from$ al posto to se ciò è possibile (ossia, $from$ è occupato e to libero). Altrimenti, stampa un messaggio di errore.

- **printBook**$()$

  Stampa il contenuto del registro (posti prenotati).

Notate che l'implementazione in C delle precedenti operazioni di alto livello può richiedere l'uso di parametri in più rispetto a quelli indicati. Riflettete quindi su quali siano i prototipi più opportuni da implementare!

Struttura dati

Per rappresentare il registro occorre usare un array `book` allocato dinamicamente in quanto la dimensione è stabilita durante l'esecuzione del programma.

Sia n la dimensione di `book`. Allora, in ogni istante del programma per ogni $0 \leq k < n$ deve valere la seguente proprietà:

- Se il posto k è prenotato da w, allora `book[k]` è l'indirizzo a un vettore contenente w.
- Altrimenti, `book[k]` vale `NULL` (indirizzo 0).

Anche se avete la tentazione di definire `book` come una variabile globale, provate a definirla nel main e a passarla come argomento alle varie funzioni.

Formato di input e output

Il programma deve leggere da standard input una sequenza di istruzioni secondo il formato nella tabella, dove k, n, $from$ e to sono interi e *name* una parola.

Riga di input	**Operazione**
`b` n	**newBook** (n)
`+` k *name*	**book**$(k, name)$
`-` k	**cancel**(k)
`m` $from$ to	**move**$(from, to)$
`p`	**printBook**$()$
`f`	Termina l'esecuzione

I vari elementi sulla riga sono separati da uno o più spazi. Quando una riga è letta, viene eseguita l'operazione associata; le operazioni di stampa sono effettuate sullo standard output, e ogni operazione deve iniziare su una nuova riga.

Si assume che l'input sia inserito correttamente. Conviene scrivere le istruzioni di input in un file `in.txt` ed eseguire il programma redirigendo lo standard input.

La lettura e l'interpretazione dei comandi può essere gestita con un ciclo contenente uno `switch`:

```
while( ( c = getchar() ) != 'f' ){
  switch(c){
    case 'b':  //  b n  -->  newBook(n)
      // ...
      break;
    case '+':  //  + k name  --> book(k, name)
      //...
      break;
    case '-':  // - k --> cancel(k)
      // ...
      break;
    case 'm':  // m from to ---> move from to
      // ..
      break;
    case 'p':  // p --->  printBook()
      // ...
      break;
    }
}
```

Esempio di funzionamento

INPUT

```
b 10
+ 1 Rossi
+ 3 Bianchi
p
m 1 5
p
+ 9 Verdi
p
- 3
p
b 20
+ 10 Mario
p
m 1 10
m 10 11
p
f
```

OUTPUT

```
REGISTER[0..9]:
1 --> Rossi
3 --> Bianchi

REGISTER[0..9]:
3 --> Bianchi
5 --> Rossi

REGISTER[0..9]:
3 --> Bianchi
5 --> Rossi
9 --> Verdi

REGISTER[0..9]:
5 --> Rossi
9 --> Verdi

REGISTER[0..19]:
10 --> Mario
move(1,10): errore

REGISTER[0..19]:
11 --> Mario
```