Lezione 13

La lezione è centrata su un esercizio per impratichirsi sui seguenti argomenti:

- realizzazione di un ambiente interattivo testuale
- alberi binari di ricerca rappresentati con puntatori

Chi notasse eventuali incoerenze o errori, oppure avesse dubbi sul contenuto di queste pagine e dei codici, è pregato di segnalarmeli per contribuire a migliorare la qualità dei materiali del corso.

Problema

Si vuole scrivere un programma `ricerca.c` che carica da un file di testo un insieme di numeri interi tutti diversi tra loro e permette all'utente di eseguire una serie di operazioni sull'insieme stesso:

- `member` i: chiede al programma di indicare se il numero intero i fa parte dell'insieme oppure no;
- `insert` i: chiede al programma di aggiungere il numero intero i all'insieme;
- `delete` i: chiede al programma di cancellare il numero intero i dall'insieme;
- `print`: chiede al programma di stampare l'insieme per valori crescenti;
- `empty`: chiede al programma di indicare se l'insieme è vuoto oppure no;
- `min`: chiede al programma di indicare l'elemento minimo dell'insieme;
- `max`: chiede al programma di indicare l'elemento massimo dell'insieme;
- `exit`: chiede al programma di terminare.

Formati di ingresso e di uscita Il file di ingresso che contiene l'insieme iniziale dei numeri interi li riporta semplicemente uno dopo l'altro separati da spazi o a capi. Per esempio:

```
95 77 748 40 242 303 793 573 247 727 87 491 832 594 302 229 967
```

All'istruzione `member` i il programma risponde stampando a video i `in T` oppure i `non in T`, a seconda del risultato (i è il numero indicato dall'utente).

All'istruzione `print` il programma risponde stampando a video i numeri in ordine crescente, su una sola riga, separati da spazi bianchi.

All'istruzione `empty` il programma risponde stampando a video `T e' vuoto` o `T non e' vuoto` secondo il risultato.

Alle istruzioni `min` e `max` il programma risponde stampando `min(T) =` i oppure `max(T) =` i, dove i è l'elemento minimo o massimo richiesto.

Traccia della risoluzione

Prima di eseguire l'esercizio, si possono scorrere interamente i lucidi per acquisire il concetto di dizionario, di albero binario di ricerca e le basi della implementazione degli alberi binari di ricerca con puntatori.

L'esercizio parte dal file `ricerca0.c`, che contiene solo la consueta funzione `InterpretaLineaComando` per recuperare il nome del file dalla linea di comando e copiarlo nella corrispondente variabile.

Prima fase (`ricerca.c`) Avviamo la realizzazione *top-down* dell'algoritmo scomponendo il problema in due fasi, risolte da corrispondenti funzioni chiamate nel `main`, dichiarate fra i prototipi prima del `main` e definite dopo il `main` (all'inizio vuote, poi da riempire):

1. `CaricaABR`: carica i numeri interi dal file indicato nella linea di comando in un albero binario di ricerca;
2. `EsegueComandi`: riconosce ed esegue i comandi forniti dall'utente da tastiera.

La prima funzione si limita ad

- aprire il file;
- creare un albero vuoto;
- leggere i numeri dal file;
- inserirli via via nell'albero;
- chiudere il file ;
- restituire l'albero.

L'implementazione è estremamente semplice, ipotizzando di aver disponibili le funzioni per creare un albero vuoto e per inserirvi numeri interi. Queste funzioni (e le altre introdotte nei lucidi) sono dichiarate nel file `abr.h`, mentre il file `abr0.c` ne contiene le definizioni vuote. La dichiarazione della struttura dati è praticamente identica a quella dell'albero binario discussa nella lezione precedente. La differenza principale è che il dato contenuto in ogni nodo è un semplice numero intero `val`, anziché un operando e/o un operatore, ma questo dipende banalmente dallo specifico problema che stiamo affrontando. Rispetto alla definizione astratta, si è aggiunta una funzione di stampa ordinata.

La seconda funzione è abbastanza interessante, perché realizza (diversamente da tutti gli esercizi svolti sinora) un programma che interagisce con l'utente per un tempo indefinito, finché questi non indica di terminare. Il programma consiste in un semplice ciclo che:

- legge da tastiera l'operazione indicata dell'utente;
- riconosce l'operazione e il suo eventuale operando;
- esegue l'operazione.

La lettura dell'operazione può avvenire con la funzione `gets`, ma è più sicuro usare la funzione `fgets(Comando,ROW_LENGTH,stdin);`, che nasce per la lettura da file di testo, ma si può adattare alla lettura da tastiera specificando `stdin` come il file da leggere, e ha il vantaggio di specificare il numero massimo di caratteri letti, in modo da evitare che si superi la dimensione della stringa di lettura `Comando`. Per riconoscere l'istruzione, osserviamo che vi sono due categorie di operazioni: quelle con un operando intero (da `member` a `delete`) e quelle senza operandi. La funzione `sscanf`, applicata sulla stringa `Comando` con la stringa di formato `"%s %d"` distingue i due casi in base al fatto che essa restituisce 2 nel primo caso e 1 nel secondo. Definita la classe cui appartiene l'operazione, si può identificarla esattamente confrontando l'operando con quelli sopra elencati. Un modo alternativo di procedere potrebbe essere di indicare esplicitamente gli operandi nella stringa di formato (per esempio, `sscanf(Comando," member %d",&val);`)[1] e usare una

[1] Lo spazio iniziale nella stringa di formato serve per riconoscere il comando anche nel caso in cui l'utente digiti qualche spazio prima della parola chiave `member`.

cascata di costrutti di selezione (`if ... else if ...`). Una volta identificata l'operazione, basta chiamare la corrispondente funzione di gestione dell'albero e stampare il risultato a video nel formato richiesto.

Seconda fase (`abr1.c`) La fase successiva dell'esercizio richiede di cominciare a realizzare la libreria per la gestione dell'albero binario di ricerca. Cominciamo con le funzioni più semplici:

- `creaABR`: crea un albero vuoto;
- `distruggeABR`: distrugge un albero binario di ricerca;
- `ABRvuoto`: restituisce `TRUE` se l'albero è vuoto, `FALSE` altrimenti;
- `insertABR`: restituisce l'albero ottenuto aggiungendo un elemento all'albero dato.

Le prime tre funzioni non riguardano il contenuto informativo, ma solo la topologia dell'albero, che è identica a quella dell'albero binario. Non stupisce quindi che siano identiche alle corrispondenti funzioni dell'albero binario. Anche la loro complessità sarà la stessa: costante per la creazione e il test di vuotezza, lineare per la distruzione.

La funzione `insertABR` e quelle che vedremo nelle fasi successive dell'esercizio, invece, tengono conto del contenuto informativo, e quindi sono abbastanza diverse da quelle dell'albero binario. In particolare, in un albero binario di ricerca la posizione degli elementi è assegnata automaticamente, anziché essere indicata in modo esplicito, come negli alberi binari. L'implementazione qui descritta tende ad avvicinarsi a quella delle dispense del corso, ma vi sono le seguenti differenze:

1. le dispense fanno in genere l'ipotesi semplificativa che l'albero su cui si opera non sia vuoto, mentre qui consideriamo il caso generale;
2. le dispense assumono che i valori da inserire nell'albero non ne facciano già parte e quelli da cancellare ne facciano parte, mentre qui consideriamo anche i casi rimanenti;
3. le dispense assumono che le funzioni passino i parametri per indirizzo, mentre in C i parametri vengono passati per valore;
4. per semplificare alcuni passaggi faremo uso della funzione `costruisceABR`, che coincide con la funzione `costruiscealbero` descritta nella lezione precedente; nelle dispense vengono invece usate le funzioni `CREA_NODO_SIN` e `CREA_NODO_DES`.

La funzione ausiliaria `costruisceABR` viene usata solo all'interno della libreria, ma non è accessibile all'utente esterno, per mantenere la libreria vicina alla struttura astratta descritta nei lucidi. Affinché possa essere chiamata dalla funzione `insertABR`, deve essere definita prima di quella, in modo che il compilatore l'abbia già letta quando arriva a occuparsi della funzione chiamante.

Procediamo con la funzione `insertABR`, al solito con un'implementazione ricorsiva:

- primo *caso base*: se l'albero è vuoto, la funzione `costruisceABR` restituisce la soluzione, cioè un nuovo albero con due sottoalberi vuoti e il nuovo dato come informazione associata alla radice;
- secondo *caso base*: se il nuovo dato coincide con la radice dell'albero corrente, si esce senza far nulla, restituendo l'albero corrente;

- *caso ricorsivo*: a seconda che il nuovo dato preceda o segua quello associato alla radice, si considera il sottoalbero sinistro o destro e si procede ricorsivamente nel sottoalbero individuato; al termine, si restituisce l'albero corrente (questo è importante, perché il nuovo dato inserito modifica la radice di uno degli alberi incontrati durante l'esecuzione e quindi bisogna accertarsi che tale radice sia modificata).

La complessità di un inserimento è proporzionale al numero di chiamate ricorsive, dato che ad ogni chiamata si esegue un numero costante di operazioni. Quindi, è proporzionale alla profondità h dell'ABR: $\Theta(h)$. L'intera costruzione di un ABR di n elementi ha quindi complessità in $\Theta(hn)$. La profondità h è compresa fra $\log_2 n$ e n, secondo che l'albero sia più o meno bilanciato. Vale il caso pessimo quando gli elementi vengono inseriti in ordine crescente o decrescente, e quindi scendono via via sempre nel sottoalbero destro o sinistro. Vale il caso ottimo quando gli elementi vengono inseriti in modo del tutto casuale, e quindi i vari sottoalberi tendono ad avere tutti la stessa profondità.

Terza fase (`abr2.c`) Nella fase seguente, realizziamo le funzioni di consultazione dell'albero, cioè:

- `stampaABR`: stampa a video gli elementi dell'albero in ordine crescente;
- `memberABR`: determina se un dato elemento appartiene o no all'albero;
- `minABR`: restituisce l'elemento minimo dell'albero;
- `maxABR`: restituisce l'elemento massimo dell'albero.

Con la funzione `stampaABR` ci mettiamo in grado di vedere l'albero caricato da file, verificando che il codice sia corretto semanticamente, e non solo sintatticamente. La funzione si riduce a una semplice visita in in-ordine, dato che, per definizione, il valore conservato nella radice di un albero è segue quelli contenuti nel sottoalbero sinistro e precede quelli contenuti nel sottoalbero destro. Ci si può chiedere se stampare qualcosa nel caso di albero vuoto: conviene non farlo perché durante la visita ogni foglia dell'albero ha due alberi vuoti, e quindi otterremmo molte stampe non desiderate. La funzione è ovviamente lineare nel numero degli elementi ($\Theta(n)$), come nel caso degli alberi binari.

Procediamo con le funzioni di interrogazione dell'albero, partendo da `memberABR`. L'implementazione è ricorsiva:

- primo *caso base*: se l'albero è vuoto, l'elemento cercato non sta nell'albero;
- secondo *caso base*: se l'elemento cercato coincide con quello conservato in radice, ovviamente sta nell'albero;
- *caso ricorsivo*: a seconda che il nuovo dato preceda o segua quello associato alla radice, si riapplica la funzione al sottoalbero sinistro o a quello destro.

Qui la complessità è proporzionale al numero di chiamate ricorsive, cioè alla profondità h dell'albero: $\Theta(h)$. Siccome la chiamata ricorsiva è terminale, è possibile ricavare facilmente e meccanicamente una semplice implementazione iterativa. Le varie chiamate ricorsive, in effetti, consistono semplicemente nel salvare sulla pila di sistema i vari sottoalberi visitati (sinistri o destri secondo l'esito del confronto fra il valore di i e quello conservato nel nodo corrente). Giunti al caso base:

- se si arriva a un sottoalbero vuoto, si restituisce `falso`;

- se la radice coincide con i, si restituisce `vero`.

La ricerca dell'elemento minimo con `minABR` ammette anch'essa un'implementazione ricorsiva di complessità $\Theta(h)$:

1. primo *caso base*: se l'albero è vuoto, restituisce un valore convenzionale corrispondente a $+\infty$;
2. secondo *caso base*: se la radice dell'albero non ha un sottoalbero sinistro, il suo valore associato è l'elemento minimo, e viene direttamente restituito;
3. *caso ricorsivo*: se l'albero ha un sottoalbero sinistro, si riapplica la funzione a tale sottoalbero.

Ancora una volta, la chiamata ricorsiva è terminale, ed è possibile ricavare un'implementazione iterativa in modo meccanico. Anziché salvare sulla pila di sistema i vari sottoalberi visitati (sempre sottoalberi sinistri). Al termine della ricorsione, si arriva al caso base, in cui la foglia non ha un sottoalbero nella direzione richiesta, e l'algoritmo restituisce il valore associato alla foglia individuata. Di conseguenza, l'implementazione iterativa di `minABR` consiste nel:

- restituire $+\infty$ se l'albero è vuoto;
- scendere lungo il sottoalbero sinistro finché si arriva a un nodo che non ha più tale sottoalbero, e restituire il valore associato al nodo.

Per la funzione `maxABR`, valgono considerazioni assolutamente analoghe a `minABR`, con la sola differenza che sia l'implementazione ricorsiva sia quella iterativa scendono sempre lungo il sottoalbero destro e che la funzione restituisce $-\infty$ se l'albero è vuoto.

Quarta fase (`abr3.c`) Per concludere, affrontiamo la funzione più complessa dell'esercizio, cioè la funzione `deleteABR`, che cancella un elemento da un albero dato. Ancora una volta la funzione è ricorsiva:

- primo *caso base*: se l'albero è vuoto, non fa nulla;
- *caso ricorsivo*: a seconda che il dato da cancellare preceda o segua quello associato alla radice, riapplica la funzione al sottoalbero sinistro o a quello destro;
- secondo *caso base*: se l'elemento da cancellare è conservato in radice, si possono distinguere due casi:
  - se la radice ha un solo nodo figlio (o nessuno), si può direttamente appendere tale figlio (se esiste) al nodo padre, "cortocircuitando" l'albero;
  - se la radice ha due figli, eliminarla compromette la struttura gerarchica dell'albero; la soluzione è (come nella cancellazione da tabelle) sovrascriverla con un altro elemento, e cancellare quest'ultimo, ma per non compromettere l'ordine totale degli elementi bisogna scegliere in modo oculato quale cancellare.

Discutiamo in maggiore dettaglio le due operazioni più complicate delineate più sopra. L'operazione che "cortocircuita" l'albero è abbastanza semplice a parole, ma presenta molte piccole differenze a seconda che il figlio esista o no, che sia a sinistra o a destra e che vi sia un nodo padre a cui appendere il figlio oppure no (se si cancella la radice, non esiste un nodo padre). Per semplicità di lettura, e per

essere fedeli alla modalità *top-down*, accorpiamo il tutto in un'apposita funzione `ABR toglienodoABR (ABRnodo *n, ABR T)`, che elimina un nodo `n` da un albero `T` restituendo il nuovo albero `T` modificato. La funzione procede come segue:

1. determina l'unico sottoalbero esistente (eventualmente nessuno);
2. se c'è un sottoalbero lo appende al padre (eventualmente assente);
3. se c'è un padre, determina in quale sottoalbero del padre sta il nodo da cancellare e sostituisce ad esso il sottoalbero del nodo cancellato;
4. finalmente, dealloca il nodo.

La funzione può essere usata solo se il nodo ha al massimo un figlio, e per efficienza non controlla che la condizione sia rispettata: lo deve fare l'utente. Non riportiamo tale funzione nel file di intestazione `abr.h` perché non è verosimile che un utente esterno lo usi per operare sull'albero e perché la funzione presuppone esplicitamente un'implementazione a puntatori nella definizione del suo secondo argomento, che è un `nodo *`. Affinché possa essere chiamata dalla funzione `deleteABR`, deve essere definita prima di quella, in modo che il compilatore l'abbia già letta quando arriva a occuparsi della funzione chiamante.

Nel caso in cui invece il nodo da cancellare abbia due figli, bisogna cercarne un altro che sia cancellabile semplicemente e che possa essere messo al posto di quello da cancellare senza compromettere l'ordinamento totale. Un possibile candidato è l'elemento massimo che precede quello da cancellare[2], dato che è certamente una foglia e che si può spostare senza cambiare l'ordine complessivo. Per trovare tale nodo, definiamo una funzione `argmaxABR` che esegue le stesse operazioni di `maxABR`, ma restituisce un nodo anziché un dato, e la applichiamo al sottoalbero sinistro del nodo da cancellare. In questo modo, troviamo il massimo del sottoalbero sinistro, che è esattamente il nodo cercato. Anche questa funzione, che è di servizio e usa esplicitamente (nel tipo del risultato) l'implementazione a puntatori, non comparirà nel file di intestazione, ma verrà semplicemente definita prima della funzione chiamante `deleteABR`. Ottenuto il nodo, ne copiamo il dato sulla radice e lo cancelliamo, dato che ora è un duplicato. Possiamo usare `toglienodoABR` senza compromettere la struttura perché il nodo certamente non ha un sottoalbero destro. Nel complesso, anche nel caso pessimo, la complessità è proporzionale al numero di chiamate ricorsive, cioè alla profondità h dell'albero: $\Theta(h)$.

[2] Un altro sarebbe l'elemento minimo che lo segue.