Alberi AVL

Algoritmi e strutture dati

C.D.L. Ingegneria dell'informatica e dell'automazione

Gigli Alessandro,
Laudenzi Guido,
Mannini Luca,
Pavani Tommaso

Indice

Capitolo 1

INTRODUZIONE

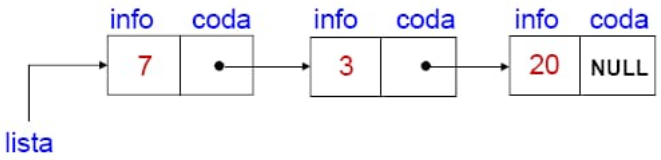
1.1 Strutture dati dinamiche

Una struttura dati è un insieme di dati organizzati secondo un preciso criterio.
Esse si dividono in due principali categorie:

- **Statiche**: in questo tipo il programmatore conosce in principio la dimensione dei dati, una volta dichiarata non è possibile modificarla lungo il corso del programma (es. Array).
- **Dinamiche**: al contrario, è qua possibile aumentare o diminuire la dimensione dei dati durante l'esecuzione del programma.

Utilizzando una struttura dati dinamica, è quindi possibile aggiungere o rimuovere gli elementi in modo da ottenere una struttura dati adeguata al programma, evitando problemi di dimensionamento. Tutto questo è reso possibile grazie all'utilizzo dei puntatori che possono essere allocati o deallocati a piacimento, inoltre essi possono modificare i collegamenti presenti tra loro, andando a creare strutture dati complesse.

Un esempio di struttura dati dinamica è una lista, formata da un insieme di nodi che sono collegati in fila. Di essi si conosce solo il puntatore alla testa (cioè il primo nodo), con il quale è possibile scorrere i rimanenti nodi della struttura facendolo avanzare.



1.2 Alberi

Un albero è una struttura dati dinamica che permette di effettuare in maniera efficiente le operazioni con un tempo di esecuzione logaritmico.
Esso è composto da:

- **Nodo**: contiene le informazioni da memorizzare; una chiave, che identifica univocamente il nodo, permettendo il loro ordinamento e quindi una rapida ricerca; le informazioni strutturali (puntatori ad altri nodi).
- **Arco**: stabilisce un collegamento ordinato fra due nodi.

In un albero, ogni nodo, può avere un numero indeterminato di archi uscenti, ma uno solo entrante. Si ha quindi una gerarchia formata dal nodo dal quale esce l'arco, chiamato **Padre**, il quale entra nel secondo, chiamato **Figlio**.

Per formare un albero occorre collegare i vari nodi in modo da averne uno, chiamato **Radice**, di cui è noto il puntatore, privo di archi entranti, ed uno, chiamato **Foglia**, privo di uscenti. I nodi compresi tra radice e foglie, sono detti **nodi interni**.

Definiamo l'altezza di un albero come il massimo della lunghezza dei suoi cammini radice-foglia.

Il **grado** di un albero equivale al numero di archi uscenti di un nodo.

1.3 Alberi binari di ricerca

Un albero binario di ricerca (binary tree search, BTS) è un albero i cui nodi hanno grado compreso tra 0 e 2; ogni "padre" ha quindi al massimo due figli chiamati sottoalbero destro e sottoalbero sinistro.

Un BTS ha, ricorsivamente per ogni nodo, due proprietà:

- Il sottoalbero sinistro di un nodo, contiene soltanto nodi con chiavi minori di quella del loro padre.
- Il sottoalbero destro, invece, contiene soltanto nodi con chiavi maggiori di quella del loro padre.

Grazie a queste proprietà, l'albero risulta **ordinato**.

N.B. Non è possibile avere due chiavi con lo stesso valore, in quanto contraddice la definizione stessa di chiave.

Un BTS è completo quando sono soddisfatte contemporaneamente le seguenti condizioni:

- Tutte le foglie hanno la stessa profondità.
- Tutti i nodi interni hanno esattamente due figli.

La complessità delle principali operazioni è $O(h)$ dove h è l'altezza massima dell'albero.

Capitolo 2

ALBERI AVL

2.1 Alberi AVL

Gli alberi AVL (creati da Adel'son-Velkskij e Londis), sono alberi binari di ricerca bilanciati in altezza, questo significa che le altezze dei sottoalberi destro e sinistro differiscono al più di uno.

Si ha un albero AVL se il fattore di bilanciamento di ogni nodo u presente sull'albero vale $-1 \leq \beta(u) \leq 1$ cioè $|\beta(u)| \leq 1$.

Il **fattore di bilanciamento** di un nodo è la differenza di altezza del suo sottoalbero sinistro e del suo sottoalbero destro.

$$\beta(u) = altezza(u \to sx) - altezza(u \to dx)$$

Un albero completo ha il fattore di bilanciamento $\beta(u) = 0$. Gli alberi bilanciati, come gli AVL, sono particolarmente adatti per realizzare strutture dati dinamiche in cui è necessario effettuare operazioni di ricerca, inserimento e cancellazione. Questo perché garantiscono una complessità $\theta(h)$ dove $h = \theta(\log_2 n)$ e n è il numero di nodi.

TEOREMA Sia T un albero AVL di n nodi e altezza h, allora $h = \theta(\log_2 n)$

Dimostrazione: Il numero di nodi n è limitato da $N(h) \leq n \leq 2^{h+1} - 1$, dove $2^{h+1} - 1$ è il numero di nodi di un albero binario completo di altezza h e N(h) è il numero minimo di nodi che un albero bilanciato di altezza h deve avere.

Dividendo in due disuguaglianze otteniamo che:

- $n \leq 2^{h+1} - 1$ diventa $n + 1 \leq 2^{h+1}$ da cui per le proprietà dei logaritmi otteniamo $\log_2 (n + 1) - 1 \leq h$ che fornisce un limite inferiore al valore di h.

- $N(h) \leq n$ fornisce un limite superiore al valore di h, si ricava utilizzando gli alberi di Fibonacci (sono un sottoinsieme degli alberi bilanciati con il minor numero di nodi a parità di altezza).

$$N(h) = \begin{cases} 1 + N(h-1) + N(h-2) & h \geq 2 \\ 1 & h = 1 \\ 2 & h = 2 \end{cases}$$

  Adel'son-Velkskij e Londis hanno dimostrato che un albero di Fibonacci di n nodi ha $h < 1,44 * \log_2(n+2) - 0,328$.

  Si ha quindi $\log_2(n+1) - 1 \leq h < 1,44 * \log_2(n+2) - 0,328$, in altre parole $h = \theta(\log_2 n)$

La struttura in C di un nodo di un generico albero AVL è la seguente.

```
struct nodo{
  int key;  //chiave del nodo
  int h;    //altezza del nodo
  struct nodo *sx;  //puntatore al sottoalbero sinistro
  struct nodo *dx;  //puntatore al sottoalbero destro
};
```

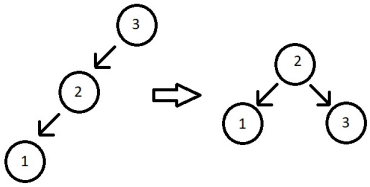
2.2 Bilanciamento

Come già accennato, l'albero AVL è un albero bilanciato in altezza. A seguito di un inserimento, o di una cancellazione, può accadere che il coefficiente di bilanciamento di un nodo risulti $|\beta(u)| \leq 1$. Occorre quindi eseguire un bilanciamento per mantenere intatte le proprietà dell'albero.

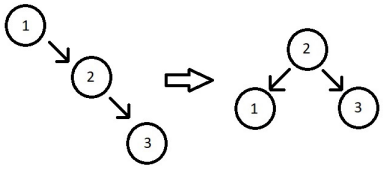
Questo bilanciamento viene eseguito tramite l'utilizzo di alcune operazioni dette **rotazioni**. Ne esistono di 4 tipi:

- **Rotazione a sinistra (ss)**: si ha a seguito di un inserimento a sinistra del sottoalbero sinistro.

```
struct nodo *ruota_ss(struct nodo *root){
  struct nodo *tmp;
  tmp=root->sx;
  root->sx=tmp->dx;
  tmp->dx=root;
  root->altezza=max(altezza_nodo(root->sx), altezza_nodo(
    root->dx))+1;
  tmp->altezza=max(altezza_nodo(tmp->sx), altezza_nodo(tmp->
    dx))+1;
  return tmp;
}
```



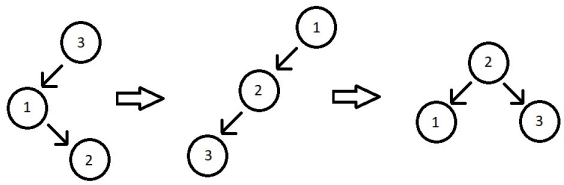
- **Rotazione a destra (dd)**: si ha a seguito di un inserimento a destra del sottoalbero destro.



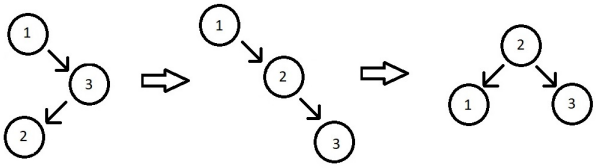

```
struct nodo *ruota_dd(struct nodo *root){
  struct nodo *tmp;
  tmp=root->dx;
  root->dx=tmp->sx;
  tmp->sx=root;
  root->altezza=max(altezza_nodo(root->sx), altezza_nodo(
    root->dx))+1;
  tmp->altezza=max(altezza_nodo(tmp->sx), altezza_nodo(tmp->
    dx))+1;
  return tmp;
}
```

- **Doppia rotazione sinistra destra (sd)**: si ha a seguito di un inserimento a destra del sottoalbero sinistro.

```
struct nodo *ruota_sd(struct nodo *root){
  root->sx=ruota_dd(root->sx);
  return ruota_ss(root);
}
```



- **Doppia rotazione destra sinistra (ds)**: si ha a seguito di un inserimento a sinistra del sottoalbero destro.




```
struct nodo *ruota_ds(struct nodo *root){
  root->dx=ruota_ss(root->dx);
  return ruota_dd(root);
}
```

La funzione max() restituisce il massimo tra due numeri interi.

```
int max(int a, int b){
  if(a>b) return a;
  else return b;
}
```

La funzione altezza nodo restituisce il valore dell'altezza memorizzato all'interno del nodo, se non è presente (p==NULL) restituisce -1.

```
int altezza_nodo(struct nodo *p){
  if(p==NULL) return -1;
  else return p->altezza;
}
```

2.3 Inserimento

La funzione di inserimento ha come argomento il puntatore alla radice, cerca ricorsivamente la posizione esatta in cui inserire il nuovo elemento e, quando trova uno spazio disponibile, crea il nodo che lo contiene.

Una volta inserito l'elemento, la funzione controlla il coefficiente di bilanciamento e, se sbilanciato, esegue un'opportuna rotazione.

La complessità dell'operazione di inserimento nel caso medio è $\theta = (\log_2 n)$.

```
struct nodo *inserisci(struct nodo *root, struct nodo *t){
  if(root==NULL){
    root=(struct nodo *)malloc(sizeof(struct nodo));
    root->key=t->key;
    root->altezza=0;
    root->sx=NULL;
    root->dx=NULL;
    return root;
  }
  else if(t->key < root->key){
    root->sx=inserisci(root->sx, t);
    if(altezza_nodo(root->sx)-altezza_nodo(root->dx)==2){
      if(t->key < root->sx->key) root=ruota_ss(root);
      else root=ruota_sd(root);
    }
  }
  else if(t->key > root->key){
    root->dx=inserisci(root->dx, t);
    if(altezza_nodo(root->dx)-altezza_nodo(root->sx)==2){
      if(t->key > root->dx->key) root=ruota_dd(root);
      else root=ruota_ds(root);
    }
  }
  root->altezza=max(altezza_nodo(root->sx), altezza_nodo(root->dx
    ))+1;
  return root;
}
```

2.4 Ricerca

L'operazione di ricerca di un elemento all'interno dell'albero è molto semplice; poiché l'albero è ordinato basta eseguire ricorsivamente un confronto con la chiave, per poi ritornare il puntatore del nodo cercato.

Anche l'operazione di ricerca di un elemento ha complessità, nel caso medio, di $\theta = (\log_2 n)$.

```
struct albero *cerca_utente(struct albero *root, int key){
  if(root==NULL){
    return root;
  }
  else if(key < root->key) cerca_utente(root->sx, key);
  else if(key > root->key) cerca_utente(root->dx, key);
  else if(key==root->key) return root;
}
```

2.5 Visita

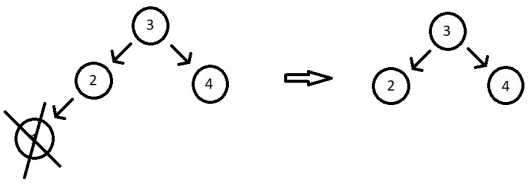
L'operazione di visita consente di scorrere tutti i nodi dell'albero per, ad esempio, stampare la chiave di ogni nodo.

```
void stampa_albero(struct nodo *root){
  if(root==NULL) return;
  printf("key=%d", root->key);
  stampa_albero(root->sx);
  stampa_albero(root->dx);
}
```

2.6 Cancellazione

La rimozione di un elemento si divide in tre casistiche:

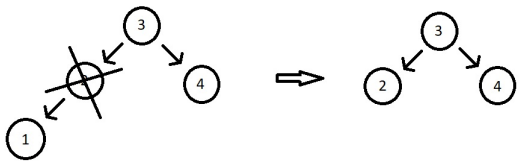
- L'elemento da cancellare è una foglia.



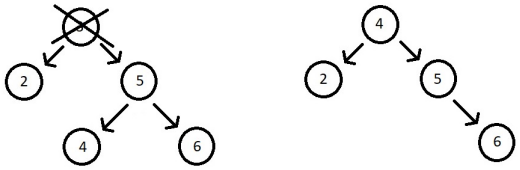
  In questo caso la soluzione è immediata, occorre semplicemente rimuovere la foglia dell'albero.

- L'elemento da cancellare ha un solo figlio.

  Anche in questo caso è relativamente semplice, basta copiare il figlio nel nodo da eliminare ed eliminare il figlio copiato.



- L'elemento da cancellare ha due figli.



Il terzo caso è il più complicato, occorre prima trovare il successore, cioè il nodo con la chiave minore fra tutti i nodi del sottoalbero destro, che nel nostro caso sarà il nodo più a sinistra del sottoalbero destro, essendo l'albero già ordinato. Trovato il successore, esso va copiato nel nodo da cancellare per poi essere eliminato.

```
1  struct albero *elimina_utente(struct albero *root, struct albero
       *t){
2    if(root==NULL) return root;
3    if(t->key < root->key){    //cerco il nodo da cancellare
4      root->sx=elimina_utente(root->sx, t);
5    }
6    else if(t->key > root->key){
7      root->dx=elimina_utente(root->dx, t);
8    }
9    else{
10     if(root->sx==NULL){
11       struct albero *tmp;
12       tmp=root->dx;
13       free(root);
14       return tmp;
```

```
15      }
16      else if(root->dx==NULL){
17        struct albero *tmp;
18        tmp=root->sx;
19        free(root);
20        return tmp;
21      }
22      struct albero *tmp;    //cerco il successore
23      tmp=root->dx;
24      while(tmp->sx!=NULL){
25        tmp=tmp->sx;
26      }
27      root->key=tmp->key;    //copio il successore nel nodo da
        cancellare
28      root->altezza=tmp->altezza;
29      root->dx=elimina_utente(root->dx, t);
30    }
31    return root;
32  }
```

Anche la cancellazione di un elemento ha complessità $\theta = (\log_2 n)$.

Capitolo 3

PROGETTO

Un'applicazione reale di un albero AVL può essere la gestione degli account utente di un videogioco online. Immaginiamo di avere dal lato server, immagazzinato grazie ad un albero, ogni account utente ordinato in ordine alfabetico; dal lato client avremo le comuni operazioni di registrazione di un nuovo utente o di login. Quando il client chiede la registrazione di un nuovo account utente, viene chiesto l'inserimento dei dati personali. A questo punto, confermando la registrazione, viene inoltrata la richiesta al server che crea un nuovo nodo dell'albero contenente il nuovo utente. Quando invece il client effettua il login viene prima effettuata una criptazione dei dati inseriti in modo da proteggere i dati sensibili, viene poi inoltrata la richiesta di login al server che ricerca l'utente fra i dati memorizzati e restituisce i suoi dati.

Una possibile realizzazione locale (simulando server e client) è questa.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//definisco la struttura dell'albero
struct albero{
  char NOME[20];
  char COGNOME[20];
  char UTENTE[20];
  char PSSWD[10];
  char key[30];
  int LVL;
  int altezza;
  struct albero *sx;
  struct albero *dx;
};

//funzione per criptare l'account
char *codifica(char UTENTE[], char PSSWD[]){
```

```
20   char *p;
21   char key[30];
22   strcpy(key, "\0");
23   strcat(key, UTENTE);
24   strcat(key, PSSWD);
25   p=key;
26   return p;
27 }
28
29 //funzione per calcolare il massimo tra due interi
30 int max(int a, int b){
31   if(a>b) return a;
32   else return b;
33 }
34
35 //restituisce l'altezza di un nodo
36 int altezza_nodo(struct albero *t){
37   if(t==NULL) return -1;
38   else return t->altezza;
39 }
40
41 //inserimento nel sottoalbero sinistro di un figlio sinistro
42 struct albero *ruota_ss(struct albero *root){
43   struct albero *tmp;
44   tmp=root->sx;
45   root->sx=tmp->dx;
46   tmp->dx=root;
47   root->altezza=max(altezza_nodo(root->sx), altezza_nodo(root->dx
       ))+1;
48   tmp->altezza=max(altezza_nodo(tmp->sx), tmp->altezza)+1;
49   return tmp;
50 }
51
52 //inserimento nel sottoalbero destro di un figlio destro
53 struct albero *ruota_dd(struct albero *root){
54   struct albero *tmp;
55   tmp=root->dx;
56   root->dx=tmp->sx;
57   tmp->sx=root;
58   root->altezza=max(altezza_nodo(root->sx), altezza_nodo(root->dx
       ))+1;
59   tmp->altezza=max(altezza_nodo(tmp->dx), tmp->altezza)+1;
60   return tmp;
61 }
62
63 //inserimento nel sottoalbero sinistro di un figlio destro
64 struct albero *ruota_sd(struct albero *root){
65     root->sx=ruota_dd(root->sx);
66     return ruota_ss(root);
```

```
67 }
68
69 //inserimento nel sottoalbero destro di un figlio sinistro
70 struct albero *ruota_ds(struct albero *root){
71     root->dx=ruota_ss(root->dx);
72     return ruota_dd(root);
73 }
74
75 //funzione per inserire un altezza_nodo
76 //viene effettuata una rotazione se il nodo inserito sbilancia l'
       albero
77 struct albero *inserisci_nodo(struct albero *root, struct albero
       *t){
78   if(root==NULL){
79     root=(struct albero *)malloc(sizeof(struct albero));
80     strcpy(root->NOME,t->NOME);
81     strcpy(root->COGNOME,t->COGNOME);
82     strcpy(root->UTENTE,t->UTENTE);
83     strcpy(root->PSSWD,t->PSSWD);
84     strcpy(root->key,t->key);
85     root->LVL=t->LVL;
86     root->altezza=0;
87     root->sx=NULL;
88     root->dx=NULL;
89     return root;
90   }
91   else if(strcmp(t->key, root->key) < 0){
92     root->sx=inserisci_nodo(root->sx, t);
93     if(altezza_nodo(root->sx)-altezza_nodo(root->dx)==2){
94       if(strcmp(t->key, root->key) < 0) root=ruota_ss(root);
95       else root=ruota_sd(root);
96     }
97   }
98   else if(strcmp(t->key, root->key) > 0){
99     root->dx=inserisci_nodo(root->dx, t);
100    if(altezza_nodo(root->dx)-altezza_nodo(root->sx)==2){
101      if(strcmp(t->key, root->key) > 0) root=ruota_dd(root);
102      else root=ruota_ds(root);
103    }
104  }
105  root->altezza=max(altezza_nodo(root->sx), altezza_nodo(root->dx
       ))+1;
106  return root;
107 }
108
109 //ricerca l'utente all'interno dell'albero e restituisce il nodo
110 struct albero *cerca_utente(struct albero *root, char key[]){
111   if(root==NULL){
112     printf("Utente o password errati\n\n");
```

```
113     return root;
114   }
115   else if(strcmp(key, root->key) < 0) cerca_utente(root->sx, key)
        ;
116   else if(strcmp(key, root->key) > 0) cerca_utente(root->dx, key)
        ;
117   else if(strcmp(key, root->key)==0) return root;
118 }
119
120 //elimina dall'albero l'utente t
121 struct albero *elimina_utente(struct albero *root, struct albero
      *t){
122   if(root==NULL) return root;
123   if(strcmp(t->key, root->key) < 0){
124     root->sx=elimina_utente(root->sx, t);
125   }
126   else if(strcmp(t->key, root->key) > 0){
127     root->dx=elimina_utente(root->dx, t);
128   }
129   else{
130     if(root->sx==NULL){
131       struct albero *tmp;
132       tmp=root->dx;
133       free(root);
134       return tmp;
135     }
136     else if(root->dx==NULL){
137       struct albero *tmp;
138       tmp=root->sx;
139       free(root);
140       return tmp;
141     }
142     struct albero *tmp;
143     tmp=root->dx;
144     while(tmp->sx!=NULL){
145       tmp=tmp->sx;
146     }
147     strcpy(root->NOME, tmp->NOME);
148     strcpy(root->COGNOME, tmp->COGNOME);
149     strcpy(root->UTENTE, tmp->UTENTE);
150     strcpy(root->PSSWD, tmp->PSSWD);
151     strcpy(root->key, tmp->key);
152     root->LVL=tmp->LVL;
153     root->altezza=tmp->altezza;
154     root->dx=elimina_utente(root->dx, t);
155   }
156   return root;
157 }
158
```

```
159 //stampo i dati dell'utente
160 void stampa_utente(struct albero *t){
161   printf("%s", t->NOME);
162   printf(" %s\n", t->COGNOME);
163   printf("%s -\t", t->UTENTE);
164   printf("Livello: %d\n", t->LVL);
165 }
166
167 //aggiorna il file
168 void crea_file(struct albero *root, FILE *pf){
169   if(root==NULL) return;
170   fprintf(pf, "%s\n", root->NOME);
171   fprintf(pf, "%s\n", root->COGNOME);
172   fprintf(pf, "%s\n", root->UTENTE);
173   fprintf(pf, "%s\n", root->PSSWD);
174   fprintf(pf, "%s\n", root->key);
175   fprintf(pf, "%d\n", root->LVL);
176   crea_file(root->sx, pf);
177   crea_file(root->dx, pf);
178 }
179
180
181 //funzione per effettuare il Login
182 //crea l'albero scansionando il file utenti.txt e, trovato l'
      utente, apre il
183 //menu utente
184 void login(){
185   int val;
186   char key[30];
187   char utente[20];
188   char psswd[10];
189   struct albero *t=NULL;
190   t=(struct albero *)malloc(sizeof(struct albero));
191   struct albero *root=NULL;
192   FILE *pf_r;
193   FILE *pf_w;
194   pf_r=fopen("utenti.txt", "r");
195   while(fscanf(pf_r, "%s", t->NOME)!=EOF){
196     fscanf(pf_r, "%s", t->COGNOME);
197     fscanf(pf_r, "%s", t->UTENTE);
198     fscanf(pf_r, "%s", t->PSSWD);
199     fscanf(pf_r, "%s", t->key);
200     fscanf(pf_r, "%d", &t->LVL);
201     fscanf(pf_r, "%d", &t->altezza);
202     root=inserisci_nodo(root, t);
203   }
204   fclose(pf_r);
205   if(root==NULL){
206     printf("Nessun utente registrato\n");
```

```
207     return;
208   }
209   t=NULL;
210   while(t==NULL){
211     printf("Inserisci nome utente: ");
212     scanf("%s", utente);
213     printf("Inserisci password: ");
214     scanf("%s", psswd);
215     strcpy(key, codifica(utente, psswd));
216     printf("\n");
217     t=cerca_utente(root, key);
218   }
219   if(t!=NULL){
220     pf_w=fopen("utenti.txt", "w");
221     stampa_utente(t);
222     printf("\n\t1- Modifica\n\t2- Cancella utente\n\t");
223     printf("3- Effettua il logut\nInserisci: ");
224     while(scanf("%d", &val)){
225       if(val==1){
226         printf("1- Modifica password\n0- Annulla\nInserisci: ");
227         while(scanf("%d", &val)){
228           if(val==1){
229             printf("Inserisci nuova password: ");
230             scanf("%s", t->PSSWD);
231             strcpy(t->key, codifica(t->UTENTE, t->PSSWD));
232             crea_file(root, pf_w);
233             return;
234           }
235           else if(val==0){
236             printf("\n\t1- Modifica\n\t2- Cancella utente\n\t");
237             printf("3- Effettua il logout\nInserisci: ");
238             break;
239           }
240         }
241       }
242       else if(val==2){
243         root=elimina_utente(root, t);
244         crea_file(root, pf_w);
245         return;
246       }
247       else if(val==3){
248         crea_file(root, pf_w);
249         break;
250       }
251     }
252   }
253   fclose(pf_w);
254 }
255
```

```
256 //controlla la lunghezza dei dati inseriti dall'utente in
      registra
257 //in caso superi la lunghezza massima consentita
258 //chiede il reinserimento del dato
259 char *controlla_lunghezza(char parola[]){
260   for(;;){
261     if(strlen(parola)>19){
262       printf("Lunghezza massima 20 caratteri\nInserire nuovamente
      : ");
263       scanf("%s", parola);
264     }
265     else{
266       char *p;
267       p=parola;
268       return p;
269     }
270   }
271 }
272
273 //funzione per registrare un nuovo UTENTE
274 //scrive su file i dati inseriti
275 void registra_nuovo(){
276   char PSSWD[10];
277   char *p;
278   FILE *pf_w;
279   pf_w=fopen("utenti.txt", "a");
280   struct albero *t=NULL;
281   t=(struct albero *)malloc(sizeof(struct albero));
282   printf("Inserisci nome: ");
283   scanf("%s", t->NOME);
284   p=controlla_lunghezza(t->NOME);
285   strcpy(t->NOME, p);
286   printf("Inserisci cognome: ");
287   scanf("%s", t->COGNOME);
288   p=controlla_lunghezza(t->COGNOME);
289   strcpy(t->COGNOME, p);
290   printf("Inserisci nome utente: ");
291   scanf("%s", t->UTENTE);
292   p=controlla_lunghezza(t->UTENTE);
293   strcpy(t->UTENTE, p);
294   printf("Inserisci password: ");
295   scanf("%s", t->PSSWD);
296   while(strlen(t->PSSWD)>9){
297     printf("Password troppo lunga, lunghezza massima 9 caratteri\
      n");
298     printf("Inserire nuovamente password: ");
299     scanf("%s", t->PSSWD);
300   }
301   for(;;){
```

```
302     printf("Inserisci nuovamente la password: ");
303     scanf("%s", PSSWD);
304     while(strlen(PSSWD)>9){
305       printf("Password troppo lunga, lunghezza massima 9
      caratteri\n");
306       printf("Inserire nuovamente password: ");
307       scanf("%s", PSSWD);
308     }
309     if(strcmp(PSSWD, t->PSSWD)==0) break;
310     printf("Errore, le password non combaciano\n");
311   }
312   strcpy(t->key,codifica(t->UTENTE, t->PSSWD));
313   t->LVL=0;
314   crea_file(t, pf_w);
315   fclose(pf_w);
316 }
317
318 //scelta iniziale tra login o registrazione
319 void scelta(){
320   int val=0;
321   printf("1- Registra un nuovo utente\n");
322   printf("2- Effettua il Login\n0- Uscire\nInserisci: ");
323   while(scanf("%d", &val)){
324     if(val==1){
325       registra_nuovo();
326       return;
327     }
328     else if(val==2){
329       login();
330       return;
331     }
332     else if(val==0){
333       return;
334     }
335     printf("Inserisci nuovamente: ");
336   }
337 }
338
339 int main(){
340   scelta();
341   return 0;
342 }
```

I dati utente sono memorizzati in un file chiamato utenti.txt del tipo

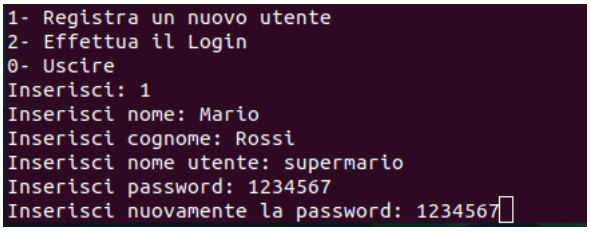
```
1 luigi
2 verde
3 killuigi
4 qwerty
5 killuigiqwerty
```

```
31
giuseppe
bianchi
white95
asdfgh
white95asdfgh
4
Mario
Rossi
supermario
1234567
supermario1234567
0
```

in cui sono salvati i dati dell'utente nell'ordine:

- Nome
- Cognome
- Nome utente
- Password
- key
- Livello

Quando viene effettuata una registrazione è richiesto l'inserimento dei dati personali che vengono aggiunti al file; il livello di un nuovo utente viene settato a 0 ed si codificano Nome utente e Password, creando la chiave. In questo programma la codifica è semplicemente Nome utente e Password scritti nella stessa stringa.



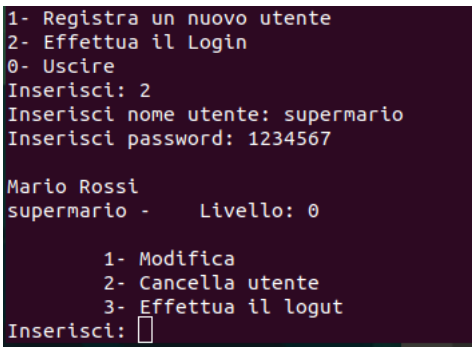
Se durante l'inserimento dei dati vengono superati i caratteri massimi consentiti, la funzione controlla lunghezza chiede il reinserimento del dato.

Effettuando il login viene scansionato il file utenti.txt creando un nodo per ogni utente. Esso viene poi inserito nell'albero ed ordinato in ordine alfabetico a seconda della chiave. Se l'inserimento di un nodo sbilancia l'albero si effettua una rotazione adeguata in modo da mantenere l'albero sempre bilanciato in altezza. Creato l'albero viene fatto l'accesso vero e proprio chiedendo di inserire Nome utente e Password, poi codificati nella chiave; la chiave viene poi usata nell'operazione di ricerca.

L'operazione di ricerca viene eseguita ricorsivamente all'interno dell'albero fino a che non viene trovato il nodo con la stessa chiave ricercata, o fino a che non si raggiunge la fine dell'albero; in quest'ultimo caso viene stampato a schermo un messaggio di errore "Utente o password errati". Se l'albero è vuoto (cioè il file è vuoto) significa che non è presente nessun utente registrato; anche in questo caso compare un messaggio di errore indicando appunto "Nessun utente registrato".

Trovato il nodo corrispondente, al login viene aperto il menu utente in cui vengono stampati i dati salvati e in cui è possibile effettuare operazioni di manutenzione dell'account:

- Modificare la password
- Cancellare l'utente



La password viene modificata aggiornando il valore contenuto nel nodo mentre la cancellazione dell'utente consiste nell'eliminare il nodo dall'albero (cancellazione di un elemento). A seguito di entrambe le operazioni viene aggiornato il file, tramite l'operazione di visita, copiando ogni nodo presente nell'albero al suo interno, sovrascrivendo tutti i dati. Al termine del programma il file sarà quindi ordinato permettendo, ad un successivo riavvio, una rapida creazione dell'albero.