Progetto di algoritmi e strutture dati

Progetto "Facebook"

Cordoni Marco 855535

L’implementazione

Il tipo di implementazione scelta per il mio progetto è l’albero 2_3, nel quale i nodi saranno strutture che ne conterranno a loro volta altre, fra cui, oltre alle strutture tipiche necessarie alle operazione all’interno dell’albero, ovvero, 3 puntatori ai nodi figli più un altro figlio ausiliario ed un puntatore al padre, vi sarà anche un puntatore ad una struttura “utente” che conterrà le informazioni degli utenti del programma.

Gli utenti saranno collegati fra loro da una lista per permettere l’utilizzo di un’ attraversamento in ampiezza degli amici di ogni utente e la visualizzazione di gruppi di amici, collegati fra loro da amicizie dirette o indirette.

Quindi ogni struttura “utente” avrà anche una lista di adiacenza di strutture “amico”, le quali indicheranno l’amico dell’utente a cui appartengono e l’anno di creazione della loro relazione.

Ho scelto di implementare il progetto come un albero2_3 in cui i nodi sono inseriti secondo il campo numerico intero positivo “id”, presente nella struttura dell’utente, che corrisponde all’identificativo univoco che contraddistingue gli utenti.

Tale scelta di implementazione è dovuta al fatto che in un albero2_3, nel caso peggiore, le operazioni di inserimento e ricerca hanno un tempo dell’ordine di O(log n), che corrisponde all’altezza dell’albero, dato che gli alberi 2_3 sono sempre bilanciati, invece che essere O(n) come per gli alberi binari, n corrisponde al numero di nodi.

Uso la notazione T(T1) per indicare il puntatore a T1 di un nodo T, oppure u(id) per il campo identificatore di un utente.

Le strutture

Tale progetto utilizzerà diversi tipi di strutture, queste avranno caratteristiche diverse a seconda della funzione.

```
1-  struct grafo{
2-          struct tree2_3 *T;
3-          struct nodo *head;
4-          struct nodo *last;
5-          int conta;
6-  }
```

La struttura “grafo” contiene diversi tipi di informazioni, ovvero, la prima è un puntatore alla radice dell’albero, a riga 3 vi è un puntatore al primo utente inserito nella relativa lista, e quindi anche nell’albero, e per velocizzare le operazioni di inserimento è presente anche un puntatore all’ultimo utente inserito, in fondo ad essa.
infine un campo di nome “conta” tiene traccia del numero di utenti presenti nel grafo.

```
1-  struct nodo{
2-          struct utente *profilo;
3-          int L;
4-          int M;
5-          struct nodo *T1;
6-          struct nodo *T2;
7-          struct nodo *T3;
8-          struct nodo *T4;
9-          struct nodo *father;
10- }
```

La struttura che costituirà i nodi dell’albero è la struttura “nodo”.
Essa è costituita da un puntatore ad una struttura utente (riga 2), i campi L ed M (righe 3 e 4) corrispondono al massimo campo id presente rispettivamente in T1 e T2, sono uguali all’id del nodo stesso solo sulle foglie.
I puntatori T1,T2,T3,T4 (righe da 5 ad 8), di cui l’ultimo è solo ausiliario durante l’inserimento, sono usati per indicare i figli di ogni nodo e si riferiscono ad altre strutture nodo.
Il puntatore al padre (riga 9) del nodo è utile nelle operazioni di ricerca della posizione di inserimento.

```
1-  struct utente{
2-          stringa nome;
3-          stringa cognome;
4-          int id;
5-          int pos;
6-          struct amico *friend;
7-          struct utente *next;
8-  }
```

La struttura “utente” contiene le informazioni delle persone, iscritte a Facebook, quali: nome, cognome ed identificativo (righe 2,3 e 4), il puntatore a questa struttura, presente nei nodi, non sarà nullo solo sulle foglie perché contengono le informazioni utili.

A riga 5 vi è un campo che memorizza l’ordine di inserimento di tale utente, uso un contatore per mantenere il numero di utenti a cui sono giunto, alla riga 6 vi è un puntatore ad una struttura “amico” che permette di accedere alla lista d’adiacenza di amici dell’utente in questione e alla riga successiva vi è un puntatore al prossimo utente.

```
1-  struct amico{
2-          struct utente *friend;
3-          int anno;
4-          struct amico *next;
5-  }
```

Infine vi è la struttura utilizzata per definire gli amici di un utente, la quale contiene a riga 2 un puntatore all’amico in questione, quindi punta alla sua struttura “utente”, l’anno in cui è nata l’amicizia ed un puntatore al prossimo amico della lista d’adiacenza dello stesso utente a riga 4.

Costo di memorizzazione dell’albero

Gli alberi binari hanno un costo di memorizzazione pari a O (n+l), dove n corrisponde al numero di nodi ed l al numero di lati, ovvero, n-1, quindi O(n+n-1) è O (n).

Le operazioni

All'avvio il programma chiederà all'utente l'inserimento di un intero positivo n, il quale rappresenterà il numero di utenti iniziali.

Successivamente, per mezzo di un ciclo, verranno inseriti gli n utenti, uno per volta, nel formato:

nome_i cognome_i identificativo_i

corrispondenti ai campi nome, cognome ed id presenti nella struttura "utente", quindi, per ogni inserimento sarà necessario creare una struttura "utente", inizializzare i suoi valori a quelli inseriti da standard input ed invocare la funzione tree2_3 insert(utente *u, grafo G), il cui valore di "u" corrisponde all'utente appena creato e "G" corrisponde ad una struttura "grafo" la quale contiene un puntatore alla radice dell'albero e potrebbe essere anche NULL, come nel caso del primo inserimento.

Infine verrà mostrato un menù fra cui scegliere una delle seguenti operazioni finché non si deciderà di uscire dal programma.

2.1)Inserire un nuovo utente Facebook

Questa operazione inizia con l'uso della funzione tree2_3 new_utente(grafo G).

```
1-  tree2_3 new_utente( grafo G){
2-      u = nuova struttura di tipo utente
3-      do{
4-          new_nome = prendi in input una stringa
5-          new_cognome = prendi in input una stringa
6-          new_id = prendi in input un intero
7-          if((lunghezza del nome)>30) OR (lunghezza del cognome)>30) OR (id<=0))
8-              stampa ("Sono presenti input errati")
9-      }while((lunghezza del nome)>30) OR (lunghezza del cognome)>30) OR (id non maggiore di zero))
10-     u(nome)=new_nome
11-     u(cognome)=new_cognome
12-     u(id)=new_id
13-     G(T)=insert(u,G)
14-     return G
15- }
```

La funzione alloca lo spazio necessario per un nuovo utente, richiede l'inserimento dei suoi dati: nome, cognome ed identificativo e lascia inizializzati a NULL tutti i suoi puntatori,tali valori verranno accettati solo quando tutti rispetteranno le caratteristiche richieste, ovvero che i campi nome e cognome non eccedano lo spazio consentito e l'identificatore sia positivo.

A riga 12 viene richiamata la funzione insert sul nuovo utente nel grafo G e ritorna l'albero eventualmente modificato.

La funzione ha un costo dell'ordine di $O((\log n)^2)$, cioè il costo per l'inserimento di un nodo(insert) in un albero2_3 perché tutte le altre operazioni sono dell'ordine di O(1).

```
1-  tree2_3 insert (utente *u, grafo G) {
2-          if (G(T) == NULL)
3-                  G(T) = create_tree2_3(u,G)
4-           else {
5-                   p = search_father(u,G(T))
6-                  if (search(u,G(T))==NULL)
7-                          append(u,p,G)
8-                  else{
9-                          stampa ("Inserimento fallito, l'identificativo e' già presente")
10-                         return G(T)
11-                 }
12-                 if(p(T4)!=NULL)
13-                         reduce(p,G)
14-                 update_thresholds(p)
15-          }
16-          return G(T)
17- }
```

Alla riga 2 viene gestito il caso in cui l'albero passato sia nullo, ovvero non è ancora presente alcun nodo, quindi viene creato un nuovo albero contenente solamente l'utente passato, per mezzo della funzione tree2_3 create_tree(utente *u, grafo G).

Se la condizione però risulta falsa allora viene eseguito il blocco di istruzioni che inizia a riga 4, il quale prevede innanzitutto la ricerca del potenziale padre attraverso la funzione nodo search_father(utente *u, tree2_3 T) a riga 5.

A riga 6 viene eseguita un'operazione di ricerca di tale nodo all'interno dell'albero con utente *search(utente *u, tree2_3 T), con lo scopo di verificare che l'elemento non esista già, quindi non esista un utente con lo stesso numero identificativo.

Solo se tale funzione ritorna NULL allora si potrà procedere con l'inserimento, altrimenti verrà visualizzato un messaggio di errore e l'inserimento non avverrà e l'albero verrà ritornato non modificato.

Nel caso in cui la condizione di riga 6 dia esito positivo si procede all'inserimento dell'utente u nel nodo p per mezzo della procedura void append(utente *u, nodo *p, grafo G), la quale inserisce tra i figli di p un nuovo nodo con u come profilo.

A riga 12, una condizione verifica che il quarto figlio di p non sia nullo, se questa condizione risulta vera significa che il precedente append ha generato un nodo con 4 figli, e se tale situazione si verifica è necessario procedere con la funzione void reduce(nodo *p, grafo G) il cui compito è di riparare l'albero in corrispondenza del nodo p.

Infine è necessario aggiornare tutte le etichette L ed M di tutti i nodi da p fino alla radice usando la funzione void update_threshold(nodo *p) (riga 14).

La funzione ritorna l'albero T del grafo G eventualmente modificato ed ha una complessità dell'ordine di $O((\log n)^2)$, perché le funzioni: search_father , search e reduce sono dell'ordine di O(log n), create_tree e append sono O(1) ma la funzione update_thresholds è dell'ordine di $O((\log n)^2)$.

In questa funzione ne sono state usate altre che verranno analizzate di seguito.

```
1-  tree2_3 create_tree(utente *u, grafo G){
2-      n1=alloca lo spazio necessario per un nodo
3-      n2=alloca lo spazio necessario per un nodo
4-      n2(profilo) = u
5-      n2(L) = ident(u)
6-      n2(M) = n2(L)
7-      G(conta)=0
8-      u(pos) = G(conta)
9-      G(conta)=G(conta)+1
10-     G(head) = u
11-     G(last)=u
12-     G(T) = n1
13-     n1(T1)=n2
14-     n1(L)=n2(M)
15-     return G(T)
16- }
```

Questa funzione alloca lo spazio necessario per un 2 nuovi nodi (n1 e n2), inizializza il valore del profilo di n2 a quello passato alla funzione ed i suoi valori di soglia al valore id dell'utente perché questo sarà una foglia, i puntatori di questo nodo saranno tutti a NULL, la sua posizione di inserimento è inizializzata al contatore degli elementi di G e tale sarà zero perché questo è il primo elemento inserito.

Il nodo n1 che avevo allocato costituirà il padre di n2, quindi il suo T1 punterà a quella foglia e la sua soglia L sarà uguale all'identificatore di quel nodo.

E' fondamentale creare un padre alla foglia perché altrimenti la funzione search_father non troverebbe alcun candidato al padre nelle sue successive chiamate, questo è l'unico caso in cui un nodo può avere un solo figlio.

Aggiorno il valore di conta e faccio puntare la testa e la fine della lista di utenti al profilo passato ed il puntatore alla radice dell'albero al nodo padre n1.
Questa funzione ha una complessità dell'ordine di O(1), cioè esegue un numero costante di istruzioni.

```
1-  int ident(utente *u)
2-      return u(id)
```

La funzione ident prende in input un utente e ritorna il suo numero identificativo, ha una complessità dell'ordine di O(1).

```
1-  nodo search_father(utente *u, tree2_3 T){
2-      if (T == NULL)
3-          return NULL
4-      i = ident(u)
5-      if (T(T1(T1)) == NULL)
6-          return T
7-      else if (i <= T(L))
8-          return search_father (u,T(T1))
9-      else if ( (i <= T(M)) OR (T(T3) == NULL) )
10-         return search_father (u,T(T2));
11-     else /* (i >T(M)) AND (T(T3) != NULL) */
12-         return search_father (u,T(T3))
13- }
```

Grazie a questa funzione è possibile stabilire dove inserire un nuovo nodo perché essa prende in ingresso il nuovo utente ed il puntatore alla radice dell'albero in cui ricercare il suo potenziale padre e ritorna NULL se l'albero passato è nullo e quindi non esiste un possibile padre, altrimenti ritorna il nodo cercato.

La funzione a riga 4 mi permette di conoscere l'identificatore dell'utente passato, il quale mi servirà per spostarmi all'interno dell'albero confrontandolo con i valori di soglia e verificando l'esistenza dei puntatori dei nodi in cui mi sposto.

A riga 5 controllo se il puntatore T1 del T1 del nodo in cui mi trovo è nullo, se tale condizione è vera significa che il prossimo nodo è una foglia perché i puntatori si riempiono in ordine da T1 a T4 e se T1 è nullo lo saranno anche gli altri, e, per definizione di foglia di un albero2_3, significa che essa si trova nell'ultimo livello dell'albero, quindi mi trovo nel padre che sto cercando, lo ritorno dalla funzione.

Se la precedente condizione è risultata falsa allora dovrò continuare a cercare richiamando la funzione stessa su uno dei figli del nodo.

Se l'identificatore dell'utente passato alla funzione è minore o uguale del valore di soglia L corrispondente al valore di soglia più grande del sottoalbero T1 allora richiamerò la funzione search_father su tale sottoalbero, altrimenti se l'id è maggiore di L ma minore o uguale ad M, che è l'id maggiore presente in T2 oppure il sottoalbero T3 è nullo quindi non si può ricercare all'interno di esso, allora richiamo la funzione su T2.

Infine, se nessuna delle precedenti condizioni risulta vera, significa che l'id è maggiore anche di M e certamente T3 non è nullo perché l'ho già verificato alla condizione precedente, quindi cerco in tale sottoalbero.

Questa funzione ha una complessità dell'ordine di O(log n) perché l'operazione di ricerca dell'identificatore è dell'ordine di O(1) ma questa operazione richiama se stessa per un numero di volte pari all'altezza dell'albero, e dato che questo è un albero bilanciato sarà sicuramente il logaritmo del numero dei nodi.

```
1-  void append(utente *u, nodo *p, grafo G){
2-        new_n=alloco lo spazio per un nuovo nodo
3-        new_n(profile)=u
4-        new_n(L)= ident(u)
5-        new_n(M)= new_n(L)
6-        new_n(father)=p
7-        u(pos) = G(conta)
8-        G(conta)=G(conta+1)
9-        inserisci new_T nella posizione corretta tra I figli di p ed esegui uno shift a destra dei figli se necessario
10-       inserisci l'utente nella lista d'adiacenza di utenti
11-       aggiorna il puntatore all'ultimo utente inserito
12- }
```

Questa operazione aggiunge un profilo utente ad un nodo, quindi è necessario innanzitutto allocare lo spazio per un nuovo nodo (riga 2), i puntatori ai figli devono essere inizializzati a NULL e ciò è corretto perché stiamo aggiungendo una foglia, tranne il padre che punterà al nodo a cui è stato inserito.

Successivamente il profilo utente del nuovo nodo allocato viene inizializzato a quello passato per argomento alla funzione, i valori di soglia devono essere posti uguali al suo identificatore ed, a riga 7, viene inserita in esso la sua posizione in ordine di inserimento ed aggiornato tale valore.

Per scegliere dove inserire il nuovo nodo bisogna controllare il valore degli identificatori tra i figli del nodo passato, se sono tutti inferiori allora andrà inserito in fondo, altrimenti è necessario effettuare uno shift a destra di una posizione dei puntatori ed inserire il nuovo nella posizione corretta.

Infine un nuovo utente viene aggiunto alla relativa lista, non scorrendola dal primo inserito perché utilizzo un puntatore all'ultimo utente immesso (last), per evitare di scorrere tutta la lista.

L' utente puntato da last avrà il campo next uguale a NULL, quindi gli verrà aggiunto il nuovo iscritto e last punterà a questo.

Questa funzione ha una complessità dell'ordine di O(1), perché le operazioni di assegnamento dei valori e modifica dei puntatori si svolgono in un tempo costante, e, poi, non bisogna scorrere la lista d'adiacenza di utenti fino a giungere all'ultimo, quindi non dovrò passare per n profili ed evito che costi O(n).

```
1-  void reduce(nodo *p1, grafo G){
2-        crea un albero formato da un nodo p2
3-        sposta(p1,p2)
4-        if( p1 era la radice dell'albero){
5-              creo una nuova radice r con figli p1 e p2
6-              G(T)=r
7-        }
8-        else( p1 non era la radice){
9-              p3=p1(father)
10-             append_interno(p3,p2)
11-       }
12-       if( p3(T4)!=NULL)
13-             reduce (p3,T)
14- }
```

Questa funzione ha il compito di aggiustare l'albero in seguito all'inserimento di un nuovo utente e la generazione di nodo con 4 figli, non conforme alle caratteristiche di un albero2_3.

Essa si occupa di creare un nuovo albero ed aggiungere ad esso gli ultimi 2 figli del nodo con l'errore(riga 3), quindi i suoi nodi T3 e T4, certamente non nulli, diventeranno i T1 e T2 del nuovo.

Successivamente bisognerà distinguere il caso in cui il nodo con 4 figli è la radice da quello in cui non lo è, nel primo caso bisognerà creare nuovamente un altro nodo detto superradice, inserire in esso i 2 precedenti e definire una nuova radice.

Altrimenti è sufficiente appendere il nuovo nodo al padre di quello che è stato corretto ma ciò può far sorgere nuovamente lo stesso problema se tale padre aveva già 3 figli e quindi sarà necessario richiamare la funzione sul padre e ripetere l'algoritmo.

Questa funzione ha una complessità dell'ordine di O(log n), perché devo effettuarla su tutti i nodi dalla foglia alla radice, ovvero su tutta l'altezza dell'albero, le operazioni di creazione di nodi sono dell'ordine di O(1), la ricerca del padre di p1 si trova semplicemente usando il suo puntatore al padre e l'operazione usata per spostare 2 sottoalberi (sposta) ed appendere un nodo interno ad un altro (append_interno) sono costanti.

```
1-  int max(nodo *p)
2-  {
3-      if(p==NULL)
4-          return NULL
5-      else if(p(T1)==NULL)
6-          return p(L)
7-      else if(p(T3)!=NULL)
8-          return max(p(T3))
9-      else /*if (p(T2)!=NULL)*/
10-         return max(p(T2))
11- }
```

Questa funzione ha il compito di individuare il massimo identificatore del sottoalbero che gli viene passato, all'inizio viene effettuato un controllo per verificare che l'albero passato non sia nullo, in tal caso viene ritornato un NULL.

A riga 5 viene controllato se Il sottoalbero T1 del nodo passato e NULL, se tale condizione è vera significa che mi trovo in una foglia e quindi ritornerò il valore di L (oppure di M), perché nelle foglie i valori di soglia sono uguali all'identificatore.

Se la precedente condizione risulta falsa allora significa che mi trovo in un nodo interno e dovrei cercare il massimo tra i figli di tale nodo, dato che gli utenti con l'identificatore maggiore vengono appesi nel sottoalbero più a destra allora rieseguo la ricerca in T3 se tale non è nullo, altrimenti in T2.

T2 sono certo che non possa essere nullo per definizione di albero2_3.

Questa funzione ha una complessità dell'ordine di O(log n), perché se la eseguo sulla radice dovrò ripeterla su un nodo per ogni livello dell'albero, quindi un numero di nodi pari all'altezza dell'albero.

```
1-  void update_threshold(nodo *p)
2-  {
3-      p(L)=max(T1)
4-      p(M)=max(T2)
5-      if(p(father)!=NULL)
6-          update_threshold(p(father))
7-  }
```

Questa funzione viene eseguita con lo scopo di aggiornare le soglie dei nodi dal nodo passato fino alla radice, ciò permette di spostarsi all'interno dell'albero correttamente.

La procedura corrente richiama "max", usata per ricavare l'identificatore maggiore di un sottoalbero e quindi assegnare tale valore alla corrispondente soglia.

A riga 5 viene controllato se il padre del nodo passato è diverso da NULL, se tale condizione è vera significa che ci sono altri nodi a cui è necessario aggiornare le soglie e quindi viene eseguita nuovamente la procedura passando come parametro il nodo padre per mezzo del puntatore.

Questa funzione ha una complessità dell'ordine di O((log n + log n)*log n), ovvero vengono eseguite 2 operazioni di ricerca del massimo tante volte quanti sono i livelli compresi tra il nodo passato la prima volta e la radice, quindi nel peggiore dei casi ciò avviene log n volte.

Quindi il costo di questa funzione è dell'ordine di $O((\log n)^2)$.

2.2)Cercare un utente Facebook

```
1-  void trova(int id, tree2_3 T){
2-      u= alloca lo spazio per un utente
3-      u=search(id,T)
4-      if(u==NULL)
5-          stampa("Utente non trovato")
6-      else
7-          stampa("u(nome) u(cognome) u(id)")
8-  }
```

La funzione "trova" riceve in ingresso un identificativo e la radice di un albero e trova il profilo dell'utente per mezzo della funzione search, se tale restituisce un NULL allora viene stampato a video un messaggio che avvisa che l'utente non è stato trovato, altrimenti viene visualizzato il suo nome, cognome e numero identificativo su una linea sola con i campi separati da spazi.

Quindi il costo di questa funzione è dell'ordine di O(log n), pari alla funzione search di riga 3, le altre sono dell'ordine di O(1).

```
1-  utente *search(int id, tree2_3 T)
2-  {
3-      if (T == NULL)
4-          return NULL
5-      if (T(T1) == NULL)
6-      {
7-          if (id == T(L))
8-              return T(profilo)
9-          else
10-             return NULL
11-     }
12-     else if (id <= T(L))
13-         return search(id,T(T1))
14-     else if (id <= T(M))
15-         return search(id,T(T2))
16-     else /*(id > T(M)) */
17-         return search(id,T(T3))
18- }
```

Questa funzione riceve in ingresso l'identificativo di un utente Facebook e la radice dell'albero in cui si vuole cercare la persona con tale dato e ritorna l'utente stesso se viene trovato e NULL altrimenti.

Se tale albero è nullo allora ritorna un NULL, altrimenti verifico se mi trovo in una foglia (riga 5), se tale condizione è vera controllo se i valori di soglia sono uguale all'id che sto cercando, se anche questa condizione dà esito positivo allora ritorno l'utente trovato, altrimenti viene ritornato un NULL perché l'id che sto cercando non esiste.

Se il nodo non era una foglia allora la ricerca continua sui figli di tale nodo, scegliendo quello appropriato sulla base dei valori di soglia.

Quindi il costo di questa funzione è dell'ordine di O(log n) perché è necessario richiamare la funzione un numero di volte pari all'altezza dell'albero.

2.3)Inserire una nuova relazione d'amicizia

```
1-  void friendship(int id1, int id2){
2-      u=search(id1,T);
3-      v=search(id2,T);
4-      if(u==NULL AND v==NULL)
5-          stampa("Utenti id1, id2 non presenti");
6-      else if(u==NULL)
7-          stampa ("Utente id1 non presente");
8-      else if(v==NULL)
9-          stampa ("Utente id2 non presente");
10-     else if(v già presente nella lista d'adiacenza di amici di u)
11-         stampa ("Amicizia gia' presente");
12-     else{
13-         stampa ("Inserisci l'anno di creazione dell'amicizia");
14-         y = inserisci l'anno di creazione dell'amicizia come intero
15-         aggiungi in testa alla lista di adiacenza di amici di u una nuova struttura amico
16-         u(friend(anno)) = year
17-         u(friend(friend)) = v
18-         aggiungi in testa alla lista di adiacenza di amici di v una nuova struttura amico
19-         v(friend(anno)) = year
20-         v(friend(friend)) = u
21-     }
22- }
```

Questa funzione permette di inserire una relazione d'amicizia tra 2 utenti il cui identificativo viene passato come parametro della funzione.

Prima di procedere all'inserimento è necessario verificare che entrambi gli utenti esistano ed in caso contrario stampare a video un messaggio di errore, verrà stampato un errore anche se l'amicizia fra i 2 fosse già stata inserita, per capire ciò bisogna scorrere la lista d'adiacenza di amici di uno dei due e cercare se l'altro è presente, essendo l'amicizia una relazione simmetrica non è necessario visitare la lista di amici anche dell'altro.

Se invece l'amicizia non vi è ancora viene richiesto l'anno di creazione di essa e successivamente verranno create 2 strutture "amico" contenenti tale valore sul campo "anno" (u(friend(anno))), l'altro amico in corrispondenza del puntatore ad un utente "friend" (u(friend(friend))) ed un puntatore al primo elemento della lista di amici e quindi conseguentemente vi sarà l'aggiornamento del puntatore al primo elemento della lista di amici presente sulle strutture "utente" ricavate dal search.

Tale aggiunta in testa alla lista permette di svolgere questa operazione in un tempo O(1), ma il costo complessivo della funzione rimane O(n) perché nel peggiore dei casi un utente è amico diretto di tutti gli n utenti e quindi per determinare se l'amicizia sia già presente bisogna scorrere una lista d'adiacenza di n amici.

2.4)Determinare i gruppi di amici

```
1-  void allgroups(grafo G){
2-        aux=alloca lo spazio per un array di interi di n elementi (n=contatore_ut)
3-        j=0;
4-        h=G(head)
5-        for(i = 0; i < G(conta); i++) {
6-              if(aux[h(pos)] != 1) {
7-                    aux[h(pos)]=1;
8-                    stampa("Gruppo  j");
9-                    j=j+1;
10-                   stampa("h(nome) h(cognome) h(id)");
11-                   bfs(h,aux);
12-             }
13-             h=h(next);
14-       }
15-       dealloca  lo spazio per l'array ausiliario
16- }
```

```
1-  void bfs(utente *u, int *aux){
2-       z=alloco lo spazio per un nodo
3-       intqueue *q = createqueue();
4-       enqueue(q,u);
5-       while(!emptyq(q)) {
6-            t = dequeue(q);
7-            aux[t(friend (pos))] = 1;
8-            for(z=t ; z!=NULL; z=z(next)){
9-                 if(aux[z(friend(pos))] != 1){
10-                     enqueue(q,z(friend));
11-                     stampa("z(friend(nome)) z(friend(cognome)) z(friend(id))")
12-                     aux[z(friend(pos))] = 1
13-                     }
14-           }
15-      }
16-      destroyqueue(q);
17- }
```

Queste 2 funzioni lavorano congiuntamente per determinare le relazioni di amicizia esistenti, ovvero, restituire dei sottoinsiemi di utenti che risultano in relazione d'amicizia diretta o indiretta.

Alla funzione principale, ovvero "allgroups", viene passato un grafo contenente un puntatore alla testa della lista d'adiacenza di utenti ed il numero di utenti.

Nell'algoritmo si fa uso di un array ausiliario di interi inizializzato al numero di utenti presenti nell'albero, questo vettore avrà il compito di marchiare gli utenti già considerati, per evitare che vengano visualizzati più di una volta ed impedire che la coda non si svuoti mai e quindi la funzione bfs entri in un loop infinito.

Ogni utente ha la sua corrispondente posizione nel vettore perché, al momento del loro inserimento, gli vengono assegnati un valore crescente, partendo da zero.

A riga 5 di allgroups vi è un ciclo con lo scopo di arrestarsi quando sono certo di aver controllato ogni utente nella lista d'adiacenza, ovvero, il valore "i" raggiunge il numero di utenti.

A riga 6 viene controllato che l'utente non sia già stato considerato e, se è vero, marchio quel nodo come già visto, definisco il primo gruppo, mando a video i suoi dati ed eseguo la visita per ampiezza dei suoi amici attraverso la funzione bfs (breadth-first search).

Quest'ultima funzione riceve in ingresso l'utente su cui effettuare la visita ed il vettore ausiliario.

Successivamente è prevista la creazione di una coda che conterrà strutture di tipo "amico" con la funzione "createqueue" di riga 3 e l'inserimento in essa degli amici dell'utente passato, con la funzione "enqueue".

Il ciclo a riga 5 terminerà solo quando la funzione "emptyq" darà esito positivo, ovvero, restituirà un booleano TRUE che indicherà che la coda è vuota e quindi ho finito di elencare un gruppo di amici.

Se la precedente condizione ha dato esito negativo allora verrà prelevato l'elemento più vecchio della lista, verrà marchiato e visitata la sua lista di amici per mezzo del ciclo for.

Quando viene visitato un amico ancora non marchiato anche la sua lista d'amici andrà aggiunta in coda.

Bisognerà ripetere il ciclo da riga 5 finché la coda conterrà almeno un elemento.

Alla fine sarà possibile distruggere la coda per liberare spazio in memoria, ritornare alla funzione "allgroups" e passare al prossimo elemento della lista d'adiacenza di amici tramite il campo "next" e ripetere la procedura da riga 5.

Al termine dell'algoritmo verranno mostrati a video i gruppi di amici, se un utente non aveva amici allora sarà inserito in un gruppo in cui lui è l'unico elemento.

Questa funzione ha un costo dell'ordine di O(|V | + |E|), cioè della somma del numero di vertici per il numero di archi.

2.5)Determinare i gruppi di amici nati dall'anno "year"

```
1-  void allgroupsyear( grafo G, int year ){
2-        aux=alloca lo spazio per un array di interi di n elementi (n=contatore_ut)
3-        j=0;
4-        h=G(head)
5-        for(i = 0; i < G(conta); i++) {
6-              if(aux[h(pos)] != 1) {
7-                    aux[h(pos)]=1;
8-                    stampa("Gruppo  j");
9-                    j=j+1;
10-                   stampa("h(nome) h(cognome) h(id)")
11-                   bfsy(h, aux, year);
12-             }
13-             h=h(next);
14-       }
15-       dealloca  lo spazio per l'array ausiliario
16  }
```

```
1-  void bfsy(utente *u, int *aux, int year){
2-       z=alloco lo spazio per un nodo
3-       intqueue *q = createqueue();
4-       enqueue(q,u);
5-       while(!emptyq(q)) {
6-            t = dequeue(q);
7-            aux[t(friend (pos))] = 1;
8-            for(z=t ; z!=NULL; z=z(next)){
9-                 if(aux[z(friend(pos))] != 1 && z(anno)>=year){
10-                     enqueue(q,z(friend));
11-                     stampa("z(friend(nome)) z(friend(cognome)) z(friend(id))")
12-                     aux[z(friend(pos))] = 1
13-                     }
14-           }
15-      }
16-      destroyqueue(q);
17- }
```

Questo algoritmo funziona similmente a quello della visita per ampiezza del punto 2.4, differisce per il fatto che in un gruppo vengono aggiunti gli amici solo se tale relazione è nata dopo l'anno passato per argomento alla funzione, quindi è stata ampliata la condizione di riga 9 della funzione bfsy.

Se una persona non ha amicizie che rispettano tale vincolo allora sarà inserita in un gruppo contenente solo lei.

Questa funzione ha un costo dell'ordine di O(|V | + |E|), cioè della somma del numero di vertici per il numero di archi.

Considerazioni finali

La maggior parte delle operazioni presenti richiedono un tempo logaritmico o costante, grazie alla presenza di numerosi puntatori che non richiedono ulteriori ricerche, pur utilizzando maggior memoria permettono di effettuare le operazioni in un tempo lineare.

Ho scelto di inserire i nuovi utenti in fondo alla lista di adiacenza di tali e non in cima perché in tale modo è possibile visualizzare facilmente l'ordine di creazione di utenti nel tempo ed eventualmente eliminare quei profili che non vengono più utilizzati e liberare spazio in memoria.

Non è possibile usare il puntatore "last" per far ciò perché le strutture utente non possiedono un puntatore al precedente elemento.

Un possibile miglioramento del programma consisterebbe nel diminuire il costo della funzione "friendship", la quale ha un costo dell'ordine di O(n), con n pari al numero di utenti.

Si potrebbe sostituire la lista d'adiacenza di amici degli utenti con un albero2_3 i cui nodi sarebbero strutture "amico", inserite sulla base del campo "pos" oppure "identificatore" dei suoi utenti.

Tale procedura aumenterebbe lo spazio occupato ma le procedure di ricerca ed inserimento avrebbero entrambe un tempo logaritmico anche nel caso peggiore.

Inoltre potrei utilizzare 2 tipi di nodi, uno per i nodi interni ed uno per le foglie, cosicché il puntatore all'utente, nullo, dei nodi interni non esista e non vi sia spreco di memoria.