Storage and Traversal of a Genelogy

Akshit Kumar (EE14B127)

30th November 2016

Abstract

This report contains the code, data structure and algorithm for storing and traversing a genelogical tree. A genelogy is a directed graph connecting parents to their children. The genelogy in the question assumes that that people don't have children with each other but only with others outside the genelogy. So each node in our genelogy can trace it's descent from the original ancestor via a unique path.

1 Data Structure to store the information in the Genelogy

1.1 Usage of Binary Tree

Usage of a Binary Tree to store the information in the genelogy is not an appropriate one because binary tree assumes that each node has atmost 2 child nodes which is not necessarily true for a genelogy and hence is only appropriate for special conditions.

1.2 Usage of 2-3 Tree

Again usage of 2-3 tree is not appropriate as it assumes that each node will have 2 or 3 child nodes, which it not necessarily true and is only appropriate from special conditions

1.3 Usage of N-ary Tree

N-ary Tree represents a generic tree data structure and is the most apt in this scenario as we are reading in a general tree. The beauty of the N-ary tree data structure lies in the structure of the Node used to store all the connectios and relevant information.

1.3.1 struct Connections

```
typedef struct Connections{
int parent_index;
char name[20];
int parent_age;
int children_index[10];
int num_children;
}Connections;
```

This Connections structure holds all the information relevant for each node. It contains the index of the parent node, indices of the children node, parent age, number of children and the node name. The N-ary tree structure is represented by an array of Connections structure. The entire genelogy is represented as an array of Connections structure.

1.3.2 Algorithm For Storing the Graph

The graph is represented as an array of structure Connections mentioned above. The algorithm for making the connections is as follows

Initialize the *Connections array* with *parent index as -1* and *number of children to 0*

Iterate through the array of Nodes

For each node in the array of nodes, *iterate* through the *edge connections*

Add revelant children indices for each node in *Connections structure*
Repeat the same for parent index

1.3.3 Implementation for making the connections

```
// Helper function to make all the connections - ie connect each node to
void make_connections(){
// Initialising the nodes
for(int i = 0; i < node_count; i++){
strcpy(connections[i].name,nodes[i].name);
connections[i].parent_index = -1; // setting parent index as -1
connections[i].num_children = 0; // setting the number of children as 0
}
// Make parent connections
// Iterate through all the nodes
for(int i = 0; i < node_count; i++){
int t = 0;
// For each node, iterate through all the edge connections
// If the node is the first name, then assign its index to all it's child
for(int j = 0; j < edge_count; j++){
if(strcmp(connections[i].name,edges[j].parent_name) == 0){
connections[i].num_children++;
connections[i].children_index[t++] = find_index_by_name(edges[j].child_name
}
}
}
// Make children connections
// Iterate through all the nodes
for(int i = 0; i < node_count; i++){
// For each node, iterate through all the edge connections
// If the node is second name, get the parent index and parent age
for(int j = 0; j < edge_count; j++){
if(strcmp(connections[i].name,edges[j].child_name) == 0){
connections[i].parent_index = find_index_by_name(edges[j].parent_name);
connections[i].parent_age = edges[j].parent_age;
}
}
}
}
```

1.3.4 Helper function used to making the connections

```
// Helper function to find the index corresponding to a name
int find_index_by_name(char name[20]){
```

```
int i = 0;
while(i < node_count){
if(strcmp(nodes[i++].name,name) == 0)
break;
}
return i-1;
}
```

1.4 Determining the Original Ancestor

The Original Ancestor of the genelogy is *James.*

1.4.1 Algorithm for determining the original ancestor

Iterate through all the Nodes in the Connections array and *find* the node with *parent_index = 1* and *num_children != 0*

1.4.2 Implementation for finding the original ancestor

```
/*
 * This is the function for finding the original ancestor
 * The algorithm is pretty straight forward.
 * Iterate through the Connections array and find a node which doesn't po
 * Return that index to find the name of the node
 */
int find_ancestor(){
int ans;
for(int i = 0; i < node_count; i++){
if(connections[i].parent_index == -1 && connections[i].num_children != 0){
ans = i;
break;
}
}
return ans;
}
```

2 Descendants for each person

2.1 Algorithm for determining the descendant of each person

The algorithm used for determining the descendant is a *Depth First Search*, which has been implemented iteratively. The algorithm is as follows:

```
push the node to stack
initialize the visited array
while the stack is not empty
```

```
    pop an element
    mark the element visited
    for each child of the element
        if the child is not visited
            push the child to stack
return count of visited node - 1
```

2.2 Time Complexity of the Algorithm

Running DFS gives a time complexity of $O(n)$ where n is the number of the descendants below. Let k be the number of generations below the node, then we can write k as a logarithmic function of n to the base m where m is the average number of people in each generation, therefore the time complexity of running the Depth First Search algorithm is $O(m^k)$. Therefore this algorithm is exponential in terms of number of generations.

2.3 Implementation of the Algorithm

```
/*
 * This is the function for finding the descendants of a given index
 * The algorithm for finding the descendants is a Depth First Search retu
 */
int count_descendants(int index){
// adding the node to the stack
push(index);
int gen_count = 0;
// initialise the visited array
bool visited[100] = {false};
// while the stack is not empty
while(top != -1){
int ele = pop(); // pop the top most element
visited[ele] = true; // mark it visited
// for all the children of that node, which are not visited, add them to
for(int i = 0; i < connections[ele].num_children;i++){
if(visited[connections[ele].children_index[i]] != true){
push(connections[ele].children_index[i]);
}
}
}
int count = 0;
// count the number of visited nodes
for(int i = 0; i < node_count; i++){
if(visited[i])
count++;
}
```

```
// return the number of nodes visited except for itself
return count - 1;
}
```

2.4 Solution to the problem

```
Printing the number of descendants
James : 48
Christopher : 9
Ronald : 36
Mary : 0
Lisa : 21
Michelle : 13
John : 6
Daniel : 0
Anthony : 0
Patricia : 11
Nancy : 5
Laura : 3
Robert : 2
Paul : 8
Kevin : 2
Linda : 0
Karen : 3
Sarah : 0
Michael : 2
Mark : 2
Jason : 2
Barbara : 1
Betty : 2
Kimberly : 2
William : 1
Donald : 2
Jeff : 2
Elizabeth : 1
Helen : 0
Deborah : 0
David : 0
George : 0
Jennifer : 0
Sandra : 0
Richard : 0
Kenneth : 0
Maria : 0
Donna : 0
Charles : 0
```

```
Steven : 0
Susan : 0
Carol : 0
Joseph : 0
Edward : 0
Margaret : 0
Ruth : 0
Thomas : 0
Brian : 0
Dorothy : 0
Sharon : 0
```

3 Getting the Great Grand Children

3.1 Algorithm and Code for obtaining the Great Grand Children

```
// Helper function to find the age of great grand father at the time of t
int find_great_grand_father_age_at_birth(int index){
int age = connections[index].parent_age;
// backtrack to its parent
index = connections[index].parent_index;
age += connections[index].parent_age;
// backtrack to its grandparent
index = connections[index].parent_index;
age += connections[index].parent_age;
return age; // return the age of the great grand father
}
// Function to find the great grand children of each node
/*
 The algorithm used is a brute force search of all the children
 There are three levels - children, grand children and great grand childr
 Iterate through all the children and find all the grand children
 Then iterate through all the grand children and find all the great grand
*/
void find_great_grand_children(int index){
int level1[100] = {-1};
int level2[100] = {-1};
int level3[100] = {-1};
int great_grand_children[100];
int level1_count = 0;
int level2_count = 0;
int level3_count = 0;
int ggc_count = 0;
```

```
// iterate through all the children
for(int i = 0; i < connections[index].num_children; i++){
level1[level1_count++] = connections[index].children_index[i];
}
// iterate through all the grand chilren
for(int j = 0; j < level1_count; j++){
for(int i = 0 ; i < connections[level1[j]].num_children;i++){
level2[level2_count++] = connections[level1[j]].children_index[i];
}
}
// iterate through all the great grand children
for(int k = 0; k < level2_count; k++){
for(int l = 0; l < connections[level2[k]].num_children;l++){
level3[level3_count++] = connections[level2[k]].children_index[l];
}
}
// print the great grand children
printf("Great Grand-Children of %s : ", nodes[index].name);
if(level3_count != 0){
for(int t = 0; t < level3_count;t++){
printf("%s ",nodes[level3[t]].name);
}
}
else{
printf("NIL");
}
if(level3_count != 0){
// check if there is overlap in the lives of great grand father and their
for(int t = 0; t < level3_count;t++){
if(find_great_grand_father_age_at_birth(level3[t]) <= nodes[index].age_of_de
great_grand_children[ggc_count++] = level3[t];
}
}
printf("\n%s lived long enough to see : ",nodes[index].name);
if(ggc_count > 0){
for(int i = 0; i < ggc_count; i++){
printf("%s ",nodes[great_grand_children[i]].name);
}
}
else{
printf("NIL");
}
}
printf("\n");
}
```

The algorithm for obtaining the great grand children is a brute force approach, where for each node we find all its children, then we iterate over all the children and find their children and then again we iterate over all the children and find their children. Then we each child we backtrack to their great grand father finding the great grand father's age while backtracking. If the age obtained is less or equal to the age of death of the node, then there was overlap in the lives of the great grand father and it's great grand child.

3.2 Time Complexity of the Algorithm

The algorithm goes 3 generations below and hence doesn't depend on k generations. If we assume that m is the average number of children per node, then the time complexity of this algorithm turns out to be $O(nm^3)$ where n is the number of nodes in the genelogy

3.3 Solution to the Great Grand Children Problem

```
Great Grand-Children of James : Kevin Linda Karen John Daniel Anthony Pat
James lived long enough to see : Kevin Linda John
Great Grand-Children of Christopher : Helen Deborah David George Jennifer
Christopher lived long enough to see : NIL
Great Grand-Children of Ronald : Michael Mark Jason Barbara Betty Kimberl
Ronald lived long enough to see : Michael Mark Jason William Donald
Great Grand-Children of Mary : NIL
Great Grand-Children of Lisa : Sandra Richard Kenneth Maria Donna Charles
Lisa lived long enough to see : NIL
Great Grand-Children of Michelle : Margaret Ruth Thomas Brian Dorothy Sha
Michelle lived long enough to see : NIL
Great Grand-Children of John : NIL
Great Grand-Children of Daniel : NIL
Great Grand-Children of Anthony : NIL
Great Grand-Children of Patricia : NIL
Great Grand-Children of Nancy : NIL
Great Grand-Children of Laura : NIL
Great Grand-Children of Robert : NIL
Great Grand-Children of Paul : NIL
Great Grand-Children of Kevin : NIL
Great Grand-Children of Linda : NIL
Great Grand-Children of Karen : NIL
Great Grand-Children of Sarah : NIL
Great Grand-Children of Michael : NIL
Great Grand-Children of Mark : NIL
Great Grand-Children of Jason : NIL
Great Grand-Children of Barbara : NIL
Great Grand-Children of Betty : NIL
```

```
Great Grand-Children of Kimberly : NIL
Great Grand-Children of William : NIL
Great Grand-Children of Donald : NIL
Great Grand-Children of Jeff : NIL
Great Grand-Children of Elizabeth : NIL
Great Grand-Children of Helen : NIL
Great Grand-Children of Deborah : NIL
Great Grand-Children of David : NIL
Great Grand-Children of George : NIL
Great Grand-Children of Jennifer : NIL
Great Grand-Children of Sandra : NIL
Great Grand-Children of Richard : NIL
Great Grand-Children of Kenneth : NIL
Great Grand-Children of Maria : NIL
Great Grand-Children of Donna : NIL
Great Grand-Children of Charles : NIL
Great Grand-Children of Steven : NIL
Great Grand-Children of Susan : NIL
Great Grand-Children of Carol : NIL
Great Grand-Children of Joseph : NIL
Great Grand-Children of Edward : NIL
Great Grand-Children of Margaret : NIL
Great Grand-Children of Ruth : NIL
Great Grand-Children of Thomas : NIL
Great Grand-Children of Brian : NIL
Great Grand-Children of Dorothy : NIL
Great Grand-Children of Sharon : NIL
```

James and *Ronald* lived long enough to see some of their great grand children.

4 Output of the Program

```
Original Ancestor : James
Printing the number of descendants
James : 48
Christopher : 9
Ronald : 36
Mary : 0
Lisa : 21
Michelle : 13
John : 6
Daniel : 0
Anthony : 0
Patricia : 11
Nancy : 5
```

```
Laura : 3
Robert : 2
Paul : 8
Kevin : 2
Linda : 0
Karen : 3
Sarah : 0
Michael : 2
Mark : 2
Jason : 2
Barbara : 1
Betty : 2
Kimberly : 2
William : 1
Donald : 2
Jeff : 2
Elizabeth : 1
Helen : 0
Deborah : 0
David : 0
George : 0
Jennifer : 0
Sandra : 0
Richard : 0
Kenneth : 0
Maria : 0
Donna : 0
Charles : 0
Steven : 0
Susan : 0
Carol : 0
Joseph : 0
Edward : 0
Margaret : 0
Ruth : 0
Thomas : 0
Brian : 0
Dorothy : 0
Sharon : 0
Great Grand-Children of James : Kevin Linda Karen John Daniel Anthony Pat
James lived long enough to see : Kevin Linda John
Great Grand-Children of Christopher : Helen Deborah David George Jennifer
Christopher lived long enough to see : NIL
Great Grand-Children of Ronald : Michael Mark Jason Barbara Betty Kimberl
Ronald lived long enough to see : Michael Mark Jason William Donald
Great Grand-Children of Mary : NIL
```

```
Great Grand-Children of Lisa : Sandra Richard Kenneth Maria Donna Charles
Lisa lived long enough to see : NIL
Great Grand-Children of Michelle : Margaret Ruth Thomas Brian Dorothy Sha
Michelle lived long enough to see : NIL
Great Grand-Children of John : NIL
Great Grand-Children of Daniel : NIL
Great Grand-Children of Anthony : NIL
Great Grand-Children of Patricia : NIL
Great Grand-Children of Nancy : NIL
Great Grand-Children of Laura : NIL
Great Grand-Children of Robert : NIL
Great Grand-Children of Paul : NIL
Great Grand-Children of Kevin : NIL
Great Grand-Children of Linda : NIL
Great Grand-Children of Karen : NIL
Great Grand-Children of Sarah : NIL
Great Grand-Children of Michael : NIL
Great Grand-Children of Mark : NIL
Great Grand-Children of Jason : NIL
Great Grand-Children of Barbara : NIL
Great Grand-Children of Betty : NIL
Great Grand-Children of Kimberly : NIL
Great Grand-Children of William : NIL
Great Grand-Children of Donald : NIL
Great Grand-Children of Jeff : NIL
Great Grand-Children of Elizabeth : NIL
Great Grand-Children of Helen : NIL
Great Grand-Children of Deborah : NIL
Great Grand-Children of David : NIL
Great Grand-Children of George : NIL
Great Grand-Children of Jennifer : NIL
Great Grand-Children of Sandra : NIL
Great Grand-Children of Richard : NIL
Great Grand-Children of Kenneth : NIL
Great Grand-Children of Maria : NIL
Great Grand-Children of Donna : NIL
Great Grand-Children of Charles : NIL
Great Grand-Children of Steven : NIL
Great Grand-Children of Susan : NIL
Great Grand-Children of Carol : NIL
Great Grand-Children of Joseph : NIL
Great Grand-Children of Edward : NIL
Great Grand-Children of Margaret : NIL
Great Grand-Children of Ruth : NIL
Great Grand-Children of Thomas : NIL
Great Grand-Children of Brian : NIL
```

```
Great Grand-Children of Dorothy : NIL
Great Grand-Children of Sharon : NIL
```

5 Source Code of the Program

```
/*
 * Name : Akshit Kumar
 * Roll No : EE14B127
 * Solution to the genelogy question in the take home endsem examination - finds the original ancestor, no. of descendants and people who lived long enough to see their great grandchildren
 */
// Inclusion of necessary libraries
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <stdbool.h>
#include <string.h>
#include <stdbool.h>

// defining the max size of the stack
#define MAXSIZE 1000

// Definition of Node struct - contains the name and age of death of the person
typedef struct Node{
  char name[20];
  int age_of_death;
}Node;

/*
 * Definition of the Connections struct
 * parent_index : stores the index of parent ie makes an edge connection to the parent
 * name[20] : contains the name of the node
 * parent_age : stores the age of the parent when that node was born
 * num_children : stores the number of children of that node
 * children_index[10] : stores the indices of the children ie makes edge connections to the children
 */
typedef struct Connections{
  int parent_index;
  char name[20];
  int parent_age;
  int children_index[10];
  int num_children;
}Connections;

// Definition of the Edge struct - contains the parent name, child name and parent age at time of child's birth
typedef struct Edge{
  char parent_name[20];
  char child_name[20];
  int parent_age;
}Edge;
```

```c

/* Implementation of stack for performing DFS*/
int stack[MAXSIZE];
int top = -1;

int isempty(){
  if(top == -1){
    return 1;
  }
  else{
    return 0;
  }
}

int isfull(){
  if(top == MAXSIZE){
    return 1;
  }
  else{
    return 0;
  }
}

int peek(){
  return stack[top];
}

int pop(){
  int data;
  if(!isempty()){
    data = stack[top];
    top = top - 1;
  }else{
    printf("Stack empty");
  }
  return data;
}

void push(int data){
  if(!isfull()){
    top = top + 1;
    stack[top] = data;
  }else{
    printf("Stack overflow");
  }
}

/* Array of structs */
Node nodes[100];
Edge edges[100];
Connections connections[100];

int node_count = 0;
int edge_count = 0;

// Helper function to read a line from the file and make a node
void make_node(char line[256]){
```

```
  char name[20];
  char age[20];
  int age_of_death;
  sscanf(line,"%s %s",name,age); // Get the name and age of the person
  if(strcmp(age,"-") == 0){
    age_of_death = INT_MAX; // If the age of death is - , assign is maximum possible value
  }
  else{
    sscanf(age,"%d",&age_of_death);
  }
  strcpy(nodes[node_count].name,name);
  nodes[node_count++].age_of_death = age_of_death;
}

// Helper function to read a line from the file and make an edge
void make_edge(char line[256]){
  char parent_name[20];
  char child_name[20];
  int parent_age;
  sscanf(line,"%s %s %d",parent_name,child_name,&parent_age); // get the name of the parent, child and parent age
  strcpy(edges[edge_count].parent_name,parent_name);
  strcpy(edges[edge_count].child_name,child_name);
  edges[edge_count++].parent_age = parent_age;
}

// Helper function to find the index corresponding to a name
int find_index_by_name(char name[20]){
  int i = 0;
  while(i < node_count){
    if(strcmp(nodes[i++].name,name) == 0)
      break;
  }
  return i-1;
}

// Helper function to make all the connections - ie connect each node to its parent and children
void make_connections(){
  // Initialising the nodes
  for(int i = 0; i < node_count; i++){
    strcpy(connections[i].name,nodes[i].name);
    connections[i].parent_index = -1; // setting parent index as -1
    connections[i].num_children = 0; // setting the number of children as 0
  }
  // Make parent connections
  // Iterate through all the nodes
  for(int i = 0; i < node_count; i++){
    int t = 0;
    // For each node, iterate through all the edge connections
    // If the node is the first name, then assign its index to all it's children nodes
    for(int j = 0; j < edge_count; j++){
      if(strcmp(connections[i].name,edges[j].parent_name) == 0){
        connections[i].num_children++;
```

```
        connections[i].children_index[t++] = find_index_by_name(edges[j].child_name);
      }
    }
  }
  // Make children connections
  // Iterate through all the nodes
  for(int i = 0; i < node_count; i++){
    // For each node, iterate through all the edge connections
    // If the node is second name, get the parent index and parent age
    for(int j = 0; j < edge_count; j++){
      if(strcmp(connections[i].name,edges[j].child_name) == 0){
        connections[i].parent_index = find_index_by_name(edges[j].parent_name);
        connections[i].parent_age = edges[j].parent_age;
      }
    }
  }
}

/*
 * This is the function for finding the original ancestor
 * The algorithm is pretty straight forward.
 * Iterate through the Connections array and find a node which doesn't point to a parent but has non zero children
 * Return that index to find the name of the node
 */
int find_ancestor(){
  int ans;
  for(int i = 0; i < node_count; i++){
    if(connections[i].parent_index == -1 && connections[i].num_children != 0){
      ans = i;
      break;
    }
  }
  return ans;
}

/*
 * This is the function for finding the descendants of a given index
 * The algorithm for finding the descendants is a Depth First Search returning the number of visited nodes
 */
int count_descendants(int index){
  // adding the node to the stack
  push(index);
  int gen_count = 0;
  // initialise the visited array
  bool visited[100] = {false};
  // while the stack is not empty
  while(top != -1){
    int ele = pop(); // pop the top most element
    visited[ele] = true; // mark it visited
    // for all the children of that node, which are not visited, add them to the stack
    for(int i = 0; i < connections[ele].num_children;i++){
```

```
      if(visited[connections[ele].children_index[i]] != true){
        push(connections[ele].children_index[i]);
      }
    }
  }
  int count = 0;
  // count the number of visited nodes
  for(int i = 0; i < node_count; i++){
    if(visited[i])
      count++;
  }
  // return the number of nodes visited except for itself
  return count - 1;
}

// Function to print all the descendants
void print_descendants(){
  printf("Printing the number of descendants\n");
  for(int i = 0; i < node_count; i++){
    printf("%s : %d\n",connections[i].name,count_descendants(i));
  }
}

// Helper function to find the age of great grand father at the time of the birth of the particular node
int find_great_grand_father_age_at_birth(int index){
  int age = connections[index].parent_age;
  // backtrack to its parent
  index = connections[index].parent_index;
  age += connections[index].parent_age;
  // backtrack to its grandparent
  index = connections[index].parent_index;
  age += connections[index].parent_age;
  return age; // return the age of the great grand father
}

// Function to find the great grand children of each node
/*
 The algorithm used is a brute force search of all the children
 There are three levels - children, grand children and great grand children
 Iterate through all the children and find all the grand children
 Then iterate through all the grand children and find all the great grand children
*/
void find_great_grand_children(int index){
  int level1[100] = {-1};
  int level2[100] = {-1};
  int level3[100] = {-1};
  int great_grand_children[100];
  int level1_count = 0;
  int level2_count = 0;
  int level3_count = 0;
  int ggc_count = 0;
  // iterate through all the children
  for(int i = 0; i < connections[index].num_children; i++){
    level1[level1_count++] = connections[index].children_index[i];
```

```
  }
  // iterate through all the grand chilren
  for(int j = 0; j < level1_count; j++){
    for(int i = 0 ; i < connections[level1[j]].num_children;i++){
      level2[level2_count++] = connections[level1[j]].children_index[i];
    }
  }
  // iterate through all the great grand children
  for(int k = 0; k < level2_count; k++){
    for(int l = 0; l < connections[level2[k]].num_children;l++){
      level3[level3_count++] = connections[level2[k]].children_index[l];
    }
  }
  // print the great grand children
  printf("Great Grand-Children of %s : ", nodes[index].name);
  if(level3_count != 0){
    for(int t = 0; t < level3_count;t++){
      printf("%s ",nodes[level3[t]].name);
    }
  }
  else{
    printf("NIL");
  }
  if(level3_count != 0){
    // check if there is overlap in the lives of great grand father and their great grand children
    for(int t = 0; t < level3_count;t++){
      if(find_great_grand_father_age_at_birth(level3[t]) <= nodes[index].age_of_death){
        great_grand_children[ggc_count++] = level3[t];
      }
    }
    printf("\n%s lived long enough to see : ",nodes[index].name);
    if(ggc_count > 0){
      for(int i = 0; i < ggc_count; i++){
        printf("%s ",nodes[great_grand_children[i]].name);
      }
    }
    else{
      printf("NIL");
    }
  }
  printf("\n");
}

// Printing the great grand children for all the nodes
void print_great_grand_children(){
  for(int i = 0; i < node_count;i++){
    find_great_grand_children(i);
  }
}

int main(int argc,char** argv){
  if(argc != 2){
    printf("Usage ./a.out <filename>\n");
```

```c
    exit(1);
  }
  FILE *file = fopen(argv[1],"r");
  if(file == NULL){
    printf("Unable to open file\n");
    exit(1);
  }
  char line[128];
  // Reading in the file and making node and edge arrays
  while(fgets(line,sizeof(line),file) != NULL){
    if(line[0] == '#' || line[0] == '\n'){
      continue;
    }
    else{
      char *pos;
      if ((pos=strchr(line, '\n')) != NULL)
        *pos = '\0';
      char str1[20];
      char str2[20];
      char str3[20];
      if(sscanf(line,"%s %s %s",str1,str2,str3) == 2){
        make_node(line);
      }
      else if(sscanf(line,"%s %s %s",str1,str2,str3) == 3){
        make_edge(line);
      }
    }
  }
  make_connections(); // make the connections
  // Printing out the solution to mentioned questions
  printf("Original Ancestor : %s\n",nodes[find_ancestor()].name);
  print_descendants();
  print_great_grand_children();
  return 0;
}
```

graph.c