CE437: Αλγόριθμοι CAD I

Homework 3

Tcl shell’s Implementation

```
C:\Users\Christos Vasileiou> tcl shell
```

By Vasileiou Christos, 1983

Files’ Structure

- customTCL.c: Includes main implemenation.
    - int main(int argc, char *argv[])
- Instructions.h: Includes Tcl instructions in a string array.
    - static char *instructions[]
- functions_1st.c: Includes 1st homework’s functions.
- functions_2nd.c: Includes 2nd homework’s functions.
    - void *commandsCreation();
    - Tcl_ObjCmdProc *cube_intersect_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[])
    - Tcl_ObjCmdProc *supercube_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *distance_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *cube_cover_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *sharp_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *my_sharp (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *sharp (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - int checkIfValid (char *checked, int size);
- Makefile: Linking and Compilation.

Files' Structure

- functions_3rd.c: Includes 3rd homework's functions.
    - void *do_read_graph (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *initIntArray (int x, int y, int array [x][y], int val);
    - void *initInt (int x, int n[x], int val);
    - int searchNodes (int x, int n[x], int n2); // return 1 if n2 is not in nD. //
    - void *printGraph (int x, int graph [x][x]);
    - void *do_write_graph (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *do_draw_graph (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - nodesDist_t find_Shortest_Explored_Node (int x, nodesDist_t *d);
    - int maximum (int a, int b);
    - int graphIsNotEmpty (nodesDist_t *n); // return 0 if n is full. //
    - void sortGraph (int x, nodesDist_t *n);
    - void *do_graph_critical_path (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - int minimum (int a, int b);
    - nodesDist_t *back_trace (nodesDist_t *Q, int arcWeight[size][size], int longest_path, int Rslack, int *previous, int maxDistanceNode, int *slack, nodesDist_t *criticalPath);

Variable’s Structure

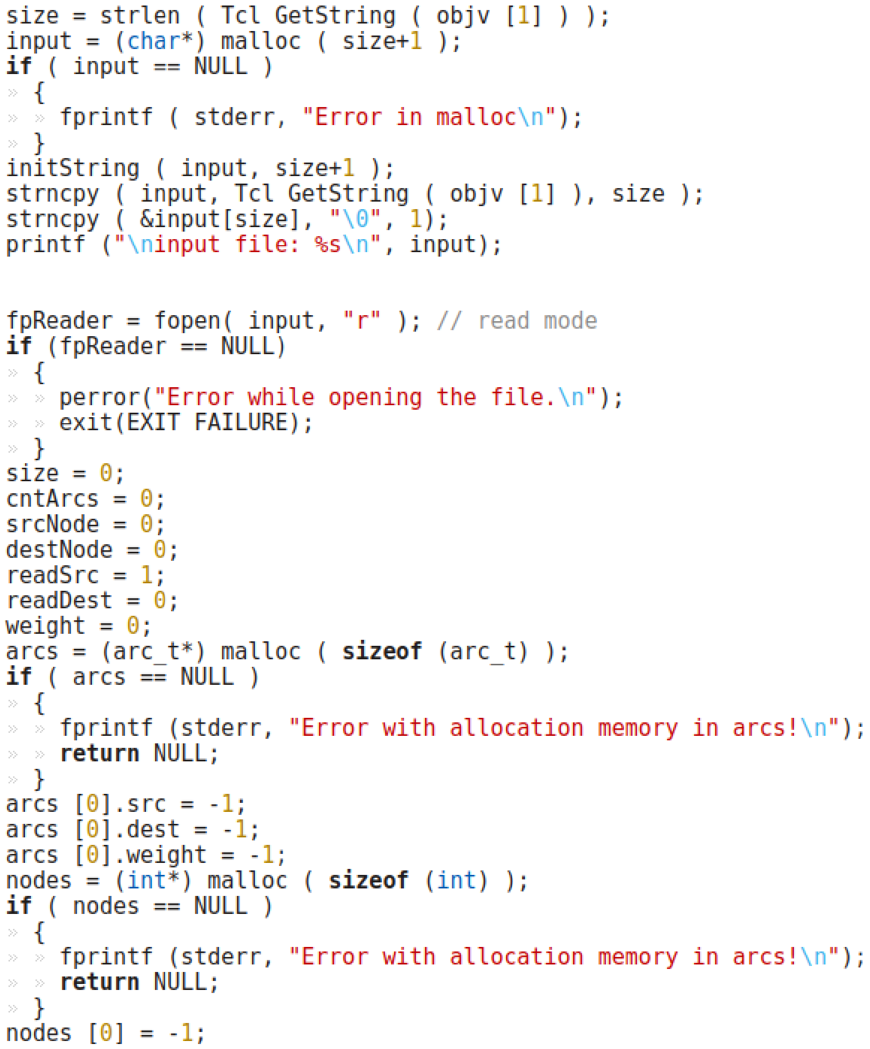
-
  ```
  enum nodeStatus
    {
          UNEXPLORED,
          EXPLORED
    };
  typedef enum nodeStatus nodeStatus_t;
  ```

-
  ```
  struct nodesDist
    {
          int node;
          int dist;
          nodeStatus_t status;
    };
  typedef struct nodesDist nodesDist_t;
  ```

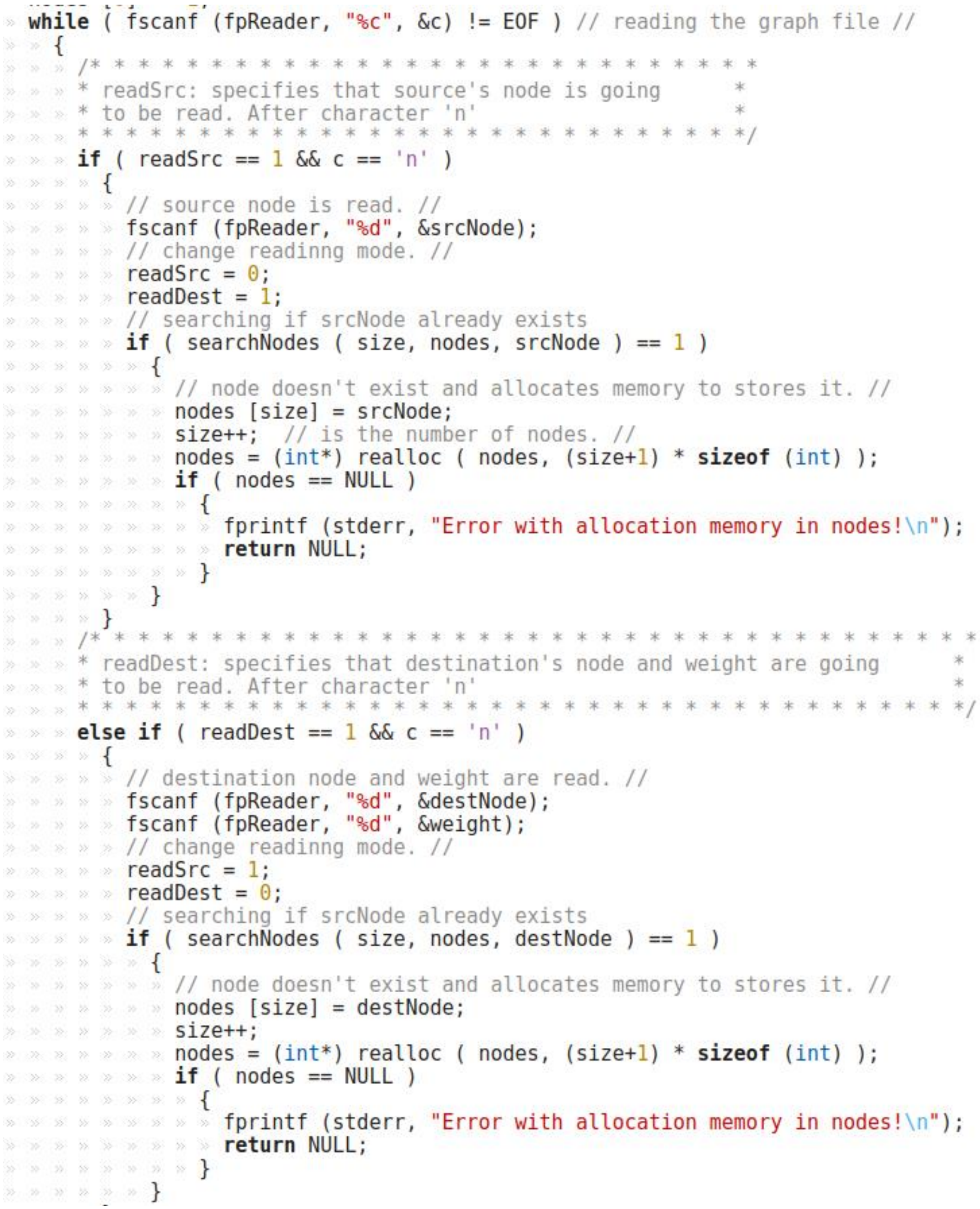
-
  ```
  struct arc
    {
          int src;
          int dest;
          int weight;
    };
  typedef struct arc arc_t;
  ```

```
arc_t *arcs;   // all graph's arcs.  //
int cntArcs;   // number of arcs.  //
int size;      // number of nodes.  //
```

Read_graph

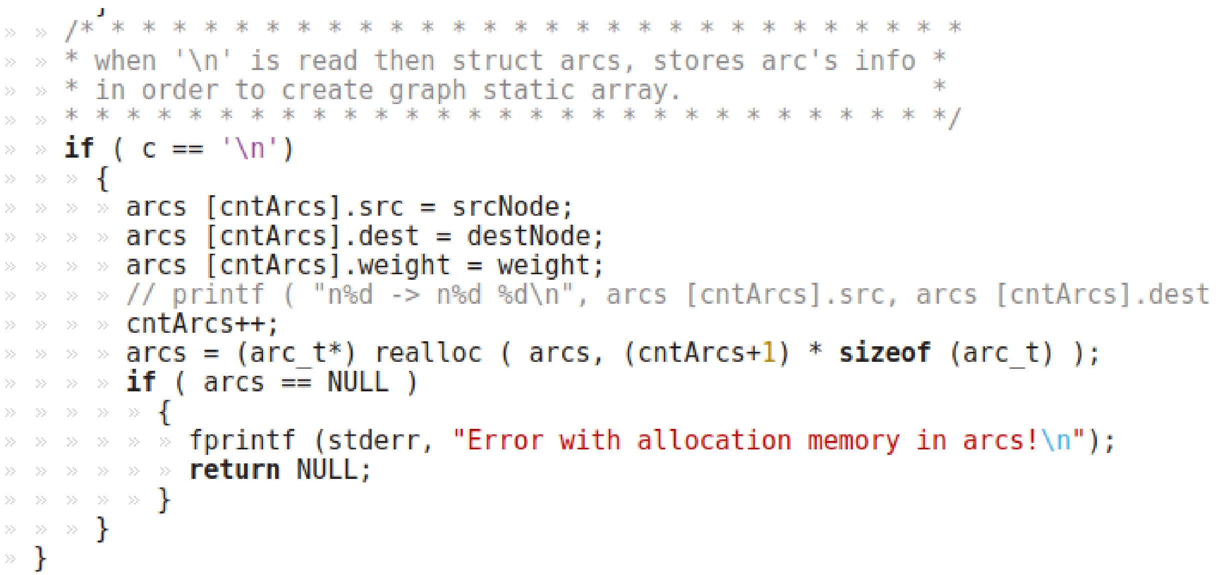


```c
size = strlen ( Tcl GetString ( objv [1] ) );
input = (char*) malloc ( size+1 );
if ( input == NULL )
    {
        fprintf ( stderr, "Error in malloc\n");
    }
initString ( input, size+1 );
strncpy ( input, Tcl GetString ( objv [1] ), size );
strncpy ( &input[size], "\0", 1);
printf ("\ninput file: %s\n", input);


fpReader = fopen( input, "r" ); // read mode
if (fpReader == NULL)
    {
        perror("Error while opening the file.\n");
        exit(EXIT FAILURE);
    }
size = 0;
cntArcs = 0;
srcNode = 0;
destNode = 0;
readSrc = 1;
readDest = 0;
weight = 0;
arcs = (arc_t*) malloc ( sizeof (arc_t) );
if ( arcs == NULL )
    {
        fprintf (stderr, "Error with allocation memory in arcs!\n");
        return NULL;
    }
arcs [0].src = -1;
arcs [0].dest = -1;
arcs [0].weight = -1;
nodes = (int*) malloc ( sizeof (int) );
if ( nodes == NULL )
    {
        fprintf (stderr, "Error with allocation memory in arcs!\n");
        return NULL;
    }
nodes [0] = -1;
```

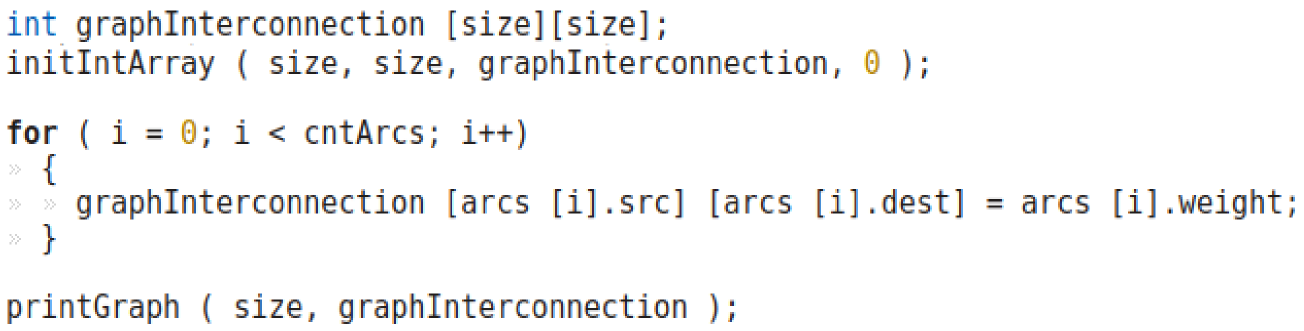


```c
while ( fscanf (fpReader, "%c", &c) != EOF ) // reading the graph file //
    {
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         * readSrc: specifies that source's node is going          *
         * to be read. After character 'n'                         *
         * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        if ( readSrc == 1 && c == 'n' )
            {
                // source node is read. //
                fscanf (fpReader, "%d", &srcNode);
                // change readinng mode. //
                readSrc = 0;
                readDest = 1;
                // searching if srcNode already exists
                if ( searchNodes ( size, nodes, srcNode ) == 1 )
                    {
                        // node doesn't exist and allocates memory to stores it. //
                        nodes [size] = srcNode;
                        size++;  // is the number of nodes. //
                        nodes = (int*) realloc ( nodes, (size+1) * sizeof (int) );
                        if ( nodes == NULL )
                            {
                                fprintf (stderr, "Error with allocation memory in nodes!\n");
                                return NULL;
                            }
                    }
            }
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         * readDest: specifies that destination's node and weight are going        *
         * to be read. After character 'n'                                         *
         * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        else if ( readDest == 1 && c == 'n' )
            {
                // destination node and weight are read. //
                fscanf (fpReader, "%d", &destNode);
                fscanf (fpReader, "%d", &weight);
                // change readinng mode. //
                readSrc = 1;
                readDest = 0;
                // searching if srcNode already exists
                if ( searchNodes ( size, nodes, destNode ) == 1 )
                    {
                        // node doesn't exist and allocates memory to stores it. //
                        nodes [size] = destNode;
                        size++;
                        nodes = (int*) realloc ( nodes, (size+1) * sizeof (int) );
                        if ( nodes == NULL )
                            {
                                fprintf (stderr, "Error with allocation memory in nodes!\n");
                                return NULL;
                            }
                    }
```

Read_graph

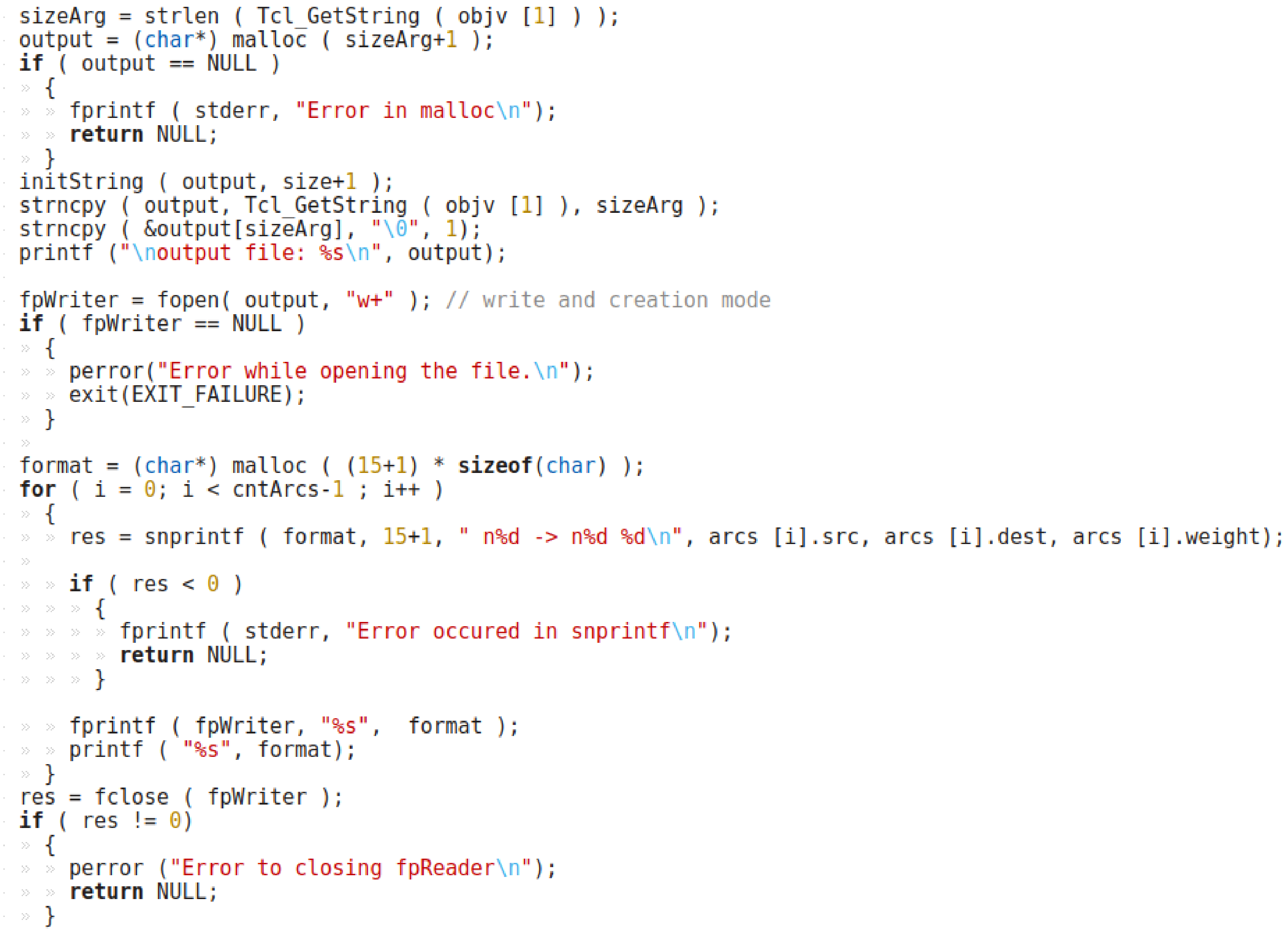


```
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        * when '\n' is read then struct arcs, stores arc's info *
        * in order to create graph static array.                 *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        if ( c == '\n')
            {
                arcs [cntArcs].src = srcNode;
                arcs [cntArcs].dest = destNode;
                arcs [cntArcs].weight = weight;
                // printf ( "n%d -> n%d %d\n", arcs [cntArcs].src, arcs [cntArcs].dest
                cntArcs++;
                arcs = (arc_t*) realloc ( arcs, (cntArcs+1) * sizeof (arc_t) );
                if ( arcs == NULL )
                    {
                        fprintf (stderr, "Error with allocation memory in arcs!\n");
                        return NULL;
                    }
            }
    }
```



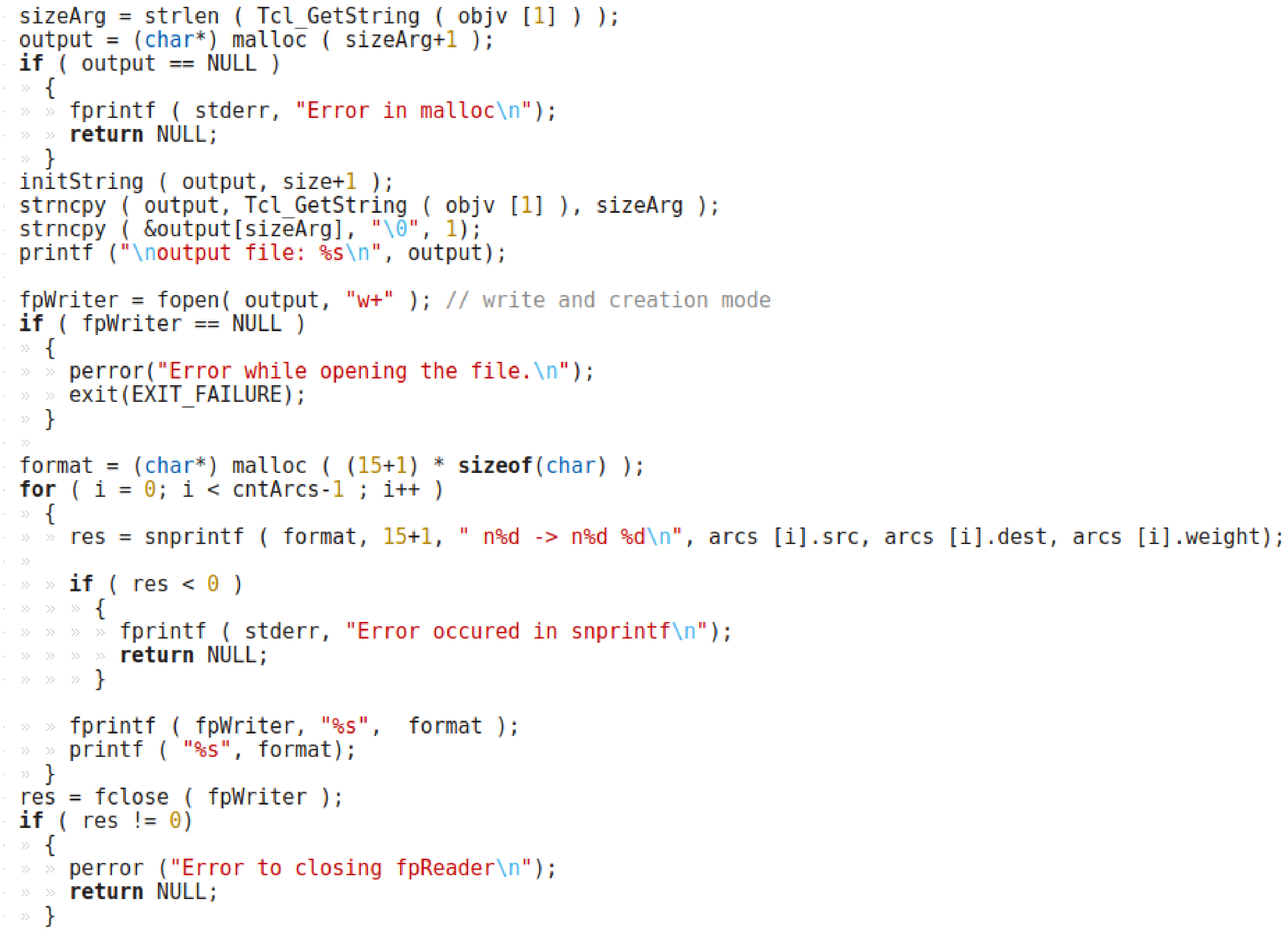
```
int graphInterconnection [size][size];
initIntArray ( size, size, graphInterconnection, 0 );

for ( i = 0; i < cntArcs; i++)
    {
        graphInterconnection [arcs [i].src] [arcs [i].dest] = arcs [i].weight;
    }

printGraph ( size, graphInterconnection );
```

Write_graph



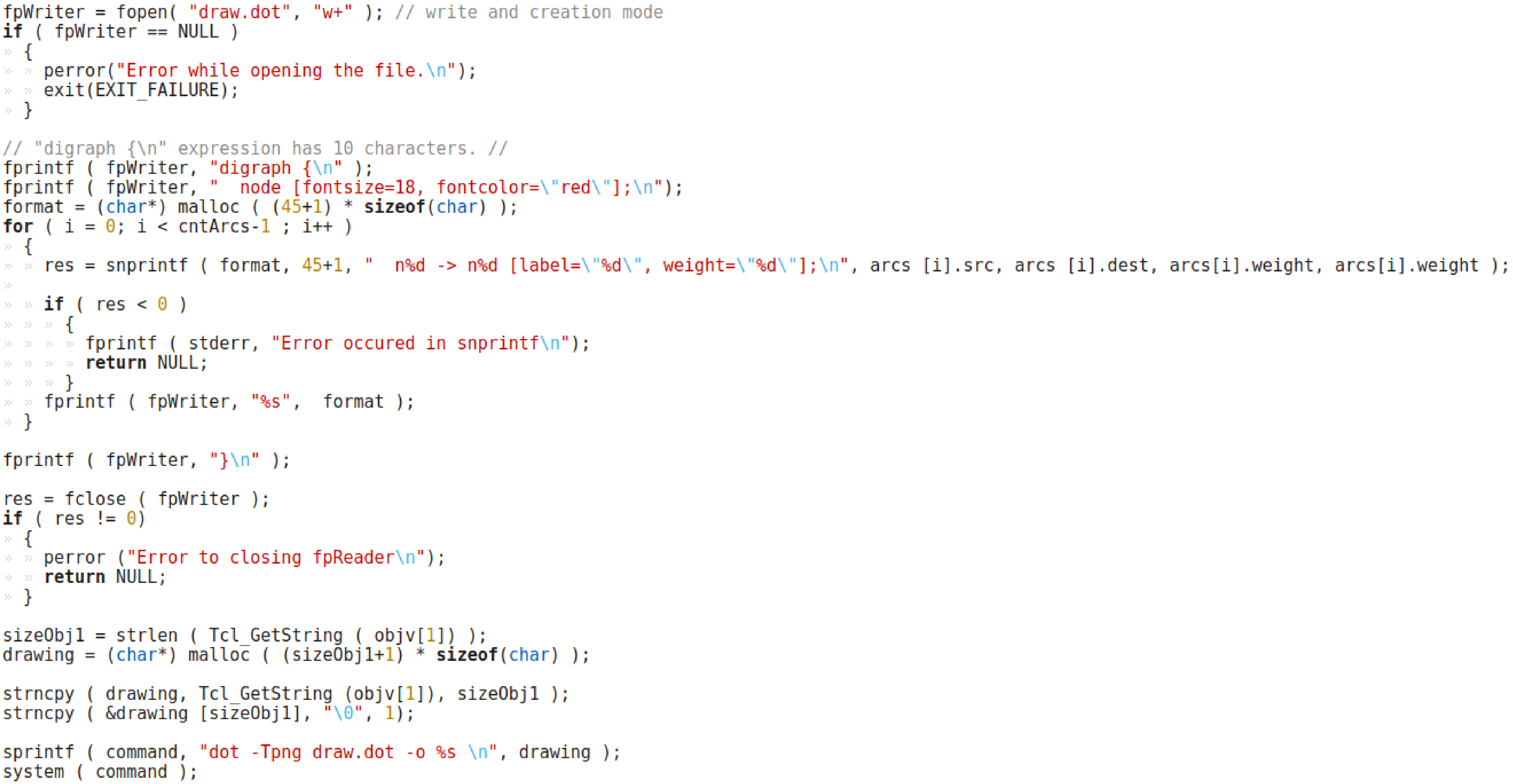
```c
sizeArg = strlen ( Tcl_GetString ( objv [1] ) );
output = (char*) malloc ( sizeArg+1 );
if ( output == NULL )
    {
        fprintf ( stderr, "Error in malloc\n");
        return NULL;
    }
initString ( output, size+1 );
strncpy ( output, Tcl_GetString ( objv [1] ), sizeArg );
strncpy ( &output[sizeArg], "\0", 1);
printf ("\noutput file: %s\n", output);

fpWriter = fopen( output, "w+" ); // write and creation mode
if ( fpWriter == NULL )
    {
        perror("Error while opening the file.\n");
        exit(EXIT_FAILURE);
    }

format = (char*) malloc ( (15+1) * sizeof(char) );
for ( i = 0; i < cntArcs-1 ; i++ )
    {
        res = snprintf ( format, 15+1, " n%d -> n%d %d\n", arcs [i].src, arcs [i].dest, arcs [i].weight);

        if ( res < 0 )
            {
                fprintf ( stderr, "Error occured in snprintf\n");
                return NULL;
            }

        fprintf ( fpWriter, "%s",  format );
        printf ( "%s", format);
    }
res = fclose ( fpWriter );
if ( res != 0)
    {
        perror ("Error to closing fpReader\n");
        return NULL;
    }
```

Write_graph



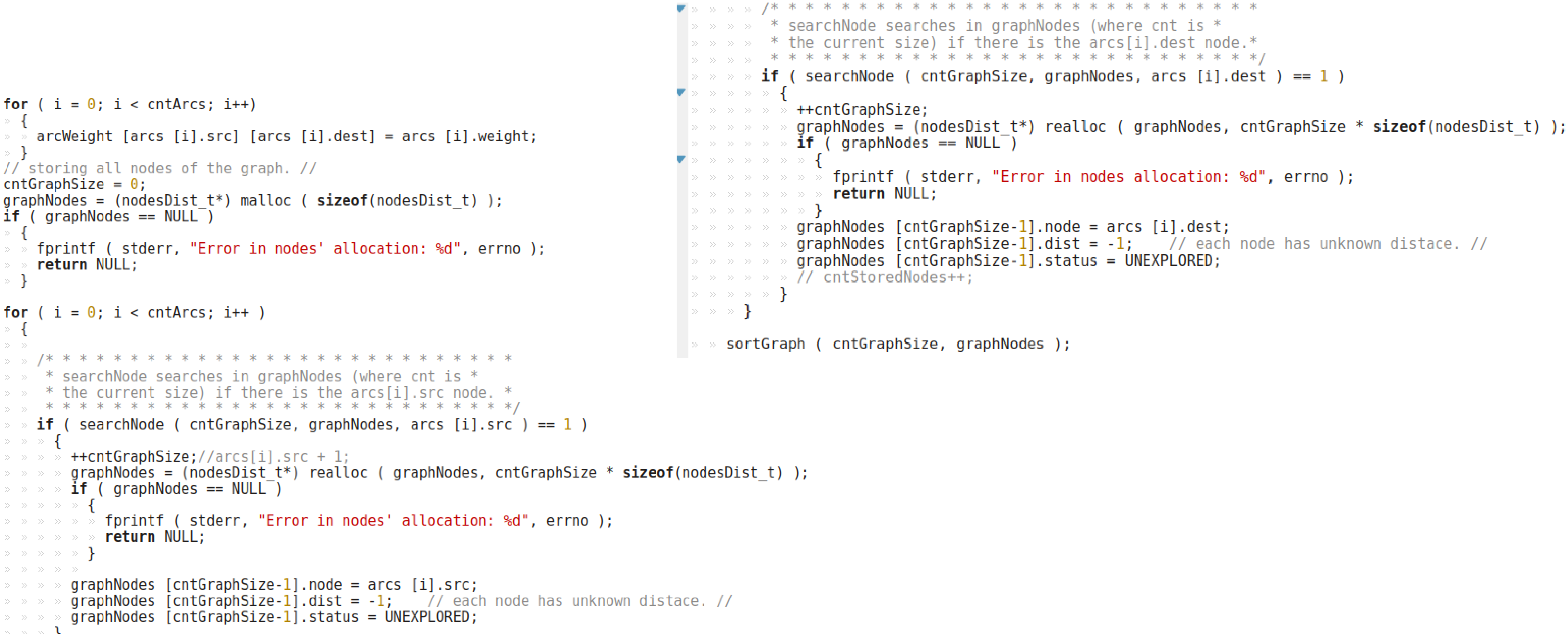
```
sizeArg = strlen ( Tcl_GetString ( objv [1] ) );
output = (char*) malloc ( sizeArg+1 );
if ( output == NULL )
    {
        fprintf ( stderr, "Error in malloc\n");
        return NULL;
    }
initString ( output, size+1 );
strncpy ( output, Tcl_GetString ( objv [1] ), sizeArg );
strncpy ( &output[sizeArg], "\0", 1);
printf ("\noutput file: %s\n", output);

fpWriter = fopen( output, "w+" ); // write and creation mode
if ( fpWriter == NULL )
    {
        perror("Error while opening the file.\n");
        exit(EXIT_FAILURE);
    }

format = (char*) malloc ( (15+1) * sizeof(char) );
for ( i = 0; i < cntArcs-1 ; i++ )
    {
        res = snprintf ( format, 15+1, " n%d -> n%d %d\n", arcs [i].src, arcs [i].dest, arcs [i].weight);

        if ( res < 0 )
            {
                fprintf ( stderr, "Error occured in snprintf\n");
                return NULL;
            }

        fprintf ( fpWriter, "%s",  format );
        printf ( "%s", format);
    }
res = fclose ( fpWriter );
if ( res != 0)
    {
        perror ("Error to closing fpReader\n");
        return NULL;
    }
```

Draw_graph



```
fpWriter = fopen( "draw.dot", "w+" ); // write and creation mode
if ( fpWriter == NULL )
    {
        perror("Error while opening the file.\n");
        exit(EXIT_FAILURE);
    }

// "digraph {\n" expression has 10 characters. //
fprintf ( fpWriter, "digraph {\n" );
fprintf ( fpWriter, "  node [fontsize=18, fontcolor=\"red\"];\n");
format = (char*) malloc ( (45+1) * sizeof(char) );
for ( i = 0; i < cntArcs-1 ; i++ )
    {
        res = snprintf ( format, 45+1, "  n%d -> n%d [label=\"%d\", weight=\"%d\"];\n", arcs [i].src, arcs [i].dest, arcs[i].weight, arcs[i].weight );

        if ( res < 0 )
            {
                fprintf ( stderr, "Error occured in snprintf\n");
                return NULL;
            }
        fprintf ( fpWriter, "%s",  format );
    }

fprintf ( fpWriter, "}\n" );

res = fclose ( fpWriter );
if ( res != 0)
    {
        perror ("Error to closing fpReader\n");
        return NULL;
    }

sizeObj1 = strlen ( Tcl_GetString ( objv[1]) );
drawing = (char*) malloc ( (sizeObj1+1) * sizeof(char) );

strncpy ( drawing, Tcl_GetString (objv[1]), sizeObj1 );
strncpy ( &drawing [sizeObj1], "\0", 1);

sprintf ( command, "dot -Tpng draw.dot -o %s \n", drawing );
system ( command );
```

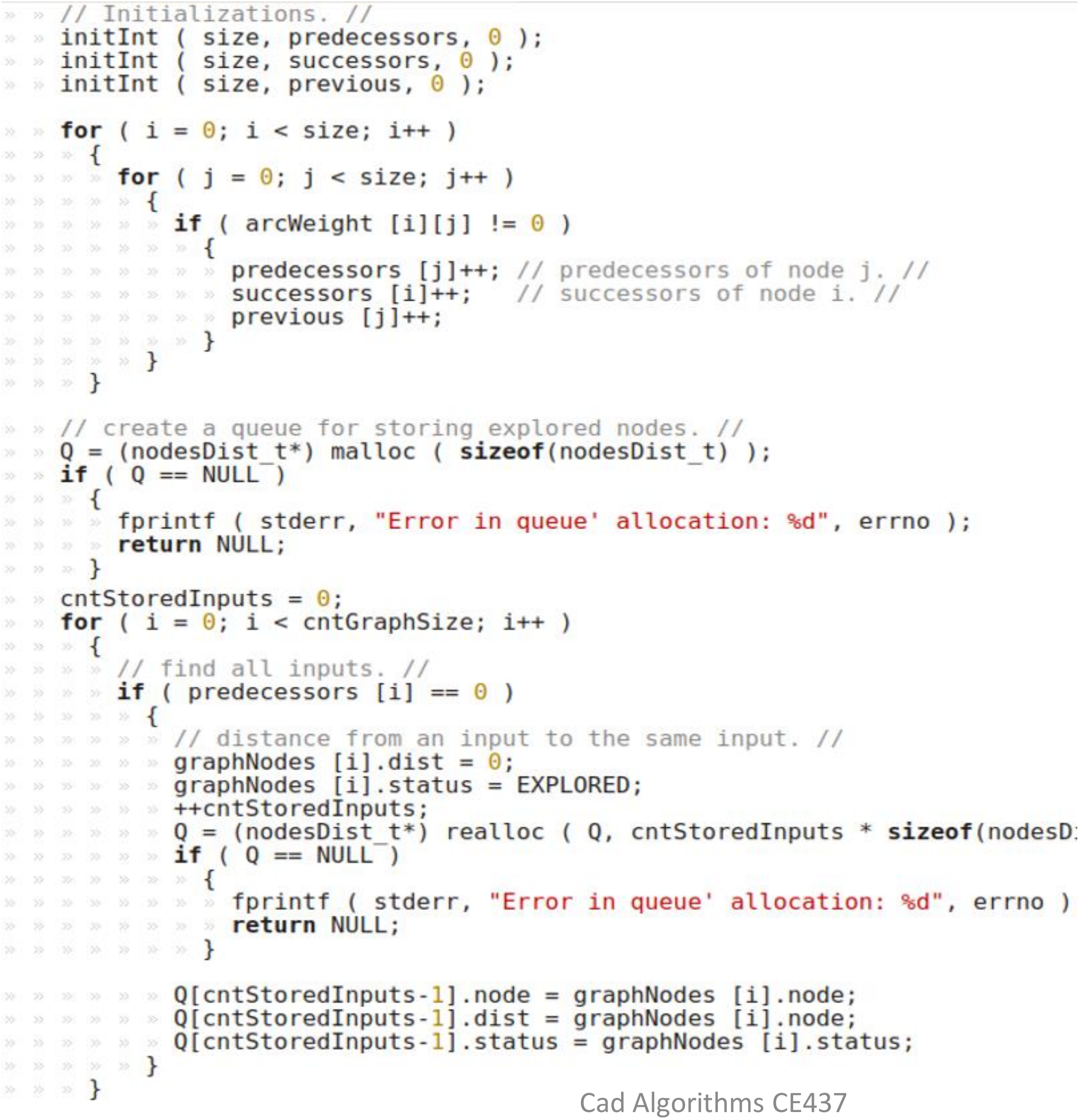
Graph_critical_path



```
for ( i = 0; i < cntArcs; i++)
  {
    arcWeight [arcs [i].src] [arcs [i].dest] = arcs [i].weight;
  }
// storing all nodes of the graph. //
cntGraphSize = 0;
graphNodes = (nodesDist_t*) malloc ( sizeof(nodesDist_t) );
if ( graphNodes == NULL )
  {
    fprintf ( stderr, "Error in nodes' allocation: %d", errno );
    return NULL;
  }

for ( i = 0; i < cntArcs; i++ )
  {

    /* * * * * * * * * * * * * * * * * * * * * * * * * * *
     * searchNode searches in graphNodes (where cnt is *
     * the current size) if there is the arcs[i].src node. *
     * * * * * * * * * * * * * * * * * * * * * * * * * * * */
    if ( searchNode ( cntGraphSize, graphNodes, arcs [i].src ) == 1 )
      {
        ++cntGraphSize;//arcs[i].src + 1;
        graphNodes = (nodesDist_t*) realloc ( graphNodes, cntGraphSize * sizeof(nodesDist_t) );
        if ( graphNodes == NULL )
          {
            fprintf ( stderr, "Error in nodes' allocation: %d", errno );
            return NULL;
          }

        graphNodes [cntGraphSize-1].node = arcs [i].src;
        graphNodes [cntGraphSize-1].dist = -1;    // each node has unknown distace. //
        graphNodes [cntGraphSize-1].status = UNEXPLORED;
      }
```

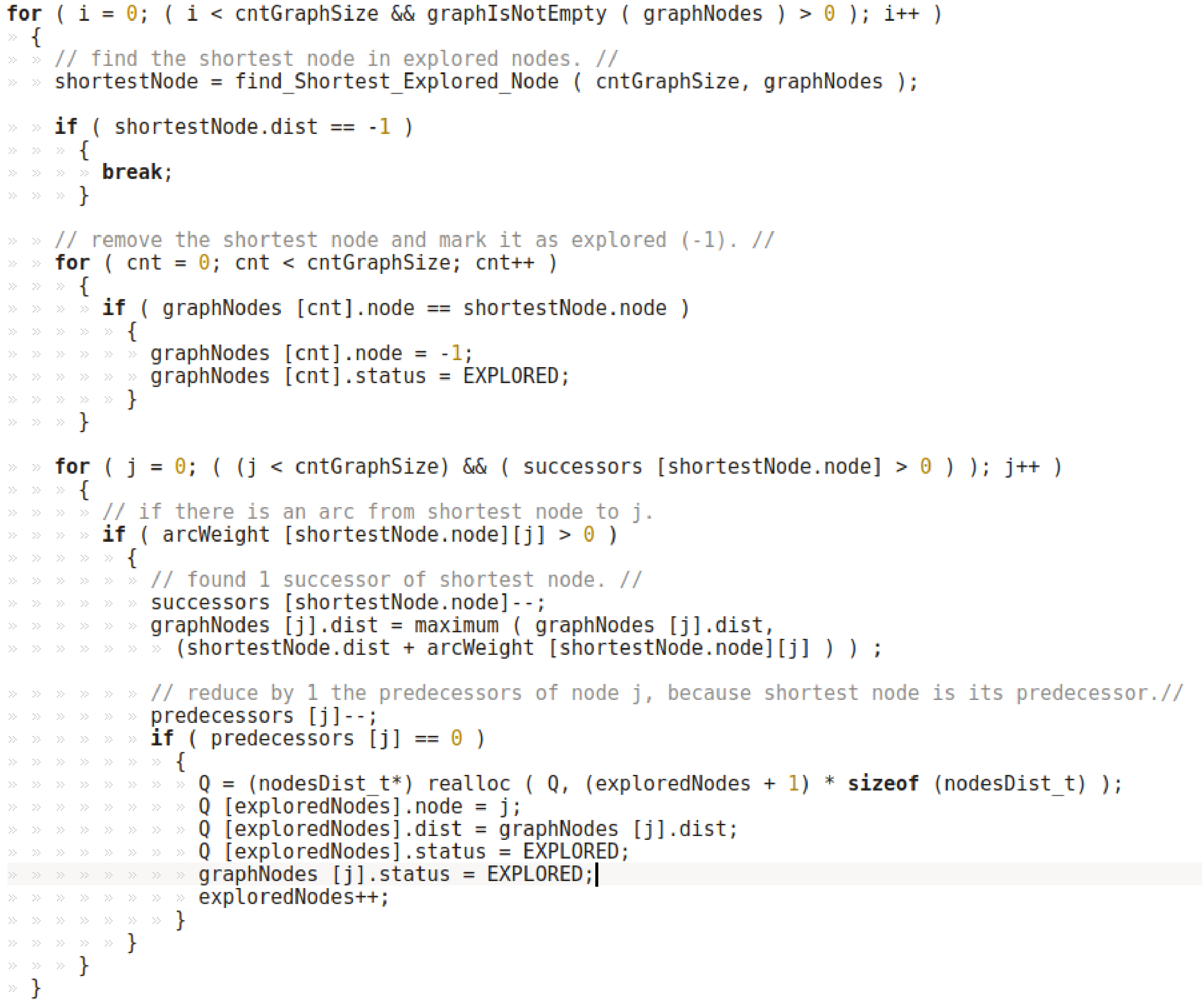
```
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * *
         * searchNode searches in graphNodes (where cnt is *
         * the current size) if there is the arcs[i].dest node.*
         * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        if ( searchNode ( cntGraphSize, graphNodes, arcs [i].dest ) == 1 )
          {
            ++cntGraphSize;
            graphNodes = (nodesDist_t*) realloc ( graphNodes, cntGraphSize * sizeof(nodesDist_t) );
            if ( graphNodes == NULL )
              {
                fprintf ( stderr, "Error in nodes allocation: %d", errno );
                return NULL;
              }
            graphNodes [cntGraphSize-1].node = arcs [i].dest;
            graphNodes [cntGraphSize-1].dist = -1;    // each node has unknown distace. //
            graphNodes [cntGraphSize-1].status = UNEXPLORED;
            // cntStoredNodes++;
          }
      }

    sortGraph ( cntGraphSize, graphNodes );
```

Graph_critical_path



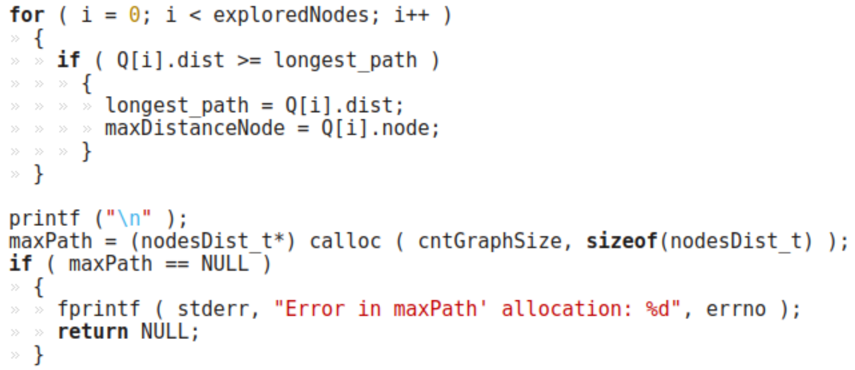
```
    // Initializations. //
    initInt ( size, predecessors, 0 );
    initInt ( size, successors, 0 );
    initInt ( size, previous, 0 );

    for ( i = 0; i < size; i++ )
      {
        for ( j = 0; j < size; j++ )
          {
            if ( arcWeight [i][j] != 0 )
              {
                predecessors [j]++; // predecessors of node j. //
                successors [i]++;   // successors of node i. //
                previous [j]++;
              }
          }
      }

    // create a queue for storing explored nodes. //
    Q = (nodesDist_t*) malloc ( sizeof(nodesDist_t) );
    if ( Q == NULL )
      {
        fprintf ( stderr, "Error in queue' allocation: %d", errno );
        return NULL;
      }
    cntStoredInputs = 0;
    for ( i = 0; i < cntGraphSize; i++ )
      {
        // find all inputs. //
        if ( predecessors [i] == 0 )
          {
            // distance from an input to the same input. //
            graphNodes [i].dist = 0;
            graphNodes [i].status = EXPLORED;
            ++cntStoredInputs;
            Q = (nodesDist_t*) realloc ( Q, cntStoredInputs * sizeof(nodesD
            if ( Q == NULL )
              {
                fprintf ( stderr, "Error in queue' allocation: %d", errno )
                return NULL;
              }

            Q[cntStoredInputs-1].node = graphNodes [i].node;
            Q[cntStoredInputs-1].dist = graphNodes [i].node;
            Q[cntStoredInputs-1].status = graphNodes [i].status;
          }
      }
```

Graph_critical_path



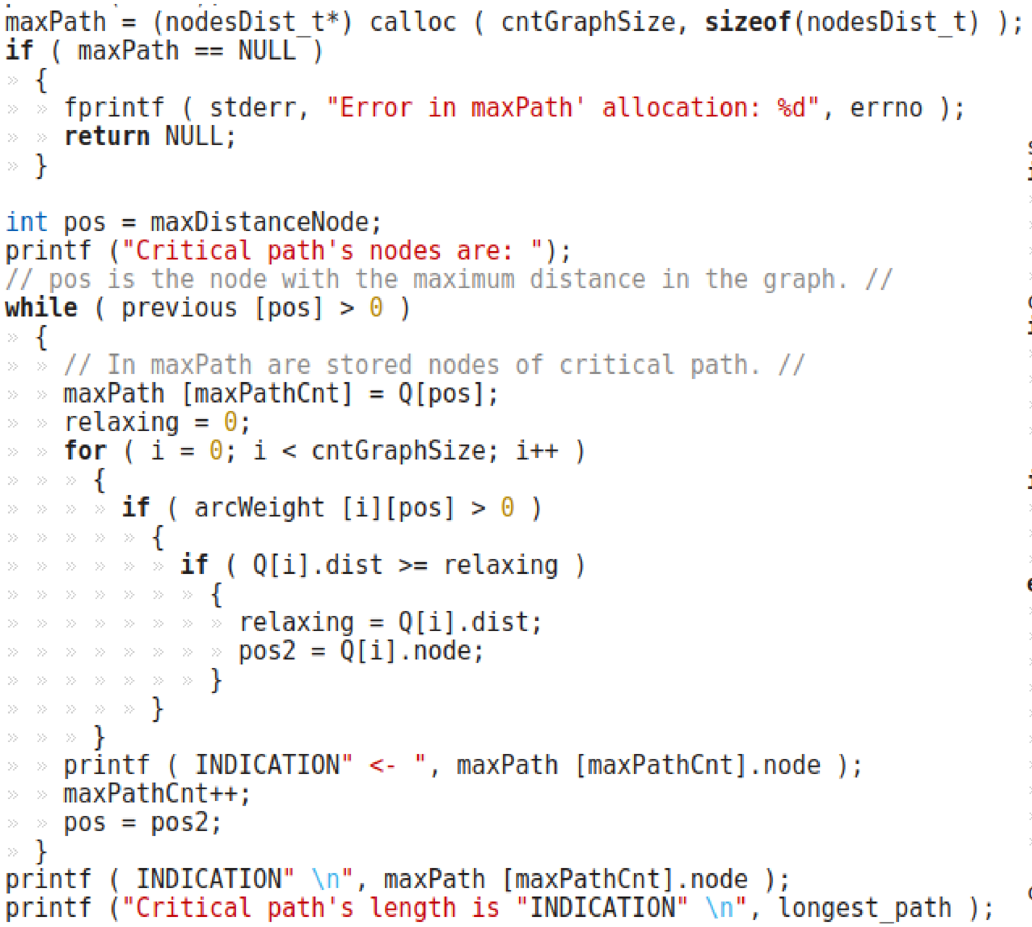
```
for ( i = 0; ( i < cntGraphSize && graphIsNotEmpty ( graphNodes ) > 0 ); i++ )
    {
        // find the shortest node in explored nodes. //
        shortestNode = find_Shortest_Explored_Node ( cntGraphSize, graphNodes );

        if ( shortestNode.dist == -1 )
            {
                break;
            }

        // remove the shortest node and mark it as explored (-1). //
        for ( cnt = 0; cnt < cntGraphSize; cnt++ )
            {
                if ( graphNodes [cnt].node == shortestNode.node )
                    {
                        graphNodes [cnt].node = -1;
                        graphNodes [cnt].status = EXPLORED;
                    }
            }

        for ( j = 0; ( (j < cntGraphSize) && ( successors [shortestNode.node] > 0 ) ); j++ )
            {
                // if there is an arc from shortest node to j.
                if ( arcWeight [shortestNode.node][j] > 0 )
                    {
                        // found 1 successor of shortest node. //
                        successors [shortestNode.node]--;
                        graphNodes [j].dist = maximum ( graphNodes [j].dist,
                            (shortestNode.dist + arcWeight [shortestNode.node][j] ) ) ;

                        // reduce by 1 the predecessors of node j, because shortest node is its predecessor.//
                        predecessors [j]--;
                        if ( predecessors [j] == 0 )
                            {
                                Q = (nodesDist_t*) realloc ( Q, (exploredNodes + 1) * sizeof (nodesDist_t) );
                                Q [exploredNodes].node = j;
                                Q [exploredNodes].dist = graphNodes [j].dist;
                                Q [exploredNodes].status = EXPLORED;
                                graphNodes [j].status = EXPLORED;
                                exploredNodes++;
                            }
                    }
            }
    }
```

Calculate the cost of longest path.

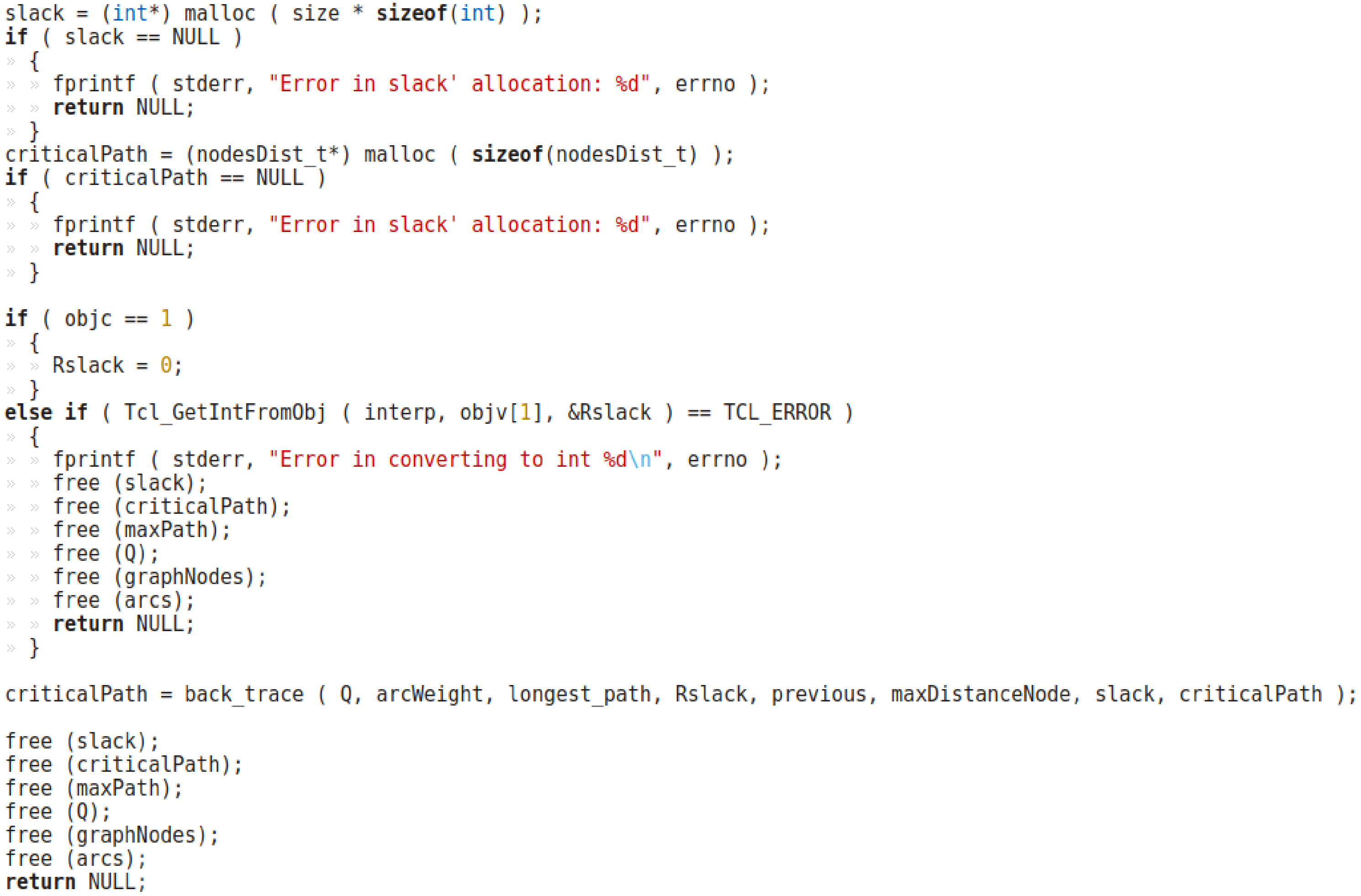


```
for ( i = 0; i < exploredNodes; i++ )
    {
        if ( Q[i].dist >= longest_path )
            {
                longest_path = Q[i].dist;
                maxDistanceNode = Q[i].node;
            }
    }

printf ("\n" );
maxPath = (nodesDist_t*) calloc ( cntGraphSize, sizeof(nodesDist_t) );
if ( maxPath == NULL )
    {
        fprintf ( stderr, "Error in maxPath' allocation: %d", errno );
        return NULL;
    }
```

Graph_critical_path



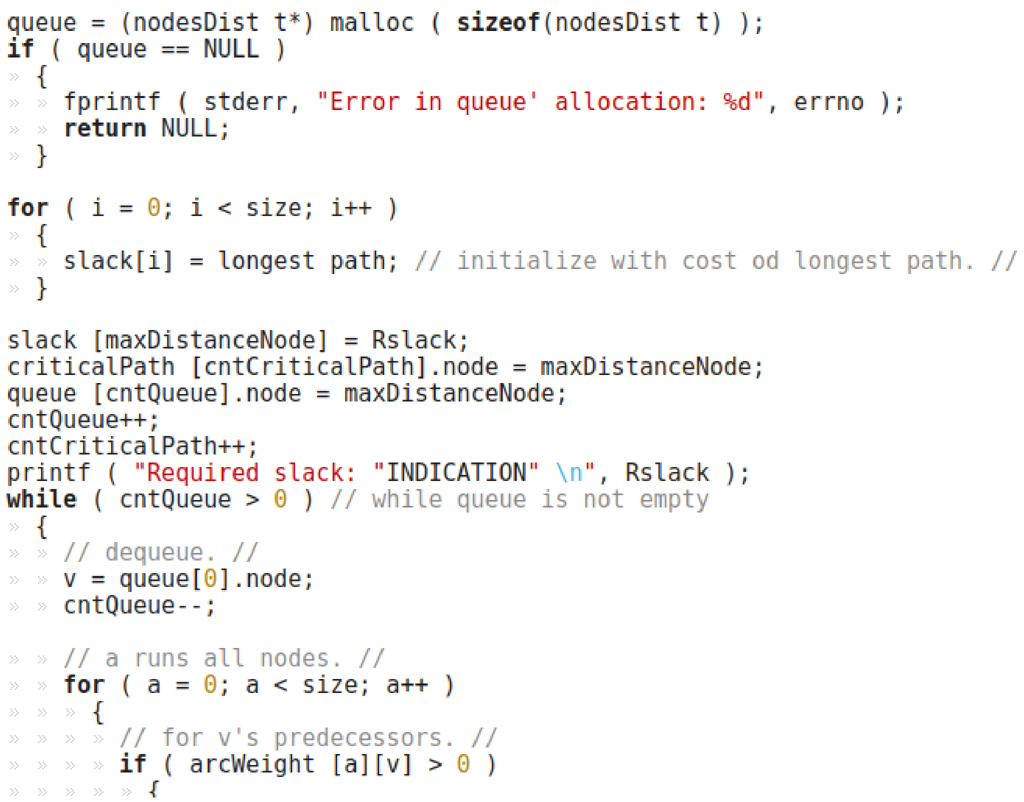
```
maxPath = (nodesDist_t*) calloc ( cntGraphSize, sizeof(nodesDist_t) );
if ( maxPath == NULL )
    {
        fprintf ( stderr, "Error in maxPath' allocation: %d", errno );
        return NULL;
    }

int pos = maxDistanceNode;
printf ("Critical path's nodes are: ");
// pos is the node with the maximum distance in the graph. //
while ( previous [pos] > 0 )
    {
        // In maxPath are stored nodes of critical path. //
        maxPath [maxPathCnt] = Q[pos];
        relaxing = 0;
        for ( i = 0; i < cntGraphSize; i++ )
            {
                if ( arcWeight [i][pos] > 0 )
                    {
                        if ( Q[i].dist >= relaxing )
                            {
                                relaxing = Q[i].dist;
                                pos2 = Q[i].node;
                            }
                    }
            }
        printf ( INDICATION" <- ", maxPath [maxPathCnt].node );
        maxPathCnt++;
        pos = pos2;
    }
printf ( INDICATION" \n", maxPath [maxPathCnt].node );
printf ("Critical path's length is "INDICATION" \n", longest_path );
```



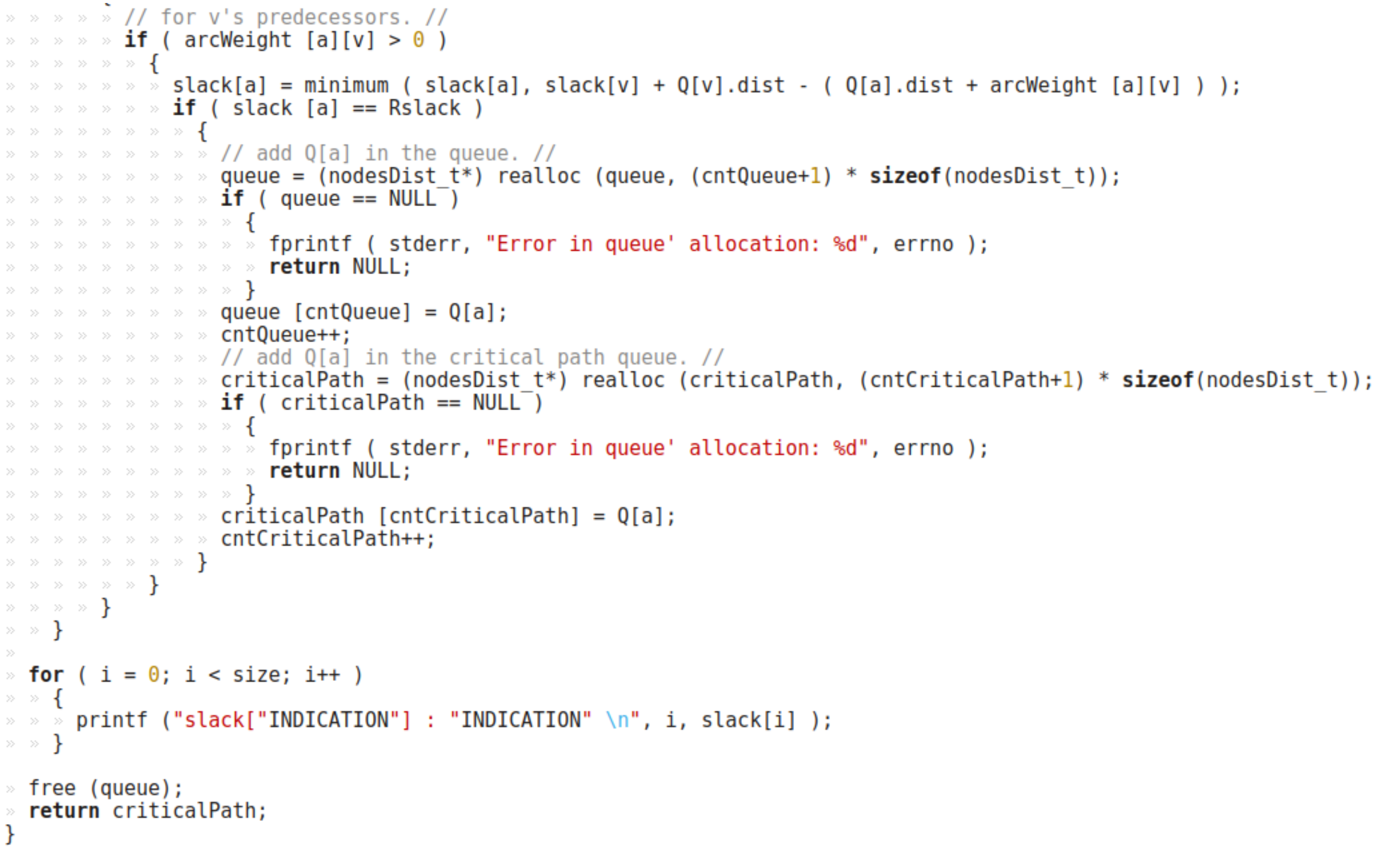
```
slack = (int*) malloc ( size * sizeof(int) );
if ( slack == NULL )
    {
        fprintf ( stderr, "Error in slack' allocation: %d", errno );
        return NULL;
    }
criticalPath = (nodesDist_t*) malloc ( sizeof(nodesDist_t) );
if ( criticalPath == NULL )
    {
        fprintf ( stderr, "Error in slack' allocation: %d", errno );
        return NULL;
    }

if ( objc == 1 )
    {
        Rslack = 0;
    }
else if ( Tcl_GetIntFromObj ( interp, objv[1], &Rslack ) == TCL_ERROR )
    {
        fprintf ( stderr, "Error in converting to int %d\n", errno );
        free (slack);
        free (criticalPath);
        free (maxPath);
        free (Q);
        free (graphNodes);
        free (arcs);
        return NULL;
    }

criticalPath = back_trace ( Q, arcWeight, longest_path, Rslack, previous, maxDistanceNode, slack, criticalPath );

free (slack);
free (criticalPath);
free (maxPath);
free (Q);
free (graphNodes);
free (arcs);
return NULL;
```

Back_trace

nodesDist_t *back_trace (nodesDist_t *Q, int arcWeight[size][size], int longest_path, int Rslack, int *previous, int maxDistanceNode, int *slack, nodesDist_t *criticalPath);



```c
queue = (nodesDist_t*) malloc ( sizeof(nodesDist_t) );
if ( queue == NULL )
    {
        fprintf ( stderr, "Error in queue' allocation: %d", errno );
        return NULL;
    }

for ( i = 0; i < size; i++ )
    {
        slack[i] = longest_path; // initialize with cost od longest path. //
    }

slack [maxDistanceNode] = Rslack;
criticalPath [cntCriticalPath].node = maxDistanceNode;
queue [cntQueue].node = maxDistanceNode;
cntQueue++;
cntCriticalPath++;
printf ( "Required slack: "INDICATION" \n", Rslack );
while ( cntQueue > 0 ) // while queue is not empty
    {
        // dequeue. //
        v = queue[0].node;
        cntQueue--;

        // a runs all nodes. //
        for ( a = 0; a < size; a++ )
            {
                // for v's predecessors. //
                if ( arcWeight [a][v] > 0 )
                    {
```



```c
                // for v's predecessors. //
                if ( arcWeight [a][v] > 0 )
                    {
                        slack[a] = minimum ( slack[a], slack[v] + Q[v].dist - ( Q[a].dist + arcWeight [a][v] ) );
                        if ( slack [a] == Rslack )
                            {
                                // add Q[a] in the queue. //
                                queue = (nodesDist_t*) realloc (queue, (cntQueue+1) * sizeof(nodesDist_t));
                                if ( queue == NULL )
                                    {
                                        fprintf ( stderr, "Error in queue' allocation: %d", errno );
                                        return NULL;
                                    }
                                queue [cntQueue] = Q[a];
                                cntQueue++;
                                // add Q[a] in the critical path queue. //
                                criticalPath = (nodesDist_t*) realloc (criticalPath, (cntCriticalPath+1) * sizeof(nodesDist_t));
                                if ( criticalPath == NULL )
                                    {
                                        fprintf ( stderr, "Error in queue' allocation: %d", errno );
                                        return NULL;
                                    }
                                criticalPath [cntCriticalPath] = Q[a];
                                cntCriticalPath++;
                            }
                    }
            }
    }

    for ( i = 0; i < size; i++ )
    {
        printf ("slack["INDICATION"] : "INDICATION" \n", i, slack[i] );
    }

    free (queue);
    return criticalPath;
}
```

End of presentation.

Thank you!