MIDDLE EAST TECHNICAL UNIVERSITY, NORTHERN CYPRUS CAMPUS

CNG213 C Programming – Programming Assignment 2

Date handed out: **Sunday 22 November 2015**

Date submission due: **Monday 7 December 2015**

"Path Finder"

This assignment aims to help you practice linked list data structure, basic linked list operations and stack abstract data type. Your main task in this assignment is to create a C program that will be used to process a map and find a path from a starting point to a given goal. Imagine that you are given a map in Figure 1 and you are asked to find a path from start point 1 to end point 16. As you can see some nodes are branching which means that there are multiple paths that you can follow. You can find a path from start point to this goal point by slowly traversing your nodes and whenever you reach to an end and there is no path to follow, then you need to go back and traverse the other alternative paths. For example, imagine that you start from 1. Then you can go to point 2. From point 2 to point 3. At point 3, you need to make a decision, you can either follow point 4 or 12. A simple approach would be to follow point 4, and keep point 3 so that you can come back to this decision point. This is known as **backtracking**. In Appendix 1, you can find a sample algorithm that makes use of a stack for finding path between start node to a goal node.

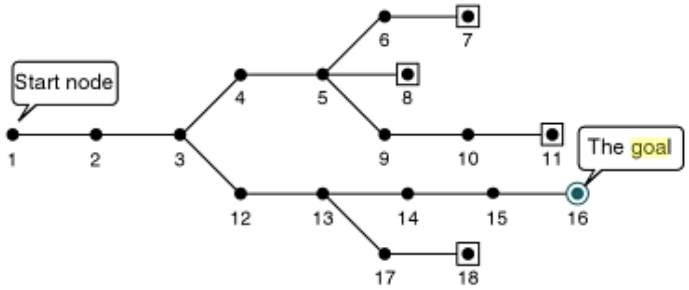


Figure 1 Input Graph

Problem Definition:

For this assignment, you need to first take your graph and represent it in a format that you can computationally process. In order to do this, you need to use a linked list representation and represent your graph with the adjacency information. Adjacency information shows the connections between points in your path. For example, graph in Figure 1 will be represented as follows:

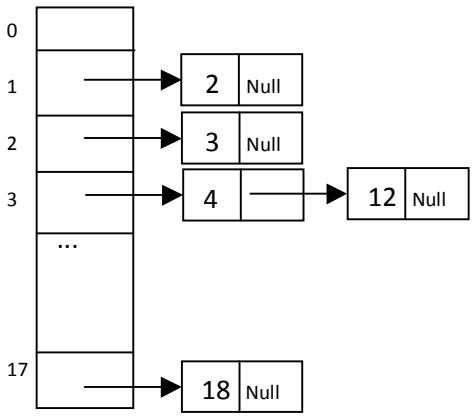


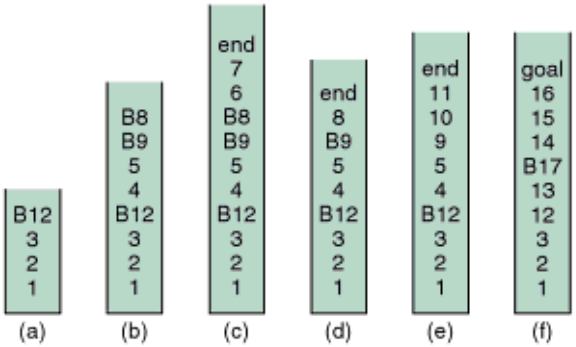
Figure 2 Adjacency representation

This representation shows that from point 1 you can go to point 2, and from point 2 you can go to point 3, etc.

Connections in a graph will be read from a file. For example, for Figure 1, your file will include the following data:

(1 2) (2 3) (3 4) (4 5) (5 6) (6 7) (5 8) (5 9) (9 10) (10 11) (3 12) (12 13) (13 14) (14 15) (15 16) (13 17) (17 18)

Once you populate your graph and create the adjacency data, then you will ask user to enter two points: a starting point and an end point. For example, when the user enters 1 and 16. You will try to find a path between these two points. In order to do that, you need to implement the algorithm in appendix one. In your execution, you need to display the content of your stack and also the path at the end.




```
The path between 1 and 16 is:

1, 2, 3, 12, 13, 14, 15, 16
```

The figure above shows the complete stack tracking of the path to the goal. Please note that B12/B8/B9, etc. denote the alternative nodes in a given decision points. For example, at node 3, we follow 4 but 12 is the alternative node that needs to be visited in case of a backtracking, please see Algorithm 1.

Programming Requirements:

First of all you need to define a data structure to represent the graph given in Figure 2. Please note that you can use one linked list data structure to represent it. Your program will need to follow these steps:

1. You need to read the data points from a file. Please note that you cannot make an assumption about the number of data points in a file. Assume that data points are represented as (d_1 d_2) (d_3 d_4) (d_5 d_6)… (d_i d_j). However, you can assume that there will be a finite number of points.

2. You will then need to create the data structure in Figure 2 to represent the adjacency of these points.

3. You will get the starting and end points for the search.

4. Finally, you will conduct the search by applying the algorithm in Appendix 1 that uses a stack to suggest a path from the starting point to an end point.

5. In order to achieve these steps you need to have the following functions:

`load_data_points():` This function will take the file name as an **input**. It will then load the data from the file to your data structure defined in Figure 2. This data structure will be **returned** by this function.

```
/*input: file name that will be used to load points

  return: a data structure to represent the adjacency data, see Figure 2*/
```

`get_start_point():` This function will get a start point from the user and return it.

```
/*input: void

  return: start point for the path that will be searched*/
```

`get_end_point():` This function will get an end point from the user and return it.

```
/*input: void

  return: end point for the path that will be searched*/
```

`goal_seek():` This function will take two three inputs: start point, end point and the data structure populated by the load_data_points() function, it will apply the algorithm given in Appendix 1 and displays the path between start point and end point.

```
/*input:start point, end point, and data structure represented in Figure 2

  return: void*/
```

Programming Style Tips!

Please follow the modular programming approach. In C we use functions referred to modules to perform specific tasks that are determined/guided by the solution. Remember the following tips!

- Modules can be written and tested separately!
- Modules can be reused!
- Large projects can be developed in parallel by using modules!
- Modules can reduce the length of the program!
- Modules can also make your code more readable!

Grading:
Your program will be graded as follows:

Grading Point	**Mark (100)**
`load_data_points()`	40 points
`get_start_point()`	5 points

`get_end_point()`	5 points
`goal_seek()`	40 points
`Main function`	10 points

NOTE: Remember to have good programming style (Appropriate comments, variable names, formulation of selection statements and loops, reusability, extensibility etc.). Each of the items above will include10% for good programming style.

Appendix 1:

The following algorithm shows how a map can be taken in and a path can be created to a goal node with backtracking. Please note that the numbers here only shows the line number in the **Pseudocode**.

```
Algorithm seekGoal (map)
This algorithm determines the path to a desired goal.
   Pre  a graph containing the path
   Post path printed
1  createStack (stack)
2  set pMap to starting point
3  loop (pMap not null AND goalNotFound)
   1   if (pMap is goal)
       1   set goalNotFound to false
   2   else
       1   pushStack (stack, pMap)
       2   if (pMap is a branch point)
           1   loop (more branch points)
               1   create branchPoint node
               2   pushStack (stack, branchPoint)
           2   end loop
       3   end if
       4   advance to next node
   3   end if
4  end loop
5  if (emptyStack (stack))
   1   print (There is no path to your goal)
6  else
   1   print (The path to your goal is:)
   2   loop (not emptyStack (stack))
       1   popStack (stack, pMap)
       2   if (pMap not branchPoint)
           1   print(map point)
       3   end if
   3   end loop
   4   print (End of Path)
7  end if
8  return
end seekGoal
```