ECE 264 Advanced C Programming

2009/04/08

Contents

1 Exam 3 and Programming Assignment 4

2 Group Discussion (Bonus Points)

Consider a typical distance table:

	Atlanta	Boston	Chicago	Denver	New York	Seattle
Atlanta	0	1115	717	1425	887	2723
Boston	1115	0	1013	2015	197	3126
Chicago	717	1013	0	1026	818	2108
Denver	1425	2015	1026	0	1807	1314
New York	887	197	818	1807	0	2927
Seattle	2723	3126	2108	1314	2927	0

From this table, the distance between Atlanta and Boston is 1115 miles; the distance between New York and Chicago is 818 miles. The table can be stored in a *2-dimensional array*, `d[n][n]`. The value `n` means the number of cities; `n` is 6 in this example. The value `d[i][j]` means the distance between the `i`th city and the `j`th city. If `i` is the same as `j`, the two cities are the same and the distance is zero.

We can modify and generalize a distance table to follow these rules:

- The distance to the same location itself, `d[i][i]`, is always zero.
- The distance from the `i`th location to the `j`th location (`i` $\neq$ `j`), `d[i][j]`, is a positive number as the distance if the two locations are **directly connected** without passing through another location in the table. If there are multiple direct routes, choose the shortest route as the distance. Because one-way streets are possible, `d[i][j]` **may be different** from `d[j][i]`.
- If there is no direct connection from the `i`th location to the `j`th location, the distance between them, `d[i][j]`, is infinity (∞). Please remember that `d[i][j]` $= \infty$ does **not** imply `d[j][i]` $= \infty$.

Suppose you are given a distance table of `n` locations.

Question 1: Write a program to find the **shortest** distance between any pair of different locations, by passing through at most `k` additional locations, $0 \leq$ `k` $\leq$ `n - 2`. Since the distances are always positive numbers, your route should never pass the same location twice.

Question 2: Write a program to find the **longest** distance between any pair of different locations, by passing through at most `k` additional locations. You can visit each location at most once.

On April 24, if your group volunteers to present a solution, each member in the group can receive 1.5 bonus point as part of class participation.

Group Assignments:

```
1 Dinkledine Aaron, adinkled@purdue.edu
1 Ko Seongwoon, ko5@purdue.edu
1 Li Hetong, li186@purdue.edu
1 Schmidt Susanne, schmidsm@purdue.edu
1 Smith Sean, smith342@purdue.edu
1 Swindler Joshua, jswindle@purdue.edu

2 Ahuja Karan, kahuja@purdue.edu
2 Faber Darrell, dmfaber@purdue.edu
2 Herdzina-Huss Darien, dhuss@purdue.edu
2 Mahmood Zaeem, zmahmood@purdue.edu
2 Mishra Ankur, amishra@purdue.edu
2 Pesyna Kenneth, kpesyna@purdue.edu

3 Bansal Nikhil, bansaln@purdue.edu
```

3 Malik Abish, amalik@purdue.edu
3 Mc Lean Ryan, rmclean@purdue.edu
3 Oliver Ian, ioliver@purdue.edu
3 Zhou Yang, zhouy@purdue.edu

4 Conaboy Michael, mconaboy@purdue.edu
4 Hall Ethan, ekhall@purdue.edu
4 Park Junhyeong, park1@purdue.edu
4 Robles Derrick, djrobles@purdue.edu
4 Wolfer Michael, mwolfer@purdue.edu

5 Chunduru Nag Varun, nchundur@purdue.edu
5 Fetter Daniel dfetter@purdue.edu
5 Lakhmani Vashisht, vlakhman@purdue.edu
5 Wetherill Julia, jwetheri@purdue.edu
5 Whyland Jon, jwhyland@purdue.edu

6 Geng Junzhe, jgeng@purdue.edu
6 Grover Animesh, grovera@purdue.edu
6 Jhajaria Krishna, kjhajari@purdue.edu
6 Mohammed Razip Ahmad Mujahid, mohammea@purdue.edu
6 Yuki Zeno, zyuki@purdue.edu

7 Al Shehhi Hamad, halshehh@purdue.edu
7 Brener Gregory, gbrener@purdue.edu
7 Christman Jacob, christjr@purdue.edu
7 Izturriaga Manuel, mizturri@purdue.edu
7 Kim Do-Hyoung, kim130@purdue.edu

8 Bajaj Arjun, abajaj@purdue.edu
8 Chen Yi-Kai, chen17@purdue.edu
8 Jesse Skylar, sjesse@purdue.edu
8 Phillips Collin, cnphilli@purdue.edu
8 Schuman Richard, rschuman@purdue.edu

9 Guo Yicheng, yguo@purdue.edu
9 Hall Loren, hallljj@purdue.edu
9 Mehta H’rsh, hmehta@purdue.edu
9 Schieler Curt, cmschiel@purdue.edu
9 Vadlamudi Ramanth, rvadlamu@purdue.edu

10 Granger William, wgranger@purdue.edu

```
10 Kim Wesley, kim265@purdue.edu
10 Neuenschwander Tyler, tcneuens@purdue.edu
10 Pulliam Stuart, spulliam@purdue.edu
10 Raj Vishwaman, vpraj@purdue.edu
```

3 Doubly Linked List

Each node in our linked lists has only one link to the next node. In such a list, we can reach any node from the first node but we cannot reach the first node from any other node. We can add another link to the **previous** node and create a *doubly linked list*. Any node can reach any other node using the previous or the next link. When handling insertion and deletion, we need to make sure both the previous and the next links are updated correctly. The following code shows how to insert a value into a doubly linked list.

```
#ifndef DLINKNODE_H
#define DLINKNODE_H
typedef struct dlinknode
{
  struct dlinknode * ln_next;
  struct dlinknode * ln_prev;
  int ln_value;
} Node;

Node * DList_copy(Node * n);
void DList_assign(Node * * n1, Node * n2);
void DList_insert(Node * * n, int v);
int DList_delete(Node * * list, int v);
void DList_print(Node * n);
void DList_destruct(Node * n);
int DList_search(Node * list, int v);
#endif

#include "dlinknode.h"
#include <stdio.h>
#include <stdlib.h>
static Node * Node_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> ln_value = v;
  n -> ln_next = 0;
  n -> ln_prev = 0;
  return n;
}

void DList_insert(Node * * n, int v)
{
  Node * p = Node_construct(v);
  Node * curr = * n;
  Node * prev = * n;
  if (( *n) == 0)
    {
      *n = p;
      return;
    }
  while ((curr != 0) && ((curr -> ln_value) < v))
    {
      prev = curr;
      curr = curr -> ln_next;
    }
  if (curr == (* n))
    {
      p -> ln_next = (* n);
      (* n) -> ln_prev = p;
      * n = p;
    }
  else
    {
      p -> ln_next = prev -> ln_next;
      p -> ln_prev = prev;
      /* Will p be the last node in the list? */
      if ((prev -> ln_next) != 0)
        { (prev -> ln_next) -> ln_prev = p; }
      prev -> ln_next = p;
    }
}
```

Exercise: This example shows how to insert. Write the delete function.

4 Divide and Conquer and Dynamic Structures

When we explained binary search, we used an array. Arrays have one major advantage: we can access any element in a single step. In contrast, in a linked list, we have to follow the links node by node. However, arrays have one major restriction: the sizes are fixed. In contrast, a linked list can expand or shrink as needed. Can we combine the performance of binary search without being restricted to arrays? A *binary search tree* is one solution.

Doubly linked list is *linear* meaning that all nodes are linked as a line. If each node has two links, the links do **not** have to be related. It is **not** necessary that any node can reach any other node using the previous or the next link. It is all right if there is only one path to any node, from one special node called *root*. We usually draw a "tree" upside down. The "root" is at the top and the "leaves" are at the bottom.