CS29206 Systems Programming Laboratory
Spring 2022
Assignment 1
Due: February 1, 2022

1. Suppose you are building a dynamic library of the following data structures. The details of the data structures and the files in which they are implemented are given below:
    a. LLIST: data type for a singly linked list of NODEs. A NODE is a C structure with two fields, value (of type int) and next (of type pointer to NODE). The LLIST type should support the following functions:
        i. LLIST createList() : creates an empty linked list, and returns it
        ii. int searchList(LLIST H, int k): searches for the value k in the list H. Returns 1 if k is found, 0 otherwise.
        iii. LLIST insertAtFront(LLIST H, int k): inserts the value k at the beginning of the list H. Returns the new list.
        iv. LLIST insertAtEnd(LLIST H, int k): inserts the value k at the end of the list H. Returns the new list.
        v. LLIST deleteFromFront(LLIST H, int *k): deletes the first element from the list and returns the value stored in that node using the pointer k. Returns the new list as the return value.
        vi. LLIST deleteFromEnd(LLIST H, int *k): deletes the last element from the list and returns the value stored in that node using the pointer k. Returns the new list as the return value.
        vii. LLIST deleteList(LLIST H, int k): deletes the value k from the list H if it is present. Returns the new list. If the value k occurs multiple times, only its first occurrence in the list is deleted.

    The LLIST data structure is to be implemented in the C file llist.c

    b. STACK: data type for a stack implemented using the LLIST data structure. The STACK type should support the following functions:
        i. STACK createStack(): Creates an empty stack, and returns it.
        ii. int isEmptyStack(STACK S): returns 1 if the stack S is empty, 0 otherwise.
        iii. STACK push(STACK S, int k): pushes a value k in the stack S, and returns the new stack.
        iv. STACK pop(STACK S, int *k): pops the top element from the stack, and returns that element using the pointer k. Returns the new stack as the return value.

    The STACK data structure is to be implemented in the C file stack.c

    c. QUEUE: data type for a queue implemented using the LLIST data structure. The QUEUE type should support the following functions:
        i. QUEUE createQueue(): Creates an empty queue, and returns it.

ii. int isEmptyQueue(Queue Q): returns 1 if the queue Q is empty, 0 otherwise.
iii. QUEUE enqueue(QUEUE Q, int k): adds a value k to the back of the queue, and returns the new queue.
iv. QUEUE dequeue(QUEUE Q, int *k): removes the element at the front and returns that element using the pointer k. Returns the new queue as the return value.

The QUEUE data structure is to be implemented in the C file queue.c

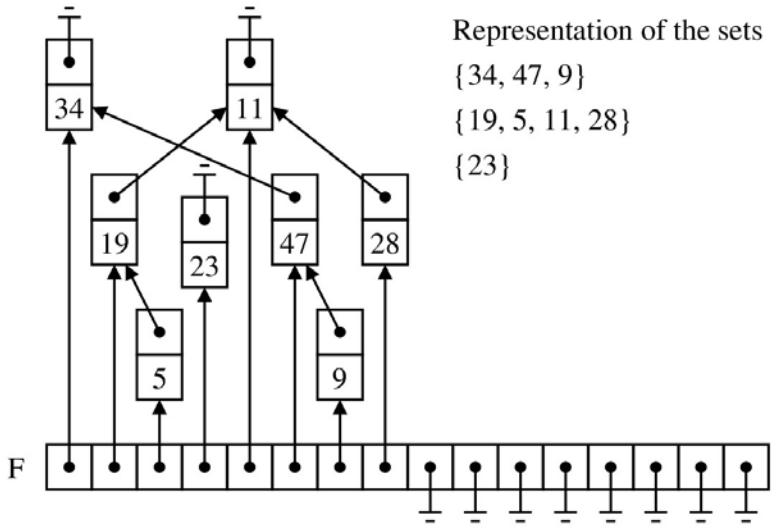
d. HEAP: data type for a min-heap. The HEAP type is to be implemented with an array of size 100 and should support the following functions:
   i. HEAP createHeap(): creates an empty heap, and returns it.
   ii. int findMin(HEAP H): returns the minimum value in the heap H.
   iii. HEAP extractMin (HEAP H): deletes the minimum value from the heap and returns the modified heap.
   iv. HEAP insertHeap(HEAP H, int k): inserts the value k in the heap H, and returns the new heap.
   v. int isFullHeap(HEAP H): returns 1 if the heap is full, 0 otherwise.
   vi. int isEmptyHeap(HEAP H): returns 1 if the heap is empty, 0 otherwise.

   The HEAP data structure is to be implemented in the C file heap.c

e. UNION_FIND: data type that implements a union-find data structure. You can assume that the data structure will only store sets of non-negative integers. Assume that NODE_PTR points to a node in the union-find forest. The union-find data structure is an array of such pointers. The data type will support the following functions:
   i. UNION_FIND createUF(int n): creates a UNION_FIND data structure that can store n node pointers. Initially, all these pointers are NULL.
   ii. UNION_FIND makeSetUF(UNION_FIND F, int x, int *i): Creates a singleton set (a node) for the element x, adds to F a pointer to that node, and returns the modified UNION_FIND structure. An int pointer is passed additionally to return the insertion index. The library is not required to choose the insertion indices in the sequence 0, 1, 2, … , but must guarantee that after this index is returned, no future efforts will be made to relocate x to another index.
   iii. NODE_PTR findUF(UNION_FIND F, int i): returns the ID of the set to which the element pointed to by the i-th pointer in F belongs. The ID is a pointer to the root node of the tree containing that element. Assume that the index i corresponding to the element being searched is known to you when you call this function.
   iv. void unionUF(UNION_FIND F, int i, int j): merges the sets containing the elements pointed to by the i-th and the j-th pointers of F. Assume again that the indices i and j corresponding to the elements are known to you when you call this function. The merging heuristic to be used is: Make the tree

with the smaller number of nodes a subtree of the tree with the larger number of nodes.

The following figure describes an example of a union-find data structure. This is capable of storing 16 elements. Only eight elements 34, 19, 5, … , 28 are inserted one by one by calling makeSetUF. You may need to store other fields in each node of the forest. The figure shows the union-find forest after some unionUF operations are carried out.



The UNION_FIND data structure is to be implemented in the C file union_find.c

All the files must have a corresponding .h files of the same name. Choose what you want to put in .h files appropriately as per discussions in class.

For each of the data structures, you can write any additional helper functions to better organize your code if you want.

First, write all the .h and .c files mentioned above to implement the above data types.

Then create a makefile that will create a dynamic library out of the files that will contain the functions of all the data types. Name your makefile makefile.basic. Your makefile should also have rules to clean all unnecessary files after the creation of the library and to install the library in the folder /usr/include/lib (assume that the makefile will be run with root permission).

2. Now suppose the above dynamic library is built. We now build a static library to implement a data type GRAPH to represent weighted, undirected graphs with positive integer weights. The GRAPH data type can represent graphs with a maximum of 100 nodes, using adjacency matrix to store the graph. For an edge existing in the graph, the corresponding entry in the matrix should store the weight of the edge. If for an entry (i, j)

no corresponding edge exists in the graph, the entry should store 0. The GRAPH data types should support the following functions:

a. GRAPH readGraph(char *FName): Reads the nodes and edges of a graph from a file with name FName (assume that the file always exists). The format of the file is specified later. The new graph is returned.
b. void DFS(GRAPH G): Does a DFS traversal of the graph G using the STACK data type (in the dynamic library), and prints out the nodes in the DFS traversal order. The implementation of this function should ignore the weights, that is, you should do the DFS as if the graph is unweighted and you do not have to print the weights.
c. void BFS(GRAPH G): Does a BFS traversal of the graph G using the QUEUE data type (in the dynamic library), and prints out the nodes in the BFS traversal order. The implementation of this function should ignore the weights, that is, you should do the BFS as if the graph is unweighted and you do not have to print the weights.
d. void MST(GRAPH G): Computes the minimum spanning tree of the graph G using Kruskal's algorithm. The function prints out the edges added to the tree (the two endpoints and the weight), each edge in a single line, followed by the weight of the MST in the last line. The function should use the UNION_FIND data structure of the dynamic library.

The functions are to be implemented in the C file graph.c with an appropriate graph.h file added.

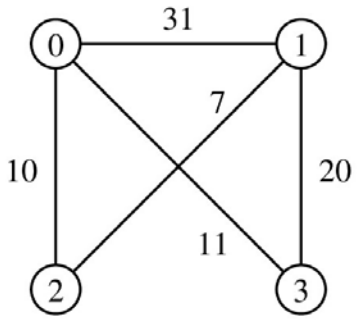
You can also write any additional helper functions to better organize your code if you want.

Write the graph.h and graph.c files first. Then write a makefile to create the static library. Name your makefile makefile.graph. Your makefile should also have rules to clean all unnecessary files after the creation of the library and to install the library in the folder /usr/include/lib (assume that the makefile will be run with root permission).

The format in which a graph will be stored in the file is given below:

- The first line will contain the number of nodes n. The nodes will be identified by the numbers 0 to n – 1.
- The second line will store the number of edges m.
- After that, there will be m lines, each specifying one edge. Each edge will be specified in a single line with the format "u v w", representing an edge between nodes u and v with weight w. Assume that the same edge is not present multiple times in the file.

An example is given below.



An undirected graph

```
4
5
0 1 31
0 2 10
0 3 11
1 2 7
1 3 20
```

File storing the graph

3. Write a program main.c with a main function that reads a file name from the user, which stores a weighted undirected graph. The program then calls readGraph to prepare a graph G. Finally, the program calls DFS, BFS, and MST on G. Now write a makefile to create an executable file called graph.out from the main.c file. Compile with whatever libraries you think you require from the earlier two parts..