# Algorithms Assignment 1
# Data Structure and Searching Algorithms

Due Date: Sep, 11, 2024

Names: Kevin Zheng UID: 010986744 Email: kevinz@uark.edu
Max Thursby UID: 010967047 Email: mathursb@uark.edu

0 Late Days

Linked List:

For the linked list implementation we started by adding the find function which created a reference node and used it to iterate through the list until a matching value was found. We then used this function in the Insert and Remove, the Insert uses the find inorder to check if the value was already in the list and if not it creates a new node with that value and places it at the root. The Remove function similarly uses the find function to locate the node that you wish to remove, it then creates temporary nodes to update the next pointers and deletes the desired node. Lastly we created a deconstructor for the linked list which iterates through the list and removes all the nodes and sets the root to nullptr.

Stack:

For the stack implementation the first thing that was done was to create the deconstructor. To do this we first make a temporary stack node that equals the head. Then loop through the stack and make a second temporary stack node that equals to the first temporary stack node. Then delete the first temporary stack node and then make that node equal to the second stack node. Then we moved on to the empty function. To implement this we just checked if the head is null. If it is null then that means the stack is empty so it'll return true. Otherwise it is not empty and returns false. Next thing we did was make the pop function for the stack. To do this we first

make the value equal to the head value. Then we make a temporary stack node that equals the head. Then we make the head of the stack point next to the head and then delete the temporary stack node and return the popped value. For the push all we did was push the value into the head. So we make a new stack node that holds the value and push it to the head.

BFS:

To start the BFS we initialized a variable to hold the number of nodes in the graph as well as two queues and two vectors, visited and trace. We start the code by putting the initial values in the vectors, fales for the visited and -1 for the trace. We then pushed the start node value into the queue and set the visited value at start to true, this is followed by the first loop to set the trace vector for the start to the passBy node. This was done by first checking if the value u is equal to the passBy, if not we start the loop to iterate through the adjacent nodes, we start this by taking a reference of the next node v and checking if it's visited, if not we set visited[v] = true and trace[v] = u , the current node, and lastly we push v into the queue. Once out of the loop we set u equal to passBy and start setting the path by going backwards from u to the start by inserting u into the path and then setting u equal to trace[u] , the previous node in the path. This chunk of code is written twice, first for the start to the passBy node and the second time to go from the passBy node to the destination. At the end of the BFS when we are setting the path the second time we insert at + 1 from the passBy inorder to not add it to the path twice.

RDFS:

For the RDFS we started by implementing the pseudo code given to us for the DFS helper function which starts with pushing the current node onto the path and sets the visited vector for that position to be true. We then checked if the current node was equal to the destination and returned 1 to exit the function if it did. We then set a checkpoint to make sure the

path size never got bigger than the numberOfBuilding, the given path size, and if it did we popped the current node off the stack, set the visited to false and returned 0 to go back into the DFS function. The next section of the helper function we implemented was the traversal and the call to renter the function, This was done by setting an int to the size of the adjacency list and then creating a node at the root. We then looped with the size of the adjacency list, through the linked list of adjacent nodes, checking if the visited value was false and if so we set a flag int value equal to the result of the DFS function and check the flag value after. Lastly we pop the current node off the list if it made it past the flag check and set visited back to false and return 0 to go back into the recursive loop.

For the main section of the RDFS function we start by getting the size of the graph (the number of nodes in the graph), and then create the visited vector using the variable saved. Lastly we start the recursive call to the DFS helper function and pass in the initialized visited vector.

DFS:

First thing we did for the dfs function was to declare and initialize the variable u. Then we make two vectors called visited and trace. Visited would be a bool and trace would be an int. After that we made an int stack called stack. Then we would loop through i to N-1 and make all the visited to be false and all the trace values be -1. After that we would push the start to the stack and make the visited vector at the start position be true. We then make a while loop that checks if the stack is not empty and in the while loop we make u be equal to the pop value from the stack. If the u value equals the destination it'll break out of the loop. After that we check if the path size is bigger than the number of buildings. If it's true then we pop once on the path vector and also visited[u] equal to false. But if it isn't it'll continue on. We then make a new variable that checks the number of adjacency nodes and also a linked list that gets the head point

of the linked list. Then we make a for loop to loop through the adjacency nodes and make a new variable v that is the adjacency node of u. After that we check if visited[v] is false then we set that visited[v] to true. Then we also make trace[v] equal to u. After that we push the value p into the stack. Out of the loop we make u equal to destination. Then we make another while loop that checks if u is not equal to -1. In that loop we insert u into the first position of the path vector. Then in the end we make u equal to trace[u].

Output:

```
10 is in the linked list
20 is not in the linked list
100 is in the linked list
The linked list has 2 nodes
100 is not in the linked list
The linked list has 1 node
Your linked list implementation is correct


Insert 0 into stack
Insert 1 into stack
Insert 2 into stack
Insert 3 into stack
Insert 4 into stack
Stack is not empty
Remove tail node (4) in stack successfully
Remove tail node (3) in stack successfully
Remove tail node (2) in stack successfully
Remove tail node (1) in stack successfully
Remove tail node (0) in stack successfully
Stack is empty
Your stack mplementation is correct
```

```
Perform unit test on your searching implementation
Path from 0 to 4 by bfs: 0 1 2 0 5 6 4
Number of nodes in the path: 7
Path from 0 to 4 by rdfs: 0 5 6 4
Number of nodes in the path: 4
Path from 0 to 4 by dfs: 0 1 2 4
Number of nodes in the path: 4


Perform unit test on your searching implementation on campus map
Path from RSWE to RCED by the bfs algorithm: RSWE -> LMK123 -> LMK124 -> LMK125 -> LMK126 -> LMK127 -> LMK101 -> RSEA -> AR
KU2 -> POST -> FPAC -> WATR -> ARSAGAS -> CORD1 -> WALK -> GEAR -> FERR -> SCEN -> PHYS -> HEAT -> MEEG -> HILL -> JBHT ->
RCED
Number of buildings in the path: 24

Path from RSWE to RCED by the rdfs algorithm: RSWE -> LMK123 -> LMK124 -> LMK125 -> LMK126 -> LMK127 -> LMK101 -> RSEA -> A
RKU2 -> POST -> FPAC -> WATR -> ARSAGAS -> CORD1 -> WALK -> GEAR -> FERR -> SCEN -> PHYS -> CHPN1 -> BELL -> CHPN2 -> DUNA
-> HAPG -> WJWH -> OTHS -> RCED
Number of buildings in the path: 27

Path from RSWE to RCED by the dfs algorithm: RSWE -> LMK123 -> LMK124 -> LMK125 -> LMK126 -> LMK127 -> LMK101 -> RSEA -> AR
KU2 -> POST -> FPAC -> WATR -> ARSAGAS -> CORD1 -> WALK -> GEAR -> FERR -> SCEN -> PHYS -> CHPN1 -> BELL -> CHPN2 -> DUNA -
> HAPG -> WJWH -> CENA -> RCED
Number of buildings in the path: 27
```