Analysis of Algorithms
Project Report

Submitted by:
Batul Zamin (nd9354), Divya Bhat (mn9386)

Rationale:

1. **Strategy:**

a. Creating Undirected Binary Tree from given input file:

- Read the input from the input.txt file line by line .
- Parse the node values to integers and add the nodes to the vector of vector structure.
- Build a tree using an adjacency list.

An adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the set of adjacent vertices of a vertex in the graph. In our case, we have an undirected binary tree which has guarantee to be connected and acyclic. Also, it is binary, which means each vertex or node has at most two children nodes, i.e., two neighboring nodes.

We represent the undirected binary tree as an adjacency list using the following information and assumptions:

1. Since the entered tree is always valid, we can guarantee that the total number of nodes in the tree is equal to the number of edges plus 1.
2. To represent the tree as an adjacency list, we assume that the node values or the data in the tree node always start from 1 onwards sequentially; their order in the tree structure can be random, however.
3. We use the node value/data as index in the adjacency list which stores a list of its neighbors (left and right children).
4. We consider the first node of the first edge in the input file as the root node of the tree and start searching for the target node from there.

b. Depth First Search:

Here we use a stack for the depth first search traversal of the undirected binary tree. The following are the steps required for the algorithm.

- First add the root to the Stack.
- Pop a node from Stack.
- If the data in the node is the same as the data in the target node, then the search is successful, and we return from the algorithm.
- Else we add its neighbors (right and left children) from the adjacency list to the stack.
- Repeat the above two steps until the stack is empty.
- If we reach this point, our search was a failure.

c. Breadth First Search:

Here we use a queue for the breadth first search traversal of the undirected binary tree. The following are the steps required for the algorithm.

- First add the root to the Queue.
- Dequeue a node from Queue.
- If the data in the node is the same as the data in the target node, then the search is successful, and we return from the algorithm.
- Else we add its neighbors (right and left children) from the adjacency list to the queue..
- Repeat the above two steps until the queue is empty.
- If we reach this point, our search was a failure.

2. Results:

The output of the search algorithms with the following inputs is as follows:

1. *Input:*

```
6,2
6,8
2,1
2,4
4,3
4,5
8,7
8,9
*
9
#
```

Output 1:

```
Enter 1 for DFS searching or 0 for BFS searching in the tree:
1

Searching the target 9
Starting DFS search in input tree:
push root 6 to stack...
pop 6 from stack...
push 2 to stack...
push 8 to stack...
pop 8 from stack...
push 7 to stack...
push 9 to stack...
pop 9 from stack...
DFS: SUCCESS
```

Output 2:

```
Enter 1 for DFS searching or 0 for BFS searching in the tree:
0

Searching the target 9
Starting BFS search in input tree:
enqueue root 6 to queue...
dequeue 6 from queue...
enqueue 2 to queue...
enqueue 8 to queue...
dequeue 2 from queue...
enqueue 1 to queue...
enqueue 4 to queue...
dequeue 8 from queue...
enqueue 7 to queue...
enqueue 9 to queue...
dequeue 1 from queue...
dequeue 4 from queue...
enqueue 3 to queue...
enqueue 5 to queue...
dequeue 7 from queue...
dequeue 9 from queue...
BFS: SUCCESS
```

2. Input:

```
6,2
2,9
2,1
6,4
4,3
4,5
8,7
8,10
1,8
1,11
*
21
#
```

Output 1:

```
Enter 1 for DFS searching or 0 for BFS searching in the tree:
1

Searching the target 21
Starting DFS search in input tree:
push root 6 to stack...
pop 6 from stack...
```

```
push 2 to stack...
push 4 to stack...
pop 4 from stack...
push 3 to stack...
push 5 to stack...
pop 5 from stack...
pop 3 from stack...
pop 2 from stack...
push 9 to stack...
push 1 to stack...
pop 1 from stack...
push 8 to stack...
push 11 to stack...
pop 11 from stack...
pop 8 from stack...
push 7 to stack...
push 10 to stack...
pop 10 from stack...
pop 7 from stack...
pop 9 from stack...
DFS: FAILURE
```

Output 2:

```
Enter 1 for DFS searching or 0 for BFS searching in the tree:
0

Searching the target 21
Starting BFS search in input tree:
enqueue root 6 to queue...
dequeue 6 from queue...
enqueue 2 to queue...
enqueue 4 to queue...
dequeue 2 from queue...
enqueue 9 to queue...
enqueue 1 to queue...
dequeue 4 from queue...
enqueue 3 to queue...
enqueue 5 to queue...
dequeue 9 from queue...
dequeue 1 from queue...
enqueue 8 to queue...
enqueue 11 to queue...
dequeue 3 from queue...
dequeue 5 from queue...
dequeue 8 from queue...
enqueue 7 to queue...
enqueue 10 to queue...
dequeue 11 from queue...
dequeue 7 from queue...
dequeue 10 from queue...
BFS: FAILURE
```

3. Input

```
6,2
2,9
2,1
1,8
4,3
4,5
8,7
6,4
8,10
1,11
*
3
#
```

Output 1:

```
Enter 1 for DFS searching or 0 for BFS searching in the tree:
1

Searching the target 3
Starting DFS search in input tree:
push root 6 to stack...
pop 6 from stack...
push 2 to stack...
push 4 to stack...
pop 4 from stack...
push 3 to stack...
push 5 to stack...
pop 5 from stack...
pop 3 from stack...
DFS: SUCCESS
```

Output 2:

```
Enter 1 for DFS searching or 0 for BFS searching in the tree:
0

Searching the target 3
Starting BFS search in input tree:
enqueue root 6 to queue...
dequeue 6 from queue...
enqueue 2 to queue...
enqueue 4 to queue...
dequeue 2 from queue...
enqueue 9 to queue...
enqueue 1 to queue...
dequeue 4 from queue...
enqueue 3 to queue...
enqueue 5 to queue...
```

```
dequeue 9 from queue...
dequeue 1 from queue...
enqueue 8 to queue...
enqueue 11 to queue...
dequeue 3 from queue...
BFS: SUCCESS
```

In general, we observe that if the target node is deeper in the tree, DFS search finds it faster than the BFS search and if the node is not deeper than BFS search finds it faster.