# CSC1001 Tutorial 14

## Tree

Frederick Khasanto (122040014)

7 December 2023

## Binary Tree

A tree where each node has 0-2 children

```
# Node class
class Node:
    def __init__(self,element,parent = None,left = None,right = None):
        self.element = element
        self.parent = parent
        self.left = left
        self.right = right

# Binary Tree class
class LBTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def __len__(self):
        return self.size

    def find_root(self):
        return self.root

    def parent(self,p):
        return p.parent

    def left(self,p):
        return p.left

    def right(self,p):
        return p.right

    def num_child(self,p):
        count = 0
        if p.left is not None:
            count += 1
        if p.right is not None:
```

```
            count += 1
        return count

    def add_root(self,e):
        if self.root is not None:
            print('Root already exists.')
            return None
        self.size = 1
        self.root = Node(e)
        return self.root

    def add_left(self,p,e):
        if p.left is not None:
            print('Left child already exists.')
            return None
        self.size += 1
        p.left = Node(e,p)
        return p.left

    def add_right(self,p,e):
        if p.right is not None:
            print('Right child already exists.')
            return None
        self.size += 1
        p.right = Node(e,p)
        return p.right

    def replace(self,p,e):
        old = p.element
        p.element = e
        return old

    def delete(self,p):
        old = p.parent
        if p.parent.left is p:
            p.parent.left = None
        if p.parent.right is p:
            p.parent.right = None
        return old
```

# Depth First Search (DFS)

A traversal/searching algorithm

Starts at the root and explores ***as deep as possible*** along each branch before backtracking

Implement with recursion

Helpful link: https://brilliant.org/wiki/depth-first-search-dfs/

```
def DFSearch(t):
    if t:
        print(t.element)
    if (t.left is None) and (t.right is None):
        return
    else:
        if t.left is not None:
            DFSearch(t.left)
        if t.right is not None:
            DFSearch(t.right)
```

## Breadth First Search (BFS)

A traversal/searching algorithm

Starts at the root and ***visit all nodes at the same depth*** before going to the next depth

Implement with Queue

Helpful link: https://brilliant.org/wiki/breadth-first-search-bfs/

```
class ListQueue:

    def __init__(self,capacity):
        self.__capacity = capacity
        self.__data = [None] * self.__capacity
        self.__size = 0
        self.__front = 0
        self.__end = 0


    def __len__(self):
        return self.__size

    def is_empty(self):
        return self.__size == 0

    def first(self):
        if self.is_empty():
            print('Queue is empty.')
        else:
            return self.__data[self.__front]

    def dequeue(self):
        if self.is_empty():
            print('Queue is empty.')
            return None
        answer = self.__data[self.__front]
        self.__data[self.__front] = None
        self.__front = (self.__front + 1) % self.__capacity
```

```
        self.__size -= 1
        return answer

    def enqueue(self,e):
        if self.__size == self.__capacity:
            print('The queue is full.')
            return None
        self.__data[self.__end] = e
        self.__end = (self.__end + 1) % self.__capacity
        self.__size += 1

    def __str__(self):
        return str(self.__data)
    def __repr__(self):
        return str(self)

def BFSearch(t):

    q = ListQueue(50)
    q.enqueue(t)

    while q.is_empty() is False:
        cNode = q.dequeue()
        if cNode.left is not None:
            q.enqueue(cNode.left)
        if cNode.right is not None:
            q.enqueue(cNode.right)
        print(cNode.element)
```

# Practice Questions

## Q1: Represent an Expression

An arithmetic expression can be represented using a binary tree, with each node being a number or an operator. Try to draw a tree to represent an expression

```
2 x (5-4)-(8/(1+3))
```

(See slides for the answer)

## Q2: Create a Tree Object

Create an object of a binary tree class, with value of each node equal to that in the binary tree of Q1.

```
tree=LBTree()
t11=tree.add_root('-')
t21=tree.add_left(t11,'*')
t22=tree.add_right(t11,'/')
```

```
t31=tree.add_left(t21,'2')
t32=tree.add_right(t21,'-')
t33=tree.add_left(t22,'8')
t34=tree.add_right(t22,'+')
t41=tree.add_left(t32,'5')
t42=tree.add_right(t32,'4')
t43=tree.add_left(t34,'1')
t44=tree.add_right(t34,'3')
```

## Q3: DFS/BFS

i) Apply the Depth First Search algorithm to the tree you create in Q2, and check the order of elements in the tree printed out.

ii) Modify the Depth First Search program a little bit, define a function that is able to evaluate an expression represented by a binary tree.

iii) You can try i) ii) using Breadth First Search algorithm.

```
print('Depth First Search:')
DFSearch(tree.root)
print('Breadth First Search:')
BFSearch(tree.root)
```

```
Depth First Search:
-
*
2
-
5
4
/
8
+
1
3
Breadth First Search:
-
*
/
2
-
8
+
5
4
1
3
```

## Q4: Path Length of a Tree

The path length of a tree T is the sum of the depths of all positions in T. Describe a linear time method for computing the path length of a tree T.

```
def pathOfTree(p,path):
    if (not p.left) and (not p.right): # leaf node (no children)
        return 0
    else:
        path = path + 1
        if p.left and p.right: # this node has 2 children (left and
right)
            return path + path + pathOfTree(p.left,path) +
pathOfTree(p.right,path)
        elif p.left and (not p.right): # this node only has left child
            return path + pathOfTree(p.left,path)
        elif (not p.left) and p.right: # this node only has right
child
            return path + pathOfTree(p.right,path)

print(pathOfTree(tree.root,0))
```

```
22
```