UCS 1312 Data Structures Lab

A5: Applications of Binary Trees

--Dr. R. Kanchana

Best Practices to be adapted

- Modular design and coding using versions
- Improve readability of code by making the program self-explanatory, giving meaningful names to your variables and functions
- Avoiding global variables

^^

Write algorithms for applications and trace them with an example. Inspect the steps using the diagrammatic representation of the tree.

1. Create an ADT for a binary tree (BinTree.h) (CO2, K3)

   a) Add the following operations:

      Construct, inorder, preorder, postorder, levelorder

   b) Write an application for the following that uses the binary tree (a5binTree.c)

      I. Given an arithmetic expression, convert to postfix (use stack.h)

      II. Represent the postfix expression as a binary tree

      III. Evaluate the expression represented by the tree (Optional – use integer stack and test with expressions involving integers not variables)

      IV. Traverse the tree in inorder, preorder, and postorder. (Level order optionally)

   c) Demonstrate the binary tree operations and applications with the following test cases

      Sample Input: (A + (B/D) * C)

      Inorder: A+B/D*C

      PostOrder: ABD/C*+

      PreOrder: +A*/BDC

      LevelOrder:

```
   +
A       *
     /     C
   B   D
```

****

Hand-out

Construction of Expression Tree:
For constructing an expression tree we use a stack. We loop through input expression and do the following for every character.

1. If a character is an operand push that into the stack
2. If a character is an operator pop two values from the stack make them its child and push the current node again.

In the end, the only element of the stack will be the root of an expression tree.

Evaluating the expression represented by an **expression tree:**

Let t be the expression tree

```
Algorithm solve(t)
If  t is not null then
    If t.value is operand then
        Return  t.value
    A = solve(t.left)
    B = solve(t.right)

    // calculate applies operator 't.value'
    // on A and B, and returns value
    Return calculate(A, B, t.value)
```

//Helper Code for constructing the expression tree and traversing in inorder

```
struct et
{
   char value;
   et* left, *right;
};

// function to check if 'c' is an operator
bool isOperator(char c)
{
   if (c == '+' || c == '-' ||
         c == '*' || c == '/' ||
         c == '^')
      return true;
   return false;
}

// function to do inorder traversal
void inorder(et *t)
{
   if(t)
   {
      inorder(t->left);
      printf("%c ", t->value);
```

```
        inorder(t->right);
    }
}

// function to create a new node
et* newNode(char v)
{
    et *temp = new et;
    temp->left = temp->right = NULL;
    temp->value = v;
    return temp;
};

// Returns root of constructed tree for given
// postfix expression
et* constructTree(char postfix[])
{
    stack<et *> st;
    et *t, *t1, *t2;

    // Traverse through every character of
    // input expression
    for (int i=0; i<strlen(postfix); i++)
    {
        // If operand, simply push into stack
        if (!isOperator(postfix[i]))
        {
            t = newNode(postfix[i]);
            st.push(t);
        }
        else // operator
        {
            t = newNode(postfix[i]);

            // Pop two top nodes
            t1 = st.top(); // Store top
            st.pop();      // Remove top
            t2 = st.top();
            st.pop();

            //  make them children
            t->right = t1;
            t->left = t2;

            // Add this subexpression to stack
            st.push(t);
        }
    }

    //  only element will be root of expression
    // tree
    t = st.top();
```

```
    st.pop();

    return t;
}
```