Practical 9

1. takes a postfix expression and produces a binary expression tree

```
//Han Yao
public static void main(String[] args){
        //String postfix = "ab*c+";
        String postfix = "abcd*+*e/";

        Stack<BinaryTree<String>> stackTree = new Stack<>();

        for(int i = 0; i < postfix.length(); i++){

            if(isOperator(postfix.charAt(i))){

                //Right SubTree
                BinaryTree<String> rightTree = stackTree.pop();

                //Left SubTree
                BinaryTree<String> leftTree = stackTree.pop();

                //Operator
                BinaryTree<String> operatorTree = new
BinaryTree<>();
                operatorTree.setTree("" + postfix.charAt(i),
leftTree, rightTree);

                stackTree.push(operatorTree);

            }else{

                BinaryTree<String> operandTree = new BinaryTree<>();
                operandTree.setTree("" + postfix.charAt(i));
                stackTree.push(operandTree);
            }
        }

        BinaryTree<String> expressionTree = stackTree.pop();

        // display root
        System.out.println("Root of tree contains " +
expressionTree.getRootData());

        // display nodes in postorder
        System.out.println("\nA postorder traversal visits nodes in
this order: ");
        expressionTree.postorderTraverse();
    }

    public static boolean isOperator(Character c){
        switch(c){
```

```
            case '*':
            case '/':
            case '^':
            case '+':
            case '-':
                return true;

            default:
                return false;
        }
    }
```

2. Implement the preorder and postorder traversals of the binary search tree in the Chapter9\binarysearchtree\BinarySearchTree class.

```
//Kah Yee
//Preorder
private class PreorderIterator implements Iterator<T>{

    private QueueInterface<T> queue = new ArrayQueue<T>();

    public PreorderIterator() {
       preorder(root);
    }

    private void preorder(Node treeNode) {
       if (treeNode != null) {
          queue.enqueue(treeNode.data);
          preorder(treeNode.left);
          preorder(treeNode.right);
       }
    }

    @Override
    public boolean hasNext() {
       return !queue.isEmpty();
    }

    @Override
    public T next() {
       if (!queue.isEmpty()) {
          return queue.dequeue();
       } else {
          throw new NoSuchElementException("Illegal call to next(); iterator is after
end of list.");
       }
    }
```

```
    }


    //Postorder
    private class PostorderIterator implements Iterator<T>{

        private QueueInterface<T> queue = new ArrayQueue<T>();

        public PostorderIterator() {
            postorder(root);
        }

        private void postorder(Node treeNode) {
            if (treeNode != null) {
                postorder(treeNode.left);
                postorder(treeNode.right);
                queue.enqueue(treeNode.data);
            }
        }

        @Override
        public boolean hasNext() {
            return !queue.isEmpty();
        }

        @Override
        public T next() {
            if (!queue.isEmpty()) {
                return queue.dequeue();
            } else {
                throw new NoSuchElementException("Illegal call to next(); iterator is after
end of list.");
            }
        }

    }

    //Kuan Xian
3.  private BinaryNode findMin(BinaryNode node){
            if (node.left != null){
                    node = findMin(node.left);
            }

            return node;
    }
```