```
/*
 * Problem 1
 */
// Reverses linked list
Node* reverse(Node* head)
{
    if (head == NULL || head->pNext == NULL)
        return head;

    // Reversing
    Node *pPrev = reverse(head->pNext);
    head->pNext->pNext = head;

    head->pNext = NULL;

        return pPrev;
    }
/*
* Problem 2
*/
// Checks for loop
int isLoop(Node* check)
{
```

```
    // To check if we found a node has been previously seen
    while (check == NULL || check->pNext == NULL)
    {
        if (check->found == 1)
            return 1;

        // Setting that we have checked that node
        check->found = 1;

        // Continuing to search
        check = check->pNext;
    }
    return 0;
}

/*
 * Problem 3
 */
// Enqueuing
void enqueue(int item)
{
    // Pushing first stack to second stack
    while (!stack1.isEmpty())
```

```
{
    // Peeking
    temp = stack1.peek();

    // Popping first stack and pushing to second
    stack1.pop();
    s2.push(temp);
}

// Pushing an item into stack1
stack1.push(item);

// Pushing it all back to the first stack
while (!stack2.isEmpty())
{
    // Peeking
    temp = stack2.peek();


    // Popping second stack and pushing to first
    stack2.pop();
    stack1.push(item);
}
```

```
}

/*
 * Problem 4
 */
// Dequeuing
void dequeue()
{
    if (stack1.isEmpty())
    {
        cout << "queue is empty" << endl;
        exit();
    }

    front = NULL;

    while (data.peek() != NULL)
    {
        // Peeking the first item
        front = data.peek();

        // Popping front item then pushing the next item stack
        data.pop();
```

```
        data.push(front);
    }
}

/*
 * Problem 5
 */
// Printing BST
void print(Node* nRoot)
{
    queue<Node*> q;

    if (nRoot == NULL)
        return;

    q.push(nRoot);

    while (!q.isEmpty())
    {
        Node *node = q.front();

        // Popping each item
        q.pop();
```

```
        if (node->pLeft != NULL)
            q.push(node->pLeft);
        if (node->pRight != NULL)
            q.push(node->pRight)
    }
}

/*
 * Problem 6
 */
// Checking if it is an anagram
bool isAna(string s1, string s2)
{
    // First checking if they have the same amount of items in the string
    if (s1.length() != s2.length())
        return false;

    // To get the size of each string
    int size1[256] = { 0 };
    int size2[256] = { 0 };

    // Traversing the strings and finding their nice
```

```
    for (int i = 0; i < s1.length(); i++)
    {
        size1[s1[i]]++;
        size2[s2[i]]++;
    }

    // Checking the size of each
    for (int i = 0; i < 256; i++)
    {
        if (size1[i] != size2[i])
        {
            return false;
        }
    }

    return true;
}
```