# Concept Practice Answer

## Question 1

The following code is supposed to insert two nodes in a linked list (with a single integer member called val and a next pointer). A) Explain why this does notwork. B) Rewrite the lines with changes to fix the bugs.

```
int addItem(iItem* head, int input){
    iItem* newItem;
    newItem = (iItem*)malloc(sizeof(iItem));
    if(newItem == NULL){return 0;}
    newItem->val = input;
    newItem->next = head;
    head = newItem;
    return 1;
}

int main(){
    int choice, input, ret;
    iItem* head = NULL;
    addItem(head, 5);
    addItem(head, 10);
    return 0;
}
```

这段代码的问题在于函数 `addItem` 中修改了传入的指针 `head`，但这个修改不会反映到函数外部。这是因为 C 语言中的参数传递是**值传递**，即在函数内部对指针的修改（如 `head = newItem;`）仅作用于函数内的局部副本，而不会影响主函数中的原始指针。要修复这个问题，我们需要将 `head` 的地址作为参数传入函数，以便在函数内部修改它。因此，应该使用指向指针的指针(`iItem** head`)或者返回新的头节点并更新主函数中的 `head`。

**法一：通过指针的指针修改头节点**：

```
int addItem(iItem** head, int input){
    iItem* newItem = (iItem*)malloc(sizeof(iItem));
    if(newItem == NULL){return 0;}
    newItem->val = input;
    newItem->next = *head;
    *head = newItem;
    return 1;
}

int main(){
    int choice, input, ret;
    iItem* head = NULL;
    addItem(&head, 5);
    addItem(&head, 10);
    return 0;
}
```

**法二：返回新节点并更新头节点：**

```
iItem* addItem(iItem* head, int input){
    iItem* newItem = (iItem*)malloc(sizeof(iItem));
    if(newItem == NULL){return head;} // 返回原头节点，表示失败
    newItem->val = input;
    newItem->next = head;
    return newItem;
}

int main(){
    iItem* head = NULL;
    head = addItem(head, 5);
    head = addItem(head, 10);
    return 0;
}
```

# Question 2

Explain **encapsulation** (scope and visibility of members) in object oriented programming with the following partial example. Which members of the class Circle are accessible from where? Does this depend on where in the program the object of the class is declared?

```
class Circle{
    float x, y;
    float radius;
public:
    Circle(float a, float b, float r){
        x = a;
        y = b;
        radius = r;
    }
    Circle(){
        x = 0;
        y = 0;
        radius = 0;
    }
    float Area(){...;};
    void Move(float, float);
    void Scale(float s);
};

void Circle::Move(float a, float b){...;}
void Circle::Scale(float s){...;}

int main(){
    Circle c(1, 1, 1);
    c.Move(10, 5);
    c.Scale(5);
    cout << "Area " << c.Area() << endl;
    return 0;
```

```
}
```

**可访问性分析**

- **私有成员（private）**：`x`、`y` 和 `radius` 只能在类内部访问。
- **公共成员（public）**：构造函数、`Area()`、`Move()` 和 `Scale()` 可以从类的外部访问。

**是否依赖对象声明的位置?**

No，可访问性完全由访问修饰符决定，而不是对象声明的位置。无论对象是在 `main()` 中还是其他函数中声明，只要尝试访问 `private` 成员都会导致编译错误。

# Question 3

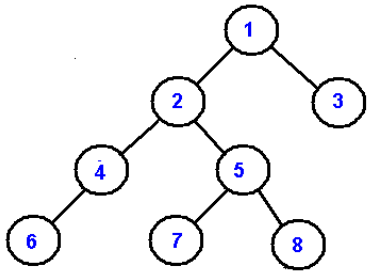
For a nested subroutine in LC-3, register **R7** must be caller-saved / callee-saved (choose one).

在 LC-3 架构中，寄存器 `R7` 是用于保存返回地址的寄存器。当调用一个子程序（subroutine）时，`R7` 会被修改以保存当前指令地址的下一条地址（即返回地址）。

如果一个子程序调用了另一个子程序（嵌套调用），则第一个子程序必须确保 `R7` 的值在调用完成后能够正确恢复。因此，`R7` **必须是 caller-saved**（调用者保存）。

# Question 4

Is the following tree a binary search tree? Explain your reason.




一个简单的方法是通过中序遍历（左 -> 根 -> 右）来检查树是否为二叉搜索树。如果中序遍历的结果是一个递增序列，则该树是二叉搜索树；否则不是。我们通过中序遍历后得到的结果是：6, 4, 2, 7, 5, 8, 1, 3 这很显然不是递增的，因此该树不是 Binary Search Tree。