Classes: A First Look

```
#include <iostream.h>

#define SIZE 10

// Declare a stack class for characters
class stack {
    char stck[SIZE]; // holds the stack
    int tos;                  // index of top-of-stack

public:
    void init();              // initialize stack
    void push(char ch); // push character on stack
    char pop();               // pop character from stack
}
```

```
// Initialize the stack
void stack::init() { tos = 0; }

// Push a character.
void stack::push(char ch) {
    if (tos==SIZE) { cout << “Stack if full”; return; }
    stck[tos] = ch;
    tos++; }

// Pop a character
char stack::pop() {
    if (tos==0) { cout << “Stack is empty”;
                  return 0; // return null on empty stack
                }
    tos--; return stck[tos]; }
```

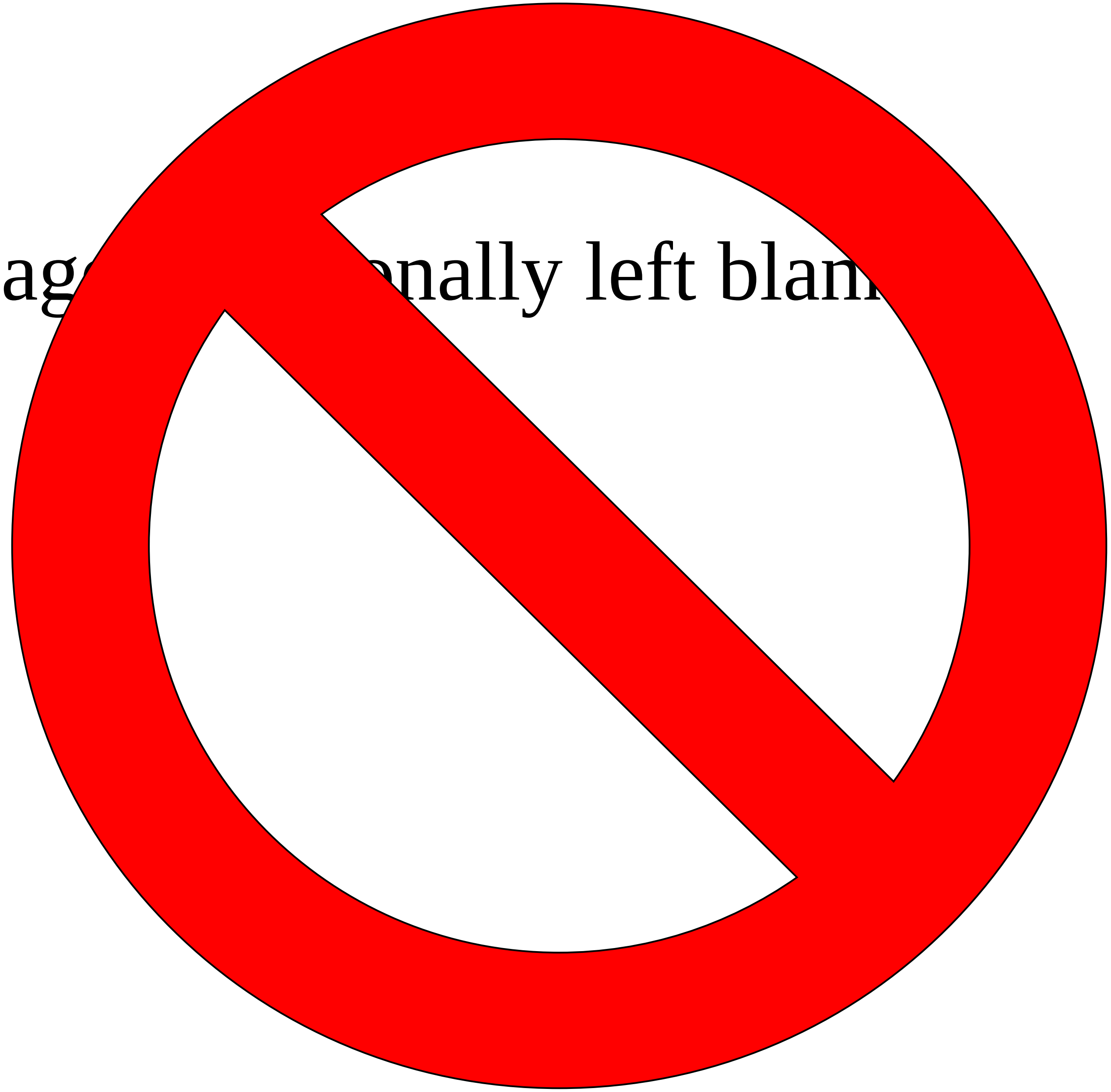
```
main() {
    stack s1, s2; // create two stacks
    int i;
    // initialize the stacks
    s1.init();
    s2.init();

    s1.push(‘a);                s2.push(‘x’);
    s1.push(‘b’);               s2.push(‘y’);
    s1.push(‘c’);               s2.push(‘z’);

    for (i=0; i<3; i++) cout << “Pop s1: “ << s1.pop() << “\n”;
    for (i=0; i<3; i++) cout << “Pop s2: “ << s2.pop() << “\n”;

    return 0;
}
```

This pag onally left blan



HW #3 (Stack, Tree & Polygon)

Part I

- A stack is an abstract data type with two basic operations: insert a new element to the stack (**push**) and remove the element that was most recently inserted to the stack (**pop**).

- In this problem you are given a linked-list implementation of a stack and you have to fill in the push and pop member functions using C++, or Java or Python if you prefer. Each push operation should allocate a new node and each pop operation should de-allocate the corresponding node.

HW #3 (2)

- You are given the definitions for class **node** and **stack.**

```
class node {
public:
    int item;
    node* next;
    node(int x, node* t) { item=x; next=t; }
};

typedef node* nodePtr;
```

HW #3 (3)

```
class stack {
private:
    nodePtr top;
public:
    stack() { top=0; }
    void empty() const {
        if (top==0) cout << "true“ << endl;
        else cout << "false“ << endl; }
    void push(int element) { ---------- // to be filled ---------------
    int pop() { ------------ // to be filled -----------------
};
```

HW #3 (4)

- The following program demonstrates the use of the stack:

```
void main() {
    stack d;
    d.empty();
    d.push(5);
    d.push(6);
    cout << d.pop() << endl;
    d.empty();
    cout << d.pop() << endl;
    d.empty(); }
```

HW #3 (5)

OUTPUT

true

6

false

5

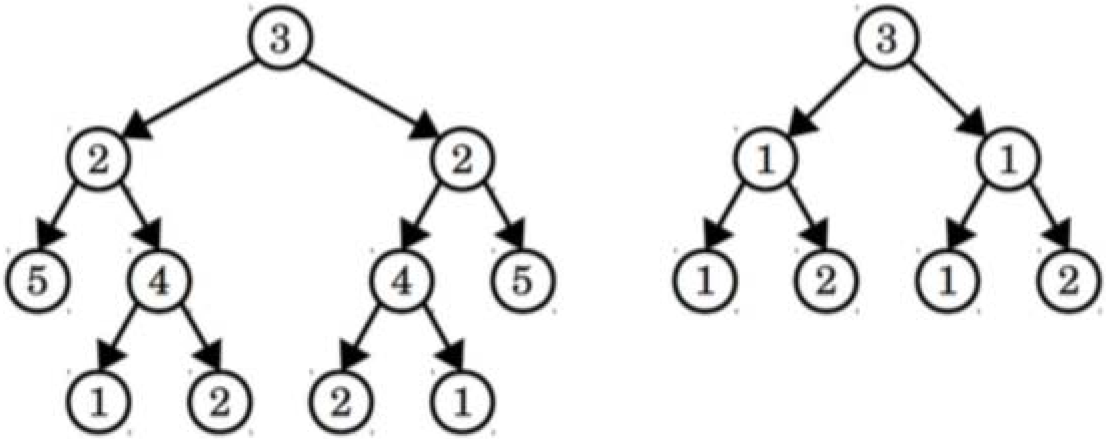
true

- Write the implementation of *void push(int element).*
- Write the implementation of *int pop().*

HW #3 (6)

Part II

- A binary tree is called a **palindromic**回文 **tree** if it is its own mirror image. For example, the tree on the left is a palindromic tree, but the tree on the right is not:



- Write a function that takes in a pointer to the root of a binary tree and returns whether it is a palindrome tree.

HW #3 (7)

```
class Tnode {
public:
    Tnode *left, *right;
    int val; };

typedef Tnode* TnodePtr;
```

- To solve this problem, we will solve a slightly more general problem: given two trees, are they mirrors of one another? We can then check if a tree is a palindrome by seeing whether that tree is a mirror of itself.

HW #3 (8)

class Btree {

private:

TnodePtr root;

bool areMirrors(TnodePtr root1, TnodePtr root2) {

/* If either tree is empty, both must be. */ ----- // to be filled -----

/* Neither tree is empty. The roots must have equal values. */

---------------- // to be filled ----------------

/* To see if they are mirrors, we need to check whether the left sub-tree of the first tree mirrors the right sub-tree of the second tree and vice-versa. */ ---------------- // to be filled ---------------- *}*

public:

bool isPalindromicTree() { return areMirrors(root, root); }

}

HW #3 (9)

Part III

- In this problem, you are required to implement member functions for a class **Polygon** to manipulate a 2D polygon.

- For example, Figure 1 shows some typical polygons. Actually, a polygon can be represented by a circular doubly linked list (see Figure 2). An edge connecting a node (Point) A and B is represented by a next-pointer in node A and a previous-pointer in node B.

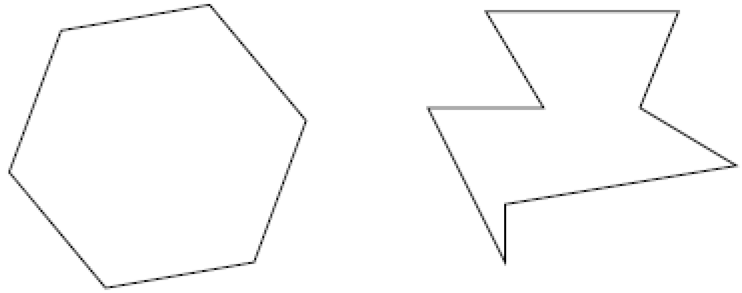
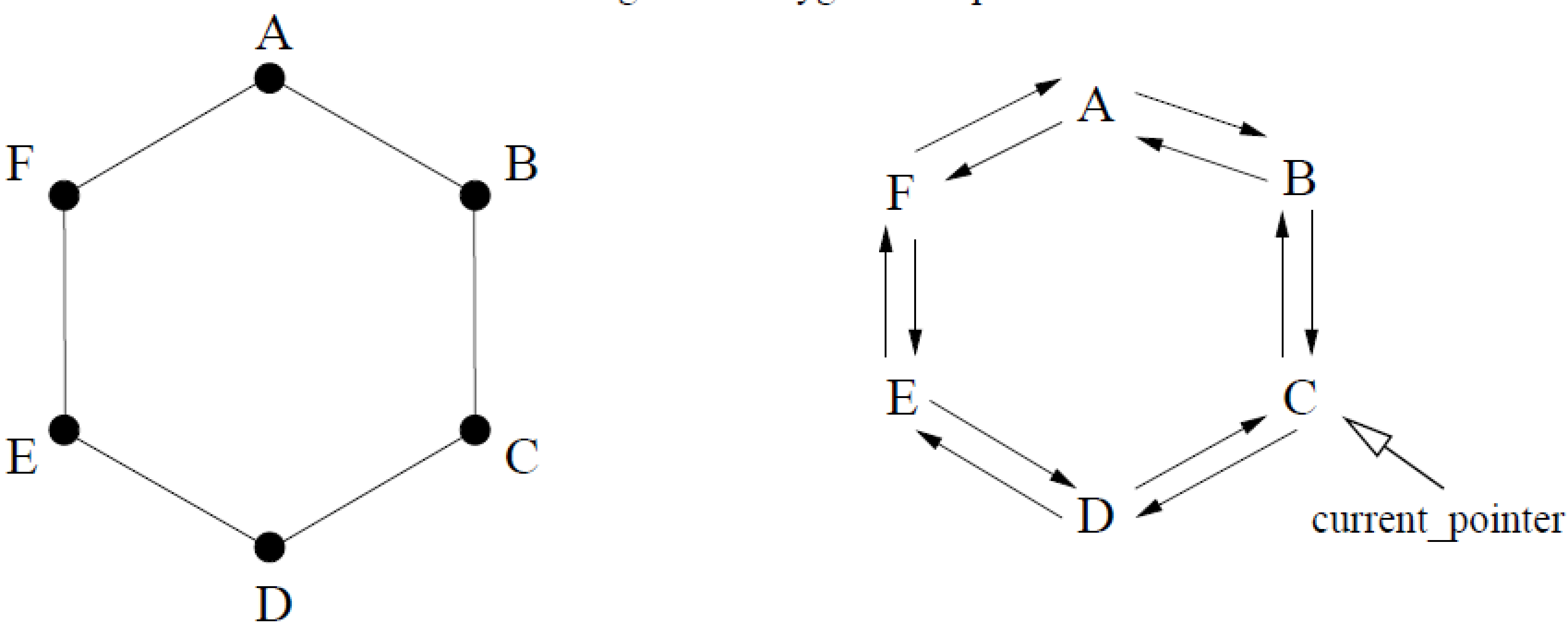


Figure 1: Polygon examples



Polygon

Doubly linked list representation of the Polygon

Figure 2: Polygon representation

HW #3 (10)

- As the result, the **Polygon** class contains a circular doubly linked list that stores the points or vertices of a polygon.
- Here is the structure of a **Point** for a 2D point or vertex:

```
struct Point {
    int x;          // x−coordinate
    int y;          // y−coordinate
};
struct Node {
    Point pt;
    Node *next, *prev; };
```

HW #3 (11)

- And, here is the definition of the **Polygon** class:

class Polygon {

public:

    Polygon();   // constructor

    ~Polygon();   // destructor

// While coding, please insert a line

//   cout << “Constructing Polygon…” << endl;

// and

//   cout << “Deleting Polygon…” << endl;

// in the body of constructor and destructor, respectively.

HW #3 (12)

```
// It takes an array of points and forms a polygon
void setPolygon( Point pts[], int size )
{
    vertexList.clear();
    for( int i=0; i<size; i++ )
    {
        vertexList.insertToNext( pts[i] );
        vertexList.pointToNext();
    }
}
```

HW #3 (13)

```
Polygon* splitPolygon();                        // To be implemented
bool isCollide( Polygon& inPolygon );           // To be implemented
// The input edge is defined by 2 end points − ptA and ptB
// This function returns true if the input edge intersects
// or touches this polygon. Otherwise, it returns false.
// Implemented for you
bool isEdgeIntersect( const Point& ptA, const Point& ptB );

private:
    LinkedList vertexList;      // The circular doubly linked list
};
```

HW #3 (14)

- As you can see, the **polygon** class contains a private variable called **vertexList** which is a **LinkedList** object. This is the circular doubly linked list.
- The definition of the **LinkedList** is defined as follows:

class LinkedList {

public:

    LinkedList();   // constructor

    ~LinkedList();   // destructor

// Please add an informative message in constructor and destructor:

//   cout << “Constructing LinkedList…” << endl;

//   cout << “Deleting LinkedList…” << endl;

HW #3 (15)

int getSize() const;

// return the number of elements (node) of the linked list

bool isEmpty() const; // return true if the list is empty

void clear(); // make the circular doubly linked list empty

void deleteCurrentNode(); // delete the current node. The current

// pointer will point to the next node of the deleted node

void pointToNext();

// make the current_pointer point to the next node

void pointToPrev();

// make the current_pointer point to the previous node

HW #3 (16)

Point getCurrentPoint() const;

// return the Point pointed by the current_pointer

void insertToNext(const Point& pt);

// insert a Point next to the current node

void insertToPrev(const Point& pt);

// insert a Point before the current node

HW #3 (17)

private:

// The current pointer. It points to the current node.
// If the linked list is empty, it equals to NULL.
Node current_pointer;*

};

HW #3 (18)

- The functionalities of the member functions of **LinkedList** are stated in the comments in the class definitions. At your best, write the code for those functions colored in red.

- In this problem, it is assumed that part of the circular doubly linked list has been implemented for you. So, you are free to use those member functions (without the need to know their implementation details).
- While implementing **splitPolygon()** and **isCollide()**, you can use, and can ***only*** use, those member functions declared in red color.

HW #3 (19)

- Implement the member function

  bool Polygon::isCollide(Polygon& inPolygon);

- This function checks whether **this** polygon collides with **inPolygon** or not. If collision occurs, this function returns **true**. Otherwise, it returns **false**.
- To know whether **this** polygon collides with **inPolygon**, one simple way is:
- If one of the polygon contains no vertices, return **false** because there must be no collision.
- For each edge in **inPolygon**, test whether the edge intersects or touches **this** polygon.

HW #3 (20)

- If one or more edges of **inPolygon** intersect or touch **this** polygon, collision occurs and you can return **true** immediately.
- Otherwise, collision does not occur.

- Implement the above pseudo-code. Write your code clearly.

HW #3 (21)

- Implement the member function

  Polygon$*$ Polygon::splitPolygon()

- This function splits the original (**this**) polygon into 2 polygons by the followings:
- Let the size of the original polygon be N. The function will copy $M = floor(N/2) + 1$ consecutive vertices from the **this** polygon (starting from the current pointer position of the **vertexList**). These M consecutive vertices will form a new polygon (by connecting the first and last node) and be returned as the return argument.

HW #3 (22)

- Among the *M* consecutive vertices in the **this** polygon, except the first and the last node, these consecutive vertices will be deleted. And, the first and the last node will be connected.
- Finally, **this** polygon is modified. You should get a new polygon from the return argument.
- When $N \leq 3$, this function returns NULL immediately, in order to prevent error.

- Implement the above pseudo-code. Write your code clearly.