CS F211

Data Structures and Algorithms

Practice Problems

Linked Lists

Problem 1

Write a class `LinkedList` that has three methods: one for adding elements, one for displaying elements, and one for counting the number of inversions in the list. An *inversion* is a pair of elements a_i and a_j such that $i < j$ but $a_i > a_j$.

```
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

class LinkedList {
private:
    Node* head, *tail;

public:
    LinkedList() : head(nullptr), tail(nullptr) {}

    ~LinkedList() {
        Node* current = head;
        while (current) {
            Node* nextNode = current->next;
            delete current;
            current = nextNode;
        }
    }

    void add(int value) {
        Node* newNode = new Node(value);
        if (!head) {
            head = newNode;
            tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
```

```
    }

    int countInversions() const {
        int inversions = 0;
        /* ??? */
        return inversions;
    }

    void display() const {
        Node* temp = head;
        while (temp) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    LinkedList list;

    list.add(5);
    list.add(3);
    list.add(2);
    list.add(1);

    cout << "Linked List: ";
    list.display();

    cout << "Number of inversions: " << list.countInversions
       () << endl;

    return 0;
}
```

Next task: Add an integer attribute that keeps track of the number of inversions. Rewrite the method for adding elements so that the value of the attribute is updated every time a new element is added. The method for counting inversions should be rewritten to simply return the value of this attribute.

Problem 2

Write a class `LinkedList` that maintains a list of `Student` objects. Each student has attributes for name and scores in each 2-2 CS CDC and methods to calculate the total score and to display the name and total score.

The linked list class must have a method that splits the list into two lists: one that stores the list of students whose score is above average while the other stores the rest. The heads of the two lists are saved as separate attributes of the class. The splitting method must make use of a helper method that calculates the average total score of the students in the list. The display method functions differently depending on whether the list has already been split or not.

```
#include <iostream>
#include <string>
using namespace std;

class Student {
/* ??? */
};

class Node {
public:
    Student data;
    Node* next;

    Node(const Student& student) : data(student), next(
        nullptr) {}
};

class LinkedList {
private:
    Node* head;
    Node* aboveAverageHead;
    Node* belowAverageHead;
    bool isSplit;

public:
    LinkedList() : head(nullptr), aboveAverageHead(nullptr),
        belowAverageHead(nullptr), isSplit(false) {}

    void addStudent(const Student& student) {
```

```
28          if (isSplit) {
29              cout << "List has already been split." << endl;
30              return;
31          }
32
33          /* ??? */
34      }
35
36      int calculateAverage() {
37          /* ??? */
38      }
39
40      void split() {
41          if (isSplit) {
42              cout << "List has already been split." << endl;
43              return;
44          }
45
46          double average = calculateAverage();
47
48          /* ??? */
49
50          isSplit = true;
51      }
52
53      void display() const {
54          if (!isSplit) {
55              /* ??? */
56          } else {
57              cout << "\nBelow average:" << endl;
58              Node* temp = aboveAverageHead;
59              while (temp) {
60                  temp->data.display();
61                  temp = temp->next;
62              }
63
64              cout << "\nAbove average:" << endl;
65              temp = belowAverageHead;
66              while (temp) {
67                  temp->data.display();
68                  temp = temp->next;
69              }
70          }
71      }
72  };
```

```
73
74 int main() {
75     LinkedList students;
76     students.addStudent(Student("Alice", 85, 90, 78));
77     students.addStudent(Student("Bob", 70, 65, 80));
78     students.addStudent(Student("Charlie", 88, 92, 95));
79     students.addStudent(Student("Diana", 60, 75, 70));
80
81     cout << "All students:" << endl;
82     students.display();
83
84     students.split();
85     students.display();
86
87     return 0;
88 }
```

Next task: Rewrite the code so that the split method creates two new `LinkedList` objects instead of modifying the links in the current list.

Problem 3

Write a `MagicArray` class that implements a doubly-linked list to allow the user to perform the following transformations.

1. Adding elements to the array.
2. Removing elements from the array.
3. Displaying the array.
4. Reverse the array.
5. Rotating the array by some amount.

```
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* prev;
    Node* next;

    Node(int value) : data(value), prev(nullptr), next(
        nullptr) {}
};

class MagicArray {
private:
    int size;
    Node* head;
    Node* tail;
    bool isReverse;

public:
    MagicArray() : size(0), head(nullptr), tail(nullptr),
        isReverse(false) {}

    void addElement(int x) {
        Node* newNode = new Node(x);
        size++;

```

```
27          if (!head) {
28              head = newNode;
29              tail = newNode;
30              head->next = head;
31              head->prev = head;
32          } else if (!isReverse) {
33              /* ??? */
34          } else {
35              /* ??? */
36          }
37      }
38
39      int popElement() {
40          if (size == 0) {
41              throw runtime_error("Cannot pop from an empty
                    array");
42          }
43
44          size--;
45          int x;
46
47          if (!isReverse) {
48              /* ??? */
49          } else {
50              /* ??? */
51          }
52          return x;
53      }
54
55      void rotate(int k) {
56          if (size == 0) return;
57
58          k %= size;
59          if (k < 0) k += size;
60
61          for (int i = 0; i < k; i++) {
62              if (!isReverse) {
63                  head = head->next;
64                  tail = tail->next;
65              } else {
66                  head = head->prev;
67                  tail = tail->prev;
68              }
69          }
70      }
```

```
71
72      void display() const {
73          if (size == 0) {
74              cout << "Magic Array is empty!\n";
75              return;
76          }
77
78          /* ??? */
79      }
80
81      void reverse() {
82          isReverse = !isReverse;
83      }
84  };
85
86  int main() {
87      MagicArray ma;
88      ma.addElement(1);
89      ma.addElement(2);
90      ma.addElement(3);
91      ma.display();
92
93      ma.reverse();
94      ma.popElement();
95      ma.addElement(4);
96      ma.addElement(5);
97      ma.display();
98
99      ma.rotate(2);
100     ma.reverse();
101     ma.display();
102
103     return 0;
104 }
```

Next task: Under `main`, add code that allows the user to interact with an initially empty `MagicArray` by performing different types of query, with each query type corresponding to a operation that can be performed on the `MagicArray`.

Problem 4

Write a `SortedArray` class that implements a doubly-linked list to always insert elements into a position such that the array remains sorted.

```
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* prev;
    Node* next;

    Node(int value) : data(value), prev(nullptr), next(
       nullptr) {}
};

class SortedArray {
private:
    Node* head;
    Node* tail;

public:
    SortedArray() : head(nullptr), tail(nullptr) {}

    void insert(int value) {
        /* ??? */
    }

    void display() const {
        if (!head) {
            cout << "Sorted Array is empty!\n";
            return;
        }
        Node* temp = head;
        while (temp) {
            cout << temp->data << ' ';
            temp = temp->next;
        }
        cout << '\n';
    }

    ~SortedArray() {
```

```
        Node* temp = head;
        while (temp) {
            Node* nextNode = temp->next;
            delete temp;
            temp = nextNode;
        }
    }
};

int main() {
    SortedArray sa;
    sa.insert(5);
    sa.insert(3);
    sa.insert(8);
    sa.insert(1);
    sa.insert(7);
    sa.display();
    return 0;
}
```

Next task: Modify the display method to include an additional parameter that allows the user to decide the order (ascending or descending) in which the elements must be printed.

Problem 5

Write a class that solves the Josephus problem. In this problem, n people numbered from 1 to n stand in a circle. Starting from person 1, k people are counted, and the last person counted is eliminated. The process then repeats, with counting beginning from the person next to the person who was previously eliminated. This process continues until only one person remains.

Your class must have a method that initializes the setting by creating a circular-linked list of n people. Another method must solve the problem by simulating the process of elimination for a given k value, and then return of the last person's number.

```
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

class JosephusCircle {
private:
    Node* head;
    int size;

public:
    JosephusCircle() : head(nullptr), size(0) {}

    void initialize(int n) {
        size = n;
        head = new Node(1);
        Node* prev = head;

        for (int i = 2; i <= n; i++) {
            Node* newNode = new Node(i);
            prev->next = newNode;
            prev = newNode;
        }
```

```
30          prev->next = head;
31      }
32
33      int solve(int k) {
34          /* ??? */
35      }
36  };
37
38  int main() {
39      JosephusCircle circle;
40      circle.initialize(7);
41      int lastPerson = circle.solve(3);
42      cout << "The last person standing is: " << lastPerson <<
            endl;
43      return 0;
44  }
```

Next task: Write code that simulates a modified version of the Josephus problem, where in instead of a fixed step size, the step size is initially 1 and keeps increasing by 1 in every subsequent turn. Also, add a method that prints the values in the circle, and use this to display the circle after each elimination.