CS F211

Data Structures and Algorithms

Practice Problems

Stacks and Queues

Problem 1

n pairs of twins stand in a circle. The people in the circle are not necessarily standing in a way that all pairs of twins are together. Now, for every pair, a rope is held so that each twin holds one end of the rope. Given this arrangement of $2n$ people, your task is to determine whether two ropes touch. Your code must run in $\mathcal{O}(n)$.

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

struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

class Stack {
private:
    Node* top;

public:
    Stack() {
        top = nullptr;
    }
    ~Stack() {
        while (!isEmpty()) {
            pop();
        }
    }
    void push(int x) {
        Node* newNode = new Node(x);
        newNode->next = top;
        top = newNode;
    }
    int pop() {
        /* ??? */
    }
    int peek() {
        if (top == nullptr) return -1;
        return top->data;
    }
    bool isEmpty() {
        return top == nullptr;
    }
};

bool ropesIntersect(int twins[], int n) {
    Stack stack;

    /* ??? */
}

int main() {
    int n = 3;
    int twins[] = {1, 2, 2, 1, 3, 3};

    if (ropesIntersect(twins, n)) {
        cout << "Intersection found\n";
    } else {
        cout << "No intersection\n";
    }

    return 0;
}
```

Next task: Modify the code so that if two ropes intersect, the numbers corresponding to the two pairs of twins holding the ropes are also output. If multiple pairs of ropes intersect, you can output just one pair.

Problem 2

You are given a 2D grid. Each cell is occupied (represented by `1`) or unoccupied (`0`). Write a program that finds the largest rectangle (by area) of occupied cells within the grid. Your code must run in $\mathcal{O}(mn)$.

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

struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

class Stack {
private:
    Node* top;
public:
    Stack() { top = nullptr; }
    ~Stack() { while (!isEmpty()) pop(); }
    void push(int x) {
        Node* newNode = new Node(x);
        newNode->next = top;
        top = newNode;
    }
    int pop() {
        if (top == nullptr) return -1;
        int val = top->data;
        Node* temp = top;
        top = top->next;
        delete temp;
        return val;
    }
    int peek() { return top ? top->data : -1; }
    bool isEmpty() { return top == nullptr; }
};

int maxRectangle(int grid[][5], int rows, int cols) {
    /* ??? */
}

int main() {
    int grid[4][5] = {
        {1, 0, 1, 1, 1},
        {1, 1, 1, 1, 1},
        {0, 1, 1, 1, 0},
        {1, 1, 0, 1, 1}
    };
    cout << maxRectangle(grid, 4, 5) << endl;
    return 0;
}
```

Next task: In the new task, you must find the largest *pure* rectangle – a pure rectangle being one that only contains occupied cells or only contains unoccupied cells.

Problem 3

There is a deque consisting of n characters. At any given moment, you can pick a character from either end of the deque. This process continues until the deque is empty, after which the characters that you have removed are arranged in the order in which they were picked. Your task is to find out whether it is possible to create a palindrome. Write a recursive function to do the same.

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

struct Node {
    char data;
    Node* next;
    Node* prev;
    Node(char val) : data(val), next(nullptr), prev(nullptr)
       {}
};

class Deque {
private:
    Node* front;
    Node* rear;
public:
    Deque() : front(nullptr), rear(nullptr) {}
    ~Deque() {
        while (!isEmpty()) popFront();
    }
    void pushFront(char x) {
        /* ??? */
    }
    void pushBack(char x) {
        Node* newNode = new Node(x);
        if (isEmpty()) front = rear = newNode;
        else {
            rear->next = newNode;
            newNode->prev = rear;
            rear = newNode;
        }
    }
    char popFront() {
        if (isEmpty()) return '\0';
        char val = front->data;
        Node* temp = front;
        front = front->next;
        if (front) front->prev = nullptr;
        else rear = nullptr;
        delete temp;
        return val;
    }
    char popBack() {
        /* ??? */
    }
    bool isEmpty() { return front == nullptr; }
};

bool isPalindrome(string s) {
    /* ??? */
}

bool canFormPalindrome(Deque& dq, string s = "") {
    /* ??? */
}

int main() {
    Deque dq;
    string input = "rearcca";
    for (char c : input) dq.pushBack(c);
    cout << (canFormPalindrome(dq) ? "YES" : "NO") << endl;
    return 0;
}
```

Next task: Modify the code so that if a palindrome is found, the series of actions (where each action is picking from left or picking from right) that lead to the palindrome is also output.

Problem 4

A hospital consists of n wards, each capable of treating one patient at a time. There are three types of patients: critical, serious, and stable. Each type of patient has a unique class that contains the patient's name and the time required for treatment. The hospital maintains three separate queues for each type of patient.

At every turn, the following happens.

1. A patient may arrive and join their respective queue.
2. A ward reduces the remaining treatment time of its current patient.
3. If a patient's treatment completes, the ward becomes empty.
4. The highest-priority patient (critical ¿ serious ¿ stable) at the front of any queue is admitted into the empty ward.
5. The process continues until all patients are treated.

Write a program that simulates this.

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

class Patient {
public:
    string name;
    int timeRequired;
    Patient *next;
    Patient(string n, int t) : name(n), timeRequired(t), next
        (nullptr) {}
};

class Queue {
public:
    Patient *front, *rear;
    Queue() : front(nullptr), rear(nullptr) {}
    void enqueue(string name, int time) {
        /* ??? */
    }
    Patient* dequeue() {
        if (!front) return nullptr;
        Patient *temp = front;
        front = front->next;
        if (!front) rear = nullptr;
        return temp;
    }
    Patient* peek() { return front; }
    bool isEmpty() { return !front; }
};

class Hospital {
    int numWards;
    Queue critical, serious, stable;
    Patient** wards;
public:
    Hospital(int n) : numWards(n) {
        wards = new Patient*[numWards]();
    }
    ~Hospital() { delete[] wards; }
    void admitPatient(int type, string name, int time) {
        /* ??? */
    }
    void process() {
        /* ??? */
    }
    void display() {
        cout << "Wards: ";
        for (int i = 0; i < numWards; i++) cout << (wards[i]
           ? wards[i]->name : "empty") << " \n"[i == (
           numWards - 1)];
    }
};

int main() {
    int cmd, type, time;
    string name;
    Hospital hospital(10);
    while (true) {
        cin >> cmd;
        if (cmd) {
            cin >> type >> name >> time;
            hospital.admitPatient(type, name, time);
        }
        hospital.process();
        hospital.display();
    }
    return 0;
}
```

Next task: Modify the code to implement `Ward` as a separate class. Also, change the patient selection logic to account for queue length instead of simply taking the highest-priority patient. Each queue is assigned a new *urgency* value, which is the queue's priority (3 for critical, 2 for serious, and 1 for stable) multiplied by the length of the queue. Now, the patient belonging to the queue with the highest urgency value is chosen when a ward empties.

Problem 5

Write a program that uses a queue to generate binary numbers from 1 to n.

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

class Node {
public:
    string data;
    Node* next;
    Node(string val) : data(val), next(nullptr) {}
};

class Queue {
private:
    Node* front;
    Node* rear;
public:
    Queue() : front(nullptr), rear(nullptr) {}

    void enqueue(string val) {
        Node* newNode = new Node(val);
        if (!rear) front = rear = newNode;
        else {
            rear->next = newNode;
            rear = newNode;
        }
    }

    string dequeue() {
        /* ??? */
    }

    bool isEmpty() { return front == nullptr; }
};

void generateBinaryNumbers(int n) {
    /* ??? */
}

int main() {
    int n;
    cout << "Enter n: ";
    cin >> n;
    generateBinaryNumbers(n);
    return 0;
}
```

Next task: Extend the code so that using an additional parameter, the function can print numbers from 1 to n in any base from base-2 to base-36. For bases greater than 10, use letters A-Z to represent digits greater than 9.