CS F211

Data Structures and Algorithms

Practice Problems

Bubble Sort, Trees, and Heaps

Problem 1

Write code that sorts the nodes of a doubly-linked list using bubble sort. Each node of the list contains a string, and the list has to be sorted in lexicographic order.

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

class Node {
public:
    string s;
    Node* prev, *next;

    Node(string _s) : s(_s), next(nullptr) {}

    bool isGreaterThan(Node* other) {
        string r = other->s;
        for (int i = 0; i < min(s.length(), r.length()); i++)
            {
            if (s[i] > r[i]) return true;
            else if (s[i] < r[i]) return false;
        }
        return s.length() > r.length();
    }
};

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

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

    void append(string s) {
        /* ??? */
    }

    void display() {
        Node* node = head;
        for (int i = 0; i < size; i++) {
            cout << node->s << " \n"[i == (size - 1)];
            node = node->next;
```

```
        }
    }

    void sort() {
        /* ??? */
    }
};

int main() {
    int n;
    cin >> n;
    LinkedList* list = new LinkedList();
    for (int i = 0; i < n; i++) {
        string s;
        cin >> s;
        list->append(s);
    }
    cout << "Before sorting: ";
    list->display();
    list->sort();
    cout << "After sorting: ";
    list->display();
    return 0;
}
```

```
>> 4
>> pilani
>> goa
>> hyderabad
>> dubai
Before sorting: pilani goa hyderabad dubai
After sorting: dubai goa hyderabad pilani
```

Next task: The strings in the list must now be sorted in order of their length. Strings of equal length should appear in the sorted list in the same order that they did in the original one.

Problem 2

You are given a rooted tree, in which each node contains 0 or 1. An agent starts at the root of the tree, and at each step, goes to one of its children with equal probability. This continues until the agent reaches a leaf. The values of all the nodes that the agent happens across (including the root and the leaf) are added up. Find the probability that this sum is odd.

The first line of input is n, which is the number of nodes in the tree. The second line contains n integers, representing the values of the nodes. The third line contains $n-1$ integer, where the ith one is the index of the parent of the $i+1$th node. A function that builds the tree in the right-child right-sibling representation is already given.

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

class Node {
public:
    int value;
    Node* child, *sibling;

    Node() : value(0), child(nullptr), sibling(nullptr) {}
};

void treeFromInput(Node** tree, int n) {
    for (int i = 0; i < n; i++) {
        int value;
        cin >> value;
        tree[i] = new Node();
        tree[i]->value = value;
    }
    for (int i = 1; i < n; i++) {
        int parent;
        cin >> parent;
        parent--;
        if (tree[parent]->child != nullptr) tree[i]->sibling
            = tree[parent]->child;
        tree[parent]->child = tree[i];
    }
}

double probabilityOdd(Node* node) {
```

```
    /* ??? */
}

int main() {
    int n;
    cin >> n;
    Node** tree = (Node**)(malloc(sizeof(Node*) * n));
    treeFromInput(tree, n);
    cout << probabilityOdd(tree[0]) << '\n';
    return 0;
}
```

```
>> 7
>> 1 0 1 0 1 0 0
>> 1 1 1 3 3 4
0.833333
```

Next task: Alter the code in such a way that the recursive DFS function calculates the expected value of the sum obtained upon reaching a leaf node.

Problem 3

You are given a rooted tree, which represents cities in India. Amazon wants to setup its warehouse at the 1st node of this tree, which can be considered to be the root of the tree. The cost of delivery is the sum of distances from the warehouse to every other city. Write code that calculates this total distance.

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

class Node {
public:
    int subtreeSize;
    Node* child, *sibling;

    Node() : subtreeSize(0), child(nullptr), sibling(nullptr)
        {}
};

void treeFromInput(Node** tree, int n) {
    for (int i = 0; i < n; i++) tree[i] = new Node();
    for (int i = 1; i < n; i++) {
        int parent;
        cin >> parent;
        parent--;
        if (tree[parent]->child != nullptr) tree[i]->sibling
           = tree[parent]->child;
        tree[parent]->child = tree[i];
    }
}

void calcSubtreeSizes(Node* node) {
    /* ??? */
}

int calcTotalDist(Node* node) {
    /* ??? */
}

int main() {
    int n;
    cin >> n;
    Node** tree = (Node**)(malloc(sizeof(Node*) * n));
```

```
    treeFromInput(tree, n);
    calcSubtreeSizes(tree[0]);
    cout << calcTotalDist(tree[0]) << '\n';
    return 0;
}
```

```
>> 7
>> 1 1 1 3 3 4
9
```

Next task: Amazon is considering moving its warehouse to another city so as to minimize the cost of delivery. Write a function to find the minimum total distance from one city to every other city.

Problem 4

A rock-paper-scissors tournament is being organized in knockout fashion. The tournament features 2^k contestants, and each has a fixed move that they play in every game. In the first round, 1 plays 2, 3 plays 4, and so on until $2^k - 1$ plays 2^k. In the next round, the winners are paired against each other in similar fashion (starting with the winner of 1 versus 2 playing against the winner of 3 versus 4). This continues till the final.

Write code that builds a binary tree whose nodes represent the games of the tournament, with the leaves representing the 2^k contestants, each internal node representing a matchup and its eventual winner, and the root representing the final. Write another function that simulates the tournament. Note that ties are broken with the first contestant being declared as winner.

Input consists of two lines. The first is a single integer k, and the second is a string of length 2^k, where the ith character represents the move that the ith player uses throughout the tournament.

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

class Node {
public:
    int idx;
    char move;
    Node* left, *right;

    Node() : idx(-1), move(0), left(nullptr), right(nullptr)
       {}
};

Node* blankTree(int k, int idx, string s) {
    Node* node = new Node();
    if (k == 0) {
        node->idx = idx + 1;
        node->move = s[idx];
        return node;
    }
    node->left = blankTree(k - 1, (idx * 2), s);
    node->right = blankTree(k - 1, (idx * 2) + 1, s);
    return node;
}
```

```
24
25 void simulateTournament(Node* tree) {
26     /* ??? */
27 }
28
29 int main() {
30     int k, n = 1;
31     cin >> k;
32     for (int k = 0; k < n; k++) n *= 2;
33     string moves;
34     cin >> moves;
35     Node* tree = blankTree(k, 0, moves);
36     simulateTournament(tree);
37     cout << '\n' << tree->idx << " wins the tournament!\n";
38     return 0;
39 }
```

```
1  >> 2
2  >> SPRP
3
4  1 vs 2
5  1 picks S
6  2 picks P
7  1 wins!
8
9  3 vs 4
10 3 picks R
11 4 picks P
12 4 wins!
13
14 1 vs 4
15 1 picks S
16 4 picks P
17 1 wins!
18
19 1 wins the tournament!
```

Next task: Modify the code to reflect a switch in strategy employed by the contestants. Each time a player wins a game and moves on to the next round, the player switches to the move that their current move would beat. For example, if the player played rock in their last game, they would switch to scissors.

Problem 5

You are given a string that represents an arithmetic expression containing addition, subtraction, multiplication, and division. Each operation is binary (performed on two numbers) and is surrounded by parentheses regardless of precedence (except the outermost expression). Each operand is a single-digit integer.

Write a function that converts this expression into a binary tree where leaves are operands and internal nodes are operations. Then, write a function that evaluates the expression using the parse tree.

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

class Node {
public:
    char data;
    int value;
    Node* left, *right;

    Node() : data(0), value(0), left(nullptr), right(nullptr)
        {}
};

void construct(string s, int* p, Node* node) {
    /* ??? */
}

int evaluate(Node* node) {
    /* ??? */
}

int main() {
    string s;
    cin >> s;
    int idx = 0;
    int* p = &idx;
    Node* node = new Node();
    construct(s, p, node);
    cout << evaluate(node) << '\n';
    return 0;
}
```

```
>> 5+((3*2)-4)
7
```

Next task: Modify the code to handle expressions that contain integers with multiple digits.

Problem 6

You are given an array of size 26, representing the frequencies in which the 26 letters of the English alphabet appear in some sample reference text. Use a min-heap to construct a Huffman encoding tree with respect to this frequency data. The tree must then be used to derive Huffman codes for each letter.

The construction algorithm begins with 26 disconnected nodes, each representing a letter. At each step, the two least frequent nodes are combined to form a new node which becomes the parent of the two nodes and whose frequency is the sum of its children's frequencies. This process repeats until only one node is left, which becomes the root of the newly constructed tree. The nodes are stored in a min-heap to ensure efficient retrieval of nodes with minimum frequency.

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

struct Node {
    char letter;
    int freq;
    Node *left, *right;
    Node(char l, int f) : letter(l), freq(f), left(nullptr),
        right(nullptr) {}
};

class MinHeap {
public:
    Node* heap[26];
    int size;
    MinHeap() : size(0) {}

    void push(Node* node) {
        int i = size++;
        while (i > 0 && node->freq < heap[(i - 1) / 2]->freq)
            {
            heap[i] = heap[(i - 1) / 2];
            i = (i - 1) / 2;
        }
        heap[i] = node;
    }

    Node* pop() {
```

```
        if (size == 0) return nullptr;
        Node* minNode = heap[0];
        heap[0] = heap[--size];
        heapify(0);
        return minNode;
    }

    void heapify(int i) {
        int smallest = i, left = 2 * i + 1, right = 2 * i + 2;
        if (left < size && heap[left]->freq < heap[smallest]->freq) smallest = left;
        if (right < size && heap[right]->freq < heap[smallest]->freq) smallest = right;
        if (smallest != i) {
            swap(heap[i], heap[smallest]);
            heapify(smallest);
        }
    }
};

Node* buildHuffmanTree(int freq[26]) {
    /* ??? */
}

void generateCodes(Node* root, string* codes, string curr = "") {
    /* ??? */
}

int main() {
    int freq[26] = {117, 44, 52, 32, 28, 40, 16, 42, 73, 5, 9, 24, 38, 23, 76, 43, 2, 28, 67, 160, 12, 8, 55, 0, 7, 0};
    Node* root = buildHuffmanTree(freq);
    string codes[26];
    generateCodes(root, codes);
    for (int i = 0; i < 26; i++) cout << (char)('A' + i) << ": " << codes[i] << '\n';
    return 0;
}
```

```
A: 011
B: 0001
C: 0011
```

```
...
X: 0100011000
Y: 0100010
Z: 0100011001
```

Next task: Rewrite the code so that instead of using the given frequency array, frequencies are calculated using sample text that is stored in an `sample.txt` file.

Problem 7

You are given an array of n integers. You must print n integers, where the ith integer represents the median of the first i integers. Use two max-heaps to do this efficiently.

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

class MaxHeap {
public:
    int* arr;
    int size;
    int capacity;

    MaxHeap(int cap) {
        capacity = cap;
        arr = new int[cap];
        size = 0;
    }

    ~MaxHeap() { delete[] arr; }

    void push(int val) {
        if (size == capacity) return;
        int i = size++;
        arr[i] = val;
        while (i > 0 && arr[(i - 1) / 2] < arr[i]) {
            swap(arr[i], arr[(i - 1) / 2]);
            i = (i - 1) / 2;
        }
    }

    int pop() {
        if (size == 0) return -1;
        int root = arr[0];
        arr[0] = arr[--size];
        heapify(0);
        return root;
    }

    int top() { return size > 0 ? arr[0] : -1; }

    int getSize() { return size; }
```

```
39
40  private:
41      void heapify(int i) {
42          int largest = i;
43          int left = 2 * i + 1;
44          int right = 2 * i + 2;
45          if (left < size && arr[left] > arr[largest]) largest
                = left;
46          if (right < size && arr[right] > arr[largest])
                largest = right;
47          if (largest != i) {
48              swap(arr[i], arr[largest]);
49              heapify(largest);
50          }
51      }
52  };
53
54  void runningMedian(int n, int arr[]) {
55      MaxHeap left(n), right(n);
56      /* ??? */
57  }
58
59  int main() {
60      int n;
61      cin >> n;
62      int arr[n];
63      for (int i = 0; i < n; i++) cin >> arr[i];
64      runningMedian(n, arr);
65      return 0;
66  }
```

```
1  >> 7
2  >> 3 7 6 1 5 2 4
3  3 3 6 3 5 3 4
```

Next task: Let us define the *k-median* to be the integer such that approximately exactly $\frac{n}{k}$ of the array's elements are less than it. By this definition, 2-median is the normal median. Modify the code to compute k-medians (for a given k) instead of normal medians.