Problem 1: Heaps

(8 Points)

Given the following heap:

The numbers in the heap represent the values stored in the heap. Use the letters next to the circles to answer the questions for this problem about where in the heap a node is located.

You will perform four operations on the heap. Answer the questions related to the state of the heap after each operation. Operations:

1. enqueue(13);
2. dequeue();
3. dequeue();
4. dequeue();

a) (1 point) Is this a min heap or a max heap? ________

b) (2 points) After operation (1), what is the position (a-o) of 13? ________

c) (2 points) After operation (2), what is the position (a-o) of 14? ________

d) (2 points) After operation (3), what is the position (a-o) of 9? ________

e) (2 points) After operation (4), what is the position (a-o) of 50? ________

f) (1 point) After operation (4), what is the position (a-o) of 25? ________
Problem 2: Trees, Evens and Odds

(10 Points)

Consider a binary search tree defined as follows:

```c
struct Tree {
    int value;
    Tree *left;
    Tree *right;
};
```

You will write two functions for this problem. First, write a function, `add()` that will add a node to a binary search tree:

```c
void add(Tree *&node, int value);
```

Second, write a recursive function `evensAndOdds()` that traverses the entire tree and populates two new binary search trees, `evens` and `odds` that hold just the even values and odd values from the original tree respectively:

```c
void evensAndOdds(const Tree *original, Tree *&evens, Tree *&odds);
```

You can assume that the original Tree pointers for `evens` and `odds` are initially set to `nullptr`. You can also assume that there will not be any duplicate values in the tree.

Please answer on the following page
// add(): adds a value to a binary search tree
void add(Tree *&node, int value) {
}

// evensAndOdds()
// Populates the evens tree with the even values from the original tree, and
// populates the odds tree with the odd values from the original tree.
void evensAndOdds(const Tree *original, Tree *&evens, Tree *&odds) {
}
Problem 3: Graph Coloring

(18 Points)

In mathematics and computer science, the *four color theorem* states that given a geographic map (e.g., of the United States, as shown below), no more than four colors are required to color the regions, so that no two regions have the same color. This theorem was first formulated in the mid-1800s, and was not proven until the 1970s, when a computer was used to aid in the proof. In fact, the proof was the first major computer-aided proof in mathematics.

Unfortunately for us, to color a geographic map with four colors, there are 1476 special cases, which you do not have time to write during a 3-hour exam. But, it is far simpler to use five colors, and maps look excellent when colored with five colors. The one other simplifying requirement is that there must be regions in the map that have less than five neighbors.

For this problem, you will color a graph using a five-color palette given in a set of colors. The graph will be undirected, and each vertex of the graph will will represent a distinct region (e.g., a State in the U.S. map), and each edge will represent two regions that share a border.

Your solution must produce a properly colored graph, with no two adjacent regions having the same color (i.e., no two neighbor Vertices may have the same color).

To aid you in writing your code, here is an overview of an algorithm that will work for the problem:

1. Choose one node (state) on the graph with less than five neighbors and remove it from the graph.
2. Choose another node from the updated graph with less than five neighbors and remove it.
3. Continue until you’ve removed all the nodes from the graph.
4. Add the nodes back into the graph in reverse order from which you removed them.
5. Color the added node with a color that is not used by any of its current neighbors.
6. Repeat the previous two steps until you have colored the entire graph.

Please write the following function:

```cpp
void colorGraph(BasicGraph &geoMap, Set<Color> &palette)
```

Notes:

- The `Color` type is an int, and you may directly compare colors using `==` and `!=`.
- You may create helper functions as necessary. Please put any helper functions above other functions that use them in your answer.
- You may use any auxiliary collections (e.g., Map, Set, Vector, Stack, Queue, etc.) to solve the problem.
The vertex class has a function to set the color of a vertex object, and it also has a visited field, which will be set to false for each vertex when your function receives the graph.

The graph you will be given will be properly constructed so that it has enough vertices with less than five neighbors, so it can be solved with a five-color algorithm.

When you remove a vertex from a graph, all of its edges are also removed. So, you must take this into consideration to solve this problem.

Although we have given you an outline of the algorithm above, you will only get full credit if you make wise choices for the data structures you create. E.g., if you pick a data structure that is more complicated than you need, that will not gain full credit.

```cpp
// colorGraph()
// colors the vertices in the graph with a 5-color theorem using the palette of colors

void colorGraph(BasicGraph &geoMap, Set &palette) {
```
Problem 4: Graph Coloring 2

(4 Points)

Given a colored graph, write a function that will return $true$ if the graph is properly colored, and $false$ if the graph is not properly colored, as defined in Problem 3.

In other words, your function should return $true$ if no vertices have the same color as any of their neighbors, and $false$ otherwise.

```cpp
// coloredProperly()
// returns true if each node is a different color
// than all of its neighbors, and returns false
// if any node is the same color as any of its
// neighbors

bool coloredProperly(BasicGraph graph) {
}```
Problem 5: Flatten Tree

(10 Points)

Given a Binary Tree, flatten the tree to a linked-list, \textit{in-place}, and following a pre-order traversal. In other words, do not create any new nodes while flattening the tree, and ensure that the resulting linked list is an in-order traversal of the original tree. The final linked list should be a tree where all the left pointers are nullptr, and the right pointers form a linked list. Given:

Your flattened tree should look like this:

Given the definition of a BST node as follows:

```c
struct Node {
    int value;
    Node *left;
    Node *right;
};
```

Write the following recursive function:

```c
void flattenTree(Node *node);
```

Note: there are iterative solutions to the problem, but you will only receive full credit for a recursive solution.
// flattenTree()
// converts a BST into a linked list in place
void flattenTree(Node *node) {

}
Problem 6: Lazy Sort

(15 Points)

In class, we discussed the fact that Selection Sort is an $O(n^2)$ sort in both best case and worst case, but that it has a nice feature of sorting the elements exactly one at a time as the algorithm runs. I.e., after the first $i$ passes through the list, the first $i$ elements have been sorted exactly from the entire list. This makes it a decent algorithm if you only need the top $i$ elements from the list and you don't care if the rest are sorted.

For this problem, you will write the functions for a LazySort class, which uses a Selection Sort algorithm to perform the work. The LazySort class is a data structure that makes it easier to know what the first N elements are in a list, and uses a Vector as the underlying representation.

The header for the class is shown below:

```cpp
class LazySort {
public:
    // constructor that initializes elements with
    // the initialVec argument
    // and performs any other necessary initialization
    LazySort(Vector<int> initialVec); // constructor

    // returns a Vector with the first N sorted elements
    Vector<int> topN(int n);

    // returns a Vector of all the elements, sorted
    Vector<int> all(); // returns the entire sorted
    // elements

private:
    // The vector of elements, only sorted
    // when necessary
    Vector<int> elements;

    // the last element in the vector that has already
    // been sorted (this should start at 0)
    int nextUnsorted;

    // private function to perform a selection sort
    // from 0->count elements.
    // e.g., if the Vector is currently {5, 1, 3, 8, 2}
    // then selectionSort(3) would only sort the first three
    // elements, and the elements vector would become
    // {1, 2, 3, 8, 5}
    void selectionSort(int count);
};
```

The LazySort class only sorts enough values to produce the Vector needed for the topN() function to be correct. However, (and very important), once elements are sorted, they don't need to be re-sorted, and should not be. In other words, the nextUnsorted variable should be used to ensure that only elements that have not been sorted yet are sorted.

As an example, if a user calls lazySortInstance.topN(20) and then later calls lazySortInstance.topN(10), the second call should not sort anything, as the elements were sorted before.
LazySort::LazySort(Vector initialVec) {

}

Vector LazySort::topN(int n) {
    // sort the top n values, if necessary

    return elements.subList(0,n);
}

Vector LazySort::all() {
    // returns the entire list sorted

    return elements;
}

void LazySort::selectionSort(int count) {
    // performs a selection sort only up to count
    // in the elements Vector