CSC 212 Practice Midterm Exam 2A
Problems marked with (*) are challenging and problems marked with (**) are hard

Your Name: _____

1. (10 points) Implement the `size` functions. Assume `m_head` is the head of a **sorted** linked list containing zero or more elements. Return the number of **unique** elements in the list. Your implementation must run in $\mathcal{O}(n)$ time.

```
class UnorderedSet {
    struct Node {
        Node* next;
        int data;
        // ...
    };

    // ...

    Node* m_head;

    // ...

    static size_t size(const Node* head) {
        // TODO: Implement this function.
    }

public:
    size_t size() const {
        // TODO: Implement this function.
    }
};
```

Solution:

```
class UnorderedSet {
    struct Node {
        Node* next;
        int data;
        // ...
    };

    // ...

    Node* m_head;

    // ...

    // `size(head)` returns the number of unique elements in the sorted
    // sublist headed by `head`.
```

```
    static size_t size(const Node* head) {
        if (head == nullptr) return 0;
        bool is_new = !head->next || head->data != head->next->data;
        return is_new + size(head->next);
    }

public:
    // `size()` returns the number of unique elements in the set.
    size_t size() const {
        return size(m_head);
    }
};
```

2. (10 points) Consider the following queue declaration. Assume the member functions are implemented as efficiently as possible using only the declared member variables. Give a Θ-bounds on the time complexity of `push`, `pop`, `pop` and `size`.

```
class Queue {
    struct Node {
        Node* next;
        int data;
        // ...
    }

    // ...

    Node* m_head;

    // ...

public:
    // ...

    void push(int data);
    void pop();

    int front() const;
    size_t size() const;
}
```

Solution: Assume `m_head` is the front of the queue:

- `push` takes $\Theta(n)$ time.
- `pop` takes $\Theta(1)$ time.
- `front` takes $\Theta(1)$ time.
- `size` takes $\Theta(n)$ time.

This is better than having `m_head` be the back of the queue since otherwise `front` takes $\Theta(n)$ time

3. (10 points) Implement the `is_sorted` function **recursively**. Assume `v` contains zero or more elements. Return `true` if and only if `v` is sorted in non-decreasing order. Your implementation must run in $\mathcal{O}(n)$ time.

```
bool is_sorted(const std::vector<int>& v) {
    // TODO: Implement this function.
}
```

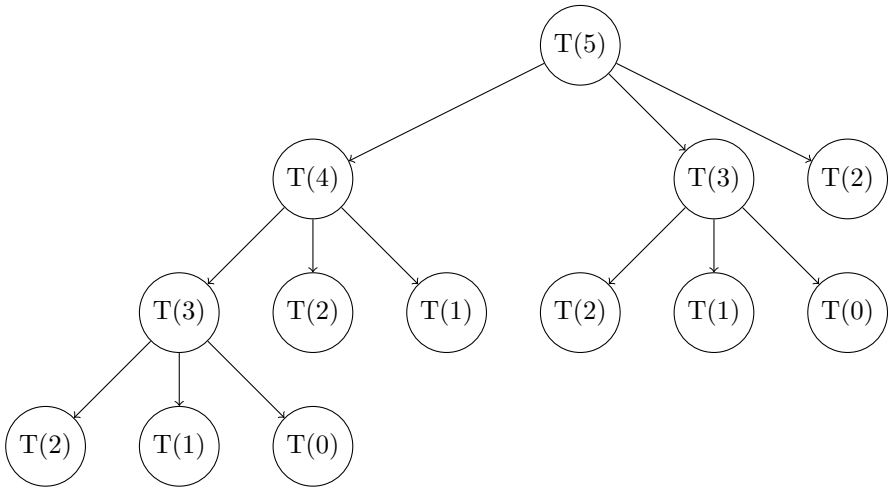
Solution:

```
// `is_sorted(v, i)` returns `true` if and only if `v[i:]` is sorted.
bool is_sorted(const std::vector<int>& v, size_t i) {
    if (i >= v.size() - 1) return true;
    return v[i] <= v[i + 1] && is_sorted(v, i + 1);
}

// `is_sorted(v)` returns `true` if and only if `v` is sorted.
bool is_sorted(const std::vector<int>& v) {
    return is_sorted(v, 0);
}
```

4. (10 points) Draw the recursion tree generated when calling `T(5)`.

```
int T(int n) {
    if (n == 1 || n == 2) return 1;
    if (n == 3) return 2;
    return T(n - 1) + T(n - 2) + T(n - 3);
}
```

Solution:



5. (10 points) Find a closed form for $T(n) = 2T(n-1) + 1$ where $T(0) = 1$ and $n \geq 0$.

Solution:
The first expansion is

$$T(n) = 2T(n-1) + 1$$

The second expansion is

$$T(n) = 2^2 T(n-2) + 2 + 1$$

The third expansion is

$$T(n) = 2^3 T(n-3) + 2^2 + 2 + 1$$

So the k-th expansion is

$$T(n) = 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i$$

To reach the base case $T(0)$, take $n - k = 0$, so $k = n$. Then

$$\begin{aligned} T(n) &= 2^n T(n-n) + \sum_{i=0}^{n-1} 2^i \\ &= 2^n T(0) + 2^{(n-1)+1} - 1 \\ &= 2^n + 2^n - 1 \\ &= 2^{n+1} - 1 \end{aligned}$$

)

6. (10 points) Give a recurrence relation and base case for $L(n)$, the number of leaves in **full** binary tree with n nodes. Assume that $n \geq 1$ and n is odd.

Solution: If two leaves share a parent, call them a leaf pair. In a full binary tree with more than one node, there is always at least one leaf pair. Removing a leaf pair decreases the number of nodes by two (since two nodes are removed), and the number of leaves by one (since two leaves are removed, and their parent becomes a leaf). The resulting tree is still full. Therefore, $L(n) = L(n-2) + 1$ for $n > 1$. When $n = 1$, there is exactly one leaf (the root), so $L(1) = 1$.

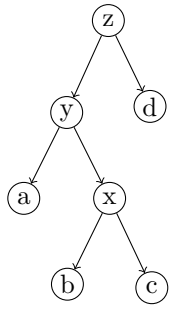
7. (10 points) (*) Implement the `mergesort` functions. Assume `v` contains zero or more elements. The `merge` function takes two sorted subarrays, `v[left:mid]` (the subarray starting at `v[left]` up to but **not** including `v[mid]`) and `v[mid:right]` (the subarray starting at `v[mid]` up to but **not** including `v[right]`), and merges them into a single sorted subarray `v[left:right]` in $\Theta(n)$ time. Your implementation must run in $\mathcal{O}(n \lg n)$ time.

```
void merge(std::vector<int>& v, size_t left, size_t mid, size_t right) {
    // ...
}

void mergesort(std::vector<int>& v, size_t left, size_t right) {
    // TODO: Implement this function.
}

void mergesort(std::vector<int>& v) {
    // TODO: Implement this function.
}
```

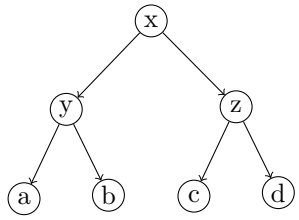
Solution:

```
// `merge(v, left, mid, right)` merges the sorted subarrays `v[left:mid]`
// and `v[mid:right]` into a single sorted subarray `v[left:right]`.
void merge(std::vector<int>& v, size_t left, size_t mid, size_t right) {
    // ...
}

// `mergesort(v, left, right)` sorts the subarray `v[left:right]`.
void mergesort(std::vector<int>& v, size_t left, size_t right) {
    if (right <= left + 1) return;
    size_t mid = left + (right - left) / 2;
    mergesort(v, left, mid);
    mergesort(v, mid, right);
    merge(v, left, mid, right);
}

// `mergesort(v)` sorts the array `v`.
void mergesort(std::vector<int>& v) {
    mergesort(v, 0, v.size());
}
```

8. (10 points) (*) An **LR-rotation** is the following transformation:



(a) Binary tree before LR-rotation at z.



(b) Binary tree after LR-rotation at z.

Implement the `lr_rotate` function. Assume that in the subtree rooted by `root`, `z`, `y`, and `x` are not `nullptr`. Return the new root after rotation. Your implementation must run in $\mathcal{O}(1)$ time.

```
struct Node {
    Node* left;
    Node* right;
    // ...
};

Node* lr_rotate(Node* root) {
    // TODO: Implement this function.
}
```

Solution:

```
struct Node {
    Node* left;
    Node* right;
    // ...
};

Node* lr_rotate(Node* root) {
    Node* z = root;
    Node* y = z->left;
    Node* x = y->right;
    Node* b = x->left;
    Node* c = x->right;
    x->left = y;
    x->right = z;
    y->right = b;
    z->left = c;
}
```

9. (10 points) Insert $4, 7, 1, 9, 0, 6, 3$ into an initially empty B-tree with $m = 3$. Draw the resulting tree after each insertion.

Solution: Check with `https://www.cs.usfca.edu/~galles/visualization/RedBlack.html`.

10. (10 points) Insert $2, 8, 5, 0, 7, 1, 4$ into an initially empty red-black tree. Draw the resulting tree, including colors, after each insertion.

Solution: Check with `https://www.cs.usfca.edu/~galles/visualization/BTree.html`.