

3. List, Sets, Stacks (13 pts. = 1 + 1 + 5 + 6)

Poor Billy Bruin is struggling. He was asked to implement a StackStr class (**stack** of strings) **using** the singly-linked list class (singly<std::string>) described on page 2 of the exam (singly-linked list with tail pointer). His implementation is below and uses composition.

```
class StackStr {
private:
    singly<std::string> dat_; // **singly-linked list with tail pointer**
public:
    StackStr() {}
    // Adds item at the end using dat_'s tail pointer
    void push(const std::string val) { dat_.push_back(val); }
    // Removes the back item
    void pop() { dat_.pop_back(); }
    // retrieves value pointed at by dat_'s tail pointer
    const std::string& top() const { return dat_.back(); }
    bool empty() const { return dat_.empty(); }
};
```

3.1) Rather than composition, Billy may have appropriately used __ **private** ____ (public / private) inheritance.

3.2) If he had used inheritance, occurrences of dat_ in the code above should be replaced by:

__ **singly<string>::_ or this->** ____

He was then asked to implement a SetStr class (**set** of strings). Against Tina Trojan's advice, Billy decided to do so using **ONLY** his StackStr class. Billy's implementation is given below and on the next page. Note: His exists() function has a bug.

```
class SetStr {
public:
    SetStr() {}
    bool empty() const; // assume is written for you
    void insert(std::string val); // given on next page
    bool exists(std::string val); // given on next page (has a bug)
    void erase(std::string val); // assume is written for you
private:
    StackStr store_;
};
```

Using the buggy version of exists(), analyze the runtime of insert() on the next page.

3.3) Examine the code for insert (and exists which has bugs and is incorrect). Neglecting the MAJOR BUG in the code that would prevent $n > 0$ items from ever existing in the set, **ASSUMING there ARE n items in the SetStr at the time insert() is called**, analyze its runtime, $T(n)$. Be sure to account for the StackStr implementation Billy used.

```
void SetStr::insert(std::string val)
{
    if( exists(val) == false) {
        store_.push(val);
    }
}
bool SetStr::exists(std::string val)
{
    while(store_.top() != val)
    {
        store_.pop();
        if(store_.empty()) return false;
    }
    return true;
}
```

Analysis of runtime $T(n)$ for insert(): - Write on the exam page directly below

[To receive any credit, you must show work with appropriate mathematical derivations similar to the homework. Remember, we know the code above has bugs, analyze its runtime as written.]

Final answer: $T(n) = \underline{\theta(n^2)}$

Work:

$$T(n) = \theta(1) + \sum_{i=1}^n i = \theta(n^2)$$

$T(\text{insert}) = T(\text{exists}) + T(\text{push})$

Worst case of exists is when it doesn't exist since it needs to look at all items in the stack.

$T(\text{exists}) \Rightarrow n * [T(\text{pop}) + T(\text{empty}) + T(\text{top})]$

Calls pop_back and on a singly linked are $T(i)$ where i is the length of the list

so first call costs n , next call costs $n-1$, then $n-2$, and so on until the list is empty.

this is the arithmetic sequence. All other operations are constant.

$T(\text{push}) = \text{constant}$

The remainder of the exam including Q3.4 on the next page requires submitting your answer on Gradescope. Do not submit further answers on this paper (it will not be graded). You may use your laptop to login to Gradescope, choose the "MT – Coding" assignment. Links to skeleton .cpp files will be provided in Gradescope (and are reprinted on this exam for visual benefit.

3.4) Now on your laptop, download the **stackset.cpp** file and write a correct implementation of `exists()`. You may **ONLY** use `StackStr`, `ints`, and `bools`. There are no runtime requirements.

// Complete the code on your laptop using the provided skeleton file and submit on Gradescope.

```
bool SetStr::exists(std::string val)
{
    StackStr temp;
    // pop until we find our data, saving in a temp stack
    while(!store_.empty() && store_.top() != val)
    {
        temp.push(store_.top());
        store_.pop();
    }

    bool result = !store_.empty();

    // Now push all the values back to store_
    while(!temp.empty())
    {
        store_.push(temp.top());
        temp.pop();
    }
    return result;

    // OR just make a copy and top/pop the copy

    StackStr temp = store_;
    // pop until we find our data, saving in a temp stack
    while(!temp.empty() && temp.top() != val)
    {
        temp.pop();
    }
    // if temp not empty, we must have found the desired val
    return (!temp.empty());
}
```