Examination 2

Name: **CS 3358, Spring 2026** **Grade:**

Instructions/directions:

(1) You should attempt to answer all questions.

(2) Closed book and closed notes.

(3) 16 questions, 70 total points.

(4) If needing more writing space for a question, use the blank page *opposite the page the question is on*.

(5) Unless specified otherwise, the following items apply to all linked list related problems:

- List is *singly-linked* and each node has the following structure:

```
struct Node
{
   int data;
   Node *link;
};
```

- A pointer-to-head-node provides the handle to the list.
- For an empty list, the pointer-to-head-node contains the null address (null pointer).
- For a non-empty list, the link field of the last node contains the null address (null pointer).
- **DO NOT** assume that there are library (toolkit) functions (of any kind) available for you to use.

1. *(3)* (***Check only one.***) Which of the following techniques best describes *templating*?

   ☐ Encapsulation. ☐ Modularization.
   ☐ Parameterization. ☐ Specialization.

2. *(4)* (***Check all that apply.***) Consider the following function template:

```
template <typename Item>
bool equal(Item a, Item b)
{
   return a == b;
}
```

   What restrictions are placed on the **`Item`** data type for a program that uses the **`equal`** function?

   ☐ The data type must be one of the built-in C++ data types.
   ☐ The data type must have a copy constructor.
   ☐ The data type must have a **<** operator defined.
   ☐ The data type must have a **==** operator defined.

3. *(3)* A container class can be a good candidate for *templating* (*i.e.*, be made a template class using C++ **template** feature) if we want to extend the capability of the class to more broadly support various container objects (created from the class) that *have matching characteristics* (behavior, data structure, features/algorithms, . . .) and *differ only in container's* _____. (***Check only one.***)

   ☐ size variability ☐ item type
   ☐ utilization frequency ☐ user habits

4. *(3)* (***Check only three.***) In the list below, indicate what the ***three*** key components of the STL (Standard Template Library) are:

   ☐ algorithms ☐ iterators
   ☐ generic type `size_t` ☐ standardized namespace `stl`
   ☐ template containers (classes) ☐ special aliases declared using `typedef`

5. *(3)* Suppose you are to use C++ class to implement a container (where order of items is significant) using some kind of underlying data structure (array, linked list, *etc.*). You want to enable the user to methodically step through the container (to process items in the container) but you don't want to expose the underlying data structure to the user (because that breaks encapsulation). Which of the following best fits the purpose? (***Check only one.***)

   ☐ Create an *alias* for *size type*.
   ☐ Create an *alias* for *value type*.
   ☐ Provide appropriate *iterator*s.

6. *(3)* Say you are to solve a problem that requires processing a big collection of data items. A key part of your algorithm involves manipulating the data items according to the *reverse* of a handling order of the data items (such as *backtracking* in orderly fashion the data items dealt with so far). Which ADT, by virtue of its characteristic operating behavior, do you think would be especially useful for implementing that key part of your algorithm? (***Check only one.***)

   ☐ queue ☐ set ☐ stack

7. *(4)* For each statement below, circle **T** or **F** to indicate whether it is **T**rue or **F**alse:
   - **[T F]** The main purpose of having *namespace* is to avoid name conflicts.
   - **[T F]** The *left-inclusive* metaphor is commonly associated with the *iterators* of the STL.

8. *(3)* [☐ True, ☐ False] Suppose you want to solve a problem as efficiently as possible. You have two alternative algorithms available and they are both $O(n^2)$. It doesn't matter which one you choose because they are equally resource efficient at solving the problem.

9. *(3)* Characterize the following code segment *as tightly as can be inferred* using **Big-O**:

```
while (n > 1000)
   n = n / 1000;
```

☐ O(1) ☐ O(log n) ☐ O(n) ☐ O(n log n) ☐ $O(n^2)$ ☐ $O(n^3)$

10. *(3)* Characterize the following code segment *as tightly as can be inferred* using **Big-O**:

```
for (j = 1; j < n; j = j*50)
   for (k = n; k > 0; k = k - 50)
      cout << k << endl;
for (i = 1; i < n; i = i + 50)
   cout << i << endl;
```

☐ O(1) ☐ O(log n) ☐ O(n) ☐ O(n log n) ☐ $O(n^2)$ ☐ $O(n^3)$

11. *(3)* Characterize the following code segment *as tightly as can be inferred* using **Big-O**:

```
for (i = 1; i < n; ++i)
   for (j = i; j >= 1; --j)
      cout << i << endl;
```

☐ O(1) ☐ O(log n) ☐ O(n) ☐ O(n log n) ☐ $O(n^2)$ ☐ $O(n^3)$

12. *(3)* Characterize the following code segment *as tightly as can be inferred* using **Big-O**:

```
for (i = 1; i <= n; ++i)
   if (i%2 != 1 && i != n/4 && i != n-50)
      for (j = 5; j < n; j += 5)
         cout << '*' << endl;
```

☐ O(1) ☐ O(log n) ☐ O(n) ☐ O(n log n) ☐ $O(n^2)$ ☐ $O(n^3)$

13. *(12)* For each linked list function below, write **A**, **B**, **C** or **D** (***only 1***) to indicate major flaw suffered.

A Head (of *original* list) unintentionally lost
B Illegal access of memory already released
C Potential null-pointer exception
D Stale pointer

(NOTE: **A**, **B**, **C** and **D** should *each be used once*.)

`bool HasMoreThanOneNode(Node* head)`
`{`
`  return head->link != 0 && head != 0;`
`}`
	Your Answer: ☐

`void FreeAllNodes(Node*& head)`
`{`
`  Node *leader = head, *trailer = 0;`
`  while (leader != 0)`
`  {`
`  trailer = leader->link;`
`  delete leader;`
`  leader = trailer;`
`  }`
`}`
	Your Answer: ☐

`void PrintList(Node*& head)`
`{`
`  while (head != 0)`
`  {`
`  cout << head->data << endl;`
`  head = head->link;`
`  }`
`}`
	Your Answer: ☐

`void RemoveHeadNodeIfListNotEmpty(Node*& head)`
`{`
`  if (head == 0) return;`
`  delete head;`
`  head = head->link;`
`}`
	Your Answer: ☐

14. *(6)* When implementing a queue with a *circular singly-linked* list (that maintains a *tail pointer*), it is most logical to have the _____ because the ***other choice*** will ***not*** allow us to do _____.

(answer for first blank - ***check only one***)
- ☐ *front* at tail node (thus *rear* at head node)
- ☐ *rear* at tail node (thus *front* at head node)

(answer for second blank - ***check only one***)
- ☐ *front operation* in *constant* time
- ☐ *pop operation* in *constant* time
- ☐ *push operation* in *constant* time

15. *(4)* In ***Assignment 5 Part 2***, you saw the use of a _____ of _____ to greatly facilitate level (breadth-first) traversal of a linked list of linked lists. (`PNode` is *P*arent *Node*; `CNode` is *C*hild *Node*)

(answer for first blank - ***check only one***)
- ☐ queue
- ☐ stack

NOTE:
This part has container in *user*/*client* view, not in *developer*/*implementor* view.

(answer for second blank - ***check only one***)
- ☐ `PNode` (→ holds `PNode` objects)
- ☐ `CNode` (→ holds `CNode` objects)
- ☐ `PNode*` (→ holds addresses of `PNode` objects)
- ☐ `CNode*` (→ holds addresses of `CNode` objects)

16. *(10)* (This question is for a *linked list* of the type described on the cover page, ***Item 5***.) Write C++ function **`FlipPositionsHeadTail`** that manipulates a given list as follows:

- If the given list has *less than 2* nodes, do nothing (*i.e.*, the given list will be unchanged).
- If the given list has *2 or more* nodes, switch the positions of the *head* and *tail* nodes.

Some examples are shown below.

Given list	List after call to function
(empty list)	(empty list)
1	1
1→2	**2→1**
5→6→7→6→5→4→3→**2**	**2**→6→7→6→5→4→3→**5**

IMPORTANT: The function should ***only manipulate pointers***, *i.e.*, it should *not* create any new nodes, destroy any existing nodes, or copy/replace the data item of any node.

NOTE 1: Creating *pointer-to-node*'s (to provide temporary storage for node addresses) does *not* constitute creating new nodes.

NOTE 2: Function should not call any other function.

```
void FlipPositionsHeadTail( _________________ head)
// first fill in above blank with the type for the only 1 argument
{  // then write code for the function body below









}
```