NATIONAL UNIVERSITY OF SINGAPORE
SCHOOL OF COMPUTING

Practical Examination (PE)
TIC2001 Data Structure and Algorithm
23 June 2018 Time Allowed: 1.5 hours

INSTRUCTIONS

1. You are advised to arrive at the venue 10 min before it starts. You will be assigned to a specific seat and the seating plan will be posted in front of your venue.
2. Please leave your student card on your desk throughout the PE.
3. Please remember your coursemology login and password. All submissions are through coursemology.
4. You are only allowed to read this cover page before the start of the PE. Do **not** flip the pages to read the questions inside until you are told to do so.
5. This is NOT an open-book exam. You may only bring one piece of A4 size paper, but **not** electronic devices, including but not limited to laptop, thumb-drive, electronic dictionary and calculator. You are to switch off/silence your mobile phone and **keep it out of view**.
6. You will be logged into a special Windows account at the beginning of the PE. **Do not log off** until the end of the PE. Do not use your own NUSNET account.
7. Please read carefully and follow all instructions in the question. If in doubt, please ask. Raise your hand and the invigilator will attend to you.
8. Any form of communication with other students or the use of unauthorised materials is considered cheating and you are liable to disciplinary action.
9. When you are told to stop, please do so **immediately**, or you will be penalised.
10. **You will be forced to log out at the time the PE ends SHARPLY**. Make sure you save and submit your work a few minutes before it ends.
11. Please check and take your belongings (especially your student card) before you leave.
12. Your program must be compilable! Or you will receive zero mark for correctness.

Important Notes

- Any variables used must be declared within some function. You are **not** allowed to use global variables (variables that are declared outside all the functions). Heavy penalty will be given (see below) if you use any global variable.
- You may write additional function(s) not mentioned in the task statement if you think it is necessary.
- Manage your time well! Do not spend excessive time on any task.
- Please save and backup your code regularly during the PE.
- Tips: It is a bad idea to do major changes to your code at the last 10 minutes of your PE.

Before we start

For this PE, your function will return "-1" if something goes wrong. E.g. if you want to return the first element of a linked list but the list is empty, it returns "-1". So, we assume that we won't insert the number -1 into our data structure.

Problem 1 Stack and Queue

In this task, we will be augmenting the Single Linked List that was similar with Lab Assignment 2 by **adding a tail pointer** and **implementing a Stack and a Queue** using the Single Linked List.

A zipped file of the solution files for MS Visual Studio is provided which contains:

- The Linked List class similar to Lab Assignment 2.
  - `linkedlist.h, linkedlist.hpp`
- The Stack class, which makes use of the Linked List class.
  - `stack.h, stack.hpp`
- The Queue class, which makes use of the Linked List class.
  - `queue.h, queue.hpp`
- The main driver class containing test codes.
  - `main.cpp`

You will submit modifications in the following three files **ONLY**:

- **`linkedlist.hpp, stack.hpp, queue.hpp`**

In our grading, we will assume all other files are not changed. You are allowed to add more helping methods and you should not change existing implemented declarations and implementations.

Task 1: Editing `insertHead()` in LinkedList

To cater for the new tail pointer, you need to augment the `insertHead()` function in `linkedlist.hpp`.

Add two lines of code so the function works properly on a single linked list with a tail pointer.

```
Task 1: Testing LinkedList<int> insertHead().
=============================================
print() Expected Value:
print() Actual Value:
getHead() Expected Value: -1
getHead() Actual Value:   -1
getTail() Expected Value: -1
getTail() Actual Value:   -1

Inserting value 10 to head of linked list.
print() Expected Value: 10
print() Actual Value:   10
getHead() Expected Value: 10
getHead() Actual Value:   10
getTail() Expected Value: 10
getTail() Actual Value:   10

Inserting value 20 to head of linked list.
print() Expected Value: 20 10
print() Actual Value:   20 10
getHead() Expected Value: 20
getHead() Actual Value:   20
getTail() Expected Value: 10
getTail() Actual Value:   10

Inserting value 30 to head of linked list.
print() Expected Value: 30 20 10
print() Actual Value:   30 20 10
getHead() Expected Value: 30
getHead() Actual Value:   30
getTail() Expected Value: 10
getTail() Actual Value:   10
```

Task 2: Editing `deleteHead()` in LinkedList

In addition, you need to augment the `deleteHead()` function in `linkedlist.hpp`.

Add two lines of code so that the function works properly on a single linked list with a tail pointer.

```
Task 2: Testing LinkedList<int> deleteHead().
==========================================
Deleting head from linked list.
print() Expected Value: 20 10
print() Actual Value:   20 10
getHead() Expected Value: 20
getHead() Actual Value:   20
getTail() Expected Value: 10
getTail() Actual Value:   10

Deleting head from linked list.
print() Expected Value: 10
print() Actual Value:   10
getHead() Expected Value: 10
getHead() Actual Value:   10
getTail() Expected Value: 10
getTail() Actual Value:   10

Deleting head from linked list.
print() Expected Value:
print() Actual Value:
getHead() Expected Value: -1
getHead() Actual Value:   -1
getTail() Expected Value: -1
getTail() Actual Value:   -1

Deleting head from empty linked list.

Inserting value 10 to head of linked list.
print() Expected Value: 10
print() Actual Value:   10
```

Task 3: Implementing Stack

With `insertHead()` and `deleteHead()` function correctly implemented, you should be able to make use of these two functions to implement a stack. Implement the `push()`, `peek()` and `pop()` operations in `stack.hpp`.

Note: `peek()` and `pop()` returns the value of a node that was peeked and popped respectively.

```
Task 3: Testing Stack<int> operations.
======================================
print() Expected Value:
print() Actual Value:
peek() Expected Value: -1
peek() Actual Value:   -1
pop() Expected Value: -1
pop() Actual Value:   -1

Pushing value 10 to top of stack.
peek() Expected Value: 10
peek() Actual Value:   10

Pushing value 20 to top of stack.
peek() Expected Value: 20
peek() Actual Value:   20
print() Expected Value: 20 10
print() Actual Value:   20 10

Popping from top of stack.
pop() Expected Value: 20
pop() Actual Value:   20
Popping from top of stack again.
pop() Expected Value: 10
pop() Actual Value:   10
print() Expected Value:
print() Actual Value:
```

Task 4: Implementing `insertTail()` in LinkedList

To implement a queue, we will need to make use of `insertTail()` function, which adds a tail node to the linked list. Implement the function in `linkedlist.hpp`.

```
Task 4: Testing LinkedList<int> insertTail().
==========================================
Inserting value 10 to tail of linked list.
print() Expected Value: 10
print() Actual Value:   10
getHead() Expected Value: 10
getHead() Actual Value:   10
getTail() Expected Value: 10
getTail() Actual Value:   10

Inserting value 20 to tail of linked list.
print() Expected Value: 10 20
print() Actual Value:   10 20
getHead() Expected Value: 10
getHead() Actual Value:   10
getTail() Expected Value: 20
getTail() Actual Value:   20

Inserting value 30 to head of linked list.
print() Expected Value: 10 20 30
print() Actual Value:   10 20 30
getHead() Expected Value: 10
getHead() Actual Value:   10
getTail() Expected Value: 30
getTail() Actual Value:   30
```

Task 5: Implementing Queue

In this solution, we will use the function `enqueue()` to indicate enqueueing an element to the tail of the linked list / queue, and the function `dequeue()` to indicate dequeueing an element from the head of the linked list / queue. Implement the `enqueue()`, `peek()` and `dequeue()` operations in `queue.cpp`.

```
Task 5: Testing Queue<int> operations.
======================================
print() Expected Value:
print() Actual Value:
peek() Expected Value: -1
peek() Actual Value:   -1
poll() Expected Value: -1
poll() Actual Value:   -1

Offering value 10 to end of queue.
peek() Expected Value: 10
peek() Actual Value:   10

Offering value 20 to end of queue.
peek() Expected Value: 10
peek() Actual Value:   10
print() Expected Value: 10 20
print() Actual Value:   10 20

Polling from start of queue.
poll() Expected Value: 10
poll() Actual Value:   10
Polling from top of stack again.
poll() Expected Value: 20
poll() Actual Value:   20
print() Expected Value:
print() Actual Value:
```

Task 6: Implementing `deleteTail()` in LinkedList

To round up the linked list data structure, you will implement the `deleteTail()` function, which deletes the tail node and set the previous node as the new tail. Implement this function in `linkedlist.hpp`.

Hint: As this is a single linked list, this function will run in O(n) time complexity, instead of the usual O(1) time complexity.

```
Task 6: Testing LinkedList<int> deleteTail().
=============================================
Deleting tail from linked list.
print() Expected Value: 30 20
print() Actual Value:   30 20
getHead() Expected Value: 30
getHead() Actual Value:   30
getTail() Expected Value: 20
getTail() Actual Value:   20

Deleting tail from linked list.
print() Expected Value: 30
print() Actual Value:   30
getHead() Expected Value: 30
getHead() Actual Value:   30
getTail() Expected Value: 30
getTail() Actual Value:   30

Deleting tail from linked list.
print() Expected Value:
print() Actual Value:
getHead() Expected Value: -1
getHead() Actual Value:   -1
getTail() Expected Value: -1
getTail() Actual Value:   -1

Deleting tail from empty linked list.

Inserting value 10 to head of linked list.
print() Expected Value: 10
print() Actual Value:   10
```

Problem 2 BST

For Tasks 7 and 8, we will add the rank function to the Binary Search Tree class, and implement operators for a custom class so that we can insert it into a Binary Search Tree.

A zipped file of the solution files for MS Visual Studio is provided which contains:

- The Binary Search Tree class, similar to Lab Assignment 4.
  - `BST.h, BST.hpp`
- The Student class
  - `Student.h, Student.hpp`

You will submit modifications in the following three files **ONLY**:

- `BST.h, BST.hpp, Student.hpp`

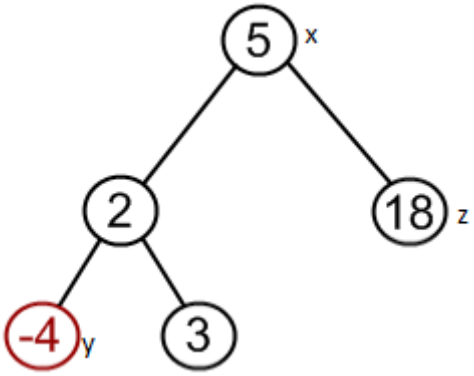
In our grading, we will assume all other files are not changed. You are allowed to add more helping methods to the Binary Search Tree class only. Other than the functions you are supposed to define, do not add or modify any functions in the Student class.

You should not change existing implemented declarations and implementations.

Task 7

For this question, you are supposed to implement the `rank` function. You may introduce new functions, but do not modify or remove any existing functions.

A node in a Binary Search Tree has rank *k* if there are precisely *k* other nodes in the binary search tree with a smaller value. For example, given the binary search tree below



Node *x* has rank 3, because there are exactly 3 nodes smaller than the value contained in node *x* (-4, 2, 3).

Similarly, node *y* has rank 0, because there are no nodes with values smaller than the value contained in node *y*.

Lastly, node *z* has rank 4, because there are exactly 4 nodes smaller than the value contained in node *z.*

Please implement the *`rank(T item)`* function, from the `BinarySearchTree` class (`BST.hpp`).

The function will return

- The rank of the node containing *item*, if such a node exists.
- -1, if such a node does not exist.

You may assume that all the nodes in the binary search tree have unique values.

A sample test case is provided in the `testRankIntegers()` function in main.cpp. For your convenience, we attach this function here, and its corresponding output if the `rank` function is correctly implemented.

```
void testRankIntegers() {

    BinarySearchTree<int> bst;

    for (int i = 0; i < 10; i++) {
        bst.insert(i * 2);
    }

    cout << "The size of the tree is " << bst.size() << endl;

    cout << "Pre-order Traversal:" << endl;
    bst.preOrderPrint();
    cout << "In-order Traversal:" << endl;
    bst.inOrderPrint();
    cout << "Post-order Traversal:" << endl;
    bst.postOrderPrint();

    for (int i = 0; i < 13; i++) {
        cout << "The rank of node containing " << i * 2 << " is " << bst.rank(i * 2) << endl;
    }
}
```

```
The size of the tree is 10
Pre-order Traversal:
6 2 0 4 14 10 8 12 16 18
In-order Traversal:
0 2 4 6 8 10 12 14 16 18
Post-order Traversal:
0 4 2 8 12 10 18 16 14 6
The rank of node containing 0 is 0
The rank of node containing 2 is 1
The rank of node containing 4 is 2
The rank of node containing 6 is 3
The rank of node containing 8 is 4
The rank of node containing 10 is 5
The rank of node containing 12 is 6
The rank of node containing 14 is 7
The rank of node containing 16 is 8
The rank of node containing 18 is 9
The rank of node containing 20 is -1
The rank of node containing 22 is -1
The rank of node containing 24 is -1
Press any key to continue . . .
```

Task 8

When submitting your answer on Coursemology, only submit the functions stated on Coursemology (`operator>`, `operator<` and `operator==`). Do not introduce or modify any other functions.

You may assume that the other functions and classes (including the Binary Search Tree class) is defined and correctly implemented.

You are provided with a `Student` class, with 3 attributes

1. `_name`: Name of the student
2. `_cap`: CAP of the student
3. `_noOfModulesTaken`: Number of modules taken by the student so far.

Your task is to implement the *<, ==,* and *>* operators, to compare two `Student` objects.

Given two `Student` objects, *stu1* and *stu2*

1. *stu1 > stu2*, if
    - *stu1's* CAP is greater than *stu2's* CAP, or
    - If the CAP for both *stu1* and *stu2* are equal, but *stu1* has taken more modules than *stu2, or*
    - If the CAP and number of modules taken by *stu1* and *stu2* are equal, but the dictionary ordering of *stu1's* name is greater than *stu2's name*
2. *stu1 < stu2,* if
    - *stu1's* CAP is less than *stu2's* CAP, or
    - If the CAP for both *stu1* and *stu2* are equal, but *stu1* has taken less modules than *stu2, or*
    - If the CAP and number of modules taken by *stu1* and *stu2* are equal, but the dictionary ordering of *stu1's* name is less than *stu2's name*
3. *stu1 == stu2,* if
    - The CAP, number of modules taken and the name of *stu1* and *stu2* are all the same.

Please implement these operators in `Student.hpp`.

Ensure that the implementation of these operators will allow *Student* objects to be inserted as Tree Nodes into a Binary Search Tree, and we can apply all the Binary Search Tree operations on these nodes.

A sample test case is provided in the `testRankStudent()` function in main.cpp. For your convenience, we attach this function here, and its corresponding output if the operators are correctly implemented.

```
void testRankStudent() {

    Student stu1("John", 3.2, 19);
    Student stu2("Andy", 4.2, 2);
    Student stu3("Joseph", 4.9, 32);
    Student stu4("Tzerbin", 5.0, 21);
    Student stu5("Dezhang", 1.0, 33);

    Student sameCap1("SameCap1", 4.0, 2);
    Student sameCap2("SameCap2", 4.0, 28);

    Student sameCapSameModule1("SameCapSameModule1", 4.0, 28);
    Student sameCapSameModule2("SameCapSameModule2", 4.0, 28);

    BinarySearchTree<Student> bsts;

    Student arr[] = { stu1, stu2, stu3, stu4, stu5, sameCap1, sameCap2, sameCapSameModule1, sameCapSameModule2 };

    for (Student s : arr) {
        bsts.insert(s);
    }

    cout << "In-order Traversal:" << endl;
    bsts.inOrderPrint();

    for (Student s : arr) {
        cout << s.name() << "'s rank = " << bsts.rank(s) << endl;;

    }

    Student notInside("Not Inside BST", 4.0, 28);
    cout << notInside.name() << "'s rank = " << bsts.rank(notInside) << endl;;

    cout << endl << endl;


}
```

```
In-order Traversal:
|| Name: Dezhang CAP: 1 No Of Modules: 33 || Name: John CAP: 3.2 No Of Modules: 19 ||
 Name: SameCap1 CAP: 4 No Of Modules: 2 || Name: SameCap2 CAP: 4 No Of Modules: 28 ||
 Name: SameCapSameModule1 CAP: 4 No Of Modules: 28 || Name: SameCapSameModule2 CAP: 4
 No Of Modules: 28 || Name: Andy CAP: 4.2 No Of Modules: 2 || Name: Joseph CAP: 4.9 N
o Of Modules: 32 || Name: Tzerbin CAP: 5 No Of Modules: 21
John's rank = 1
Andy's rank = 6
Joseph's rank = 7
Tzerbin's rank = 8
Dezhang's rank = 0
SameCap1's rank = 2
SameCap2's rank = 3
SameCapSameModule1's rank = 4
SameCapSameModule2's rank = 5
Not Inside BST's rank = -1
```

Again, the test cases on Coursemology assumes that the Binary Search Tree functions (including `rank`) is correctly implemented. In other words, even if you do not manage to define the `rank` function correctly in Task 7, you should be able to pass the test cases on Coursemology for Task 8 if your operators are correctly defined.