TIC2001 Data Structure and Algorithm PE
12th Nov 2020 Time Allowed: 2 hours

General

- Please make sure your code can be compiled and runnable after you submitted to Couresmology. Any crashing code will be zero mark.
- You should stop modifying your code and start submitting 10 min before the end of the PE in order to avoid the "jam" on Coursemology. Actually, you should submit a version after every time you finished a task. It's your own responsibility to submit the code to Coursemology on time.
- You cannot add more functions or global variables. You cannot include other extra libraries such as STL.

Project/File Settings

You will receive the following files in your project file:

- `simpleLinkedListTemplate.{h,cpp}`, `Graph.{h,cpp}`, `StackAndQueue.h`, `book.h`, and,
- `main.cpp` (you only need to modify this)

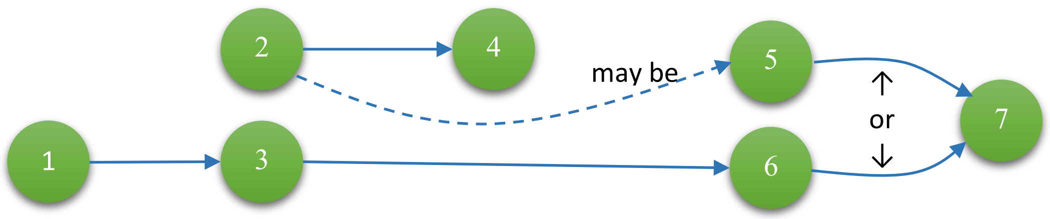
You only need to submit the "zoned" portion in the `main.cpp`. Please look for the comments with "`Submission Zone`" in it. More specifically, you have to submit the following functions *only*:

- `List<T>::insertTail`, `Stack<T>::push()`, `Stack<T>::pop()`, `Queue<T>::enqueue()`, `Queue<T>::dequeue()`, `qBalancedBrackets()`, `Graph::DFS()`, `Graph::BFS()` and `Graph::nComponents()`.
- You have to submit the function `exist()` and `removeTail()` in `List<T>` but you can submit it without modification if you do not use them. These last two functions will not be graded.

And you have the following **SEVEN** tasks to do in this PE. **Please plan your strategy accordingly**.

1. Implement `insertTail()` in the Linked List. (10 marks)
2. Implement `push()` and `pop()` for the template class `Stack`. (10 marks)
3. Implement `enqueue()` and `dequeue()` for the template class `Queue`. (10 marks)
4. Check if a math expression has balanced brackets, namely '`(`', '`)`', '`{`' and '`}`'. (20 marks)
5. Implement DFS in an *unweighted undirected* graph. (20 marks)
6. Implement BFS in an *unweighted undirected* graph. (20 marks)
7. Implement the function to compute the number of connected components in an undirected graph. (10 marks)

To help you, this is the suggested *task dependency graph*, namely, what task(s) you need to finish before which. E.g. the *first* tasks you can start with are Tasks 1, 2 or 5.



In `main()` in `main.cpp`, you will find a lot of test case functions. You can uncomment them one by one to test your Tasks. (But you do not submit them onto Coursemology.)

Given Code/Class Descriptions

The Linked List class `List<T>`

It's basically the same as your Assignment 2. With the following additions:

- The iterator functions `start()`,`next()`,`end()`,`current()` as in Assignment 5
- A tail pointer `_tail`. The pointer will point to the last node of the list. We already set the pointer correctly for you in all the functions except `insertTail()`.

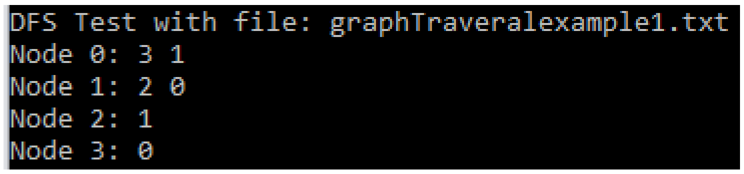
Not important changes:

- Corrected the bug in the function `headItem()`. Now it will correctly return the item of the node instead of the node itself.
- Changed the display function `print()` to print with or without spaces for the list by the parameter.
- Added assertions for you to know which functions went wrong if your code crashed in this part

You can assume our test cases will never make you return the head item or remove a node if the list is empty.

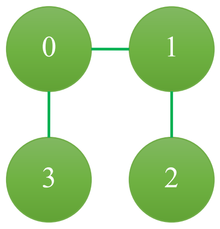
The Graph class `Graph`

Basically it's the same class in Assignment 5. However, we are using ***unweighted undirected graph*** in this PE so we will add two directed edges for an edge in an undirected graph. Our test function will read a file and set up the class for you and you do not have to worry about this. The graphs use adjacency lists and we will print out for you in each test cases like this:



```
DFS Test with file: graphTraveralexample1.txt
Node 0: 3 1
Node 1: 2 0
Node 2: 1
Node 3: 0
```

$\Leftrightarrow$



So each line is the adjacency list of that node. For example, the neighbors of Node 1 are Nodes 0 and 2. However, the order in the adjacency list is **important** because that will **decide** the order of traversal in our DFS and BFS. E.g. Starting from Node 0, you should traverse to Node 3 before Node 1 *according to the order in the adjacency list*.

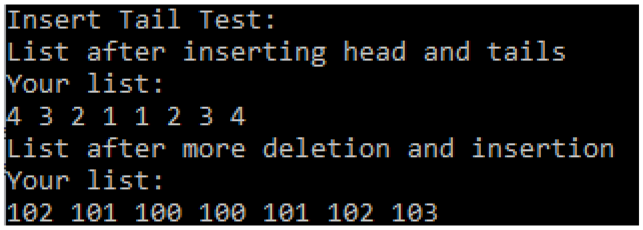
Also, to help you, we added an array `_visited` in the class `Graph` for you to keep track of the node visitation. The value of `_visited[i]` is set to be `true` if Node `i` is visited and `false` otherwise.

- `_nVisited`: The number of entry in the array `_visited` that is set to one. Namely, keep track of how many nodes are visited.
- `_resetVisited()`: To reset the array such that all nodes are not visited.
- `_setVisisted(int i)`: Set Node `i` to be visited
- `_isVisited(int i)`: return `true` if Node `i` is visited, `false` otherwise.

You can assume the graphs provided in this PE are set correctly for you, e.g. if there are only 10 nodes in the graph, we will not have an edge from Node 20 to 30.

Task 1 `InsertTail()` (10 marks)

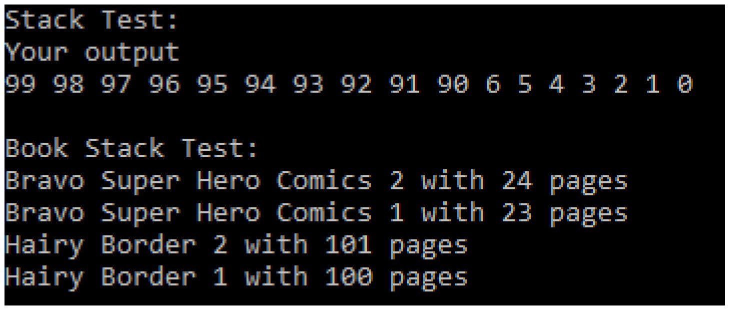
Implement the function `insertTail(T i)` in the template class `List` such that it will insert the item `i` at the end of the list. Here is the sample output of `insertTailTest()`:



```
Insert Tail Test:
List after inserting head and tails
Your list:
4 3 2 1 1 2 3 4
List after more deletion and insertion
Your list:
102 101 100 100 101 102 103
```

Task 2 Implement the Template Class Stack (10 marks)

Implement the template class by filling in the body for `push()` and `pop()` of `Stack<T>`. Here are the sample output for `stackTest()` and `bookStackTest()`:

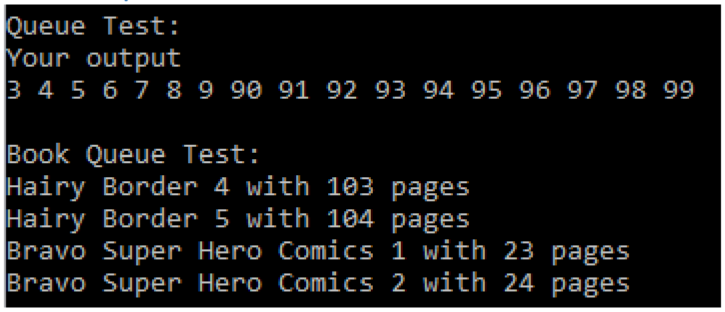


```
Stack Test:
Your output
99 98 97 96 95 94 93 92 91 90 6 5 4 3 2 1 0

Book Stack Test:
Bravo Super Hero Comics 2 with 24 pages
Bravo Super Hero Comics 1 with 23 pages
Hairy Border 2 with 101 pages
Hairy Border 1 with 100 pages
```

Task 3 Implement the Template Class Queue (10 marks)

Implement the template class by filling in the body for `enqueue()` and `dequeue()` of `Queue<T>`. Here are the sample output for `queueTest()` and `bookQueueTest()`:

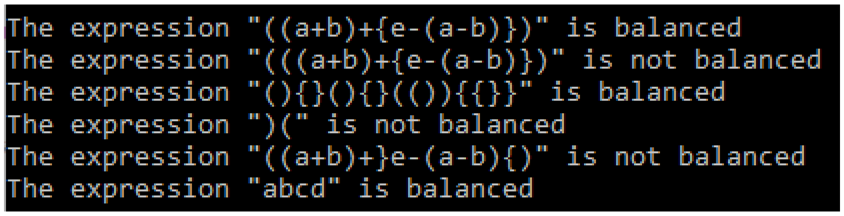


```
Queue Test:
Your output
3 4 5 6 7 8 9 90 91 92 93 94 95 96 97 98 99

Book Queue Test:
Hairy Border 4 with 103 pages
Hairy Border 5 with 104 pages
Bravo Super Hero Comics 1 with 23 pages
Bravo Super Hero Comics 2 with 24 pages
```

Task 4 Checking Balanced Brackets (20 marks)

Given an expression string `exp`, write a function `int qBalancedBrackets(string exp)` to examine whether the pairs and the orders of "`{`", "`}`", "`(`", "`)`" are correct in `exp`. Return 1 if it's balanced and 0 otherwise.



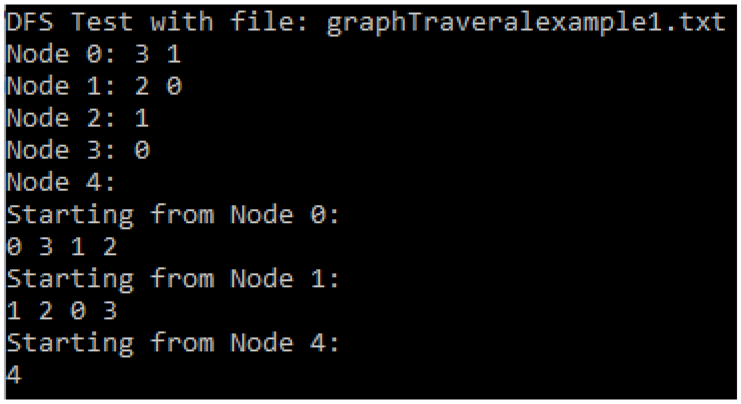
```
The expression "((a+b)+{e-(a-b)})" is balanced
The expression "(((a+b)+{e-(a-b)})" is not balanced
The expression "(){}(){}(()){{}}" is balanced
The expression ")(" is not balanced
The expression "((a+b)+}e-(a-b){)" is not balanced
The expression "abcd" is balanced
```

You only have to check the brackets, no need to make sure the rest are correct.

Task 5 DFS From a node within its own component. (20 marks)

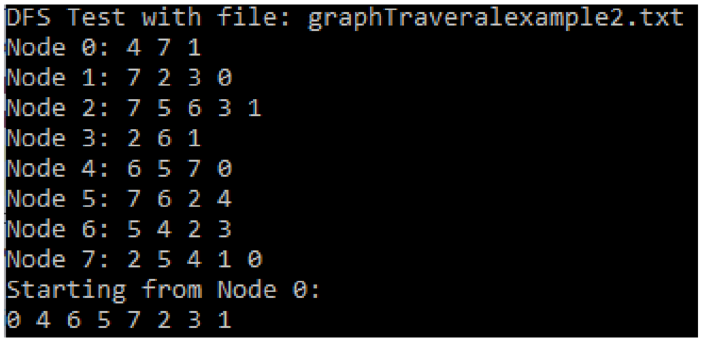
Given a graph, implement Depth First Search starting from a node `s` within its ***own component*** in the function `Graph::DFS()`. Namely, you do not need to traverse the whole graph if the graph has more than one component. For example, this example graph on the right has two components and Node 4 is isolated.

For the graph on the right, if we start DFS from Node 0, we will follow its neighbor Node 3 (dead end) > Node 1 > Node 2 (dead end) so the order is "0 3 1 2". Notice that ***the order in the adjacency list is important***. E.g. when we start with Node 0, we will go to Node 3 before Node 1 so that the final DFS traversal order will be "0 3 1 2" but NOT "0 1 2 3".

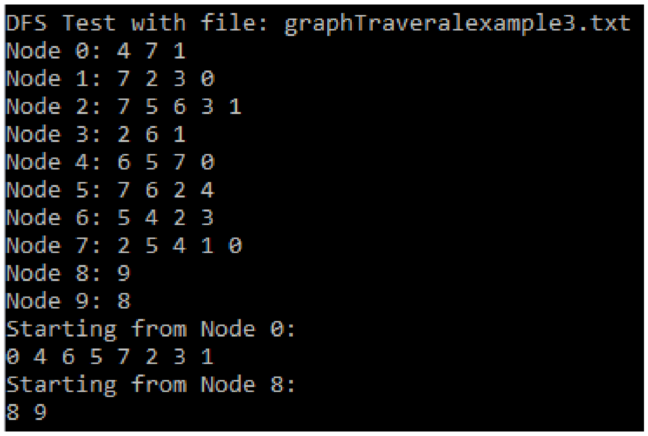


```
DFS Test with file: graphTraveralexample1.txt
Node 0: 3 1
Node 1: 2 0
Node 2: 1
Node 3: 0
Node 4:
Starting from Node 0:
0 3 1 2
Starting from Node 1:
1 2 0 3
Starting from Node 4:
4
```

Here are some more sample output:



```
DFS Test with file: graphTraveralexample2.txt
Node 0: 4 7 1
Node 1: 7 2 3 0
Node 2: 7 5 6 3 1
Node 3: 2 6 1
Node 4: 6 5 7 0
Node 5: 7 6 2 4
Node 6: 5 4 2 3
Node 7: 2 5 4 1 0
Starting from Node 0:
0 4 6 5 7 2 3 1
```



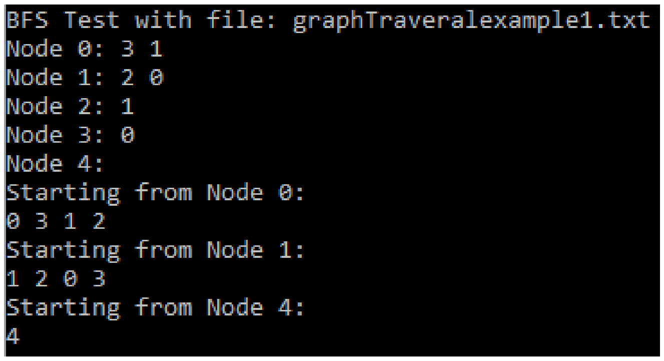
```
DFS Test with file: graphTraveralexample3.txt
Node 0: 4 7 1
Node 1: 7 2 3 0
Node 2: 7 5 6 3 1
Node 3: 2 6 1
Node 4: 6 5 7 0
Node 5: 7 6 2 4
Node 6: 5 4 2 3
Node 7: 2 5 4 1 0
Node 8: 9
Node 9: 8
Starting from Node 0:
0 4 6 5 7 2 3 1
Starting from Node 8:
8 9
```

The three parameters of the function DFS are:

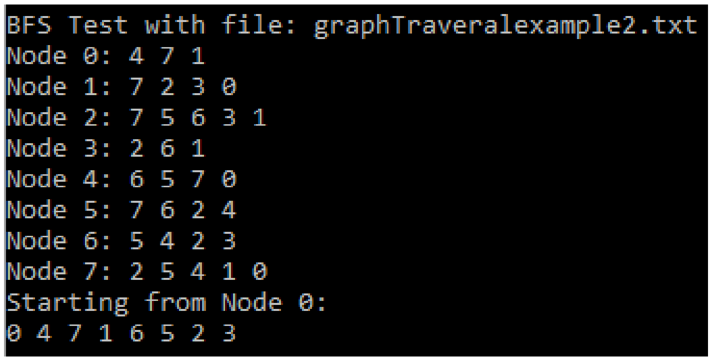
- `int s`: the starting node of the traversal
- `List<int>& output`: Your output. You should put the order of the DFS traversal in this list output such that `output.print()` will print your DFS traversal order. You can assume `output` was empty before DFS.
- `bool resetVisited`: Help you to decide if you want to reset the `_visited` array to be all unvisited or not when `DFS()` is called. We assume we will call your function with `resetVisited` equals to `true` to start with.

Task 6 BFS From a node within its own component. (20 marks)

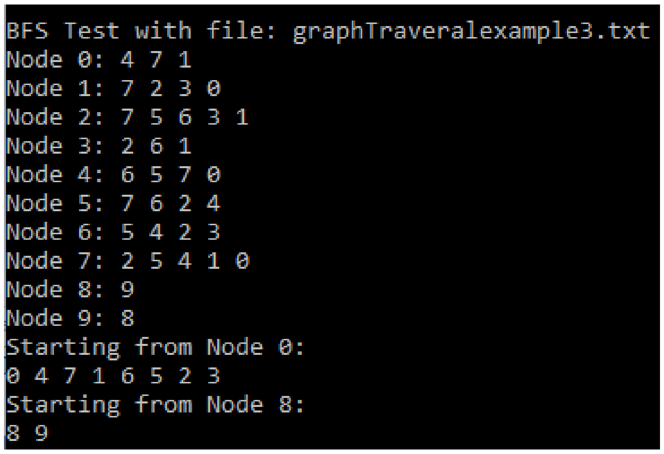
Implement Breath First Search in a graph similar to the previous task. Here are the sample outputs.



```
BFS Test with file: graphTraveralexample1.txt
Node 0: 3 1
Node 1: 2 0
Node 2: 1
Node 3: 0
Node 4:
Starting from Node 0:
0 3 1 2
Starting from Node 1:
1 2 0 3
Starting from Node 4:
4
```



```
BFS Test with file: graphTraveralexample2.txt
Node 0: 4 7 1
Node 1: 7 2 3 0
Node 2: 7 5 6 3 1
Node 3: 2 6 1
Node 4: 6 5 7 0
Node 5: 7 6 2 4
Node 6: 5 4 2 3
Node 7: 2 5 4 1 0
Starting from Node 0:
0 4 7 1 6 5 2 3
```



```
BFS Test with file: graphTraveralexample3.txt
Node 0: 4 7 1
Node 1: 7 2 3 0
Node 2: 7 5 6 3 1
Node 3: 2 6 1
Node 4: 6 5 7 0
Node 5: 7 6 2 4
Node 6: 5 4 2 3
Node 7: 2 5 4 1 0
Node 8: 9
Node 9: 8
Starting from Node 0:
0 4 7 1 6 5 2 3
Starting from Node 8:
8 9
```

Task 7 Computing the Number of Component in a graph (10 marks)

Implement the function `nComponents()` to return the number of connected components of a graph. The numbers of connected components in the three example files, “`cexample1.txt`”, “`cexample2.txt`” and “`cexample3.txt`” are 2, 1 and 3 respectively.

- End of Paper -