CS2040C Data Structures and Algorithm PE AY2023/24 Sem 2
Time Allowed: 2 hours

General

- Please make sure your code can be compiled and run before you submit it to Examplify and Couresmology. Any crashing code will receive a zero mark.
- You cannot add global variables.
- We have already included the necessary libraries for you in each part. Do not include any further libraries (i.e. do not submit any #include directives).
  - Specifically for Parts 1 and 2 only fundamental elements of C++ (primitives, pointers, and string) are allowed, **no STL in parts 1 and 2**.
  - For **Part 3 only, you may additionally use the STL queue library**. We have included the usage of the queue library in the Appendix.
  - Your unit test code may use STL or system libraries at your discretion. You will notice that the provided unit tests make use of some STL libraries. This is not required, but may be helpful.
- You must download the skeleton zip file 2324S2_PE.zip. **Password**: `KnightQuest2024`

Allowed Resources

- Build environment that supports CMake, C++17, and googletest without using the internet.
  - You may use any build environment of your choice (e.g. MSVC, XCode, gcc, llvm, clang, etc) including debuggers, but it is your responsibility to ensure they are compatible before starting the exam. If your build environment worked for the assignments it will work for the exam.
- 1 A4 size cheat sheet and 1 blank piece of paper.
- Lectures and/or lab slides, and **your own** lecture notes and past assignments.
- Local build system documentation

Disallowed Resources

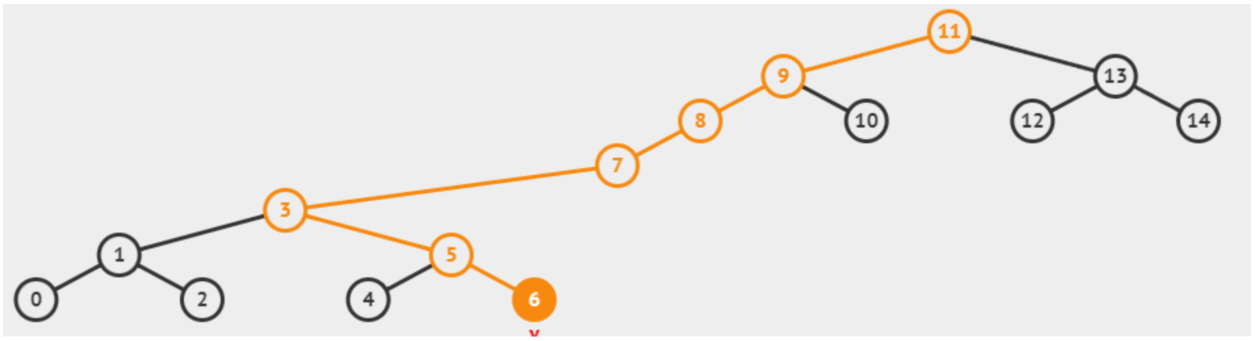
- Internet access
- Any electronic device other than your laptop (e.g. phones, watches, calculators, etc)
- Anyone else's notes, textbooks, internet resources etc.
- Any method of communication with other students.
- Generative coding tools of any kind.
  - Single-word code completion (e.g. IntelliSense) is acceptable
  - CoPilot, TabbyCat etc are strictly forbidden

You are responsible for ensuring that no disallowed resources are available / accessed during the exam.

Part 1 Level Order Traversal of A BST Tree (25 marks)

You will find in the code package base code that is a subset of the BST you implemented in Assignment 3. You do NOT need to do any balancing. Your task is to do a level order traversal of a given tree. The test code will insert elements into the BST and then call the function `levelOrderTraveral(T* arr)` with a pointer to an array. You are to write to the array the elements of the tree in level order. An example is given below.

For example, if you insert the `{ 11, 9, 8, 10, 13, 12, 14 , 7, 3, 1, 0, 2, 5, 4, 6}`, into a BST, the tree will look like this:



and the level order traversal would be:

```
11 9 13 8 10 12 14 7 3 1 5 0 2 4 6
```

Output Format

Your function must copy the elements of type T from the tree nodes to the array in the correct order. You may assume that the array will always have enough space for all nodes in the tree.

Submission Instruction

For the submission of Part 1, you are required to submit the contents of your *part1.cpp* solution onto Examplify (this should be only the `levelOrderTraversal()` function, but you may add helper functions if you desire). Please remember to remove any styling formats before submission (You can copy and paste it onto an **empty** *Notepad*, before copy and pasting from it onto Examplify submission textbox).

Part 2 Sorters (35 marks)

You are given a class `Sorter`. You are to implement the following member functions within the class:

- `void insertionSort(int* arr, int start,int end);`
- `int partition_first(int* arr, int start, int end);`
- `void quickSort(int* arr,int start,int end,int useISwhenLength=0);`

Each function performs the according action in the integer array `arr` from indices `start` to `end-1`.

In this part, you can assume that the input array contains only integers and there are no duplicates.

Task 1 Insertion Sort

Implement the function `insertionSort()` to apply the Insertion Sort algorithm to sort the elements from indices `start` to `end-1`. For example, if the input array is

```
int arr [15] = {12,14,5,3,9,10,7,6,11,2,0,8,1,4,13};
```

When `sorter.insertionSort(arr,0,10)` is called, the following steps are performed. Note that the brackets indicate which index was inserted into.

```
12 (14) 5 3 9 10 7 6 11 2

(5) 12 14 3 9 10 7 6 11 2

(3) 5 12 14 9 10 7 6 11 2

3 5 (9) 12 14 10 7 6 11 2

3 5 9 (10) 12 14 7 6 11 2

3 5 (7) 9 10 12 14 6 11 2

3 5 (6) 7 9 10 12 14 11 2

3 5 6 7 9 10 (11) 12 14 2

(2) 3 5 6 7 9 10 11 12 14
```

After the function completes, the array will be set to: `2 3 5 6 7 9 10 11 12 14`.

In order to verify that insertion sort is being done correctly, your code must call **`insertionSort_Step(arr,start,end,n)`** for each step if insertion sort, where *n* is the index of the newly slotted element. For example: using the example array above, and `sorter.insertionSort(arr,3,10)` is called, the following Step function will be called:

```
3 (9) 10 7 6 11 2     // insertionSort_Step(arr, 3, 10, 4);

3 9 (10) 7 6 11 2     // insertionSort_Step(arr, 3, 10, 5);

3 (7) 9 10 6 11 2     // insertionSort_Step(arr, 3, 10, 4);

3 (6) 7 9 10 11 2     // insertionSort_Step(arr, 3, 10, 4);

3 6 7 9 10 (11) 2     // insertionSort_Step(arr, 3, 10, 8);
```

```
        (2)  3  6  7  9  10  11      // insertionSort_Step(arr, 3, 10, 3);
```

And the input array will be modified to (the underlined part is sorted by the insertion sort):

```
12  14  5  2  3  6  7  9  10  11  0  8  1  4  13
```

Please note that the input parameter `start` and `end` are not necessary 0 and the length of array - 1 respectively. But you can assume `start >= 0` and `end` < the length of the array.

Task 2 Partition

Write a function `partition_first()` to partition the array by the first element as the pivot ***according to our lecture slides***. Here is the pseudo code in our slides:

```
partition(A[1..n], n)
      pivot = A[1]
      low = 2;
      high = n+1;
      while (low < high) {
            while (A[low]  < pivot) and (low < high) do low++;
            while (A[high] > pivot) and (low < high) do high--;
            if (low < high) then swap(A[low], A[high]);
      }
      swap(A[1], A[low-1]);
      return  low - 1;
```

Please note that the above is just pseudo code for you to reference only. Your actual code will deviate a bit. E.g. the pseudo code partitions the array range from 1 to n but our code part partitions the range from `start` to `end-1`.

You are to call `partition_first_Step(arr,start,end,n)` before returning from the function, where *n* is the index of the pivot element within the entire array `arr`.

E.g. The output for the array in Task 1 by calling `sorter.partition_first(arr,0,15)` will perform the following:

```
1  4  5  3  9  10  7  6  11  2  0  8  (12)  14  13
// partition_first_Step(arr, 0, 15, 12);
```

And if call `sorter.partition_first(arr,4,12)`, then the output will be

```
2  8  7  6  0  (9)  11  10
// partition_first_Step(arr, 4, 12, 9);
```

And it will change the array accordingly.

Task 3 QuickSort

Use the functions in the previous tasks to perform quick sort. However, there is one more parameter "`useISwhenLength`" . It means the recursion will switch to insertion sort when `end-start <= useISwhenLength`. Your program should perform the followings if you call the function `sorter.quickSort(arr,0,15,3)` :

```
1 4 5 3 9 10 7 6 11 2 0 8 (12) 14 13
// partition_first_Step(arr,0,15,12)

0 (1) 5 3 9 10 7 6 11 2 4 8
// partition_first_Step(arr,0,12,1)

2 3 4 (5) 7 6 11 10 9 8
// partition_first_Step(arr,2,12,5)

2 (3) 4    // insertionSort_Step(arr,2,5,3)
2 3 (4)    // insertionSort_Step(arr,2,5,4)

6 (7) 11 10 9 8
// partition_first_Step(arr,6,12,7)

8 10 9 (11)
// partition_first_Step(arr,8,12,11)

8 (10) 9   // insertionSort_Step(arr,8,11,9)
8 (9) 10   // insertionSort_Step(arr,8,11,9)

(13) 14    // insertionSort_Step(arr,13,15,13)
```

And you should change the array `arr` to `0 1 2 3 4 5 6 7 8 9 10 11 12 13 14`.

There is no Step function for quicksort, and all the necessary Step calls are done by `insertionSort()` & `partition_first()`.

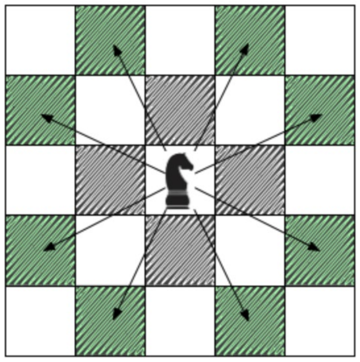
Submission Instruction

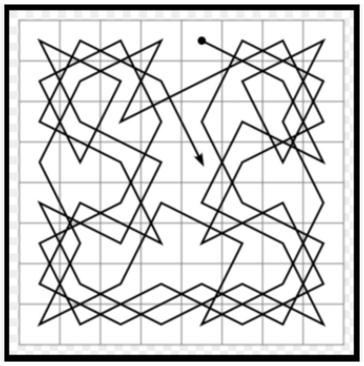
For the submission of Part 2, you are required to submit the contents of your *part2.cpp* solution onto Examplify. Please remember to remove any styling formats before submission (You can copy and paste it onto an **empty** *Notepad*, before copy and pasting from it onto Examplify submission textbox).

NB: The unit tests for Part 2 are somewhat more complicated as a result of the internal step checks. You are encouraged to write simpler tests to guide your development initially.

Part 3 Knight Movement (40 marks)

In international chess, the knights move in an "L-shape"—that is, they can move two squares vertically followed by one square horizontally, or two squares horizontally followed by one square vertically. The knight can move past any obstacles so long as the destination square is free. The traditional chess board has size is 8 x 8. However, in this question, your knight will move around in any chess board of size n x n with n > 0. In this question we want to determine how many moves the knight needs to move in order to go from a starting location to a destination without landing on a blocked square.





Task 1: Finding Minimal Moves

If your chess board size is n x n, the row and column indices are ranging from 0 to n-1. A knight's position can be specified by a xy-tuple `(a,b)` with `0 <= a,b < n`. A knight with position `(a,b)` can move to any position `(a ± 2, b ± 1)` or `(a ± 1, b ± 2)` within the chessboard. Given a starting position `(a,b)` and a destination position `(c, d)`, write the function `findMinNumOfMoves(startx, starty, endx, endy)` to return the minimum number of steps for a knight to move from the starting to the ending position. E.g. if the chess board size is 8x8 and the function `ks0.findMinNumOfMoves(0,0,0,4)`should return 2 because the path will be:

```
(0,0) > (1,2) > (0,4)
```

To begin, consider only the case that there are no blocked squares.

Task 2: Blockages

A number of helper functions are provided, including `isBlocked(x,y)` to determine if a square is blocked. Improve your function from Task 1 such that it finds the minimum number of moves to go from the starting position to the destination while avoiding landing on any blocked squares. For example, for the above query, if we set a blockage at position `(1,2)` then the function will return 4, because the path will be changed to:

```
(0,0) > (2,1) > (4,2) > (2,3) > (0,4)
```

The number of blockages is arbitrary, and you can assume the starting position of the knight will not be any of those blocked positions. E.g. if we set the blockages to be `(1,2)`, `(2,3)` and `(2,5)`, then the minimal path will be 6.

However, there is a possibility that the knight cannot reach the destination position with some configurations in both tasks. In that case, your function should return -1. E.g. if we set the blockage to be `(1,2)` and `(2,1)`, there will be no way to reach from (0,0) to (0,4).

Output Format

The output of the method is simply an integer with the number of moves, or -1 if there is no such path.

Submission Instruction

For the submission of Part 3, you are required to submit the contents of your *part3.cpp* solution onto Examplify. Please remember to remove any styling formats before submission (You can copy and paste it onto an **empty** *Notepad*, before copy and pasting from it onto Examplify submission textbox).

Appendix Usage of queue

<queue>

std::**queue**

```
template <class T, class Container = deque<T> > class queue;
```

FIFO queue

queues are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

queues are implemented as ***containers adaptors***, which are classes that use an encapsulated object of a specific container class as its ***underlying container***, providing a specific set of member functions to access its elements. Elements are ***pushed*** into the ***"back"*** of the specific container and ***popped*** from its ***"front"***.

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

```
empty
size
front
back
push_back
pop_front
```

The standard container classes deque and `list` fulfill these requirements. By default, if no container class is specified for a particular `queue` class instantiation, the standard container deque is used.

fx Member functions

(constructor)	Construct queue (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
front	Access next element (public member function)
back	Access last element (public member function)
push	Insert element (public member function)
emplace	Construct and insert element (public member function)
pop	Remove next element (public member function)
swap	Swap contents (public member function)

Example usage of queue<int>:

```
// CPP code to illustrate Queue operations in STL
// Credits go to Divyansh Mishra --> divyanshmishra101010
#include <iostream>
#include <queue>

using namespace std;

// Print the queue
void print_queue(queue<int> q)
{
	queue<int> temp = q;
	while (!temp.empty()) {
		cout << temp.front()<<" ";
		temp.pop();
	}
	cout << '\n';
}

// Driver Code
int main()
{
	queue<int> q1;
	q1.push(1);
	q1.push(2);
	q1.push(3);

	cout << "The first queue is : ";
	print_queue(q1);

	queue<int> q2;
	q2.push(4);
	q2.push(5);
	q2.push(6);

	cout << "The second queue is : ";
	print_queue(q2);


	q1.swap(q2);

	cout << "After swapping, the first queue is : ";
	print_queue(q1);
	cout << "After swapping the second queue is : ";
	print_queue(q2);

	cout<<q1.empty(); //returns false since q1 is not empty

	return 0;
}
```