**Deadline & Submission**

1. **At least one team member should submit the compressed group solution as zip file containing the programs and report under acadox –> tasks.**
2. **The deadline for submitting the solution of this phase assignment is Friday 1st April 2016 @ 11:00 pm.**

**Objectives**

1. Learn about linked lists.
2. Employ data structures in a real application.

**About this assignment**

1. This assignment will be solved in teams of 3 students.
2. Each team member should work on one of these tasks and the team should integrate their code together in one program.
3. The weight of the phase is 3.5 marks
4. All code should compile and run with standard C++11.
5. **Assume any missing details and search for information yourself.**
6. In case of cheating in any part of the work, **all the team will be responsible / punished.**

**Introduction**

It's very important to have a to-do list in your life. In this to do list you should write everything you want to do in the very near future. There are many applications which we can use to manage our tasks. But in this project you and your great team will implement a program which any user can use to manage daily or weekly tasks.

Note: Regardless of this project, you should always remember that you should have your own to do list. Use something like sticky notes to write your tasks. To do list will help you to form a good vision for your goal. And If you haven't a good vision for your goal you will probably work for someone else goal

**Problem statement**

You are required to implement a to-do list manager program. The program keeps track of all your tasks and the progress of each task. Also it allows the user to categorize tasks into projects so each task may be a subtask in big project

**Task1**

Create a file with your tasks with all your tasks. The file is called "MyTasks.txt" each task is described by five attributes: name, description, priority, due date and the number of days it takes. Each piece of info is represented on a separate line. So if you have a data structure assignment you can describe it as a task as follows

Data Structure Assignment 2

Work in groups of 3 to solve problems in sorting algorithms and complexity.

5

15/3/2016

2

Line three represents the priority which ranges from 5 (to priority) to 1 (least priority)

You are required to build the following:

1. Build a class Task that represents a task with its attributes and any necessary functions. Overload operator<< to print task info.
2. Build a Task List using an **ordered linked list** with all tasks sorted according to their **priority** from the most important tasks to the least important ones. Tasks with priority 5 come first. Use FCISLL given in the class (available under acadox) to build your ordered linked list. An ordered linked list keeps the tasks sorted all the time. Tasks are sorted by priority. And add any necessary functions. You will need also to overload operator<< to be able to print the list.
3. Write a To Do List Management program that allows the user to **load** his tasks from MyTasks.txt and **display** them and **save** them again in file.

**Task 2**

After reading the tasks list we want to allow user to enter new tasks to our tasks list. Remember tasks should be always sorted as explained above.

You are required to build the following:

1. For the given template class FCISLL, add the following methods which all return the items removed:

* **T removeHead ()**
* **T removeTail ()**
* **T removeFromIndex (int i) // Make sure the index is valid**
* **T removeItem (T item) // Remove first instance of item**

1. Add a method **removeWithPredicate (bool (\*predicate)(T& item))** which takes a predicate function and removes from the list all nodes were the predicate holds. E.g.,

**// A fun to return true if date of task t passed already**

**bool isPassed (Task t) { ... }**

**// remove all past tasks from list**

**list.removeWithPredicate (task) // remove all past tasks from list**

1. Write a small program to test and demo the methods you completed. Make sure you test all different possible cases, including the case of a list with one node, or invalid index.

**Task 3**

We want to make our application more dynamic. So we want to make the sort process of tasks depends on a dynamic function.

Current sorting depends on priority. But we want to allow user to sort tasks depending on **task duration** or **due date** not priority. Also we want to allow user to choose if the tasks should be sorted ascending or descending.

We also like to be able to merge to lists, but adding another list to an existing one and re-sorting all the tasks together.

Your task will be:

1. For the given template class FCISLL, add the following method:

* **sort (bool (\*comp)(T& item1, T& item2), int order)**

which takes a comparator function that takes two items and returns true based on the result of their comparison. Then sort method sorts the linked list based on this comparator. A comparator is a generalization of < operator since it can take any two items. For example, 3 < 5 is true but we need to define if task1 < task2 is true or not based on a criterion for comparing tasks (name, priority, duration, etc.). You can use any sorting algorithm suitable for working on linked lists. So, in your code your should be able to write something like this:

**bool orderByDuration (Task task1, Task task2) {**

**return (task1.getDuration() < task2.getDuration);**

**}**

**....**

**myTaskList.sort (orderByDuration, ASCENDING);**

1. Which sorting algorithms are more suitable for sorting linked lists? And Why? And which are less suitable for this task? And why?
2. Add another method:

* **Merge (FCISLL<T> anotherList)**

It allows the user to merge to lists and keeping them sorted. This should be a very easy task using the methods you created before.

1. Write a program to test and demo the methods you wrote. Test all different possible cases, including empty lists and lists with one node.

**Guidelines:**

1. **Originality**:

* All work submitted by students must be original and their own work.
* Under no circumstances submit a solution that you have taken from a friend, from the net, from a book or any other resources.
* Submission of copied work will result in severe punishment and negative mark.

1. **Style**

* All code submitted must follow proper C++ coding style.

1. **Submission to Acadox Account:**

* Team leader should collect all the programs, integrate them together and make sure the overall program works and organize the solutions of bonus parts.
* Use your Acadox account to upload your answer before the deadline. Write a cover page of the report should have your full details. **No submission is accepted outside acadox. You can upload before the deadline and then continue working and upload a better version.**
* The team leader should put all submissions in one zip file named:
* ProjectPart1\_TA\_YourGroup\_YourID1\_YourID2\_YourID3.zip and upload it to acadox by deadline.
* Create and account under GitLab and upload the code there.
* Your code must compile and run without errors.
* **Code that does not compile or files that are not named correctly will be rejected.**

1. **Marking Criterion**

* **In case of copied work or giving the book to other students, all the team gets -7 marks in the entire project phase 1.**
* 2 marks are given for each individual Task. The **task developer** gets full mark for fully working and tested functionality with no error. He gets part of the mark if work is not fully functional.
* 1.5 marks for **the entire team** for integrating the three tasks together and having a program that works well.
* 1 bonus mark **for the entire team** if they provide answers to Bonus 1 questions and respond to questions in discussion.
* 1 bonus mark **for the entire team** if they provide answers to Bonus 1 questions and respond to questions in discussion.

**Bonus 1: Theoretical and LL Coding Questions**

These are (1) theoretical problems that do not require coding, although you may code some of them to further understand and investigate concepts and (2) LL coding exercises. **Your team can get one bonus mark if they answer these questions and provide answers in *handwriting* يقدم الحل بخط اليد to their TA.**

1. (Coding) In the class FCISLL, overload the operator == to deeply check if two linked lists have the same content. (They have the same num of elements and each element is equal to the corresponding element. Elements do not have to be in order.) There are a number of ways to do this task, using other functions that you already developed in the project. **Think if a creative solution.** Write a program to test your solution.

Note: Lists 1->2->3-> and 2->3->1 and 3->1->2 are similar and have the same content but are not similar to list 1->1->2->3.

1. Linked lists do not have to be implemented with pointers. Think of an alternative implementation of linked lists that does not use nodes lined by pointers. Draw this list in the suggested implementation. What is the O(n) of the basic list operations in this implementation.

+---+---+ +---+---+ +---+----+ +---+----+

+------>| 9 | o---->| 4 | o---->| 5 | o---->| 8 |NULL|

| +---+---+ +---+---+ +---+----+ +---+----+

+-|-+

head | o |

+---+

1. Discuss three possible implementations of Priority Queue ADT type and their pros and cons. This ADT is a queue in which people receive the service (dequeue) based on their priority. Show the O(n) of each of important queue operations for each one of the possible implementations.
2. Given the following class Node. Suppose in some application, the following structure/data has been built. Assuming head is declared as **Node\***.

**class Node**

**{**

**public:**

**// constructor**

**Node(int val=0, Node\* next\_ptr=NULL)**

**{ data = val; next = next\_ptr; }**

**// data members**

**int data;**

**Node \*next;**

**};**  +---+--+ +---+--+ +---+--+ +---+----+

+--->| 9 | o-->| 4 | o-->| 3 | o-->| 5 |NULL|

| +---+--+ +---+--+ +---+--+ +---+----+

+-|-+

head | o |

+---+

1. Draw a picture after each of the following lines are executed. Note those lines are NOT executed sequentially
2. **head = head­->next-­>next;**
3. **head-­>next = new Node(2, head-­>next);**
4. (Coding) Starting from the original list shown above, write a program that **reverses** the list (or any SLL) to be:

+---+--+ +---+--+ +---+--+ +---+----+

+--->| 5 | o-->| 3 | o-->| 4 | o-->| 9 |NULL|

| +---+--+ +---+--+ +---+--+ +---+----+

+-|-+

head | o |

+---+

1. Answer the following questions
2. Draw the state of a self-organizing doubly linked list using (1) transpose method, (2) move-to-front method and (3) count method after each line of the following operations. (each line is applied to the result of the previous one) Show the links too.

insertHead(a), insertHead(b), insertHead(c),

search(a),

search(a),

search(b),

search(c),

search(c),

1. Under what condition will a self-organizing list perform better than an ordinary linked list?
2. Compare lists self-organizing methods (transpose, move-to-front and count) with each other in terms of:
3. Need for extra storage
4. Ease of implementation
5. This exercise is designed to help get the reader familiar with choosing the right data structure for the right problem. If implemented, the parts of this exercise should be done by making use of an implementation of the relevant ADT (Stack, Queue, Lists) in the STL. Solve the following problems by reading a text ﬁle one line at a time and performing operations on each line in the appropriate data structure(s). Your implementations should be fast enough that even ﬁles containing a million lines can be processed in a few seconds. Suggest an ADT and an algorithm for each of these tasks:
6. Read the input one line at a time and then write the lines out in reverse order, so that the last input line is printed ﬁrst, then the second last input line, and so on.
7. Read the ﬁrst 50 lines of input and then write them out in reverse order. Read the next 50 lines and then write them out in reverse order. Do this until there are no more lines left to read, at which point any remaining lines should be output in reverse order. In other words, your output will start with the 50th line, then the 49th, then the 48th, and so on down to the first line. This will be followed by the 100th line, followed by the 99th, and so on down to the 51st line. And so on. Your code should never have to s store more than 50 lines at any given time.
8. Read the input one line at a time. At any point after reading the first 42 lines, if some line is blank (i.e., a string of length 0) then output the line that occurred 42 lines prior to that one. For example, if Line 242 is blank, then your program should output line 200. This program should be implemented so that it never stores more than 43 lines of the input at any given time.
9. Read the input one line at a time and write each line to the output if it is not a duplicate of some previous input line. Pay attention so that a file with many duplicate lines does not use more memory than what is required for the number of unique lines.
10. Read the input one line at a time and write each line to the output only if you have already read this line before. (The end result is that you remove the ﬁrst occurrence of each line.) Take special care so that a ﬁle with a lot of duplicate lines does not use more memory than what is required for the number of unique lines.
11. Read the entire input one line at a time. Then output all lines sorted by length, with the shortest lines ﬁrst. In the case where two lines have the same length, resolve their order using the usual “sorted order.” Duplicate lines should be printed only once.
12. Do the same as the previous question except that duplicate lines should be printed the same number of times that they appear in the input.
13. Read the entire input one line at a time and then output the even numbered line (starting with the ﬁrst line, line 0) followed by the odd-numbered lines.
14. Read the entire input one line at a time and randomly permute the lines before outputting them. To be clear: You should not modify the contents of any line. Instead, the same collection of lines should be printed, but in a random number.

**Bonus 2: Problems on Stacks and Queues**

These are coding problems on stacks and queues. **Each Individual Student can get one bonus mark if s/he answers one of these questions and provides the solution program.**

**Question 1 – Stacks – Delimiter Matching**

**Objective:** Learn about the uses of stacks and lexical analysis of programs by compilers.

1. Implement **delimiterMatching** algorithm in explained in the lecture to check for the balance of the following types of delimiters:

**( ) [ ] < > { } /\* \*/ " "**

1. Make sure to remove or ignore single line comments because any brackets inside them should not be considered like this one:

**// Anything here is ignored ]([(>/\***

1. Also check " …. ". Make sure to ignore anything between them including **\"**
2. Take care the cases of **<** which is considered in some cases and ignored in other cases:
   * **cout << x << endl; // 2 together are not checked**
   * **for (i = 0; i < 5; ++i) // < operator is not checked**
   * **#include <iostream> // with #include, check <...>**
   * **queue<int, list<int> > q2; // check here, > > must have space**
3. Write 10 test cases to cover all possibilities.

**Question 2 – Stacks – Very Large Number**

**Objective:** Learn about the uses of stacks and lexical analysis of programs.

1. Write a function to implement **addingLargeNumbers** algorithm explained in lecture. You can use the **LargerInteger** class implemented in assignment 1 or assume the integers come as strings.
2. Write a function to implement an algorithm **addingLargeFloats** to add floats. You can use the previous function in implementing this one.
3. Write a program to test and demo the functionality of these functions and consider all the different cases.

**Question 3 – Queues and Stacks**

Data is stored in a stack

isEmpty

clear

enqueue

dequeue

first

**Objective:** Learn about how to implement a data structure using another one.

**Queue**

1. Implement a queue in terms of a stack. Use the stack in STL. This means that the underlying data structure for your queue will be a stack. You will need to implement **isEmpty, clear, first,** **enqueue** and **dequeue** operations in terms of stack’s **pop** and **push** and other operations. In the operations **enqueue** and **dequeue** you may need to use extra data structures (e.g., another stack) to perform the operation.
2. Is it a good idea to use a stack as the underlying data structure to build a queue using it? Why yes or why not?

**Question 4 – Priority Queue**

**Objective:** Learn about the possible implementations of priority queues.

1. Implement a class **PriorityQueue** that uses FCISLL as the underlying data structure.
2. Implement all the standard queue operations discussed in the class.
3. Notice that stored items will have a value and a priority that defines their importance.
4. Develop a demo / testing program to show the functionality of your program.