Data Structures and Algorithms - Spring 2023
Assignment 3

Due Date and Time: 7th April, 2023 (11:59 A.M, morning) Marks: 100

Instructions:

1. *Late assignment will not be accepted.*
2. *Few questions are handwritten and must be submitted as a scanned pdf along with the coding questions.*
3. *There will be no credit if the given requirements are changed.*
4. *Please mention the question number and its part (if any) before writing down its solution. Use the same conventions used in the assignment's statement document.*
5. *Understanding the questions of assignment is part of the assignment.*
6. *Plagiarism may result in zero marks in the whole assignments category (all assignments) regardless of the percentage plagiarized.*

Solve question no. 1 on the page and also prepare a scanned copy of it with your name and registration number.

Question no.1 **10 marks**

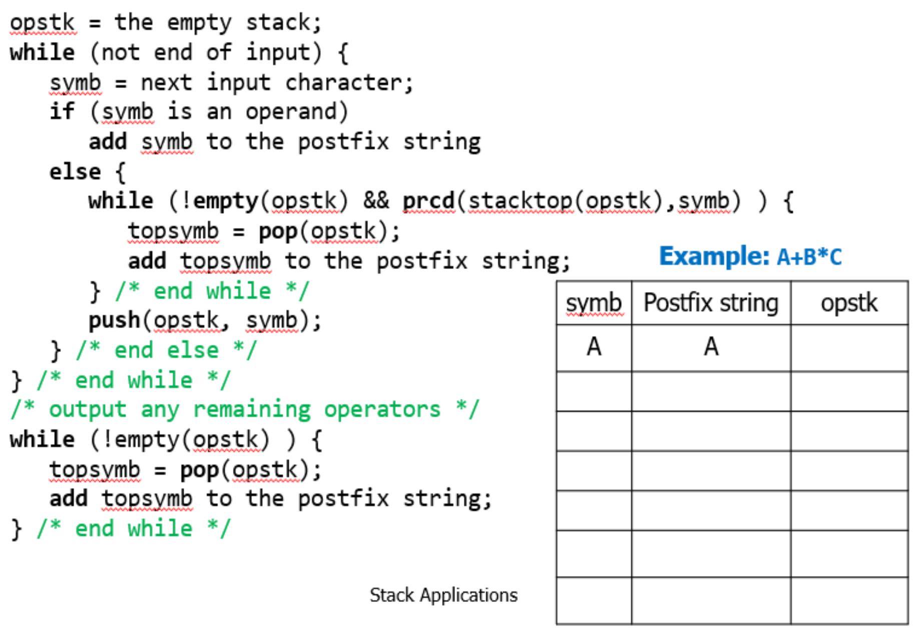
Part a: Convert the following expressions into Polish and Reverse Polish Notations. This should be done directly but show all steps involved in the derivation.

1. (A * (B / C – D) ^ E) + (E / F / (G + H))
2. A + B & (((C + D % E) * F) / G)

Part b: Convert the following directly to infix notation.

1. A B C D % E + F * G / * +
2. +A +* B C /^ D E *F G

Part c: Dry run the simple Infix to Postfix algorithm. Also mention explicitly the statement of the pseudocode that caused changes in Output Stack/String or Operator Stack.



```
opstk = the empty stack;
while (not end of input) {
   symb = next input character;
   if (symb is an operand)
      add symb to the postfix string
   else {
      while (!empty(opstk) && prcd(stacktop(opstk),symb) ) {
         topsymb = pop(opstk);
         add topsymb to the postfix string;
      } /* end while */
      push(opstk, symb);
   } /* end else */
} /* end while */
/* output any remaining operators */
while (!empty(opstk) ) {
   topsymb = pop(opstk);
   add topsymb to the postfix string;
} /* end while */
```

Example: A+B*C

symb	Postfix string	opstk
A	A	

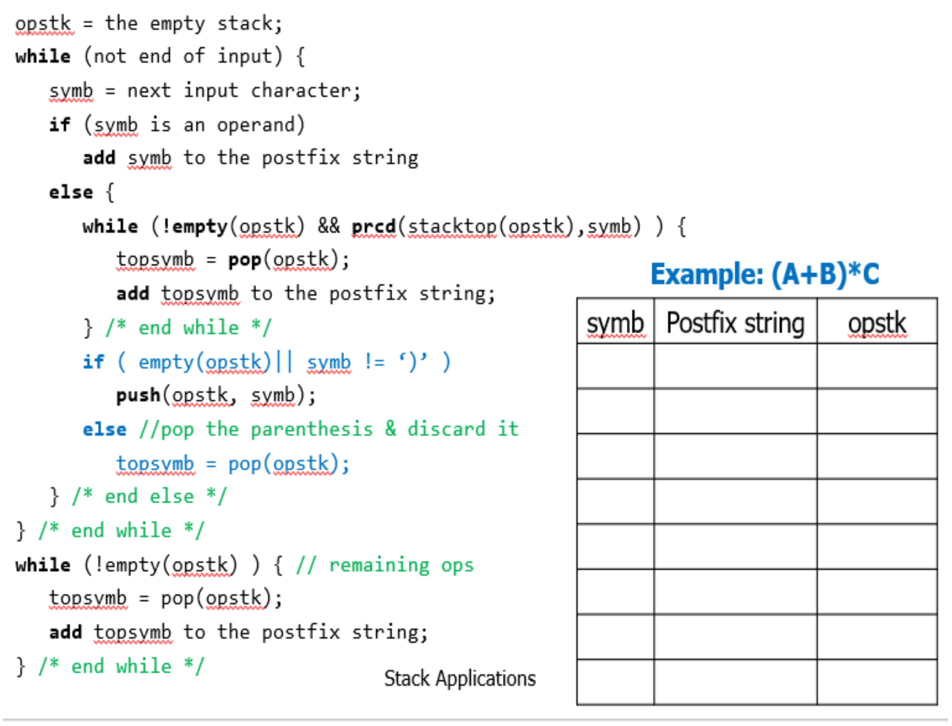
Stack Applications

Expression: A + B * C ^ D ^ E % F * G

Symb	Postfix string	Opstk	Statement

Part d: Dry run the Infix to Postfix algorithm. Also mention explicitly the statement of the pseudocode that caused changes in Output Stack/String or Operator Stack.

Expression: A + B * C ^ (((D ^ E) % F) * G)



```
opstk = the empty stack;
while (not end of input) {
   symb = next input character;
   if (symb is an operand)
      add symb to the postfix string
   else {
      while (!empty(opstk) && prcd(stacktop(opstk),symb) ) {
         topsymb = pop(opstk);
         add topsymb to the postfix string;
      } /* end while */
      if ( empty(opstk)|| symb != ')' )
         push(opstk, symb);
      else //pop the parenthesis & discard it
         topsymb = pop(opstk);
   } /* end else */
} /* end while */
while (!empty(opstk) ) { // remaining ops
   topsymb = pop(opstk);
   add topsymb to the postfix string;
} /* end while */
```

Example: (A+B)*C

symb	Postfix string	opstk

Stack Applications

Symb	Postfix string	Opstk	Statement

Question no.2 **20 marks**

Make proper classes and implement functions for the following parts.

Part a: Implement the Infix to Postfix algorithm considering the following constraints and all other rules remain the same.

1. Precedence of / is higher than *
2. Precedence of % is higher than ^ but less than *

3. Precedence of + is higher than – but less than ^
4. ABC (case-insensitive) is a new operator having the highest precedence.
5. Maintain an Array based Stack (size 30) for String generated during the processing of the algorithm.
6. Maintain a Doubly Linked List based Stack for operators stored during the processing of the algorithm.

Part b: Implement the Simple Infix to Postfix algorithm considering the following constraints and all other rules remain the same.

1. Maintain a String object for String generated during the processing of the algorithm.
2. Maintain a Linked List based Stack using Queues for operators stored during the processing of the algorithm. There are multiple ways to do but use the below logic to implement Stack using Queue.
    a. Push (E element) if q1 is empty, enqueue E to q1. if q1 is not empty, enqueue all elements from q1 to q2, then enqueue E to q1, and enqueue all elements from q2 back to q1.
    b. Pop, dequeue an element from q1.

Part c: Implement the Infix to Prefix algorithm using the below mentioned details and all other details remain the same as of Part b.

- Reverse the infix string.
    - ➢ Adjust parenthesis, i.e., make every '(' as ')' and every ')' as '('
- Perform infix to postfix algorithm on reversed string.
- Reverse the output postfix expression to get the prefix expression.

Part d: Implement the Postfix Evaluation Algorithm to work on any given expression and its values.

Question no. 3 **70 marks**

Part a: Design a Text Processing Software (TPS) integrated with C++ console. All text editing will be done using the console screen, but the final outcome should be stored in a text file once the program ends. Implement Linked List based Stacks and Queue to handle Undo/Redo operations in TPS.

Initially the program opens an associated file or creates a new text file (when program runs first time ever). Suppose the text file is included in the same working directory where you have saved this C++ program. In any normal text editor you can add a word or character and even undo/redo it anytime you want. Here is the basic algorithm:

1. Maintain two stacks: undoStack and redoStack.
2. When a change is made to the document, push the current state of the document onto the undoStack.
3. If undoStack is not empty and the user requests an undo operation, pop the top state from undoStack and push it onto redoStack. Set the current state of the document to the top state of undoStack.
4. If redoStack is not empty and the user requests a redo operation, pop the top state from redoStack and push it onto undoStack. Set the current state of the document to the top state of redoStack.
5. If the user makes a new change to the document after undoing some changes, clear the redoStack.
6. If the user makes a new change to the document after redoing some changes, clear the undoStack.
7. When the document is closed, clear both undoStack and redoStack.

The above-mentioned steps only maintain a Single-level undo/redo operation. This is further elucidated with example no.1. In Multi-level undo/redo step no. 5 and 6 triggers Mega State save action in a Queue. Any time during the editing user can fully shift to any of the saved Mega State. The is explained below in the next paragraph after the example no.1.

There is no proper undo/redo facility in a normal text editor (notepad) available with windows. This task provides you an opportunity to inculcate some advanced text editing features in a normal text editor assisted with C++ console.

Rules:

Only full-stop, next line, undo/redo and save operations trigger the changes and on changes the state is saved. There is no capitalization of the first letter. Implement only the mentioned functionality however it is to be noted that the actual undo/redo functionality of MS Word is more complex than this.

1. State is saved on every full stop. Full stop is inclusive in the text.
2. State is saved on every next line operation.
3. Undo/redo also save a State.
4. Save, program end and file close operations also save the State.
5. Undo/redo only works when file is open and till closed.

Example no.1:

One session of typing:	**Actions in Sequence:**
This course is.	Typed full stop so state is saved
This course is. Good.	Typed full stop so state is saved
This course is.	Undo pressed
This course is. Good.	Redo pressed
This course is.	Undo pressed
This course is. Excellent.	Typed full stop and new text added after an undo
This course is. Excellent.	Redo pressed but no changes
This course is.	Undo pressed

From the example no.1, it is evident that there is no way to trace back to "This course is. Good.", when "Excellent." is added as new text after undo in the example. In case of Multi-level undo/redo this glitch can be avoided by additionally saving a complete State in a Queue. User can restore the first lost trace of "This course is. Good." using the deque operation once and same is applicable if multiple words were added in the same manner. If a State is restored back from the Multi-level undo/redo then it is same as closing the file with a new State and reopening such that no single or multi-level undo/redo option is available anymore.

In simple words the State representing pre-glitch text can be restored. The rules defined in this text editor may sound unrealistic but indeed every software seems odd until users get use to of it. The implementation of this text-editor introduces new and existing functionalities that are missing in the notepad of windows.

Console Help:

Suppose console has initially the text read from the text file. In next line ask for input. Input can be a space, character/s or symbols. ***Symbols have a special purpose as defined in the table.***

/	***Next line***
.	***Full stop***
@	***Undo***
#	***Redo***
!	***Text saved***
*	***File closed***
$	***File opened or reopened***
%	***Program ends and everything auto saves***
^	***Restore to first glitch if any***
^^	***Restore to second glitch if any***
^^	***Restore to third glitch if any (there can be any number of glitches)***

The console showing initial text "This is the text." along with some later editing is displayed below. The text in bold is action/reaction of system and in normal font is your interaction through typing.

```
This is the text.
Please enter text or symbol: acha.                          🡐 . pressed so state saved
This is the text. acha.
Please enter text or symbol: kia hal hai!                   🡐 ! pressed so text saved
This is the text. acha. kia hal hai
Please enter text or symbol: ye wali assignment to easy hai/   🡐 / pressed so next line
This is the text. acha. kia hal hai ye wali assignment to easy hai

Please enter text or symbol:done
This is the text. acha. kia hal hai ye wali assignment to easy hai
done
Please enter text or symbol:*                          🡐 * file closed but program not ended
```

Part b: Add an additional feature of Version Control in TPS. By utilizing version control, several individuals can work on a single project simultaneously. Each person can make modifications to their own version of the files and decide when to synchronize those modifications with the team.

The TPS keeps running all the time in this part and different users can interact with the TPS using their own ID. Suppose TPS is used by multiple users and each user has a unique ID. Whenever the program initiates, user generates a new ID and stores in a separate text file specifically associated with the original text file on which different users are contributing. Your system has multiple users and each time a user interacts with TPS to update the text file a relevant version of it is maintained. There is one admin who is also a user of TPS in the same System. Admin has the privilege to access the history of the document for each user of the TPS including the admin. All functionalities presented in the part (a) are also available to each user of the TPS contributing to the text file.

Only admin can save or restore any version of the text document before closing the program. Once a program is closed the text file is permanently stored with the latest version.

You are required to maintain an additional Doubly Linked List based Stack (only one stack is allowed) to reflect the behavior of undo/redo at the user level. Part (a) is also inclusive in it. Remember in part (a) undo/redo was for a single user whereas here it is assisting multi-users.

Part c: This part is in addition to part (a) and (b). All functionalities of previous parts are included in this part and some additional working is also required in part (a/b) for this part.

Implement a Searching mechanism to find the traces of all changes made to the text file by a user or users. The admin has an additional control to not just view different versions of the text file but also navigate to different versions of text file created by each user. In part (a), undo/redo history was only available until the file was opened but for this part it is also required to keep the back-up of the latest session of undo/redo for each user. You are allowed to use or create multiple files for this specific task.

There are two searching algorithms that are required to be implemented. One is Breadth First Search (BFS) and second is Depth Fist Search (DFS). BFS will search all changes made by the first user before moving to the next user. DFS on the other hand will search first version of text file by all users before moving to the second version of all users.

Admin has the option to perform text searching either in BFS or DFS order using Queues and Stacks.

The steps involved in the BFS algorithm are:

1. Initialize a Linked List based queue data structure and add the root node to the queue.
2. Mark the root node as visited.
3. While the queue is not empty, dequeue the next node from the queue and visit all its neighbors.
4. For each unvisited neighbor, mark it as visited and add it to the end of the queue.
5. Repeat steps 3-4 until the queue is empty.

The steps involved in the DFS algorithm are:

1. Initialize a Linked List based stack data structure and add the root node to the stack.
2. Mark the root node as visited.
3. While the stack is not empty, pop the top node from the stack and visit all its neighbors.
4. For each unvisited neighbor, mark it as visited and push it onto the stack.
5. Repeat steps 3-4 until the stack is empty.

The searching facility only finds the given text in the first occurrence of the oldest version of the text file either in BFS or DFS. Admin may be interested to search changes of a single user before moving to other users or first version of all users before moving to the second version of all users.

Ask user to press either 1 or 2 after verifying their IDs.

1. Start editing the document.
2. Enter text to search. (Only valid for admin)
    a. Press 3 to search in BFS order.
    b. Press 4 to search in DFS order.