# NATIONAL INSTITUTE OF TECHNOLOGY SIKKIM
## Department of Computer Science and Engineering

# CS13201 Data Structure and Algorithms Laboratory

**Odd Semester, July–December 2024**

# Laboratory Assignment #3

NB: Solve the following programming problems according to instructions given during the laboratory sessions. Along with the requirements mentioned in the text of the questions, additional instructions may be given by the course instructor to produce the desired output. All programs should be written in the C programming language. Do not use any header other than *stdio.h* and *stdlib.h*.

1. Write a program to implement the following operations on a given array *list* (of any data type) of size *k*, where *list* and *k* are taken as input from the user. Each operation should be implemented by a function according to the definition of the operation given below. After initially taking *list* and *k* as input from the user, the user will be shown a menu-based interface where the user will choose to execute an operation. The menu will be repeatedly displayed until the user chooses to quit. Do not use any external variable.

   Insert(*list, x, p, k*) := Inserts element *x* in array *list* at position *p* and updates size *k* of *list*.

   Delete(*list, p, k*) := Delete element *x* from position *p* of array *list*, updates size *k* of *list*, and returns *x*.

   Empty(*list, k*) := Deletes all elements of array *list* and updates size *k* of *list*.

   Display(*list, k*) := Display array *list* of size *k* in the correct order (from beginning to end).

2. Write a program to implement a stack with an array. Given below are abstract definitions of operations that need to be implemented by writing a function for each operation, where *s* is a stack, *x* is an element that can be stored in *s*, *top* is the end of stack where push and pop operations are allowed, *bottom* is the other end of the stack. After initially creating a stack *s* with *k* elements and with maximum size *max*, taking *k*, *max*, and the elements of *s* as input from the user, your program will show a menu-based interface, similar to Question no. 1, where the user can choose from the following options: i) Push, ii) Pop, iii) Display, iv) Quit. The menu will be displayed repeatedly until the user chooses option (iv). The Push and Pop functions must call the functions IsFull and IsEmpty, respectively, to check the overflow and underflow conditions, for which appropriate messages should be displayed. Do not use any external variable.

   IsEmpty(*s*) := returns True if *s* is empty, otherwise returns False.

   IsFull(*s*) := returns True if *s* is full, otherwise returns False.

   Push(*s, x*) := pushes *x* to *s*.

   Pop(*s*) := pops an element *x* from *s* and returns *x*.

   Display(*s*) := prints all elements of *s* in the order starting from *top* and ending at *bottom*.

3. Write a program to implement a queue with an array, assuming that it is a linear queue. Given below are abstract definitions of operations that need to be implemented by writing a function for each operation, where *q* is a queue, *x* is an element that can be stored in *q*, *front* is the end of queue where only the delete operation is allowed, *rear* is the other end of the queue where only the add operation is allowed. After initially creating a queue *q* with *k* elements and with maximum size *max*, taking *k*, *max*, and the elements of *q* as input from the user, your program will show a menu-based interface, similar to Question no. 2, where the user can choose from the following options: i) Add, ii) Delete, iii) Display, iv) Quit. The menu will be displayed repeatedly until the user chooses option (iv). The Add and Delete functions must call the functions IsFull and IsEmpty, respectively, to check the overflow and underflow conditions, for which appropriate messages should be displayed. Your program should be able to perform the Add operation as long as there is empty space available in the array. Do not use any external variable.

   IsEmpty(*q*) := returns True if *q* is empty, otherwise returns False.

IsFull($q$) := returns True if $q$ is full, otherwise returns False.

Add($q$, $x$) := adds $x$ to the rear of $q$.

Delete($q$) := deletes an element $x$ from the front of $q$ and returns $x$.

Display($q$) := prints all elements of $q$ in the order starting from *front* and ending at *rear*.

4. Similar to Question no. 3 above, write a program to implement a circular queue with an array. All other requirements remain the same as in Question no. 3.

5. Write a program to implement a series of $n$ stacks with an array of structures where each instance of a structure represents one stack (hint: declare a stack, its top, and its maximum size as the members of the structure), and the stacks are indexed with the numbers 1 to $n$, i.e. the $i^{th}$ stack has index number $i$. The maximum size of the $i^{th}$ stack is denoted by $max_i$. After taking the value of $n$ and the $n$ values of $max_i$ as input from the user, the program will display elegantly the contents of the $n$ stacks (which are initially empty) along with a menu which will have the following options: i) Push, ii) Pop, iii) Transfer_Next, iv) Transfer_Anywhere, v) Quit. The contents of the $n$ stacks along with the menu will be displayed repeatedly in a loop until the user chooses to quit. The abstract definitions of the four operations for the menu options (i) to (iv) respectively are given below, where $S$ is a series of $n$ stacks, and $x$ is an element that can be stored in a stack. Do not use any external variable. The data type of the elements of the stacks are left to the programmer's discretion.

   Push($x$, $S$) := pushes $x$ to the $1^{st}$ stack of $S$.

   Pop($S$) := pops an element $x$ from the $n^{th}$ stack of $S$ and returns $x$.

   Transfer_Next($i$) := pops an element $x$ from the $i^{th}$ stack of $S$ and pushes $x$ to the $(i+1)^{th}$ stack of $S$.

   Transfer_Anywhere($i$, $j$) := pops an element $x$ from the $i^{th}$ stack of $S$ and pushes $x$ to the $j^{th}$ stack of $S$.

6. Sikkim Wine Company has a policy of selling the newest wine bottles first and the oldest wine bottles last, since old wine bottles are considered more flavourful and can be sold at a higher price. So, the company maintains a stack to store the information of the wine bottles, where the newest bottle to be pushed to the stack is the first one to be removed from the stack and sold. The oldest bottle in the stack is the last one to be removed from the stack and sold.

   Write a program to create a stack, using an array of structures, to store the information of wine bottles so that each element of the stack is a structure object containing the following details of one wine bottle:

   i) Serial number of the bottle, *srl*
   ii) Date of manufacture of the bottle, *date*
   iii) Name of the vineyard for the bottle, *vinyrd*
   iv) Price of the bottle, *price*

   The price of each wine bottle is initially set and then later updated according to the following scheme: when a bottle is pushed to the stack, its initial *price* is 500 rupees, and then every time a new bottle is pushed to the stack, the *price* of all the previously existing bottles of the stack (i.e. all bottles except the new bottle which is being pushed to the stack) is increased by 50 rupees.

   Your program will initially set the stack to be empty, after taking the maximum size of the stack *max* as input from the user, and provide a menu-based interface containing the following options:

   i) "Add New Bottle", which pushes a new bottle into the stack after taking its *srl*, *date*, *vinyrd*, and initial *price* as input from the user,
   ii) "Sell Bottle", which removes one bottle from the stack and shows the final selling *price* of the bottle, along with its *srl*, *date*, and *vinyrd*, in a reader-friendly format resembling a bill,
   iii) "Display Stock", which shows the *srl*, *date*, *vinyrd*, and current *price* of all bottles in the stack in a reader-friendly tabular format, and

iv) "Quit", which lets the user quit the menu which otherwise is repeatedly displayed in a loop.

After the user quits the menu, display as the final output of your program the total profit of the company if the manufacturing cost of every bottle in the stack is 450 rupees. Note that the "Add New Bottle" option and the "Sell Bottle" option should handle the overflow condition and the underflow condition respectively, and display appropriate messages.

7. Write a program to create the game of Tower of Hanoi. The game consists of three stacks, say *s1*, *s2*, *s3*, where every stack has its elements arranged in a non-decreasing order from top to bottom, i.e. in any stack *s*, a push operation can be performed only if the element *x* to be pushed to *s* is less than to equal to the top element *t* of *s* (i.e. if $x > t$, then push is not allowed in *s*). Initially, *s1* has *n* elements, and *s2* and *s3* are empty. The objective of the game is to transfer all the *n* elements from *s1* to *s3* by a series of moves where a move is defined as popping an element *x* from a stack $s_i$ and then pushing *x* to another stack $s_j$ (where, $1 \le i \le 3$, $1 \le j \le 3$, and $i \ne j$). Note that only one move can be made at a time. Your program will initially take the value of *n* and the elements of *s1* as input from the user, and then will repeatedly display the contents *s1*, *s2*, and *s3*, along with a menu where the user is given the option to make a move. Each time the menu is displayed, the user will choose the source stack for the pop operation and destination stack for the push operation for one single move. The menu and the stack contents will be displayed repeatedly until the game is solved or the user chooses to quit. If the game is solved, your program will print the number of moves taken by the user to solve the game. Take the elements of the stack to be of *int* type, and do not use any external variable.

8. Write a program to implement a deque (double-ended queue) with an array. Given below are abstract definitions of operations that need to be implemented by writing a function for each operation, where *d* is a deque, *x* is an element that can be stored in *d*, *front* is one end of the deque, and *rear* is the other end of the deque. After initially creating a deque *d* with *k* elements and with maximum size *max*, taking *k*, *max*, and the elements of *d* as input from the user, your program will show a menu-based interface, similar to Question no. 2, where the user can choose from the following options: i) Add At Front, ii) Add At Rear, iii) Delete From Front, iv) Delete From Rear, v) Display, vi) Quit. The menu will be displayed repeatedly until the user chooses option (vi). The two add operations and the two delete operations must check the overflow and underflow conditions, for which appropriate messages should be displayed. Do not use any external variable.

IsEmpty(*d*) := returns True if *d* is empty, otherwise returns False.

IsFull(*d*) := returns True if *d* is full, otherwise returns False.

AddAtFront(*d*, *x*) := adds *x* to the *front* of *d*.

AddAtRear(*d*, *x*) := adds *x* to the *rear* of *d*.

DeleteFromFront(*d*) := deletes an element *x* from the *front* of *d* and returns *x*.

DeleteFromRear(*d*) := deletes an element *x* from the *rear* of *d* and returns *x*.

Print(*d*) := prints all elements of *d* in the order starting from *front* and ending at *rear*.

9. Write a program to solve the valid parentheses problem. Take an infix expression with brackets as input from the user through the command line where each token of the infix expression is entered as a command-line argument in the correct order, as shown in the sample execution given below (the tokens are separated by blank spaces). If the brackets in the expression are correctly matched, then your program reports that it is a valid expression, otherwise your program reports that it is an invalid expression. Assume that all three types of brackets—parentheses/round brackets, braces/curly brackets, and square brackets—are allowed in the entered infix expression.

Sample Execution

```
$ ./check_brackets.out a + [ b * ( c - d / { e - f ) + g ] / h } ↵
Invalid expression (brackets not matched).
```

```
$ ./check_brackets.out a + [ b * { c – d / ( e – f ) + g } / h ] ↵
Valid expression (brackets are matched).
```

10. Write a program to evaluate an expression written in the infix notation where the tokens (i.e. the operands and the operators) of the expression are entered by the user as command-line arguments. Each token (an operator or an operand) will be separated from another token by a blank space at the command line, i.e. each token is a command-line argument. Your program should display the result of the expression as the final output. Assume that only the five operators '+', '-', '*', '/', '%' are allowed in the infix expression, that only integers are allowed as operands in the expression, and that no parentheses are allowed in the expression. The aforementioned five operators have their usual meaning, precedences, and associativity rules (as in C programming language).

    Sample Execution

    ```
    $ ./evaluate_infix.out 15 + 17 * 10 - 150 / 3 % 20 +  5 * 15 ↵
    Result: 250
    ```

11. Write a program to evaluate an expression written in the postfix notation where the tokens (i.e. the operands and the operators) of the expression are entered by the user as command-line arguments. Each token (an operator or an operand) will be separated from another token by a blank space at the command line, i.e. each token is a command-line argument. Your program should display the result of the expression as the final output. Assume that only the five operators '+', '-', '*', '/', '%' are allowed in the postfix expression, and that only integers are allowed as operands in the expression. The aforementioned five operators have their usual meaning, precedences, and associativity rules (as in C programming language).

    Sample Execution

    ```
    $ ./evaluate_postfix.out 15 17 10 * + 150 3 / 20 % - 5 15 * + ↵
    Result: 250
    ```

12. Write a program to transform an infix expression, containing no brackets, into a postfix expression. Take the infix expression as input from the user as command-line arguments where each token of the infix expression is entered as a command-line argument in the correct order, as shown in the sample execution given below (the tokens are separated by blank spaces). Your program will print the equivalent postfix expression as the final output. Assume that only the five operators '+', '-', '*', '/', '%' are allowed in the infix expression, that any real number or variable (denoted by letters from the English alphabet) is allowed as an operand in the expression, and that no parentheses are allowed in the expression. The aforementioned five operators have their usual meaning, precedences, and associativity rules (as in C programming language).

    Sample Execution

    ```
    $ ./infix_to_postfix.out a * b – c + d / e ↵
    a b * c – d e / +
    ```

13. Write a program to transform an infix expression, with brackets, into a postfix expression. Take the infix expression as input from the user as command-line arguments where each token of the infix expression is entered as a command-line argument in the correct order, as shown in the sample execution given below (the tokens are separated by blank spaces). Your program will print the equivalent postfix expression as the final output. Assume that only the five operators '+', '-', '*', '/', '%' are allowed in the infix expression, that any real number or variable (denoted by letters from the English alphabet) is allowed as an operand in the expression, and that brackets (parentheses, braces, and square brackets) are allowed in the expression. The aforementioned five operators have their usual meaning, precedences, and associativity rules (as in C programming language).

Sample Execution

```
$ ./infix_to_postfix.out a * [ b + { ( c - d ) / e + f } * g ] - h ↵
a b c d - e / f + g * + * h -
```

14. Write a program to implement two stacks with a single array. Take the elements of two stacks, *s1* and *s2*, as input from the user, and then store *s1* and *s2* in an array *A* such that the left end of *A* corresponds to the bottom of *s1* and the right end of *A* corresponds to the bottom of *s2*. Elements can be pushed into *s1* and *s2* as long as $s1_{count} + s2_{count} \leq A_{max}$, where $s1_{count}$ is the number of elements in *s1*, $s2_{count}$ is the number of elements in *s2*, and $A_{max}$ is the maximum size of *A*. Create a menu-based interface, similar to Question no. 2, with the following options: i) Push to s1, ii) Push to s2, iii) Pop from s1, iv) Pop from s2, v) Display s1, vi) Display s2, vii) Quit. The menu options are self-explanatory, and their definitions are similar to the ones given in Question no. 2. Do not use any external variable.

15. Write a program to implement multiple stacks with a single array. Your program will store *k* stacks, taking *k* as input from the user, in an array *A* of maximum size *max*, where *k* * *n* = *max* for some positive integer *n*. The *k* stacks will be initially stored by dividing *A* into *k* equal segments, each of size *n*, with the $i^{th}$ segment representing the $i^{th}$ stack, where *i* = 1 to *k*. Therefore, in the initial arrangment, the bottom of the $1^{st}$ stack will be stored in the $0^{th}$ index of *A*, the bottom of the $2^{nd}$ stack will be stored in the $(0+n.1)^{th}$ index of *A*, the bottom of the $3^{rd}$ stack will be stored in the $(0+n.2)^{th}$ index of *A*, and so on. Your program will create a menu-based interface where the user can choose to perform push and pop operations on any of the *k* stacks. The program should be able to perform push operations in a stack as long as there is free space available in *A*, implying that stacks may need to be shifted from their initial arrangement to accommodate a new element for a particular stack as long as there is free space available in *A* to make the shift operation possible (refer to the method of checking for free space in an array representing multiple stacks given in the textbook by Horowitz et al.). A stack will be considered empty if the index of its top element is less than the index of its bottom element. Your program will repeatedly display the menu until the user chooses to quit.

----------x----------