**Stacks and Queues**

**2A.** Aim: Write a program to reverse the given string using a stack.

**Algorithm:**

Step 1: Initialize a char array to store the string and input the string from the user.

Step 2: Get the length of the string by traversing it and incrementing counter till a \0 is

encountered.

Step 3: Allocate memory for a stack to push the characters in the string. Initialize top of stack to

-1.

Step 4: Traverse the characters in the string, and push each character to the stack.

Step 5: While pushing the characters in the stack, pre-increment the top of stack variable.

Step 6: After all the characters are pushed to the stack, pop all the characters and display the

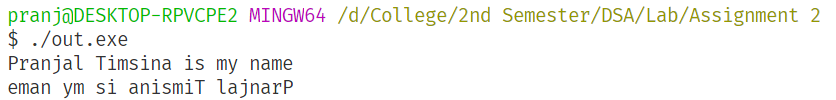
character until top of stack is equal to -1.

Step 7: End the program.

**Program:** [In next page]



**Output:**



**Results:**

Thus, the program to reverse the string using a stack is implemented.

**2B.** Aim: Write a program to implement a stack using a linked list to keep student's details such as name, reg no, age, etc.

**Algorithm:**

Step 1: Initialize variables to temporarily store the name, registration number and age the user

inputs.

Step 2: Initialize the head pointer for the linked list, which will also be the pointer to the top of

the stack.

Step 3: Until the user enters 0, ask the user to input data.

Step 4: Store the data that the user entered in a struct consisting of attributes name, registration

number, age, and a pointer to the next element in the linked list.

Step 5: Change the head pointer to point at the most recent data and the new head should point to

the previous head.

Step 6: Go to step 3.

Step 7: Pop the data from the stack until the head points to a NULL pointer.

Step 8: Display the data that was popped, update the head pointer and free the memory for the

data that was popped.

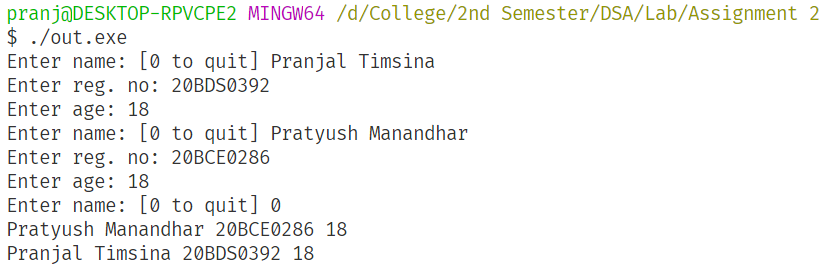
Step 9: End program

**Program:** [In next page]



****

**Output:**



**Results:**

Thus, the program to store student details using a stack implemented using a linked list was written.

**2C.** Aim: Write and implement a program to convert the given infix expression to a postfix expression.

**Algorithm:**

Step 1: Input the infix expression in a string variable.

Step 2: Initialize a stack to store the operators. Initialize top of stack to -1.

Step 3: Surround the infix expression with ‘(’ and ‘)’ to reduce complexity.

Step 4: Traverse the infix expression.

Step 5: If a number is encountered, append the number to the postfix expression.

Step 6: If a left parenthesis is encountered, push the left parenthesis to the stack.

Step 7: If a right parenthesis is encountered, pop all the operators until the matching left

parenthesis is found and append the operand to the postfix expression.

Step 8: If an operator is found and if the stack is empty push the operand to the stack.

Step 9: If an operator is found and if it has greater precedence than the operator on the top of the

stack, push the new operator to the stack.

Step 10: If an operator is found and if it has lower or equal precedence than the operator on the

top of the stack, pop all operators until the stack is empty or the operators on top of the stack has greater precedence that the new operator. Append the popped operator to the

postfix expression.

Step 11: Display the postfix expression.

**Program:** [In next page]

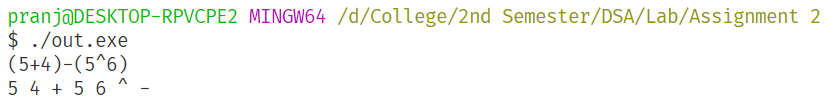








**Output:**

****

**Results:**

Thus, the program to change infix to postfix is implemented.

**2D.** Aim: Write and implement a program to evaluate infix expressions

**Algorithm:**

Step 1: Input the infix expression in a string variable.

Step 2: Initialize a stack to store the operators. Initialize top of stack to -1.

Step 3: Surround the infix expression with ‘(’ and ‘)’ to reduce complexity.

Step 4: Traverse the infix expression.

Step 5: If a number is encountered, append the number to the postfix expression.

Step 6: If a left parenthesis is encountered, push the left parenthesis to the stack.

Step 7: If a right parenthesis is encountered, pop all the operators until the matching left

parenthesis is found and append the operand to the postfix expression.

Step 8: If an operator is found and if the stack is empty push the operand to the stack.

Step 9: If an operator is found and if it has greater precedence than the operator on the top of the

stack, push the new operator to the stack.

Step 10: If an operator is found and if it has lower or equal precedence than the operator on the

top of the stack, pop all operators until the stack is empty or the operators on top of the stack has greater precedence that the new operator. Append the popped operator to the

postfix expression.

Step 11: Create another stack to store the operands. Initialize the top of stack to -1.

Step 12: Traverse the postfix expression from left to right.

Step 13: If a number is encountered, push the number to the stack.

Step 14: If an operator is encountered, pop two numbers from the stack and apply the operator,

and push the result back in the stack.

Step 15: Repeat step 13 and 14 until top of stack is -1.

Step 16: Pop and display the last number in the stack.

Step 17: End program.

**Program:** [In next page]

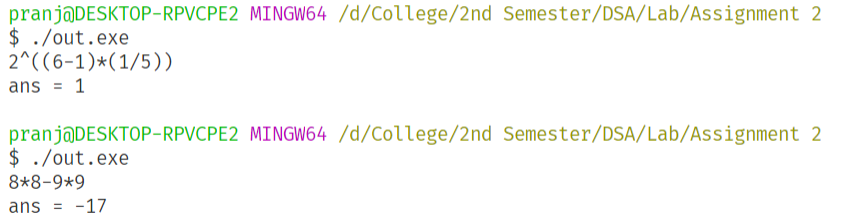








**Output:**

****

**Results:**

Thus, the program to evaluate infix expression is implemented.

**2E.** Aim: Implement a data structure to keep students’ details in the same order of their admission, with provision to add and remove in the same of their entry in the list.

**Algorithm:**

Step 1: Define a struct ‘node’ with attributes name, registration number, age and the pointer to

the next node.

Step 2: Initialize the head and tail of the queue to a NULL pointer.

Step 3: Implement the function enqueue which allocates memory to a new node, and stores the

relevant data to the new node. If the head pointer was previously NULL, make the head

pointer and the tail pointer to point to the new node. Set the next node attribute of the current new node to be NULL. If the head pointer was not NULL, make the tail of the queue point to the new node, and set the next node attribute of the current new node to be NULL.

Step 4: Implement the function dequeue which sets the value of head to what is pointed by it

currently. Print the details of the current head. Then free the memory allocated to the

current node.

Step 5: In the main program. While the user does not input 0, ask them to choose to enqueue, dequeue or exit.

Step 6: If the user chooses to enqueue, input the relevant data from the user and enqueue the data.

Step 7: If the user chooses to dequeue, dequeue and display the data pointed by the tail pointer.

Step 8: Go to step 5 until the user chooses to exit.

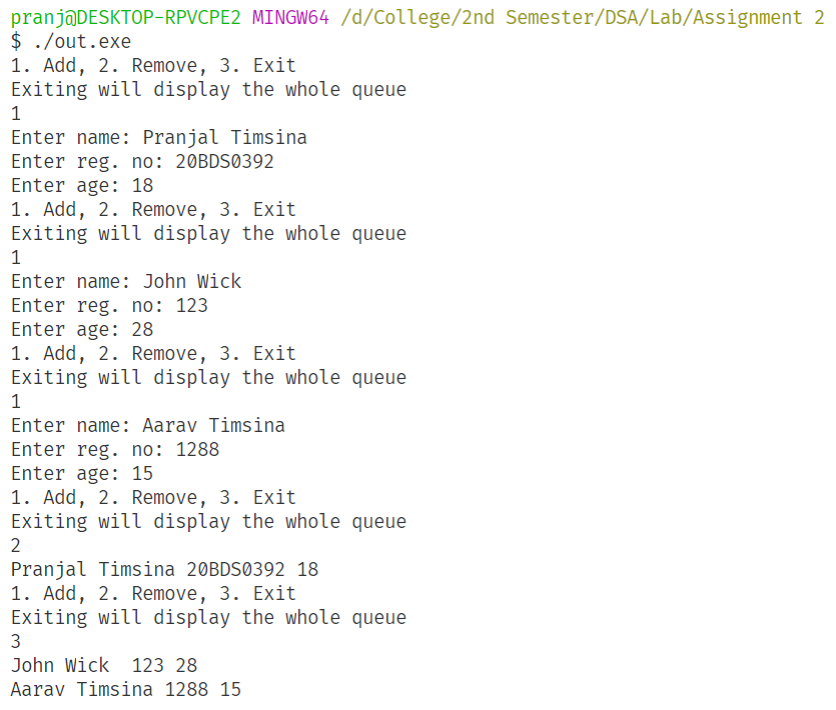
**Program:**







**Output:**



**Results:**

Thus, the program to [do something] is implemented with time complexity .

**2F.** Aim: Implement a data structure to keep sales details in the order of their entry. The number of entries is unknown and dynamically changing.

**Algorithm:**

Step 1: Define a struct ‘node’ with attributes product name, product id, price and the pointer to

the next node.

Step 2: Initialize the head and tail of the queue to a NULL pointer.

Step 3: Implement the function enqueue which allocates memory to a new node, and stores the

relevant data to the new node. If the head pointer was previously NULL, make the head

pointer and the tail pointer to point to the new node. Set the next node attribute of the current new node to be NULL. If the head pointer was not NULL, make the tail of the queue point to the new node, and set the next node attribute of the current new node to be NULL.

Step 4: Implement the function dequeue which sets the value of head to what is pointed by it

currently. Print the details of the current head. Then free the memory allocated to the

current node.

Step 5: In the main program. While the user does not input 0, ask them to choose to enqueue, dequeue or exit.

Step 6: If the user chooses to enqueue, input the relevant data from the user and enqueue the data.

Step 7: If the user chooses to dequeue, dequeue and display the data pointed by the tail pointer.

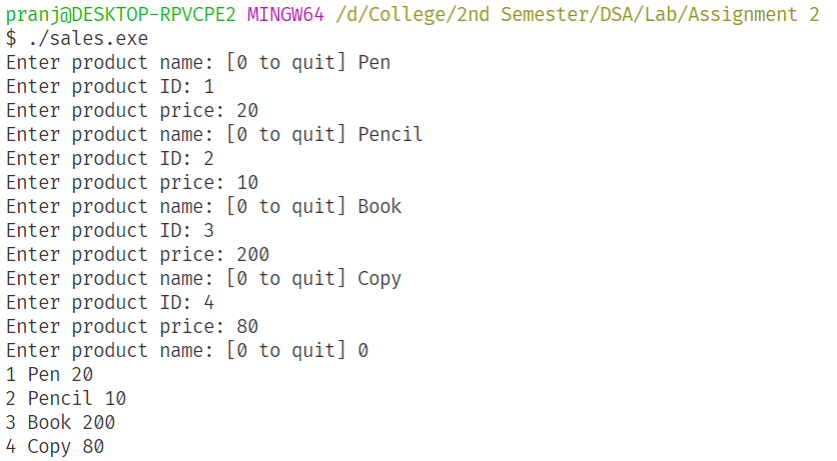
Step 8: Go to step 5 until the user chooses to exit.

**Program:** [In next page]





**Output:** [In next page]



**Results:**

Thus, the program to store the details of unknown number of sales entries is implemented in the order they are made is implemented using a queue.

**2G.** Aim: Implement a circular queue with functionality to add and remove data

**Algorithm:**

Step 1: Implement function enqueue that takes head, tail, size, data, and an array-based queue as

the arguments. If , then it initializes head and tail both to 0 then inserts data at queue [0]. Else if the queue is not full, then it sets , and inserts new data at queue [tail].

Step 2: Implement function dequeue that takes head, tail, size, and the array-based queue as the arguments and returns the element at the head of the queue. Then it sets the

if there still remain other elements in the queue, else sets it to -1.

Step 3: Implement function display queue that iterates cyclically from head to tail and prints the elements.

Step 4: Input the size of queue from the user and initialize an array of size N.

Step 5: Initialize the head and tail of the queue to -1.

Step 6: While the user does not enter 0, display the menu, and then add/remove/display data as

per the user’s input.

Step 7: End program.

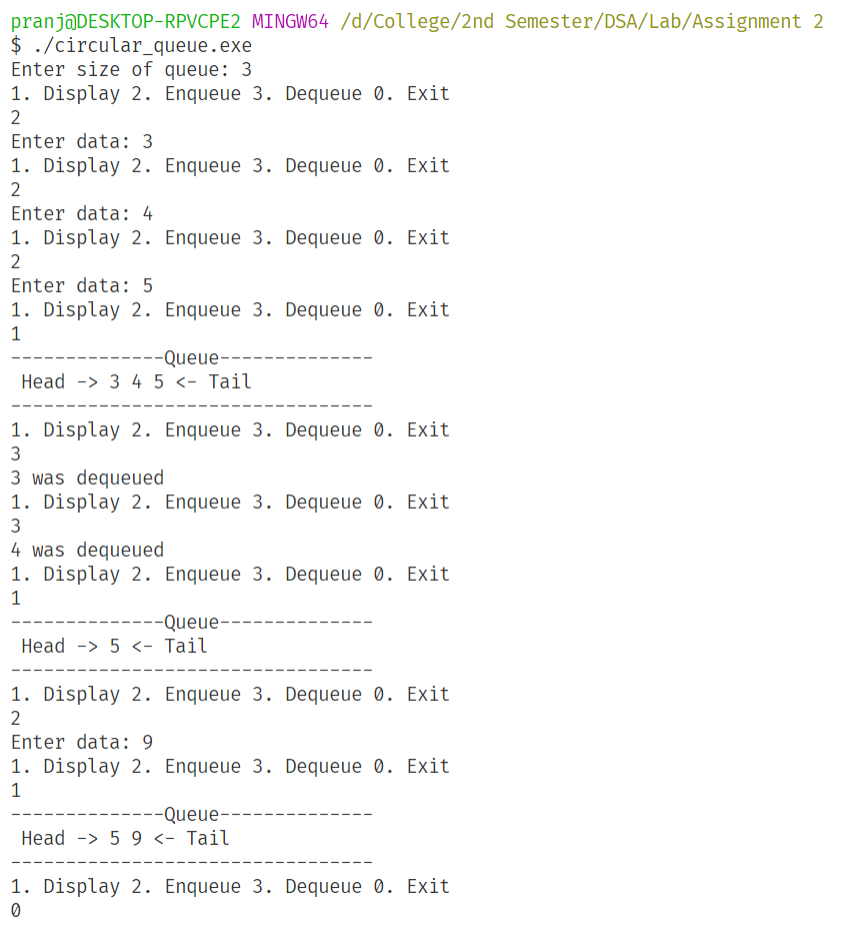
**Program:** [In next page]



[Continued in next page]



**Output:**



**Results:**

Thus, a circular queue is implemented.

**2H.** Aim: Implement a solution to Josephus problem using a circular linked list to identify and print the id of the winner. Users can get the number of players and the id of the starter.

**Algorithm:**

Step 1: …

Step 2: …

Step 3: …

Step 4: …

Step 5: …

Step 6: …

**Program:**

**Output:**

[Insert Output Here]

**Results:**

Thus, a circular queue is implemented with time complexity .

**2I.** Aim: Implement the Tower of Hanoi problem using recursion. User can give the number of disks; print each step of disk movement.

**Algorithm:**

Step 1: Input the number of discs

Step 2: Implement a recursive function shift described below.

Step 3: The shift function takes in arguments: number of discs, from tower, to tower, and

auxiliary tower.

Step 4: If the number of discs is 1 display Move N discs from *‘from tower’* to *‘to tower’*.

Step 5: Else shift n-1 discs from tower *‘from’* to tower *‘aux’* using tower *‘to’* as an auxiliary

tower. Then display Move N discs from *‘from tower’* to *‘to tower’.*

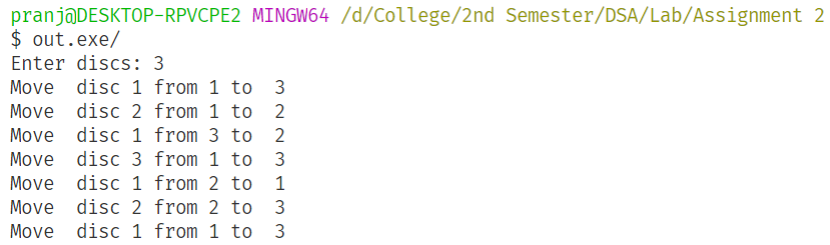
Step 6: Shift n-1 discs from tower *‘aux’* to tower *‘to’*.

Step 7: End program.

**Program:**



**Output:**



**Results:**

Thus, a solution to the Towers of Hanoi problem is implemented.

**2J.** Aim: Implement a program to perform polynomial addition using a linked list to represent polynomials.

**Algorithm:**

Step 1: …

Step 2: …

Step 3: …

Step 4: …

Step 5: …

Step 6: …

**Program:**

**Output:**

[Insert Output Here]

**Results:**

Thus, a circular queue is implemented with time complexity .