# Stack using linked list

Step 1 : start

Step 2 : Declare the node and the required variables

Step 3 : Declare the functions for push Pop display and search

Step 4 : Read the choice from the user to push, pop display or search an element

Step 5 : If the user choose to push an element then read the element to be pushead and call the functions to push the element by passing the value to the functions.

Step 5 : 1 : Declare the new node and allocate memory for the new node.

Step 5.2 : Set newnode → data = value.

Step 5.3 : Check if Top = = null.
then set newnode → next = null

Step 5.4 : Else set newnode → next = top

Step 5.5 : Set top = newnode and then
print insertion is successful.

Step 6 : If the user chooses to pop
an element from the stack
then call the function
to pop the element.

Step 6.1 : check if top = = null then
print stack is empty.

Step 6.2 : Else declare a pointer
variable temp and initialize
it to top.

Step 6.3 : print the element that is
being deleted.

Step 6.4 : set temp = temp → next.

Step 6.5 : free the temp.

Step 7 : If the user choose to display
the element in the stack
then call the function to
display the element in the
stack.

step 7.1 : check if top = = null the print
stack is empty

step 7.2 : Else declare a pointer variable
temp and initialize its top

step 7.3 : Repeate step 7.4 and 7.5
while temp -> next != null.

step 7.4 : print temp -> data

step 7.5 : set temp = temp -> next

step 8 : If the user choose to search
an element from the stack
then call the function to
search an element.

step 8.1 : Declare a pointer variable ptr
and other necessary variable.

step 8.2 : Initialize ptr = top.

step 8.3 : Check if ptr = null. then
print stack is empty.

step 8.4 : Else read the element to
be searched from user

step 8.5 : Repeate the step 8.6 and to
8.8 while ptr ! = null.

Step 8.6 : Check if ptr → data = = item then print element found and to location and set flag = 1

Step 8.7 : else set flag = 0.

Step 8.8 : Increment i by 1 and set ptr = ptr → next.

Step 8.9 : check if flag = = 0. then print element not found.

Step 9 : Stop.

(Enqueue, Dequeue) Circular Queue

Step 1 : Start.

Step 2 : Declare the queue and required variables.

Step 3 : Declare the function for enqueue, dequeue, display and search.

Step 4 : Read the choice from the user to enqueue, dequeue display and search.

Step 5 : If the user choose the option enqueue then read the element to be inserted from the user, the call the function enqueue and pass the value to the function.

Step 5.1 : check if front = = -1 and rear = = -1 then set front = 0, rear = 0 and set queue [rear] = element.

Step 5.2 : else if rear+1 mod max == front or front = = rear +1 then Print queue is overflow.

Step 5.3 : Else set rear = rear+1 mod max and set queue [rear] = element.

Step 6 : If the user choose the option dequeue then call the function dequeue

Step 6.1 : check if front = = -1 and rear = = -1 then print queue is underflow.

Step 6.2 : Else check if front == rear then print the element is to be deleted. Then set front = -1 and rear = -1.

Step 6.3 : Else print the element to be dequeued set front = front+1 mod max.

Step 7 : If the user choose the option to display the queue the call the function display.

Step 7.1 : check if front = = -1 and rear = = -1 then print queue is empty.

Step 7.2 : Else repeate the step 7.3 While i <= rear.

Step 7.3 : print queue [i] and set i = i+1 mod max.

Step 8. : If the user choose to search an element in the queue then call the function to search an element in queue.

8.1 : Read the element to be searched in the queue.

Step 8.2 : check if item = = queue[i] then print item found and its position and increment c by 1.

Step 8.3 : Check if c = = 0. then print item not founded.

Step 9 : End.

# Merging

Step 1 : start.

Step 2 : Delare the variables.

Step 3 : Read the size of the first array.

Step 4 : Read the element of first array in sorted order

Step 5 : Read the size of the second array.

Step 6 : Read the elements of second array in sorted order.

Step 7 : Repeat step 8 and 9 while $i < m$ & $j > n$

Step 8 : check if $a[i] >= b[j]$ then $c[k++] = b[j++]$

Step 9 : Else c [k++] = a [i++]

Step 10 : Repeat step 11 while i < m.

Step 11 : c [k++] = a [i++].

Step 12 : Repeat step 13 while j < n.

Step 13 : c [k++] = b [j++].

Step 14 : print the first array.

Step 15 : print the second array.

Step 16 : print the merged array.

Step 17 : End.

# Doubly linked list Operation

Step 1 : Start.

Step 2 : Declare a structure and related variables.

Step 3 : Declare functions to create node, insert a node in the beginning, at the end and given position, display the list and search an element in the list.

Step 4 : Define function to create a node, declare the required variables.

Step 4.1 : Set memory allocated to the node = temp.
then set temp → prev = null
and temp → next = null.

Step 4.2 : Read the value to be inserted to the node.

Step 4.3 : Set temp -> n = data and decrement count by 1.

Step 5 : Read the choice from the user to perform different operations on the list.

Step 6 : If the user choose to perform the insertion operation at the beginning then call the function to perform the insertion.

Step 6.1 : Check if head = = null then call the function to create a node, perform step 4 and 4.3.

Step 6.2 : Set head = temp and temp 1 = head.

Step 6.3 : Else call the function to create a node. perform step 4 to 4.3 then set temp -> next = head, set head -> prev = temp and head = temp.

Step 7 : If the user choice is to perform insertion at the end of the list, then call the function to perform the insertion at the end.

Step 7.1 : Check if head = = null then call the function to create a newnode then set temp = head and then set head = temp1.

Step 7.2 : Else call the function to creat a new node then set temp1 -> next = temp, temp -> prev = temp1 and temp1 = temp.

Step 8 : If the user choose to perform insertion in the list at any position then call the function to perform the insertion operation.

set head -> prev = temp and head = temp.

Step 7 : If the user choice is to perform insertion at the end of the list, then call the function to perform the insertion at the end.

Step 7.1 : Check if head = = null then call the function to create a newnode then set temp = head and then set head = temp1.

Step 7.2 : Else call the function to creat a new node then set temp 1 -> next = temp, temp -> prev = temp1 and temp1 = temp.

Step 8 : If the user choose to perform insertion in the list at any position then call the function to perform the insertion operation.

set head → prev = temp and head = temp.

Step 7 : If the user choice is to perform insertion at the end of the list, then call the function to perform the insertion at the end.

Step 7.1 : Check if head = = null then call the function to create a newnode then set temp = head and then set head = temp1.

Step 7.2 : Else call the function to creat a new node then set temp 1 → next = temp, temp → prev = temp1 and temp1 = temp.

Step 8 : If the user choose to perform insertion in the list at any position then call the function to perform the insertion operation.

Step 8.1 : Declare the necessary variable.

Step 8.2 : Read the position where the node and need to the inserted, set temp 2 = head.

Step 8.3 : Check if pos < 1 or pos >= count + 1. then print the position is out of range.

Step 8.4 : Check if head == null and pos = 1. then print "Empty list cannot insert and other other than 1st position.

Step 8.5 : Check if head == null. And pos = 1 then call the function to create newnode, then set temp = head and head = temp1.

Step 8.6 : While i < pos then set temp 2 = temp 2 -> next then increment i by 1.

Step 8.7 : Call the function to create a new node and then set temp -> prev = temp 2.

temp → next = temp 2 → next

→ prev = temp.

temp 2 → next = temp.

Step 9 : If the user choose to perform deletion operation is the list then all the function to perform the deletion operation.

Step 9.1 : Declare the necessary variable

Step 9.2 : Read the position where node need to be deleted set

temp 2 = head.

Step 9.3 : Check if pos < 1 or pos > = count + 1. then print position out of range.

Step 9.4 : Check if head = = null then print the list is empty.

Step 9.5 : While i < pos then temp 2 = temp 2 → next and increment i by 1.

temp → next = temp 2 → next

→ prev = temp.

temp 2 → next = temp.

Step 9 : If the user choose to perform deletion operation in the list then all the functions to perform the deletion operation.

Step 9.1 : Declare the necessary variable

Step 9.2 : Read the position where node need to be deleted set temp 2 = head.

Step 9.3 : Check if pos < 1 or pos > = count + 1. then point position out of range.

Step 9.4 : Check if head = = null then point the list is empty.

Step 9.5 : While i < pos then temp 2 = temp 2 → next and increment i by 1.

Step 9.6 : check if i = = 1 then check if temp 2 -> next = = null then print node deleted free (temp 2) set temp 2 = head = null.

Step 9.7 : check if temp 2 -> next = = null then temp 2 -> prev -> next = null. then, free (temp 2) then print node deleted.

Step 9.8 : temp 2 -> next -> prev = temp 2 -> prev. then check if i ! = 1 then temp 2 -> prev -> next = temp 2 -> next.

Step 9.9 : check if i = = 1 then head = temp 2 -> next. then print node deleted then free temp 2 and decrement count by 1.

Step 10 : If the user choose to perform the display operation then call the function to display the list.

Step 10.1 : set temp 2 = n.

Step 10.2 : Check if temp 2 = null then print list is empty

Step 10.3 : While temp 2 → next = null then print temp 2 → n then temp 2 = temp 2 → next.

Step 11 : If the user choose to perform the search operation then call the function to perform search operation.

Step 11.1 : Declare the necessary variable.

Step 11.2 : Set temp 2 = head.

Step 11.3 : check if temp 2 = = null then print the list is empty.

Step 11.4 : Read the value to be searched.

Step 11.5 : While temp 2 ! = null. the check if temp 2 → n = = data then print element found at position count + 1.

Step 11.6 : Else set temp 2 = temp 2 ->
next and increment count by 1.

Step 11.7 : Print element not found
in the list

Step 12 : End.

# Set Operations

Step 1 : Start.

Step 2 : Delare the necessary variables.

Step 3 : Read the choice from the user to perform set operation.

Step 4 : If the user chouse to perform union :

Step 4.1 : ~~If the users Read~~ the cardinality of 2 sets

Step 4.2 : Check if $m_1 = n$ then print cannot perform union.

Step 4.3 : else read the elements in both the sets.

Step 4.4 : Repeat the step 4.5 to 4.7 until $i < m$.

Step 4.5 : $C[i] = A[i] | B[i]$

Step 4.6 : print $C[i]$

Step 4.7 : Increment i by 1.

Step 5 : Read the choice from the user to perform intersection.

Step 5.1 : Read the cardinality of 2 sets.

Step 5.2 : check if m!=n then print cannot perform intersection.

Step 5.3 : Else read the elements in both the sets.

Step 5.4 : Repeat the step 5.5 to 5.7 until i<m.

Step 5.5 : C[i] = A[i] & B[j]

Step 5.6 : print C[i].

Step 5.7 : Increment i by 1.

Step 6 : If the user choose to perform set difference operation.

Step 6.1 : Read the cardinality of 2 sets.

Step 6.2 : check if m != n then print can not perform set difference operation.

Step 6.3 : Else read the element in both sets

Step 6.4 : Repeat the step 6.5 to 6.8 Until $i < n$.

Step 6.5 : Check if $A[i] == 0$. then $C[i] = 0$.

Step 6.6 : Else if $B[i] = = 1$ then $C[i] = 0$.

Step 6.7 : Else $C[i] = 1$

Step 6.8 : Increment i by 1.

Step 7. : Repeat the step 7.1 and 7.2 Until $i < m$.

Step 7.1 : print $C[i]$.

Step 7.2 : Increment i by 1.

# Binary Search Tree

Step 1 : Start.

Step 2 : Declare a structure and structure pointers for insertion deletion and search operations and also declare a function for inorder traversal.

Step 3 : Declare a pointer as root and also the required variable.

Step 4 : Read the choice from the user to perform insertion, deletion, searching and inorder traversal.

Step 5 : If the user choose to perform insertion operation then read the value which is to be inserted to the tree from the user.

Step 5.1 : pass the value to the insert pointer and also the root pointer

Step 5.2 : check if ! root then allocate memory for the root

Step 5.3 : set the value to the info part of the root and then set left and right part of the root to null and return root.

Step 5.4 ; check if root -> info > x then call the insert pointer to insert to left of the root

Step 5.5 : check if root -> info x. then call the insert pointer to insert to the right of the root.

Step 5.6 : Return the root.

Step 6 : If the user choose to perform deletion operation then read the element to be deleted from the tree pass the root pointer and the item to the delete pointer

step 6.1 : check if not ptr then print node no node found.

step 6.2 : Else if ptr → info ex the call delete pointer by passing the ~~left~~ right pointer and the item.

Step 6.3 : else by passing the right pointer.

6.4 : check if ptr → info == item then check if ptr → left == ptr → right then free ptr and return null.

step 6.5 : Else if ptr → left == null. set P1 . ptr → right and free ptr, return P1.

step 6.6 : else if ptr right == null set P1 = ptr → left and free ptr, return P1.

Step 6.7 : Else set P1 = ptr → right and P2 = ptr → right.

Step 6.8 : While P1 → left not equal to null, set P1 → left. ptr → left and free ptr, return P2.

Step 6.9 : Return ptr.

Step 7 : If the user choose to perform search operation the call the pointer to perform search operation.

Step 7.1 : Declare the necessary pointers and variables.

Step 7.2 : Read the element to be searched.

Step 7.3 : while ptr check if item > ptr → info then ptr = ptr → right.

Step 7.4 : Else if item < ptr → info then ptr = ptr → left.

step 7.5 : else break.

step 7.6 : check if ptr then print that the element is found.

step 7.7 : else print element not found in tree and return root.

Step 8 : If the user choose to perform traversal then call the traversal function and pass the root pointers

Step 8.1 : If root not equals to null recursively call the function by passing root -> left

Step 8.2 : print root -> Info.

Step 8.3 : Call the traversal function recursively by passing root -> right.