MERGING

Step 1 : Start

Step 2 : Declare the variables

Step 3 : Read the size of first array

Step 4 : Read elements of first array in sorted order

Step 5 : Read the size of 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 $c[k+1] = a[i++]$

Step 10 : Repeat step 11 while $i<m$

Step 11 : $c[k++] = a[j++]$

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.

STACK OPERATION

Step 1 : start

Step 2 : Declare the node and the required variables

Step 3 : Declare the functions for push, pop, display and search an element

Step 4 : Read the choice from the user

Step 5 : if the user choose to push an element, then read the element to be pushed & call the function to push the element by passing the value to the function.

Step 5.1 : Declare the newnode & allocate memory for the newnode

Step 5.2 : set newNode → data = value

Step 5.3 : check if top = = null then set newNode → next = null

Step 5.4 : set newNode → next = top

Step 5.5 : set top = newNode & then print insertion is successful

Step 6 : if user choose 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 being deleted

Step 6.4 - Set temp = temp → next

Step 6.5 - free the temp

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

Step 7.1 : Check if top == Null then print stack is empty

Step 7.2 : Else declare a pointer variable temp & initialize it to top.

Step 7.3 : Repeat steps below 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 : ~~If~~ 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 empty

Step 8.4 : Else read the element to be searched

Step 8.5 : Repeat step8.6 to 8.8 while ptr != null

Step 8.6 - check if ptr → data == item then print

element founded and to be located 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 the element not found

Step 9 : End

Circular Queue Operation

step 1 : start

step 2 : Declare the queue and other variables.

step 3 : Declare the functions for enqueue - dequeue search and display.

step 4 : Read the choice from the user

step 5 : if the user choose the choice enqueue then Read the element to be inserted from the user and call the enqueue function by passing the value.

step 5-1 : check if front == -1 && rear == -1 then set front = 0, rear = 0 and set queue [rear] = element.

step 5-2 : Else if rear + 1 % max == front or front = rear + 1 then print Queue is overflow.

step 5-3 : Else set rear = rear + 1 % max and set queue [rear] = element.

step 6 : if ~~the~~ user choice is 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 % max

Step 7 : If the user choice is to display the Queue then call the function display

~~Step 7.1 : If the user choice is to display~~

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

Step 7.2 : Else repeat the step 7.3 while i <= rear

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

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

Step 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 i by 1

Step 8-3 : check if c == 0 then print item not found

Step 9 : ~~Else~~ End.

Doubly linked list Operation

step 1 : start

step 2 : Declare a structure and related variables

step 3 : Declare functions to create a node, insert a node in the begening 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 increment count by 1

step 5 : Read the choice from the user to perform different operation on the list

step 6 : if the user choose to perform 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 to 4.3

step 6.2 : set head = temp and temp1 = head

step 6-3 : Else call the function to create a node perform step4 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 create a new node then set temp1→next = temp, temp→prev = temp1 and temp1 = temp

step 8 : if the user choose to perform insertion on 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 need to the inserted, set temp2 = 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 other than 1st position.

Step 5-5 : check if head == null and pos = 1 then call the function to create newnode, then set temp = head and head = temp 1

Step 5-6 : while i < pos then set temp 2 = temp2 → next then increment i by 1.

Step 5-7 : 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 in the list then all the function to perform the deletion operation.

Step 9-1 ; Declare the necessary variables

Step 9-2 ; ~~Declare~~ 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 : checks 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

step 9-6 : check if i==1 then check or if temp2->next == null then print node deleted Free(temp2) Set temp2 = head = null

step 9-7 : check if temp2 -> next == null then temp2 -> prev -> next = null then free(temp2) then print node deleted.

step 9-8 : temp2 -> next -> prev = temp2 -> prev then check if c1 = 1 then temp2 -> prev -> next = temp2 -> next

step 9-9 : check if i=1 then head = temp2 -> 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 temp2 = null then print list is empty

step 10-3 : while temp 2 -> next1 = null then print temp2 -> 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 to necessary variables.

step 11-2 : Set temp2 = 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 temp2 != null the check if temp 2 -> n = ~~t~~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 : Declare the necessary variable

step 3 : Read the choice from the user to perform set operation

step 4 : If the user choose to perform union

step 4-1 : Read the cardinality of 2 sets

step 4-2 : check if m1 = 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 : Read check if m1=n then print cannot perform intersection

step 5-3 : Else read the elements is 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 m1=n then print cannot 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 : ~~check~~ 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 requirred 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 ~~from~~ 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 opeation 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 not found

step 6-2 : else if ptr → info < x then call deleted pointer by passing the right pointer and the item.

step 6-3 : else if ptr → info > x then call delete pointer by passing the left pointer and the item

step 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 then set P1 . ptr → right and free ptr, return Pi

step 6-6 : else if ptr → right == null then set Pi = ptr → left and free ptr, return Pi

step 6.7 : else set P1 = ptr → right and P2 = ptr → right

step 6.8 : while Pi → left not equal to null, set Pi → 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 panter 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~~ 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 functions by passing root → left

step 8-2 - print root → info

step 8-3 - call the traversal function recursivly by passing root → right.

step 1 - Start

step 2 - Declare the structure and related structure variable

step 3 - Declare a function makeSet()

step 3.1 - Repeat step 3.2 to 3.4 until i<n

step 3.2 - dis.parent[i] is set to i

step 3.3 - set dis.rank[i] is equal to 0

step 3.4 - increment i by 1

step 4 - Declare a function display set

step 4.1 - Repeate step 4.2 and 4.3 until i<n

step 4.2 - print dis.parent[i]

step 4.3 - increment i by 1

step 5 - Declare a function find and pass x to the function.

step 5.1 - check if dis.parent[n] != x then set the return value to dis.parent[x]

step 5.2 - return dis.parent[n]

step 6 - Declare a function union and pass two variables x and y

step 6.1 - set x set to find(x)

step 6.2 - Set y set to find (y)

step 6.3 - check if xset == yset then return

step 6.4 - Check if dis.rank[xset] < dis.rank[yset] then

step 6.5 - set yset = dis.parent[yset]

step 6.6 - set -1 to dis.rank[xset]

step 6.7 - Else if check dis.rank[xset] > dis.rank [yset]

step 6.8 - Set xset to dis.~~rank~~parent[yset]

step 6.9 - Set -1 to dis.~~parent~~rank[yset]

step 6.10 - Else dis.parent[yset] = xset

step 6.11 - Set dis.rank[yset]+1 to dis.rank[xset]

step 6.12 - Set -1 to dis.rank[yset]

step 7 - Read the number of elements

step 8 - call the function make set

step 9 - Read the choise from user to perform union find and display operation

step 10 - If the user choice to perform union operation. read the element to perform union operation.

Step 11 - If the user choose to perform Find operation read the element to check if connected

Step 11.1 - check if find(x) == find(y) then print connected component

Step 11.2 - Else print not connected Component

Step 12 - If the user choose to performs display operation to call the function displayset

Step 13 - End.