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 scond array.

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

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

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 : Stop.

Stack Operations

Step 1 : start

Step 2 : Declare the node and the required variables.

Step 3 : Declare the function for push, pop, display and search

Step 4 : Read the choice from user.

Step 5 : If the user chose to push an element then read the element to be pushed and call the function to push element by passing 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 successfull.

Step 6 : If user chose 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 intialise 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 in stack

step 7.1 : check if top = = Null then print stack is empty.

step 7.2 : else declare the pointer variable temp and initialise 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 chose 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 variables.

step 8.2 : initialise ptr = top

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

step 8.4 : else read the element to be searched

step 8.5 : Read step 8.6 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

Scanned by TapScanner

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 : stop.

Circular Queue Operations

Step 1 : Start

step 2 : Declare the que and other variables

Step 3 : Declare the function 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 function dequeue

Scanned by TapScanner

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 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 it's position and increment i by 1 step 8.3 : check if c = 0 then print item not found

step 9 : END.

Doubly Linked List Operation

Step1: start

Step2: Declare a structure and related variables

step3: Declare function to create a node, insert a node in the begining, at end and given position, display the list and search an element in the list.

step4: Define function to create a node declare the required variables

step 4.1: ~~Define~~ function ~~to create a~~ node, set memory allocated to node = temp then set temp-> prev = null and temp->next = null

step4.2: Read the values to be inserted to the node

step4.3: set temp->n = data and increment count by 1

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

step 6: If the user choose to perform insertion operation at the begining then call the function to perform insertion

step 6.1: check if head == null then call the function to create a node, perform ~~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 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 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 set head = temp1

step 7.2: Else call the function to create a newNode 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

step 8.1: Declare the necessary variables.

step 8.2: Read the position where the node need to be inserted, step set temp2 = head.

step 8.3: check if pos<1 or pos>= count+1, then print position is out of range.

step 8.4: check if head = = null and pos = 1 then print "empty list can't insert other than 1'st 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 newNode 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 call the function to perform the deletion operation.

step 9.1: Declare the necessary variables.

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 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 temp2 -> next = = null then print node deleted free (temp2). set temp2 = head = null

step 9.7 : Check if temp 2 -> next = = null then temp 2 -> prev -> next = null then free (temp2) 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 temp2 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 temp2 -> n then temp 2 = temp2 -> next.

step11: If the user chose to perform the search operation then call the function to perform search operation.

step11.1 : Declare the necessary variables.

step11.2 : Set temp2=head.

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

step 11.4 : Read the values to be searched.

step 11.5 : while temp 2!=null then check if temp2->n == data then print element found at position count+1

step 11.6 : Else set temp 2 = temp2-> next and increment count by 1

step11.7 : Print element not found in the list.

step 12 : End.

Scanned by TapScanner

SET Operation

step1 : Start

step2 : Declare the necessary variables.

step3 : Read the choice from the user to perform set operations.

step 4 : If user chose to perform union

step4.1 : Read cardinality of two sets.

step4.2 : Check if m!=n then print "operation Not possible".

step4.3 : Else read the elements in both the set.

step 4.4 : Repeat step 4.5 to 4.7 until i<m

step4.5 : c[i] = A[i] | B[i]

step4.6 : print c[i]

step 4.7 : increment i by 1

step5 : Read the choice from user to perform intersection

step 5.1 : Read cardinality of 2 sets

step 5.2 : check if m!=n then print operation not possible

step 5.3 : Else read the elements in both sets

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

step5.5 : c[i] = A[i] & B[i]

step5.6 : print c[i]

step 5.7: incrament 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 "operation not possible".

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 RB[i]=1 then c[i]=0

step 6.7: else c[i]=1

step 6.8: incrament i by 1

step 7: Repeat the step 7.1 and 7.2 until i<m

step 7.1: print [i]

step 7.2: incrament i by 1

Binary Search Tree

Step1: start.

Step2: Declare a structure and structure pointers for insertion deletion and search operations and also declare a function for inorder traversal.

Step3: Declare a pointer as root and also the required variable.

Step4: Read the choice from the user to perform insertion deletion, searching and inorder traversal

Step5: If the user choice is to perform insertion operation then read the value which is to be inserted to the tree from the user.

Step5.1: Pass the value to the insert pointer and also the root pointers.

Step5.2: check if !root then allocate memory for root.

Step5.3: set the value to the info part of the root and then set left and right part of root to null and return root.

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

Scanned by TapScanner

step 5.5: check if root->info < x then call insert pointer to insert to the right of 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 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 call the delete 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 P1.

step 6.6 : Else if ptr->right = =null then 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 elements 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 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 equal to null recursively call

the functions by passing root -> left.

step 8.2: Print root -> info

step 8.3: call the traversal function recursively by Passing root -> right