MERGING

Step 1: Start

Step 2: Declare the variables

Step 3: Read the size of the 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 $k < m$ & $i < n$

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 $k < m$

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 operations

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 the pushed & call then function to push the element by passing the value to 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 -> data = value

Step 5.5 : set top = newnode & then print insertion is successfull

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 pop == NULL then print stack is empty

Step 6.2 : else declare a pointer variable temp and initialise 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 the stack.

Step 7.1 : check if top == NULL then print stack is empty

Step 7.2 : else declare a pointer variable temp & initialise it to top

Step 7.3 : Repeat steps below ~~free~~ while temp -> next != NULL

Step 7.4 : print temp -> data

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 : initialise 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 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

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 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 deleted 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 position and increment i by 1.

Step 8.3: check if c==0 then print item not found

Step 9: end.

Linked list operation

Step 1: Start

Step 2: declare a structure and related variables.

Step 3: declare function to create a 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: design 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 by 1

Step 5: Read the above 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 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 step 4 and 4.3 then set temp → next = head, head → prev = temp and head = temp.

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

Step 7.1 : ~~If the user~~

check if head == null then call the function to create a new node then set temp = head and then set head = temp1

Step 7.2 : else ~~call~~ 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 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 need to be 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 call push "empty list cannot insert 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 temp2 = temp2->ne
then increment by 1.

Step 8.7 : call the function to create a new node and then set temp->prev = temp2,
temp->next = temp2->next->prev
temp2->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 neceessary variables.

Step 9.2: Read the position where node need to be deleted set temp2 = 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 temp2 = temp2 -> 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: chek 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 i!=1 then temp2 -> prev -> next = temp2 -> next.

Step 9.9 : check if i=1 then head = temp2 → next then point 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 temp2 = head

Step 10.2: check if temp2 = null then print list is empty

Step 10.3: while temp2 → next != null then print temp2 → n then temp2 = temp2 → 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 variables

Step 11.2 : set temp = head

Step 11.3 : Check if temp2 == null the print list is empty

Step 11.4 : Read the value to be searched.

Step 11.5 : while temp2 != null then check if temp2 → n == data then print element found at position count+1

Step 11-6 : else set temp2 = temp2 → next and

movement count by 1

Step 11-7: point element not found in the list

Step 12 : end.

Set operations

Step 1 : start

Step 2 : declare the variables

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

Step 4 : If user choose to perform union

Step 4.1 : Read the cardinality of the two sets

Step 4.2 : check if m!=n then print cannot perform union.

Step 4.3 : else read the elements in both the set

Step 4.4 : Repeat the ~~setet~~ step 4-5 to 4.7 until l<m

Step 45 : to check 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 sets.

Step 5.2 : If m1=0 the point cannot perform intersection.

Step 5.3 : else read the element is both the sets

Step 5.4 : Repeat the step 5.5 to 5.7 until k=0

Step 5.5 : c(i) = a[i] & b[i]

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=0 then point cannot perform Set ~~area~~ difference operation.

Step 6.3 : else read the element in both sets

Step 6.4 : Repeat the step 6-5 to 6-8 until k=0.

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

step : end

search tree

Step 1: Start

Step 2: declare a structure and structure pointer 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 choice to perform insertion operation then read the value which is to be inserted to the tree from the user.

Step 5.1: Store value to be inserted ~~the~~ the pointed and ~~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 the call insert pointer to insert to right. of the root.

Step 5.6: return root

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

Step 6.1: check if root ptr then print node not found

Step 6.2: else if ptr->info < x then call delete pointer by passing the right pointer and the item.

Step 6.3 else if ptr -> info > x the call delete pointer by using 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 ptr = ptr->right and free ptr, return p)

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 the ptr below p2.

Step 6.9: return ptr

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

Step 7.1 declare the 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 point that the element is found.

Step 7.7: else point element not found in the tree and return 0006.

Step 8: 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 recursively by passing root→right.

Step 9: end

disjoint, set

Step 1: start

Step 2: declare structure and defined structure variable

Step 3: declare a function makeset()

Step 3.1 Repeat step 3.0 to 3.4 until i<n

Step 3.2: set dis.parent[] is set to i

Step 3.3: set dis.rank[] is is equal to 0

Step 3.4: increment i by 1.

Step 4.1: declare a function display set.

Step 4.1: Repeat step 4.2 and 4.3 until i<n

Step 4.2: Print dis.parent[i]

Step 4.3: Increment i by 1

Step 4.4: Repeat step 4.5 and 4.6 until i<n

Step 4.5: Print dis.rank[i]

Step 4.6: Increment i by 1.

5: declare a function find and pass x to the function

Step 5.1: check if dis.parent[x]!=i then set the return value to dis.parent[x]

Step 5.2: return dis.parent[x]

Step 6.3: declare a function and 1000 variable

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]

Step 6.5 : set yset = dis.parent [y set]

Step 6.6 : set -1 to dis rank [x set]

Step 6.7 : else if check dis.rank [xset] > dis.root [yset]

Step 6.8 : Set x set to disparent [y set]

Step 6.9 : set -1 to d.s.rank [yset]

Step 6.10 : and the dis.parent [ysef] = x setc.

Step 6.11 : set dis.rank [x.set] +1 to dis.rank [xset]

Step 6.12 : set -1 to dis.rank [y set]

Step 7 : Read no of elements.

Step 8 : Call the function makeset

Step 9 : Read the choice from to perform union and find display operation

Step 10 : If the user chooses to perform union operation read the elements to perform union operation. Then call the function to perform union operation.

Step11 : 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 components

Step 11.2 : else print the not connected componet

Step 12 : If the user choose to perform display operation call the function display set

Step 13 : end .