MERGING

Step 1 : Start

Step 2 : Declare the Variables

Step 3 : Read the size of First array

Step 4 : Read element of first array in Sorted order.

Step 5 : Read the size of Second array.

Step 6 : Read the element 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[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 function for push, pop, display and search an element.

Step 4: Read the choice from user.

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

Step 5.1: Declare the newnode and 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 and 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 pop == 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 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 and initialise it to top.

Step 7.3: Print temp -> data.

Step 7.4: Set temp = temp -> next

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

Step 8.1: Declare the 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 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 for choice enqueue, then read the element to be inserted from the user & 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

Date / /

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 : 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 & its position and increment i by 1.

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

Step 9 : 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 begining, 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.

Camlin Page
Date / /

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 & temp1 = head

step 6.3: else call the function to create a node. Perform step 4 - 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

Camlin Page
Date / /

Perform the insertion at the end.

step 7.1 : check if head = = null then call the function create a newnode then set temp = head & 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 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 the inserted, set temp2 = head

Camlin Page
Date / /

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 & pos=1 then print "Empty list cannot insert other than 1st position"

step 8.5 : check if head==null & 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->next then increment i by 1.

step 8.7 : call the function to create a new node and then set
temp->prev=temp2.
temp->next=temp2->next->prev=temp.
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 necessary variable.

step 9.2: Read the position where node need to be deleted set temp2=head

step 9.3: check if pos<1 or pos>=count1. 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=temp->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 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 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 temp2 = n

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 operations operations

Camlin Page
Date / /

step 11.1 : Declare the necessary variables
step 11.2: set temp2 = head
step 11.3 : check if temp2 == 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 == data then print
element found at position count
+1
step 11.6 : Else set temp2 = temp2 -> next
and increment count by 1
step 11.7 : Print element not found in the
list
step 12 : End

Camlin Page
Date / /

SET OPERATIONS

step 1 : start

step 2 : Declare the neccessary Variable

step 3 : Read the choice from 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 m!=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 - 5.7 until i<m

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 m!=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 - 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

step 8 : stop

Camlin Page

Date / /

BINARY 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 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 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 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 then call the pointer to perform search operation.

step 7.1 : Declare the neccessary Pointer 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

Camlin Page
Date / /

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 pointer.

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 recursively by passing

Root -> Right

Step 9 : Stop.

DIS JOINT

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 - 3.4 until i<n
step 3.2 : dis.parent[x] is set to i
step 3.3 : Set dis.rank[c] 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 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

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

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

step 5.2: return dis.parent[x]

step 6: Declare a function union & pass two variable x & y

step 6.1: set x set to find(x)

step 6.2: set y set to find(y)

step 6.3: check if $x$set == $y$set then return.

step 6.4: check if dis.rank[$x$set] < dis.rank[$y$set]

step 6.5: set $y$set = dis.parent[$y$set]

step 6.6: set -1 to dis.rank[$x$set]

step 6.7: Else if check dis.rank[$x$set] > dis.rank[$y$set].

step6.8: Set x set to dis.parent[y set]
Step 6.9: set -1 to dis.rank[ySet]
Step 6.10: else dis.parent[y set] = x set
step 6.11: set dis.rank[x set] + 1 to dis.rank[x set]
Step 6.12: Set -1 to dis.rank[y set]
step 7: Read the number of elements
step 8: Call the function makeset
step 9: Read the choice from user to perform union find and display operation.
step 10: If the user choose to perform union operation read the element to perform union operations.
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 perform display operation call the function display set

step 13 : End.