

MERGING

Step 1 : Start

Step 2 : Declase the variable

Step 3 : Read the size of first array.

Step 4 : Read the 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 & 9 while $i \leq m$ & $j \leq n$

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

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

Step 10 : Repeat step 11 while $i \leq m$

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

Step 12 : Repeat step 13 while $j \leq n$

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 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 & call the function to push the element by passing the value to the function

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

Step 5.2: Set newnode \rightarrow data = Value

Step 5.3: Check if top = null then set newnode \rightarrow next = null

Step 5.4: Set newnode \rightarrow 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 \rightarrow next

Step 6.5: free the temp

Step 7: If the user choose the display then call the functions 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 & initialize it to top

Step 7.3: Repeat steps below while temp \rightarrow next != null

Step 7.4: Print temp \rightarrow data

Step 7.5: Set temp = temp \rightarrow next

Step 8: If the user choose to search an element from the stack then call the functions 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 $\text{ptr} = \text{null}$ 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 $\text{ptr} \neq \text{null}$

Step 8.6: Check if $\text{ptr} \rightarrow \text{data} == \text{item}$ 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 $\text{ptr} = \text{ptr} \rightarrow \text{next}$.

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 variable

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 $\text{front} == -1 \& \& \text{rear} == -1$ then set
 $\text{front} = 0$, $\text{rear} = 0$ and set $\text{queue}[\text{rear}] = \text{element}$.

Step 5.2 : Else if $\text{rear} + 1 \% \text{max} == \text{front}$ or $\text{front} == \text{rear} + 1$
then print queue is overflow

Step 5.3 : Else set $\text{rear} = \text{rear} + 1 \% \text{max}$ and set
 $\text{queue}[\text{rear}] = \text{element}$.

Step 6 : If the user choice is the option dequeue then
call the function dequeue .

Step 6.1 : Check if $\text{front} == -1$ and $\text{rear} == -1$ then print
queue is under flow .

Step 6.2 : Else check if $\text{front} == \text{rear}$ then print the element
is to be deleted then set $\text{front} = -1$ and
 $\text{rear} = -1$

Step 6.3: Else print the element to be dequeued Set
 $\text{front} = \text{front} + 1 \% \max$

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

Step 7.1: Check if $\text{front} = -1$ and $\text{rear} = -1$ then print
the queue is empty.

Step 7.2: Else repeat the step 7.3 while $i < \text{rear}$

Step 7.3: print $\text{queue}[i]$ 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 $\text{item} == \text{queue}[i]$ 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: 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 beginning , at the end and given position, display the list and search an element in the list.

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

Step 4.1: Set memory allocated to the node = temp , then set $\text{temp} \rightarrow \text{prev} = \text{null}$ and $\text{temp} \rightarrow \text{next} = \text{null}$

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

Step 4.3: Set $\text{temp} \rightarrow \text{n} = \text{data}$ 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 operations at the beginning then call the functions to perform the insertion .

Step 6.1: Check if $\text{head} == \text{null}$ then call the functions to create a node, perform step 4 to 4.3.

Step 6.2: Set $\text{head} = \text{temp}$ and $\text{temp1} = \text{head}$

Step 6.3: Else call the function to create a node, perform step 4 to 4.3 then set $\text{temp} \rightarrow \text{next} = \text{head}$, set $\text{head} \rightarrow \text{prev} = \text{temp}$ and $\text{head} = \text{temp}$.

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 $\text{head} == \text{null}$ then call the function to create a newnode then set $\text{temp} = \text{head}$ and then set $\text{head} = \text{temp}$

Step 7.2: Else call the function to create a newnode then set $\text{temp} \rightarrow \text{next} = \text{temp}$, $\text{temp} \rightarrow \text{prev} = \text{temp}$ and $\text{temp} \rightarrow \text{next} = \text{temp}$.

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 $\text{temp} \rightarrow \text{head}$.

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

Step 8.4: Check if $\text{head} == \text{null}$ and $\text{pos} == 1$ then print "empty list" cannot insert other than 1st position.

Step 8.5: Check if $\text{head} == \text{null}$ and $\text{pos} = 1$ then call the function to create newNode , then set $\text{temp} = \text{head}$ and $\text{head} = \text{temp}$

Step 8.6: while $i < \text{pos}$ then set $\text{temp}^2 = \text{temp}^2 \rightarrow \text{next}$ then increment i by 1

Step 8.7: Call the function to create a newNode and then set $\text{temp} \rightarrow \text{prev} = \text{temp}^2$, $\text{temp} \rightarrow \text{next} = \text{temp}^2 \rightarrow \text{next} \rightarrow \text{prev} = \text{temp}$, $\text{temp}^2 \rightarrow \text{next} = \text{temp}$.

Step 9: If the user choose to perform deletion operation is 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 $\text{temp}^2 = \text{head}$.

Step 9.3: Check if $\text{pos} < 1$ or $\text{pos} >= \text{count} + 1$, then print position out of range

Step 9.4: check if $\text{head} == \text{null}$ then print the list is empty.

Step 9.5: while $i < \text{pos}$ then $\text{temp}^2 = \text{temp}^2 \rightarrow \text{next}$ and increment i by 1

Step 9.6: check if $i = 1$ then check if $\text{temp}^2 \rightarrow \text{next} = \text{null}$, then print node deleted. $\text{free}(\text{temp}^2)$
Set $\text{temp}^2 = \text{head} = \text{null}$.

Step 9.7: Check if $\text{temp2} \rightarrow \text{next} = \text{null}$ then $\text{temp2} \rightarrow \text{prev} \rightarrow \text{next} = \text{null}$, then free(temp2), then print node deleted.

Step 9.8: $\text{temp2} \rightarrow \text{next} \rightarrow \text{prev} = \text{temp2} \rightarrow \text{prev}$, then check if $i1 = 1$ then $\text{temp2} \rightarrow \text{prev} \rightarrow \text{next} = \text{temp2} \rightarrow \text{next}$.

Step 9.9: Check if $i = 1$ then $\text{head} = \text{temp2} \rightarrow \text{next}$ 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 $\text{temp2} = n$

Step 10.2: Check if $\text{temp2} = \text{null}$ then print list is empty

Step 10.3: While $\text{temp2} \rightarrow \text{next1} = \text{null}$ then print $\text{temp2} \rightarrow n$ then $\text{temp2} = \text{temp2} \rightarrow \text{next}$.

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 $\text{temp2} = \text{head}$

Step 11.3: Check if $\text{temp2} = \text{null}$ then print the list is empty.

Step 11.4: Read the value to be searched.

Step 11.5: While $\text{temp}^{\rightarrow 1} = \text{null}$ then check if $\text{temp}^{\rightarrow n} == \text{data}$
then print element found at position count + 1.

Step 11.6: Else set $\text{temp}^{\rightarrow 2} = \text{temp}^{\rightarrow 2} \rightarrow \text{next}$ 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 operations.

Step 4 : If the user choose to perform union

Step 4.1 : Read the cardinality of 2 sets.

Step 4.2 : Check if the min 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 : $c[i] = A[i] \cup B[i]$

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 min 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: $C[i] = A[i] \& B[i]$

Step 5.6: print $C[i]$

Step 5.7: Increment i by 1

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

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 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 & 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 variables.

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

Step 5 : If the user choose the 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 root pointer and also to 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 $\text{root} \rightarrow \text{info} > x$ then call the insert pointer to insert to left of the root.

Step 5.5 : Check if $\text{root} \rightarrow \text{info} < x$ then call the insert pointer to insert to the right of the root.

steps.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 $\text{ptr} \rightarrow \text{info} == \text{item}$ then call delete pointer
by the passing the right pointer and the item.

Step 6.3: Else if $\text{ptr} \rightarrow \text{info} > \text{x}$ then call delete pointer
by passing the left pointer and the item

Step 6.4: Check if $\text{ptr} \rightarrow \text{info} == \text{item}$ then check if $\text{ptr} \rightarrow \text{left} == \text{ptr} \rightarrow \text{right}$ then free ptr and
return null.

Step 6.5: Else if $\text{ptr} \rightarrow \text{left} == \text{null}$ then set $P1 = \text{ptr} \rightarrow \text{right}$
and free ptr, return P1.

Step 6.6: Else if $\text{ptr} \rightarrow \text{right} == \text{null}$ then set $P1 = \text{ptr} \rightarrow \text{left}$
and free ptr, return P1.

Step 6.7: Else set $P1 = \text{ptr} \rightarrow \text{right}$ and $P2 = \text{ptr} \rightarrow \text{right}$.

Step 6.8: while $P1 \rightarrow \text{left}$ not equal to null, set $P1 \rightarrow \text{left} = \text{ptr} \rightarrow \text{left}$
and free ptr, return P2

Step 6.9: Return P1.

Step 7 : If the user choose to perform search operation
 then 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 $\text{ptr} \neq \text{NULL}$ check if $\text{item} > \text{ptr} \rightarrow \text{info}$ then
 $\text{ptr} = \text{ptr} \rightarrow \text{right}$

Step 7.4 : Else if $\text{item} < \text{ptr} \rightarrow \text{info}$ then $\text{ptr} = \text{ptr} \rightarrow \text{left}$

Step 7.5 : Else break

Step 7.6 : Check if $\text{ptr} \neq \text{NULL}$ then point 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 functions and pass the root
 pointers.

Step 8.1 : If root not equals to NULL recursively call all the
 functions by passing $\text{root} \rightarrow \text{left}$.

Step 8.2 : Print $\text{root} \rightarrow \text{info}$

Step 8.3 : Call the traversal functions recursively by
 passing $\text{root} \rightarrow \text{right}$.