

## 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  $i < n$  &  $k < n$
- Step 8: Check if  $a[i] \leq b[j]$  then  $c[k++] = b[j++]$
- Step 9: Else  $c[k++] = a[i++]$
- Step 10: Repeat step 11 while  $i < n$
- 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: End.



## Singly linked Stack

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 newnode

Step 5.2: Set  $\text{newNode} \rightarrow \text{data} = \text{value}$

Step 5.3: Check if  $\text{top} == \text{null}$  then set  $\text{newNode} \rightarrow \text{next} = \text{null}$

Step 5.4: Set  $\text{newNode} \rightarrow \text{next} = \text{top}$

Step 5.5: Set  $\text{top} = \text{newNode}$  & 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  $\text{top} == \text{null}$  then print stack is empty

Step 6.2: Else declare a pointer variable temp & initialize it to top

Step 6.3: print the element that being deleted



- Step 6.4 : set  $\text{temp} = \text{temp} \rightarrow \text{next}$
- Step 6.5 : free the temp.
- Step 7 : if the user choose to display then call the function to display the element in the stack
- Step 7.1 : check if  $\text{top} = \text{null}$  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  $\text{temp} \rightarrow \text{next} \neq \text{null}$
- Step 7.4 : print  $\text{temp} \rightarrow \text{data}$ .
- Step 7.5 : Set  $\text{temp} = \text{temp} \rightarrow \text{next}$
- Step 8 : if the user choose to search an element from the stack then call the function to search an element.
- Step 8.1 : Declare a pointer variable ptr and other necessary variable
- Step 8.2 : initialize  $\text{ptr} = \text{top}$
- 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

(4)

Step 8.7 : Else Set  $\text{flag} = 0$

Step 8.8 : increment  $i$  by 1 and set  $\text{ptr} = \text{ptr} \rightarrow \text{next}$

Step 8.9 : check if  $\text{flag} \neq 0$  then print the element not found.

Step 9 : End



## Circular Queue operation

(5)

Step 1 : Start

Step 2 : Declare the queue and other variable

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 the function dequeue.

Step 6.1 : check if  $front == -1$  and  $rear == -1$  then print queue is underflow.

⑥

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  $\text{front} = \text{front} + 1 \% \text{max}$ .

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 Queue is empty.

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

Step 7.3 : print  $\text{queue}[i]$  and set  $i = i + 1 \% \text{max}$

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

(7)

- Step 1 : Start
- Step 2 : Declare a Structure and related variable
- 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 function to create a node, declare the required variables
- Step 4.1 : Set memory allocated to the node = temp then  
Set temp  $\rightarrow$  prev = null and temp  $\rightarrow$  next = null
- Step 4.2 : Read the value to be inserted to the node
- Step 4.3 : Set temp  $\rightarrow$  data = 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 step 4.3
- Step 6.2 : Set head = temp and temp1 = head



Step 6.3 : Else call the function to create a node. (8)  
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 new node then Set  $\text{temp} = \text{head}$  and Set  $\text{head} = \text{temp}$

Step 7.2 : Else call the function to create a new node then Set  $\text{temp}_1 \rightarrow \text{next} = \text{temp}$ ,  
 $\text{temp} \rightarrow \text{prev} = \text{temp}_1$  and  $\text{temp}_1 = \text{temp}$

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 and to be inserted, Set  $\text{temp}_2 = \text{head}$ .

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 at the 1st position"



Step 8.5 : check if head == null and pos = 1 then (9)  
call the function to create newNode. then  
Set temp = head and head = temp.

Step 8.6 : while  $i < pos$  then set  $temp2 = temp2 \rightarrow next$   
the increment  $i$  by 1.

Step 8.7 : call the function to create a newNode and  
then set  $temp \rightarrow prev = temp2$ ,  $temp \rightarrow next = temp2$   
 $next \rightarrow prev = temp$ ,  $temp2 \rightarrow next = temp$

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 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  $temp2 = temp2 \rightarrow next$   
And increment  $i$  by 1



Step 9.6 : check if  $c=1$  then check if  $temp2 \rightarrow next == null$  (b)  
then print node deleted free (temp2) set  
 $temp2 = head = null$

Step 9.7 : check if  $temp2 \rightarrow next == null$  then  $temp2 \rightarrow prev \rightarrow$   
 $next = null$  then free (temp2) then print node  
deleted.

Step 9.8 :  $temp2 \rightarrow next \rightarrow prev = temp2 \rightarrow prev$  then check if  
 $1 \neq 1$  then  $temp2 \rightarrow prev \rightarrow next = temp2 \rightarrow next$ .

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  $temp2 \rightarrow next \neq null$  then print  
 $temp2 \rightarrow n$  then  $temp2 = temp2 \rightarrow next$

Step 11 : if the user choose to perform the search  
operation then call the function to perform search  
operations

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  $\text{temp2} \neq \text{null}$  the & check if  $\text{temp2} \rightarrow \text{data} == \text{data}$   
then print element found at position  $\text{Count} + 1$

Step 11.6: Else set  $\text{temp2} = \text{temp2} \rightarrow \text{next}$  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 operation.

Step 4: If the user choose to perform union.

Step 4.1: Read the cardinality of 2 sets.

Step 4.2: check if  $m_1 - n$  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  $i \leq m$

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  $m_1 - n$  then print Cannot perform Intersection.

Step 5.3: Else read the elements in both the sets.

Step 5.4 : Repeat the step 5.5 to 5.7 until  $i \leq n$

Step 5.5 :  $c[i] = A[i] \wedge B[i]$

Step 5.6 : print  $c[i]$

Step 5.7 : Increment  $i$  by 1

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

Step 6.1 : Read the cardinality of 2 sets.

Step 6.2 : check if  $m=0$  then print cannot perform set difference operation

Step 6.3 : Else read the elements in both sets

Step 6.4 : Repeat the step 6.5 to 6.8 until  $i \leq n$

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  $i \leq n$

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 operation 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 ~~root~~ tree from the user.

Step 5.1: The value to be insert pointer and also the root pointer.

Step 5.2: check if ! root then allocate memory for the root

Step 5.2: 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  $\rightarrow$  info  $> x$  then call the insert pointer to insert to left of the root

Step 5.5: check if root  $\rightarrow$  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  $\rightarrow$  info  $< x$  then call delete pointer by passing the right pointer and the item.

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

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

Step 6.5: Else if ptr  $\rightarrow$  left  $=$  null then set pl = ptr  $\rightarrow$  right and free ptr, return pl



Step 6.6 : Else if  $ptr \rightarrow right == null$  then Set  
 $p1 = ptr \rightarrow left$  and free  $pb$ , return  $p1$

Step 6.7 : Else set  $p1 = pb \rightarrow right$  and  $p2 = pb \rightarrow right$

Step 6.8 : while  $p1 \rightarrow left$  not equal to null, Set  
 $p1 \rightarrow left = ptr \rightarrow left$  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 element to be searched.

Step 7.3 : while  $pb$  check if  $item > pb \rightarrow info$  then  
 $ptr = pb \rightarrow right$

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

Step 7.5 : Else break

Step 7.6 : check if  $pb$  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  $\rightarrow$  left

Step 8.2: print root  $\rightarrow$  info

Step 8.3: call the traversal function recursively by passing root  $\rightarrow$  right.

Step 9: End



## Disjoint sets.

Step 1: Start

Step 2: Specify the height of the tree by representing the Set  $\text{rank}[i]$ .

Step 3: Create  $n$  single item sets. call  $\text{int } n$ .

Step 4: Make set by calling  $\text{dis.parent}[i] = i$  and  $\text{dis.rank}[i] = 0$

Step 5: display set by calling  $\text{dis.parent}[i]$  and  $\text{dis.rank}[i]$ .

Step 5.1: Find set of given item  $x$ .

Step 6: Find the representative of the set that  $x$  is an ~~element~~ element by  $\text{dis.parent}[x] \neq x$

Step 6.1: recursively call find on its parent and move it's node directly under the representative of this set  $\text{dis.parent}[x] = \text{find}(\text{dis.parent}[x])$ .

Step 7: Do union of two set represented by  $x$  and  $y$

Step 8: Find current set of  $x$  and  $y$ .  
 $\text{int } x\text{set} = \text{find}(x); \text{ and } \text{int } y\text{set} = \text{find}(y);$

Step 9: check if they are already in same set  
by if (xset == yset)

Step 10: put smaller ranked item under bigger  
ranked item if ranks are different.

Step 11: if rank are same, then increment rank

Step 12: End.