MERGING

Step1: Start

Step2: Declare the variables

Step3: Read the size of first array

Step4: Read elements of first array in sorted order

Step5: Read the size of second Array

Step6: Read the elements of second Array in sorted order

Step7: Repeat steps 8 and 9 while i<m & k<n

Step8: check if a[i]>=b[j] then c[k++] = b[j++]

Step9: else c[k++] = a[i++]

Step10: Repeat step 11 while i<m

Step11: c[k++] = a[j++]

Step12: Repeat step 13 while j<n

Step13: c[k++] = b[j++]

Step14: print the first Array

Step15: print the second Array

Step16: print the Merged Array

Step17: End.

Singly linked Stack

Step1: Start

Step2: Declare the node and the required variables

Step3: Declare the functions for push, pop, display and search an element.

Step4: Read the choice from the user

Step5: 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

Step5.1: Declare the newnode & allocate memory for newnode

Step5.2: Set newNode → data = value

Step5.3: check if top == null then set newNode → next = NULL

Step5.4: Set newnode → next = top

Step5.5: Set top = newNode & then print insertion is successful.

Step6: If user choose to pop an element from the stack then call the function to pop the element

Step6.1: check if top == null then print stack is Empty

Step6.2: Else declare a pointer variable temp & initialize it to top

Step6.3: print the element that being deleted

Step6.4 : set temp = temp → next

Step6.5 : free the temp.

Step 7 : if the user choose the display then call the function to display the element in the Stack

Step7.1 : check if top = Null then print stack is empty.

Step7.2 : else declare a pointer variable temp & initialize it to top

Step7.3 : Repeat steps below while temp → next != Null

Step7.4 : print temp → data.

Step7.5 : Set temp = temp → next

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

Step8.1 : Declare a pointer variable ptr and other neccessary 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 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.

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 & 7 : Read the element to be searched in the queue

Step 8.2 : check if item == queue [i] then print item found and is 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 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 → 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

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. 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 the 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 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 neccassary variable.

Step 8.2 : Read the position where the node and 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 print "empty list cannot insert other than 1st posi

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 → next then increment i by 1.

Step 8.7 : call the function to create a newNode 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 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 c=1 then check if temp2 → next == null then print node deleted free (temp2) set temp2 = head = null ⑥

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 1 != 1 then temp2 → prev → next = temp2 → 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 → 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

Step 11.1 : Declare the neccessary 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 temp2->data == 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.

Set operations

Step 1 : start

Step 2 : Declare the neccessary 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 m1-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 m1-n then print cannot perform Intersection.

Step 5.3 : Else read the elements is 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 seperation.

Step 6.1 : Read the cardinality of 2 sets

Step 6.2 : check if m1=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<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.

Binary Search tree

Step1: start

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

Step3: 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 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 the 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
the call the pointer to perform search oper

Step 7.1 : Declare the neccessary 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 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 all 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 sets.

Step 1: Start

Step 2: Specify the height of the tree by representing the set rank[i].

Step 3: Create n single item sets. call int n.

Step 4: Make set by calling dis.parent[i]=i and dis.rank[i]=0

Step 5: display set by calling dis.parent[i] and dis.rank[i].

Step 5.1: find set of given item x.

Step 6: find the representative of the set that x is an ~~eleme~~ element by dis.parent[x] != x

Step 6.1: recursively call find on its parent and move i's node directly under the representative of this set dis.parent[x] = find(dis.parent[x]);

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

Step 8: find current set of x and y.
int x set = find(x); and int y set = 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.

Graph Traversal Technique DFS

Step1 : Define a stack of size total number of vertices

Step2 : Select any vertex as starting point for traversal visit that vertex and push it onto stack

Step3 : visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it onto stack.

Step4 : Repeat step 3 until there are no new vertex to be visited from vertex on top of the stack

Step5 : when there is no new vertex to be visited use back tracking and pop one vertex from stack

Step6 : Repeat step 3, 4 and 5 until stack become empty

Step7 : when stack become empty, produce final spanning tree by removing unused edges from the graph.

Graph Traversal Techniques BFS

Step 1: Define a queue of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal visit that vertex and insert it into queue.

Step 3: Visit all adjacent vertices of the vertex which is front of queue which is not visited and insert them into the queue.

Step 4: When there is no new vertex to be visit from the vertex at front of the queue then delete that vertex from the queue.

Step 5: Repeat Step 3 & Step 4 until queue becomes empty.

Step 6: Produce final Spanning tree by removing unused edges from the graph.

Topological Sorting

Step 1: Comput in-degree (number of incoming edges) for each of the vertex. and Initialize count of visited nodes as zero

Step 2: Pick all the vertices with in-degree as 0 and add them into a queue. (Enqueue)

Step 3: Remove a vertex from que. (Dequeue)

Step 3.1: Increment count of visited nodes by 1

Step 3.2: Decrease in-degree by 1 for all its neighbouring nodes

Step 3.2: If in-degree of a neighbouring node is reduced to zero, then add it to the queue

Step 4: Repeat Step 3 until the queue is empty.

Step 5: If count of visited node is not equal to the number of nodes in the graph then topological sort is not possible for given graph.