CSE104 DATA STRUCTURES LABORATORY

<u>List of Exercises</u>

1. Implementation of sorting techniques (Bubble Sort and Insertion Sort)
2. Implementation of searching techniques (Sequential search, Binary search and Hash search)
3. Demonstration of Stack and Queue operations using arrays
4. Implementation of Singly Linked List
5. Demonstration of Stack and Queue operations using linked linear list
6. Demonstration of Infix to Postfix conversion using stack
7. Implementation of Doubly Linked List
8. Demonstration of Circular DLL operations
9. Demonstration of BST operations
10. Graph traversal using BFS
11. Graph traversal using DFS
12. Finding Minimum Weight Spanning Tree employing Prim's algorithm

Ex. No: 01 Implementation of sorting techniques (Bubble Sort and Insertion Sort)

Pre Lab: Implementation knowledge on Arrays and Functions.

a) Bubble Sort

Algorithm bubbleSort (list, last)

Purpose sort an array using bubble sort, Adjacent elements are compared and exchanged until list is completely ordered.

Pre List must contain at least one item last contains index to last element in the list

Post List has been rearranged in sequence low to high

1. set current to 0
2. set sorted to false
3. loop (current <= last and sorted false)
   1. Each iteration is one sort pass
   2. set walker to last
   3. set sorted to true
   4. loop (walker >current)
      1. if (walker data <walker-1 data)
         1. set sorted to false
         2. exchange (list,walker,walker-1)
      2. end if
      3. decrement walker
   5. end loop
   6. increment current
4. end loop

End bubbleSort

b) Insertion Sort

Algorithm InsertionSort (list, last)

Sort list array using insertion sort. The array is divided into sorted and unsorted lists. With each pass, the first element in the unsorted list is inserted into the sorted list.

Pre list must contain atleast one element last is an index to last element in the list

Post list has been rearranged

1. set current to 1
2. loop (until last element sorted)

        1. move current element to hold
        2. set walker to current-1
        3. loop (walker>=0 and hold key <walker key)
            1. move walker element right one element
            2. decrement walker
        4. end loop
        5. move hold to walker+1 element
        6. increment current
3. end loop

End Insertion sort

Ex. No 02 Implementation of searching techniques (Sequential search, Binary search and Hash search)

a. Sequential Search

Algorithm seqSearch (list, last, target, locn)

Purpose: Locate the target in an unordered list of elements.
Pre: list must contain at least one element
last is index to last element in the list target contains the data to be located locn is address of index in calling algorithm
Post: if found: index stored in locn & found true if not found: last stored in locn & found false Return found true or false

1. set looker to 0
2. loop (looker < last AND target not equal list[looker])
    1. increment looker
3. end loop
4. set locn to looker
5. if (target equal list[looker])
    1. set found to true
6. else
    1. set found to false
7. end if
8. return found

end **seqSearch**

Algorithm RecursiveLinearSearch(List,index,last, locn,target)

Purpose: Locate the target in an unordered list of elements.
Pre: list must contain at least one element
index is the first element in the list
last is index to last element in the list target contains the data to be located
locn is address of index in calling algorithm
Post: if found: index stored in locn & found true if not found: last stored in locn & found false Return found true or false

1.if(index>last)
   1.locn=-1
   2.return flase
2.end if
3.if(target==List[index])

    1.locn= index
    2.return true
4.end if
5.return RecursiveLinearSearch(List,index+1,last, locn,target)
End **RecursiveLinearSearch**

b. <u>Binary Search</u>

Algorithm binarySearch (list, last, target, locn)

Purpose:	Search an ordered list using Binary Search
Pre:	list is ordered; it must have at least 1 value
	last is index to the largest element in the list
	target is the value of element being sought
	locn is address of index in calling algorithm
Post:	if target found: locn assigned index to target element
	if target not found: locn assigned element below or above target
Return :	found as true or false

1. set begin to 0
2. set end to last
3. loop (begin <= end)
    1. set mid to (begin + end) / 2
    2. if (target > list[mid])
        1. set begin to (mid + 1)
    3. else if (target < list[mid])
        1. set end to mid - 1
    4. else
        1. set begin to (end + 1)
    5. end if
4. end loop
5. set locn to mid
6. if (target equal list [mid])
    1. set found to true
7. else
    1. set found to false
8. end if
9. return found

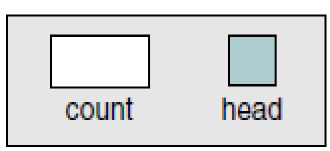
end **binarySearch**

Algorithm BinarySearch_Recursion(List, first, last,target,Loc)

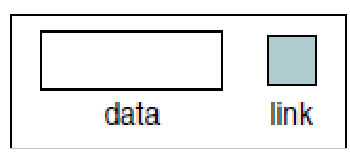
Purpose: Search an ordered list using Binary Search
Pre: list is ordered; it must have at least 1 value
first is index to the smallest element in the list
last is index to the largest element in the list
target is the value of element being sought
locn is address of index in calling algorithm
Post: if target found: locn assigned index to target element
if target not found: locn assigned element below or above target
Return : found as true or false

1. If last > first then
    1. Loc=-1
    2. return false
2. Else
    1. Mid = (first + last) / 2
    2. If target< List[Mid] then
        1. return BinarySearch_Recursion (List, first, Mid-1, target,Loc)
    3. Else If target >List[Mid] then
        1. return BinarySearch_Recursion (List, Mid+1, last, target,Loc)
    4. Else
        1. Loc=Mid
        2. return true
    5. End if
3. End if
4. End BinarySearch_Recursion

c.Hash Search



hash



node

Algorithm node createNode(key)

Purpose: Insert a new key into hashTable
Pre: key is the element to be inserted
Post: newNode is created

1. allocate newNode
2. set newNode data as key

3. set newNode link as null
4. return newNode

end **createNode**

Algorithm insertToHash(hashTable, size, key)

Purpose: Insert a new key into hashTable
Pre: hashTable is a array of SLL head nodes
size is the maximum size of hashTable
key is the element to be inserted
Post: key is inserted into hashTable

1. set hashIndex as key modulo size
2. create newnode with createNode(key)
3. set newnode link as hashTable[hashIndex] head
4. set hashTable[hashIndex] head to newnode
5. increment hashTable[hashIndex] count by 1
6. return

end **insetToHash**

Algorithm deleteFromHash(hashTable, size, dltKey)

Purpose: Delete key from hashTable
Pre: hashTable is an array of SLL head nodes
size is the maximum size of hashTable
key is the element to be deleted
Post: key is deleted from hashTable

1. set hashIndex to dltKey modulo size
2. set delNode to hashTable[hashIndex] head
3. set temp as null
4. loop (delNode is not equal to null)
    1. if (delNode data equals to dltKey)
        1. if (temp is null)
            1. set hashTable[hashIndex] head as delNode link
        2. else
            2. set temp link as delNode link
        3. end if
        4. decrement hashTable[hashIndex]count by 1
        5. return
    2. end if
    3. set temp as delNode;
    4. set delNode as delNode link
5. end loop
6. write ”Given key is not present in hash Table”
7. return

end **deleteFromHash**

Algorithm searchInHash(hashTable, size, key)

Purpose: Searching an element in the hashTable
Pre: hashTable is an array of SLL head nodes
size is the maximum size of hashTable
key is the element to be searched
Post: key is searched in the hashTable

1. set hashIndex to key modulo size
2. set searchNode to hashTable[hashIndex] head
3. loop (searchNode not equals to null)
    1. if (searchNode key equals to key)
        1. print searchNode key
        2. return
    2. end if
    3. set searchNode to searchNode link
4. end loop
    1. write "Search element unavailable in hash table"

end **searchInHash**

Algorithm display(hashTable, size)
Purpose: Display the elements in the hashTable
Pre: hashTable is an array of SLL head nodes
size is the maximum size of hashTable
Post: All elements in hashTable are printedtraversed

1. set i as 0
2. loop (i less than size)
    1. set temp as hashTable[i] head
    2. loop (temp not equals to null)
        1. print temp data
        2. set temp as temp link
    3. increment i by 1
3. end loop

end **display**

Ex. No 03 Demonstration of Stack and Queue operations using array

Pre Lab: Implementation knowledge on Arrays and Functions.

Stack operations:

a) Push:

Algorithm PushStack(stack, data)
1. if (stack is full)
    1. set success to false
2. else
    1. increment top
    2. store data at stack[top]
    3. set success to true
3. end if
4. return success

End PushStack

b) Pop

Algorithm PopStack (stack, dataOut)

1 if (stack empty)
  1. set success to false

2 Else
  1. set dataOut to data in top node
  2. decrement top
  3. set success to true

3 end if

4 return success

End PopStack

c) StackTop

Algorithm StackTop (stack, dataOut)

1 if (stack empty)
  1. set success to false

2 Else
  1. set dataOut to data in top node
  2. set success to true

3 end if

4 return success
End StackTop

d) EmptyStack

Algorithm EmptyStack(queue)
1 if (Top is -1)
    1 set empty to true
2 else
    1 set empty to false
3 end if
4 return empty
end EmptyStack

e) FullStack

Algorithm FullStack(Stack)
1 if (Top equals to MAXSIZE)
        1 set full to true
2 else
        1 set full to false
3 endif
4 return full
End FullStack

Queue operations

a) Enqueue

Algorithm enqueue(queue, dataIn)
1 if (queue is full)
    1 set success to false
2 else
    1 increment rear
3 set queue[rear] to data in dataIn
2 set success to true
3 return success
End enqueue

b) *Dequeue*

Algorithm dequeue(queue, dataOut)

1 if (queue is empty)
    1 set success to false
2 else
    1 set dataOut to data in queue[front]
    2 increment front
    3 set success to true
3 end if
4 return success

End dequeue

c) *QueueFront*

Algorithm QueFront (queue, dataOut)

1 if (queue empty)
    1. set success to false
2 else
    1. set dataOut to data in front node
    2. set success to true
3 end if
4 return success

End QueFront

d) *QueueRear*

Algorithm QueueRear (queue, dataOut)

1 if (queue empty)
    1. set success to false
2 else
    1. set dataOut to data in Rear node
    2. set success to true
3 end if
4 return success

End QueueRear

e) *EmptyQueue*

Algorithm EmptyQueue(queue)

1 if (front and rear are -1)
    1 set empty to true
2 else

1 set empty to false
3 end if
4 return empty
End EmptyQueue

f) FullQueue

Algorithm FullQueue(queue)
1 if (rear equals to MAXSIZE)
    1. set full to true
2 else
    1. set full to false
3 endif
4 return full
End FullQueue

g) QueueCount

Algorithm QueueCount(Queue)
1 if (Queue Empty)
    1 Display ‘Queue is Empty’
    2 Return
2 Else
    1 Return rear – front + 1
3 Endif
End Display

Ex. No: 04 Implementation of Singly Linked List

a) Create list

Algorithm createList (list)
Initializes metadata for list.
Pre: list is metadata structure passed by reference
Post: metadata initialized

1. allocate (list)
2. set list head to null
3. set list count to 0

end createList

b) Insert node

Algorithm insertNode (list, pPre, dataIn)
Inserts data into a new node in the list.
Pre list is metadata structure to a valid list
pPre is pointer to data's logical predecessor
dataIn contains data to be inserted
Post data have been inserted in sequence
Return true if successful, false if memory overflow

1. allocate (pNew)
2. set pNew data to dataIn
3. if (pPre null)
   1 set pNew link to list head
   2 set list head to pNew
4. else
   1 set pNew link to pPre link
   2 set pPre link to pNew
5. end if
6. return true

end insertNode

c) Delete node

Algorithm deleteNode (list, pPre, pLoc, dataOut)
Deletes data from list & returns it to calling module.
Pre list is metadata structure to a valid list
Pre is a pointer to predecessor node
pLoc is a pointer to node to be deleted
dataOut is variable to receive deleted data
Post data have been deleted and returned to caller

1 move pLoc data to dataOut
2 if (pPre null)
    1. set list head to pLoc link
3 else
    1. set pPre link to pLoc link
4 end if
5 recycle (pLoc)

end deleteNode

d)Search list

Algorithm searchList (list, pPre, pLoc, target)
Searches list and passes back address of node containing target and its logical predecessor.

Pre list is metadata structure to a valid list
pPre is pointer variable for predecessor
pLoc is pointer variable for current node
target is the key being sought

Post pLoc points to first node with equal/greater key-or- null if target > key of last node
pPre points to largest node smaller than key-or- null if target < key of first node

Return true if found, false if not found

1. set pPre to null
2. set pLoc to list head
3. loop (pLoc not null AND target > pLoc key)
    1. set pPre to pLoc
    2. set pLoc to pLoc link
4. end loop
5. if (pLoc null)
   //Set return value
    1. set found to false
6. else
    1. if (target equal pLoc key)
        1. set found to true
    2. else
        1. set found to false
    3. end if
7. end if
8. return found

end searchList

e) Retrieve Node

Algorithm retrieveNode (list, key, dataOut)
Retrieves data from a list.

Pre list is metadata structure to a valid list
key is target of data to be retrieved
dataOut is variable to receive retrieved data
Post data placed in dataOut -or- error returned if not found
Return true if successful, false if data not found

1. set found to searchList (list, pPre, pLoc, key)
2. if (found)
   1. move pLoc data to dataOut
3. end if
4. return found

end retrieveNode

f) Empty List

Algorithm emtpyList (list)
Returns Boolean indicating whether the list is empty.
Pre list is metadata structure to a valid list
Return true if list empty, false if list contains data */

1. if (list count equal 0)
   1. return true
2. else
   1. return false

end emptyList

g) List Count

Algorithm listCount (list)
Returns integer representing number of nodes in list.
Pre list is metadata structure to a valid list
Return count for number of nodes in list */
1 return (list count)
end listCount

h) Traversal and Display

Algorithm getNext (list, fromWhere, dataOut)
Traverses a list. Each call returns the location of an element in the list.
Pre list is metadata structure to a valid list from Where is 0 to start at the first element
dataOut is reference to data variable
Post dataOut contains data and true returned -or- if end of list, returns false
Return true if next element located false if end of list

1. if (empty list)
   1. return false

2. if (fromWhere is beginning)
    1. set list pos to list head
    2. move current list data to dataOut
    3. return true
3. else
4. if (end of list)
    1. return false
5. else
    1. set list pos to next node
    2. move current list data to dataOut
    3. return true
6. end if

2. end if

end getNext

i) DestroyList

Algorithm destroyList (pList)
Deletes all data in list.
Pre list is metadata structure to a valid list
Post All data deleted

1. loop (not at end of list)
    1. set list head to successor node
    2. release memory to heap
2. end loop
3. set list pos to null
4. set list count to 0

end destroyList

Ex. No: 05 Demonstration of Stack and Queue operations using linked linear list

a. Stack Operations

i. Create Stack

Algorithm createStack

Purpose: Creates and initializes metadata structure.
Pre: Nothing
Post: Structure created and initialized
Return: stack head

1 allocate memory for stack head
2 set count to 0
3 set top to null
4 return stack head
end createStack

ii. Push Stack

Algorithm pushStack (stack, data)

Purpose: Insert (push) one item into the stack.
Pre: stack passed by reference
data contain data to be pushed into stack
Post: data have been pushed in stack

1. allocate new node
2. store data in new node
3. make current top node the second node
4. make new node the top
5. increment stack count

end pushStack

iii. Pop Stack

Algorithm popStack (stack, dataOut)

Purpose: This algorithm pops the item on the top of the stack and returns it to the user.
Pre: stack passed by reference
dataOut is reference variable to receive data
Post: top most data has been returned to calling algorithm and that node is deleted

Return: true if successful; false if underflow

1. if (stack empty)
   1. set success to false
2. else
   1. set dataOut to data in top node
   2. make second node the top node
   3. decrement stack count
   4. set success to true
3. end if
4. return success

end popStack

iv. **Stack top**

Algorithm stackTop (stack, dataOut)

Purpose: This algorithm retrieves the data from the top of the stack without changing the stack.
Pre: stack is metadata structure to a valid stack
dataOut is reference variable to receive data
Post: top most data has been returned to calling algorithm
Return: true if data returned, false if underflow

1. if (stack empty)
   1 set success to false
2. else
   1 set dataOut to data in top node
   2 set success to true
   3 end if
3. return success

end stackTop

v. **Empty Stack**

Algorithm emptyStack (stack)

Purpose: Determines if stack is empty and returns a Boolean.
Pre: stack is metadata structure to a valid stack
Post: returns stack status
Return: true if stack empty, false if stack contains data

1. if (stack count is 0)
   1. return true
2. else
   1. return false
3. end if

end emptyStack

vi. **Destroy Stack**

Algorithm destroyStack (stack)

Purpose: This algorithm releases all nodes back to the dynamic memory.
Pre: stack passed by reference
Post: stack empty and all nodes deleted

1. if (stack not empty)
    1. loop (stack not empty)
        1. delete top node
    2. end loop
2. end if
3. delete stack head

end destroyStack

b. Queue Operations

i) Create Queue

Algorithm createQueue

Purpose: Creates and initializes queue structure.
Pre: queue is a metadata structure
Post: metadata elements have been initialized
Return: queue head

1 allocate queue head
2 set queue front to null
3 set queue rear to null
4 set queue count to 0
5 return queue head
end createQueue

ii) Enqueue

Algorithm enqueue (queue, dataIn)

Purpose: This algorithm inserts data into a queue.
Pre: queue is a metadata structure
Post: dataIn has been inserted
Return: true if successful, false if overflow

1. if (queue full)
    1. return false
2. end if
3. allocate (new node)
4. move dataIn to new node data
5. set new node next to null pointer
6. if (empty queue)
    1. set queue front to address of new data
7. else
    1. set next pointer of rear node to address of new node
8. end if
9. set queue rear to address of new node
10. increment queue count
11. return true

end enqueue

iii) Dequeue

Algorithm dequeue (queue, item)

Purpose:	This algorithm deletes a node from a queue.
Pre:	queue is a metadata structure
	item is a reference to calling algorithm variable
Post:	data at queue front returned to user through item and front element deleted
Return:	true if successful, false if underflow

1. if (queue empty)
    1. return false
2. end if
3. move front data to item
4. if (only 1 node in queue)
    1. set queue rear to null
5. end if
6. set queue front to queue front next
7. decrement queue count
8. return true

end dequeuer

iv) Retrieve data from front

Algorithm queueFront (queue, dataOut)

Purpose:	Retrieves data at the front of the queue without changing queue contents.
Pre:	queue is a metadata structure
	dataOut is a reference to calling algorithm variable
Post:	data passed back to caller
Return:	true if successful, false if underflow

1. if (queue empty)
    1. return false
2. end if
3. move data at front of queue to dataOut
4. return true

end queueFront

v) Retrieve data from rear

Algorithm queueRear (queue, dataOut)

Purpose: Retrieves data at the rear of the queue without changing queue contents.
Pre: queue is a metadata structure
dataOut is a reference to calling algorithm variable
Post: data passed back to caller
Return: true if successful, false if underflow

1. if (queue empty)
    1. return false
2. end if
3. move data at rear of queue to dataOut
4. return true

end queueRear

vi) Empty Queue

Algorithm emptyQueue (queue)

Purpose: This algorithm checks to see if a queue is empty.
Pre: queue is a metadata structure
Return: true if empty, false if queue has data

1. if (queue count equal 0)
    1. return true
2. else
    1. return false

end emptyQueue

vii) Count Queue

Algorithm queueCount (queue)

Purpose: This algorithm returns the number of elements in the queue.
Pre: queue is a metadata structure
Return: queue count

1. return queue count

end queueCount

viii) Destroy Queue

Algorithm destroyQueue (queue)

Purpose: This algorithm deletes all data from a queue.
Pre: queue is a metadata structure
Post: all data have been deleted

1. if (queue not empty)
   1. loop (queue not empty)
      1. delete front node
   2. end loop
2. end if
3. delete head structure

end destroyQueue

Ex. No: 06 Demonstration of Infix to Postfix conversion using stack

<u>Priority for Operators:</u>

Priority 3: ^
Priority 2: * and /
Priority 1: + and –
Priority 0: (

Algorithm inToPostFix(formula)
Convert infix formula to postfix.
Pre formula is infix notation that has been edited to ensure that there are no syntactical errors
Post postfix formula has been formatted as a string
Return postfix formula

1. createstack(stack)
2. loop(for each character in formula)
   1 if (character is open parenthesis)
      1. pushStack(Stack, character)
   2 elseif (character is close parenthesis)
      1. popStack(Stack, character)
      2. loop(character not open paranthesis)
         1. concatenate character to postFixExpr
         2. popStack(Stack, character)
      3. end loop
   3. elseif (character is operator)
      1 Test priority of token to token at top of stack

2 stackTop(Stack, topToken)
3 loop(not emptyStack(Stack) AND priority(character) <= priority(topToken))
    1. popStack (Stack, tokenOut)
    2. concatenate tokenout to postFixExpr
    3. stackTop (Stack, topToken)
4 end loop
5 pushstack (Stack, Token)
4. else
    1 character is operand
    2 Concatenate Token to postfix
5. end if
3. end loop
4. loop (not emptyStack(Stack))
    1. popStack (Stack, character)
    2. concatenate Token to postFixExpr
5. end loop
6. return postfix

end inToPostFix

Ex. No: 07 Implementation of Doubly Linked List

a. <u>Creating an Empty DLL</u>

Algorithm createList(list**)**

Purpose: Initializes metadata for list
Pre: list is metadata structure passed by reference
Post: metadata initialized

1. allocate (list)
2. set list head to null
3. set list rear to null
4. set list count to 0

end createList

b. <u>Inserting an element into a DLL</u>

Algorithm insertDLL (list, dataIn)

Purpose: This algorithm inserts data into a doubly linked list.
Pre: list is metadata structure to a valid list
dataIn contains the data to be inserted
Post: The data have been inserted in sequence
Return: 0: failed--dynamic memory overflow
1: successful

2: failed--duplicate key presented

1. if (full list)
    1. return 0
2. end if
3. set found to searchList(list, predecessor, successor, dataIn key)
4. if (not found)
    1. allocate new node
    2. move dataIn to new node
    3. increment list count by 1
    4. if (predecessor is null)
        1. set new node back pointer to null
        2. set new node fore pointer to list head
        3. set list head to new node
    5. else
        1. set new node fore pointer to successor
        2. set new node back pointer to predecessor
        3. set predecessor fore pointer to new node
    6. end if
    7. if (successor is null)
        1. set list rear to new node
    8. else
        1. set successor back to new node
    9. end if
    10. return 1
5. end if
6. return 2

end insertDLL

c. <u>**Deleting an element from DLL**</u>

Algorithm deleteDLL (list, deleteNode)

Purpose: This algorithm deletes a node from a doubly linked list.
Pre: list is metadata structure to a valid list
deleteNode is a pointer to the node to be deleted
Post: node deleted

1. if (deleteNode null)
    1. abort ("Impossible condition in delete double")
2. end if
3. if (deleteNode back not null)
    1. set predecessor to deleteNode back
    2. set predecessor fore to deleteNode fore
4. else

        1. set list head to deleteNode fore
    5. end if
    6. if (deleteNode fore not null)
        1. set successor to deleteNode fore
        2. set successor back to deleteNode back
    7. else
        1. set list rear to deleteNode back
    8. end if
    9. recycle (deleteNode)
    10. decrement list count by 1

end deleteDLL

d. <u>Searching for the address of an element and its predecessor node in the DLL</u>

Algorithm searchList (list, pPre, pLoc, target)

Purpose: Searches list and passes back address of node containing target and its logical predecessor.

Pre: list is metadata structure to a valid list
pPre is pointer variable for predecessor
pLoc is pointer variable for current node
target is the key being sought

Post: pLoc points to first node with equal/greater key -or- null if target > key of last node
pPre points to largest node smaller than key -or- null if target < key of first node

Return: true if found, false if not found

1. set pPre to null
2. set pLoc to list head
3. loop (pLoc not null AND target >pLoc key)
    1. set pPre to pLoc
    2. set pLoc to pLoc link
4. end loop
5. if (pLoc null)
    1. set found to false
6. else
    1. if (target equal pLoc key)
        1. set found to true
    2. else

1. set found to false
3. end if
7. end if
8. return found

end searchList

e. Check for Empty DLL

Algorithm emtpyList (list)

Returns: Boolean indicating whether the list is empty.
Pre: list is metadata structure to a valid list
Return: true if list empty, false if list contains data

1 if (list count equal 0)
    1 return true
2 else
    1 return false
end emptyList

f. Counting number of nodes in DLL

Algorithm dllCount (list)

Purpose: count for number of nodes in list
Pre: list is metadata structure to a valid list
Returns: integer representing number of nodes in list.

1. return (list count)

end dllCount

g. Displaying elements of DLL from Head to Rear

Algorithm displayHeadToRear(list)

Purpose: displaying the elements in list from head to rear
Pre: list is metadata structure to a valid list
Returns: print the data of nodes in list.

1. set pWalker to list head
2. loop (pWalker not null)
    1. write (pWalker data)
    2. set pWalker to pWalker fore pointer
3. end loop

end displayHeadToRear

h. Displaying elements of DLL from Rear to Head

Algorithm displayRearToHead(list)

Purpose: displaying the elements in list from rear to head
Pre: list is metadata structure to a valid list
Returns: print the data of nodes in list.

1. setpWalker to list rear
2. loop (pWalker not null)
    1. write (pWalker data)
    2. set pWalker to pWalker back pointer
3. end loop

end displayRearToHead

i. Deleting the entire DLL

Algorithm destroyDLL (list)

Purpose: Deletes all data in list.
Pre: list is metadata structure to a valid list
Post: All data deleted

1. loop (not at end of list)
    1. set list head to successor node
    2. release memory to heap
2. end loop
3. set list pos to null
4. set list count to 0

end destroyDLL

j. Searching the position of target from the head of the list

Algorithm searchDLLFromHead (list,target)

Purpose: Searches list and passes back the position of the target node from head.
Pre: list is metadata structure to a valid list
target is the key being sought
Return: pos: found
-1 : not found

1. set pLoc to list head
2. set pos to 1
3. loop (pLoc not null AND target >pLoc data)
    1. increment pos by 1
    2. set pLoc to pLoc fore
4. end loop
5. if (pLoc null)
    1. return -1
6. else
    1. if (target equal pLoc data)
        1. return pos
    2. else
        1. return -1
    3. end if
7. end if

endsearchDLLFromHead

k. <u>Searching the position of target from the rear of the list</u>

Algorithm searchDLLFromRear (list,target)

Purpose: Searches list and passes back the position of the target node from head.
Pre: list is metadata structure to a valid list
target is the key being sought
Return: pos: found
-1 : not found

1. set pLoc to list rear
2. set pos to 1
3. loop (pLoc not null AND target <pLoc data)
    1. increment pos by 1
    2. set pLoc to pLoc back
4. end loop
5. if (pLoc null)
    1. return -1
6. else
    1. if (target equal pLoc data)
        1. return pos
    2. else
        1. return -1
    3. end if
7. end if
8. endsearchDLLFromRear

Ex. No: 08 Demonstration of Circular DLL operations

a. Creating an Empty CDLL

Algorithm createCDLL(list**)**

Purpose: Initializes metadata for list
Pre: list is metadata structure passed by reference
Post: metadata initialized

1. allocate (list)
2. set list head to null
3. set list rear to null
4. set list count to 0

end createCDLL

b. Inserting an element into a CDLL

Algorithm insertCDLL (list, dataIn)

Purpose: This algorithm inserts data into a circular doubly linked list.
Pre: list is metadata structure to a valid list
dataIn contains the data to be inserted
Post: The data have been inserted in sequence
Return: 0: failed--dynamic memory overflow
1: successful
2: failed--duplicate Data presented

1. if (full list)
    1. return 0
2. end if
3. set found to searchList(list, predecessor, successor, dataIn Data)
4. if (not found)
    1. allocate new node
    2. move dataIn to new node
    3. increment list count by 1
    4. if (predecessor is null)
        1. if(list is empty)
            1. 1 set list head to newnode
            2. 2 set list rear to newnode
        2. else
            1. 1set list head back to newnode
            2. 2 set list rear fore to newnode
        3. endif

        4. set newnode back to list rear
        5. set newnode fore to list head
        6. set list head to newnode
    5. else
        1. set new node fore pointer to successor
        2. set new node back pointer to predecessor
        3. set predecessor fore pointer to new node
        4. if(successor is equal to list head)
            1. 1set list rear to newnode
        5. endif
        6. set successor back to newnode

    6. end if
    7. return 1
5. else
    1. return 2
6. endif

end insertCDLL

c. <u>Deleting an element from CDLL</u>

Algorithm deleteCDLL (list, target)

Purpose: This algorithm deletes a node from a circular doubly linked list.
Pre: list is metadata structure to a valid list
Target is delete node data
Post: node deleted

1. set predecessor to null
2. set deletenode to null
3. set found to searchList(list, predecessor, successor, target)
4. if(found)
    1. if(list count is 1)
        1. set list head to null
        2. set list rear to null
    2. else
        1. if(predecessor is not null)
            1. set predecessor fore to deletenode fore
            2. set deletenode fore back to predecessor
            3. if(deletenode is equal to list rear)
                a. 1 set list rear to predecessor
            4. endif
        2. else
            1. set list head to deletenode fore
            2. set deletenode fore back to deletenode back
            3. set list rear fore to deletenode fore

                3. endif
        3. endif
        4. recycle deletenode
        5. decrement list count by 1
5. else
        1. write "data not found"
6. endif
7. end deleteCDLL

d. Searching for the address of an element and its predecessor node in the CDLL

Algorithm searchList (list, pPre, pLoc, target)

Purpose: Searches list and passes back address of node containing target and its logical predecessor.
Pre: list is metadata structure to a valid list
pPre is pointer variable for predecessor
pLoc is pointer variable for current node
target is the Data being sought
Post: pLoc points to first node with equal/greater Data -or- null if target > Data of last node
pPre points to largest node smaller than Data -or- null if target < Data of first node
Return: true if found, false if not found

1. set pPre to null
2. set pLoc to list head
3. if(list is empty)
    1. return false
4. endif
5. if(pLoc data is target)
    1. return true
6. endif
7. if(target < pLoc data)
    1. return false
8. endif
9. set ppre to pLoc
10. set pLoc to pLoc fore
11. loop (pLoc not list head AND target > pLoc Data)
    1. set pPre to pLoc
    2. set pLoc to pLoc link
12. end loop
13. if (pLoc is list head)
    1. set found to false
14. else
    1. if (target equal pLoc Data)

            1. set found to true
        2. else
            1. set found to false
        3. end if
    15. end if
    16. return found
end searchList

e. <u>Check for Empty CDLL</u>

Algorithm emtpyCDLL (list)

Purpose: Check whether the list is empty.
Pre: list is metadata structure to a valid list
Return: true if list empty, false if list contains data

    1. if (list count equal 0)
        1. return true
    2. else
        1. return false
end emptyCDLL

f. <u>Check for Full CDLL</u>

Algorithm fullCDLL (list)

Purpose: Check whether the list is full.
Pre: list is metadata structure to a valid list
Return: false if room for new node; true if memory full

    1. if (memory full)
        1. return true
    2. else
        1. return false
    3. end if
    4. return true
end fullCDLL

g. <u>Counting number of nodes in CDLL</u>

Algorithm cdllCount (list)

Purpose: Counting number of nodes in list.
Pre: list is metadata structure to a valid list
Return: count for number of nodes in list

1. return (list count)

end cdllCount

h. <u>Displaying elements of CDLL from Head to Rear</u>

Algorithm displayHeadToRear(list)

Purpose: displaying the elements in list from head to rear
Pre: list is metadata structure to a valid list
Returns: print the data of nodes in list.

1. set pWalker to list head
2. if(pWalker is not null)
    1. write pWalker data
    2. set pWalker to pWalker fore
3. endif
4. loop (pWalker not list head)
    1. write (pWalker data)
    2. set pWalker to pWalker fore pointer
5. end loop

end displayHeadToRear

i. <u>Displaying elements of CDLL from Rear to Head</u>

Algorithm displayRearToHead(list)

Purpose: displaying the elements in list from rear to head
Pre: list is metadata structure to a valid list
Returns: print the data of nodes in list.

1. set pWalker to list rear
2. if(pWalker is not null)
    1. write pWalker data
    2. set pWalker to pWalker back
3. endif
4. loop (pWalker not list head)
    1. write (pWalker data)
    2. set pWalker to pWalker back pointer
5. end loop

end displayRearToHead

j. <u>Deleting all the elements of CDLL</u>

Algorithm destroyCDLL (list)

Purpose: Deletes all data in list.
Pre: list is metadata structure to a valid list
Post: All data deleted

1. set temp to list head
2. set temp1 to null
3. if(list count in not empty)
    1. set temp1 to temp
    2. set temp to temp fore
    3. recycle temp1
    4. loop (temp is not list head)
        1. set temp1 to temp
        2. set temp to temp fore
        3. recycle temp1
    5. end loop
4. endif
5. set list head to null
6. set list rear to null
7. set list count to 0

end destroyCDLL

k. <u>Searching the positin of target element in CDLL from Head</u>

Algorithm searchCDLLFromHead (list,target)

Purpose: Searches list and passes back the position of the target node from head.
Pre: list is metadata structure to a valid list
target is the Data being sought
Return: pos: found
-1 : not found

1. set pLoc to list head
2. set pos to 1
3. if(list is empty)
    1. return -1
4. endif
5. if(target is pLoc data)
    1. return pos
6. endif
7. set pLoc to pLoc fore
8. increment pos by 1
9. loop (pLoc not list head AND target > pLoc data)
    1. increment pos by 1
    2. set pLoc to pLoc fore
10. end loop
11. if (target equal pLoc data)

1. return pos
12. else
    1. return -1
13. end if
end searchCDLLFromHead

l. <u>Searching the positin of target element in CDLL from Rear</u>

Algorithm searchCDLLFromRear (list,target)

Purpose: Searches list and passes back the position of the target node from head.
Pre: list is metadata structure to a valid list
target is the Data being sought
Return: pos: found
-1 : not found

1. set pLoc to list rear
2. set pos to 1
3. if(list is empty)
    1. 1 return -1
4. endif
5. if(target is pLoc data)
    1. 1 return pos
6. endif
7. set pLoc to pLoc back
8. increment pos by 1
9. loop (pLoc not list rear AND target > pLoc data)
    1. increment pos by 1
    2. set pLoc to pLoc back
10. end loop
11. if (target equal pLoc data)
    1. return pos
12. else
    1. return -1
13. end if
end searchCDLLFromRear

Ex. No: 09 Demonstration of BST operations

a. **Algorithm addBST (root, newNode)**

Purpose: Insert node containing new data into BST using recursion.

Pre: root is address of current node in a BST
newNode is address of node containing data
Post: newNode inserted into the tree
Return: address of potential new tree root

1. if (empty tree)
    1. set root to newNode
    2. return newNode
2. end if
3. if (newNode < root)
    1. return addBST (leftSubTree, newNode)
4. else
    1. return addBST (rightSubTree, newNode)
5. end if

end addBST

b. **Algorithm deleteBST (root, dltKey)**

Purpose: This algorithm deletes a node from a BST.

Pre: root is reference to node to be deleted
dltKey is key of node to be deleted
Post: node deleted
if dltKey not found, root unchanged
Return: true if node deleted, false if not found

1. if (empty tree)
    1. return false
2. end if
3. if (dltKey < root)
    1. return deleteBST (leftSubTree, dltKey)
4. else if (dltKey > root)
    1. return deleteBST (rightSubTree, dltKey)
5. else
    1. If (no leftSubTleftSubTreeree)
        1. make rightSubTree the root
        2. return true
    2. else if (no rightSubTree)
        1. make leftSubTree the root
        2. return true

3. else
    1. save root in deleteNode
    2. set largest to largestBST (left subtree)
    3. move data in largest to deleteNode
    4. return deleteBST (leftSubTree of deleteNode, key of largest)
4. end if

6. end if

end deleteBST

c. **Algorithm searchBST (root, targetKey)**

Purpose: Search a binary search tree for a given value.

Pre: root is the root to a binary tree or subtree
targetKey is the key value requested
Return: the node address if the value is found
null if the node is not in the tree

1. if (empty tree)
    1. return null
2. end if
3. if (targetKey < root)
    1. return searchBST (leftSubTree, targetKey)
4. else if (targetKey > root)
    1. return searchBST (rightSubTree, targetKey)
5. else
    1. return root
6. end if

end searchBST

d. **Algorithm findLargestBST (root)**

Purpose: This algorithm finds the largest node in a BST.
Pre: root is a pointer to a nonempty BST or subtree
Return: address of largest node returned

1. if (rightSubTree empty)
    1. return (root)
2. end if
3. return findLargestBST (rightSubTree)

end findLargestBST

e. **Algorithm findSmallestBST (root)**

Purpose: This algorithm finds the smallest node in a BST.
Pre: root is a pointer to a nonempty BST or subtree

Return: address of smallest node

1. 1 if (leftSubTree empty)
    1. return (root)
2. end if
3. return findSmallestBST (leftSubTree)

end findSmallestBST

f. **Algorithm preOrder (root)**

Purpose: Traverse a binary tree in node-left-right sequence.
Pre: root is the entry node of a tree or subtree
Post: each node has been processed in order

1. if (root is not null)
    1. process (root)
    2. preOrder (leftSubTree)
    3. preOrder (rightSubTree)
2. end if

end preOrder

g. **Algorithm inOrder (root)**

Purpose: Traverse a binary tree in left-node-right sequence.
Pre: root is the entry node of a tree or subtree
Post: each node has been processed in order

1. if (root is not null)
    1. inOrder (leftSubTree)
    2. process (root)
    3. inOrder (rightSubTree)
2. end if

end inOrder

h. **Algorithm postOrder (root)**

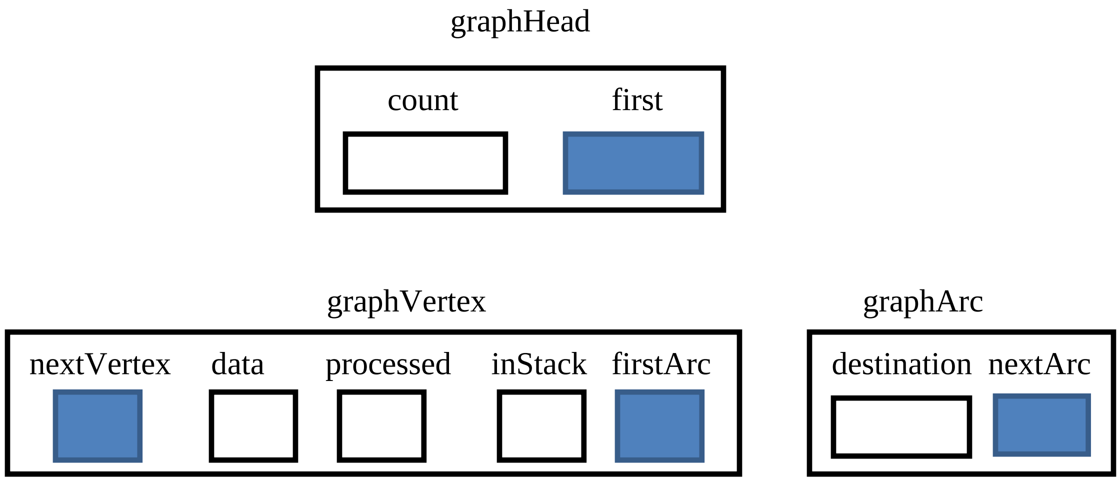
Purpose: Traverse a binary tree in left-right-node sequence.
Pre: root is the entry node of a tree or subtree
Post: each node has been processed in order

1. if (root is not null)
    1. postOrder (leftSubTree)
    2. postOrder (rightSubTree)
    3. process (root)
2. end if

end postOrder

Ex. No: 10 Graph traversal using BFS

Graph Data Structure and its creation



Algorithm createGraph

Purpose: Initializes the metadata elements of a graph structure.
Pre: graph is a reference to metadata structure
Post: metadata elements have been initialized

1. allocate memory for graphHead
2. set count to 0
3. set first to null
4. return graphHead

end createGraph

Algorithm insertVertex(graph, dataIn)

Purpose: Allocate memory for a new vertex and copies the data to it.
Pre: graph is a reference to graphHead structure
dataIn contains data to be inserted into vertex
Post: new vertex inserted and data copied

1. allocate memory for new graphVertex structure
2. store dataIn in graphVertex
3. initialize metadata elements in newVertex
4. increment count in graphHead structure
5. if (empty graph)
    1. set graph first to newVertex
6. else
    1. search for insertion point

2. if (inserting before first vertex)
    1. set graph first to newVertex
3. else
    1. insert newVertex in sequence
4. end if

7. endif

end insertVertex

Algorithm insertArc(graph, fromKey, toKey)

Purpose: Adds an arc between two vertices
Pre: graph is reference to graphHead structure
fromKey is the data of the originating vertex
toKey is the data of the destination vertex
Post: arc added to adjacency list
Return:
+1 if successful
-2 if fromKey not found
-3 if toKey not found

1. allocate memory for newArc
2. search and set fromVertex
3. if (fromVertex not found)
    1. return -2
4. endif
5. search and set toVertex
6. if (toVertex not found)
    1. return -3
7. end if
8. set arc destination to toVeretex
9. if (fromVertex arc list empty)
    1. set fromVertex firstArc to newArc
    2. set new arc nextArc to null
    3. return 1
10. end if
11. find insertion point in arc list
12. if (insert at beginning of arc list)
    1. set fromVertex firstArc to new arc
13. else
    1. insert in arc list
14. end if
15. return 1

end insertArc

<u>Breadth First Search:</u>

The breadth first traversal of a graph processes a vertex, and then processes all of its adjacent vertices. A queue data structure is used in implementing the process.

Algorithm breadthFirst(graph)

Purpose: Processes the nodes in the given graph in breadth first manner
Pre: graph is pointer to the graph data structure
Post: nodes are processed and displayed

```
1  if(empty graph)
    1. return
2  end if
3  createQueue(queue)
4  loop(through all vertices)
      1. set vertex processed to null
5  end loop
6  loop(through all vertices)
    1 if (vertex processed not set)
        1. if (vertex enqueued not set)
            1. enqueue(queue,walkPtr)
            2. set vertex to enqueued
        2. end if
        3. loop(not emptyQueue(queue))
            1. set vertex to dequeue(queue)
            2. process(vertex)
            3. set vertex to processed
            4. loop(through adjacency list)
                1. if (destination not enqueued or not processed)
                    1. enqueue(queue,destination)
                    2. set destination to enqueued
                2. end if
                3. get next destination
            5. end loop
        4. end loop
    2 end if
    3 get next vertex
7  end loop
8  destroyQueue(queue)
```

end breadthFirst

Ex. No: 11 Graph traversal using DFS

Represent the graph as same in previous exercise

Algorithm depthFirst (graph)

Purpose: Process the keys of the graph in depth-first order
Pre: graph is a pointer to a graphHead structure
Post: vertices “processed”

```
1.  if (empty graph)
      1.return
2.  end if
3.  set walkPtr to graph first
4.  loop (through all vertices)
      1.set processed to 0
5.  end loop
6.  createStack (stack)
7.  loop (through vertex list)
      1.if (vertex not processed)
            1.if (vertex not in stack)
                  1.  pushStack(stack, walkPtr)
                  2.  set walkPtr processed to 1
            2.endif
            3. loop (not emptyStack(stack)
                  1.  set vertex to popStack(stack)
                  2.  process(vertex)
                  3.  set vertex to processed
                  4.  loop (through arc list)
                        1.if (arc destination not in_stack)
                              1.  pushStack(stack, destination)
                              2.  set destination to in_stack
                        2.end if
                        3.get next destination
                  5.  end loop
            4.end loop
      2.end if
      3.get next vertex
8.  end loop
9.  destroyStack(stack)
end depthFirst
```

Ex. No: 12 Minimum Weight Spanning Tree

Algorithm SpanningTree(graph)

Purpose: Determine the minimum spanning tree of a network
Pre: graph contains a network
Post: Spanning Tree determined

```
1. If (empty graph)
   1. Return
2. end if
3. loop (through all vertices)
   1. set vertex intree flag to false
   2. loop (through all edges)
      1. Set edge intree flag to false
      2. get next edge
   3. end loop
   4. get next vertex
4. end loop
5. set first vertex to in tree
6. set treeComplete to false
7. loop (not treeComplete)
      1. set treeComplete to true
      2. set minEdge to maximum integer
      3. set minEdgeLoc to null
      4. loop (through all vertices)
         1. if (vertex in tree and vertex outdegree > 0)
            1. loop (through all edges)
               1. if (destination not in tree)
                  1. set treeComplete to false
                  2. if (edge weight  < minEdge)
                     1. set minEdge to edge weight
                     2. set minEdgeloc to edge
                  3. end if
               2. end if
               3. get next edge
            2. end loop
         2. end if
         3. get next vertex
      5. end loop
      6. if (minEdgeLoc not null)
         1. set minEdgeLoc inTree flag to true
         2. set destination inTree flag to true
      7. end if
```

8. end loop

end SpanningTree

Additional Exercises

- Ternary Search
- Circular Queue implementation using Arrays
- Evaluation of postfix expression using stack
- Operations on circular singly linked list
- Operations on AVL Tree
- MST using Kruskal’s algorithm