CS301
Data Structure and Algorithms
Lecture 7: Linked list - Circularly and doubly linked lists

Pandav Patel
Assistant Professor

Computer Engineering Department
Dharmsinh Desai University
Nadiad, Gujarat, India

Objective

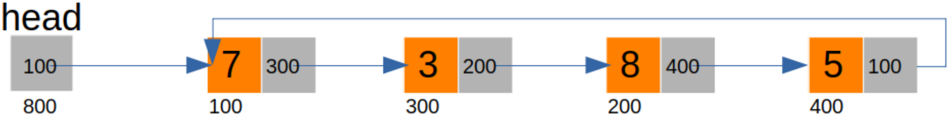
- To understand circularly linked list
- To understand doubly linked list

Overview

1. Objective
2. Circularly linked list
   - Introduction to circularly linear linked list
   - Insert at front (List with list head)
   - Insert at front (List without list head)
   - More circular list algorithms for practice
3. Doubly linked list
   - Introduction
   - Insert node to left of given node
   - Delete a given node
   - Doubly linked list as a queue
   - Doubly linked circular list

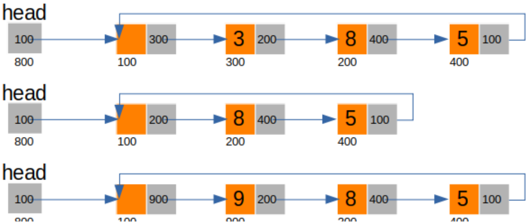
Introduction to circularly linear linked list

- So far we have seen only linked linear list
- If last node of linked linear list points to first node instead of storing NULL link, then it is called *circularly linked linear list* or simply *circular list*
- One of the advantages of circular list is that all nodes can be reached from any given node



Introduction to circularly linear linked list (cont...)

- However special care should be taken to identify end of the list otherwise there is possibility of ending up with infinite loop
  - One way to achieve this is by using special node as *list head*. Such node will not contain any INFO.
    - This makes sure that HEAD is never changed - neither on insertion of a node at the front of the list nor on deletion of the node from the front of the list
    - And because of this LINK of last node will not need to be updated in case of insertion/deletion of node at/from the front of the list



Circular list: Insert at front (List with list head)

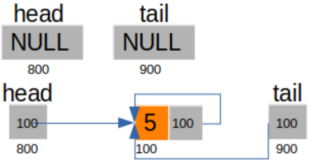
- When circular list (with list head) is empty, LINK(HEAD) = HEAD.
- Steps for Insertion of a node (with its INFO as X) at front of such list are straight forward

```
1 NEW ←Create new node
2 INFO(NEW) ←X
3 LINK(NEW) ←LINK(HEAD)
4 LINK(HEAD) ←NEW
```

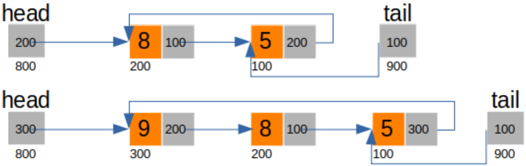
- Can above steps handle insertion of very first node (apart from list head node) in the list?

Circular list: Insert at front (List without list head)

- Cases
  - Empty list



  - Non-empty list



Circular list: Insert at front (List without list head) (cont...)

- We will assume that TAIL pointer is maintained
- We will also assume that arguments are passed by reference
- Steps
    1. Create a new node and check if it is created properly
    2. Initialize INFO part of the new node
    3. Handle empty list case - set LINK of NEW and then initialize HEAD and TAIL
    4. Insert into non-empty list - Make sure LINK(TAIL) is updated

Circular list: Insert at front (List without list head) (cont...)

Algorithm: CINSERT_AT_FRONT(X, HEAD, TAIL)

ASSUME that args are passed by ref

Insert value X at the front of the list

HEAD, TAIL: Pointers to first and last node resp

NEW: Pointer to newly created node

```
1. [ Create a new node ]
       NEW ←Create a new node
       If NEW = NULL then
           Write(”New node not created”)
           return
2. [ Set INFO of new node ]
       INFO(NEW) ←X
3. [ Insert in empty list ]
       If HEAD = NULL then
           LINK(NEW) ←NEW
           HEAD ←TAIL ←NEW
           return
4. [ Insert in non-empty list ]
       LINK(NEW) ←HEAD
       HEAD ←NEW
       LINK(TAIL) ←NEW
       return
```

Circular list: Insert at front (List without list head) (cont...)

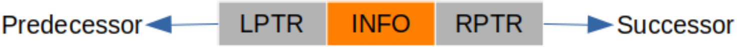
- Why did we assume that arguments are passed by reference?
  - Because we can not return HEAD and TAIL if both needs to be changed (e.g. When first node is inserted)
- If TAIL was not maintained then we would need to traverse entire list to reach last node to update its LINK
  - How would you find last node in that case?
- In step 3, why did we do following?
  - LINK(NEW) ←NEW
- Is there need to call above algorithm as follow?
  - HEAD ←CINSERT_AT_FRONT(X, HEAD, TAIL)

Try on your own

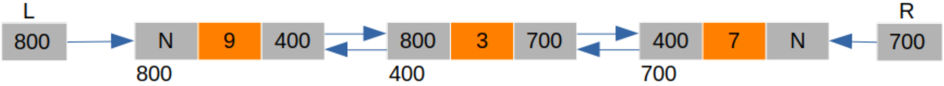
- Insertion at the end of the circular list (with list head)
- Insertion at the end of the circular list (without list head)
- Deletion of a node whose address is given by X (list with list head)
- Deletion of a node whose address is given by X (list without list head)

Introduction to doubly linear linked list

- Node structure



- Example of doubly linked list

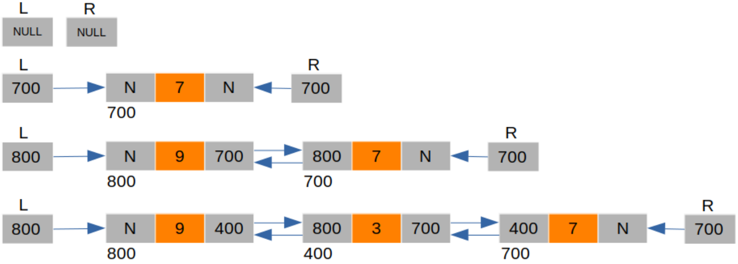


- L and R are pointers to left most and right most nodes respectively
- **Advantage** of doubly linked list is that it can be traversed in both the directions equally efficiently
- **Disadvantage** of doubly linked list is that it requires storage for two pointers for each node

Doubly linked list: Insert node to left of given node

Cases

- Empty list
- Non-empty list
  - Insert at the front (to left of node pointed by L)
  - Insert in the middle
  - Insert at the end (to right of node pointed by R)??? NOT POSSIBLE IN THIS CASE



Doubly linked list: Insert node to left of given node (cont...)

Steps

1. Create a new node. Return if node is not created
2. Initialize INFO of new node with given value
3. If list is empty
   - Initialize L and R to point to new node
   - Set left and right pointer of new node to NULL
   - return
4. Handle insertion at front of list
   - Make left pointer of first node (pointed by L) to point to new node
   - Initialize right and left pointer of new node to address of first node and NULL resp.
   - Change L to point to new node
   - return
5. Otherwise insert new node in the middle
   - Make left and right pointers of new node to point to nodes which will be its predecessor and successor node after insertion
   - Change right pointer of predecessor and left pointer of successor to point to new node

Algorithm: DINSERT_TO_LEFT(L, R, M, X)

Assumptions: M is NULL or has valid address. And args are passed by ref

Insert value X on left of node with address M

L, R: Pointers to left-most and right-most nodes

NEW: Pointer to newly created node

```
1. [ Create a new node ]
    NEW ←Create a new node
    If NEW = NULL then
        Write("New node not created")
        return
2. [ Set INFO of new node ]
    INFO(NEW) ←X
3. [ Insert in empty list ]
    If L = NULL then
        L ←R ←NEW
        LPTR(NEW) ←RPTR(NEW) ←NULL
        return
4. [ Handle insertion at left end ]
    If M = L then
        LPTR(NEW) ←NULL
        RPTR(NEW) ←M
        LPTR(M) ←NEW
        L ←NEW
        return
5. [ Insert in middle ]
    LPTR(NEW) ←LPTR(M)
    RPTR(NEW) ←M
    LPTR(M) ←NEW
    RPTR(LPTR(M)) ←NEW
    return
```

Doubly linked list: Insert node to left of given node (cont...)

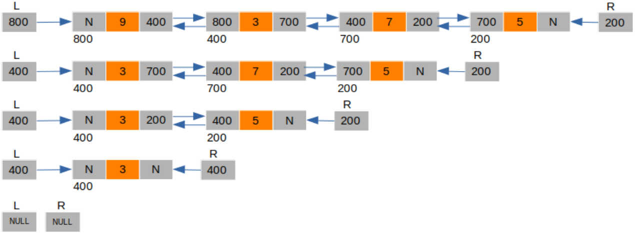
Discussion

- In step 4, can we replace $LPTR(NEW) \leftarrow NULL$ with $LPTR(NEW) \leftarrow LPTR(M)$
- Do you observe redundancy in step 4 and step 5?
    - Can you rewrite algorithm to take out common instructions from step 4 and step 5?
    - Only second last instruction is different in both steps

Doubly linked list: Delete a given node

Cases

- Empty list (INVALID case)
- Non-empty list
    - First node
        - List has only one node
    - Middle node
    - Last node



Doubly linked list: Delete a given node (cont...)

Steps

1. Check if list has only one node
   - Set L to NULL
   - Set R to NULL
   - Free the memory of node to be deleted
   - Return
2. Handle deletion of left-most node
   - Set L to point to second node in the list
   - Set LPTR of second node to NULL
   - Free the memory of node to be deleted
   - Return
3. Handle deletion of last node
   - Set R to point to second last node
   - Set RPTR of second last node to NULL
   - Free the memory of node to be deleted
   - Return
4. Handle deletion of middle node
   - Set RPTR of predecessor to successor
   - Set LPTR of successor to predecessor
   - Free the memory of node to be deleted
   - Return

Algorithm: DDELETE(L, R, M)

Assumptions: Node with address M exists. And args are passed by ref

Delete node with address M

L, R: Pointers to left-most and right-most nodes

```
1. [ Does list have only one node? ]
    If L = R then
        L ←R ←NULL
        FREE(M)
        return
2. [ Deleting left-most node? ]
    If L = M then
        L ←RPTR(M)
        LPTR(L) ←NULL
        FREE(M)
        return
3. [ Deleting right-most node? ]
    If R = M then
        R ←LPTR(M)
        RPTR(R) ←NULL
        FREE(M)
        return
4. [ Deleting middle node ]
    RPTR(LPTR(M)) ←RPTR(M)
    LPTR(RPTR(M)) ←LPTR(M)
    FREE(M)
    return
```

Doubly linked list: Delete a given node (cont...)

Discussion

- Do you observe redundancy of instructions in above algorithm?
  - Can rewrite algorithm after taking out common instructions to free memory and return?

Doubly linked list as a queue

- Operations
  - Insert at right end
  - Remove from left end

- Alternate Operations
  - Insert at left end
  - Remove from right end

Doubly linked list as a queue (cont...)

Insert at right end

Algorithm: QINSERT(L, R, X)

Assumptions: Args are passed by ref

Insert value X at right end of the queue

L, R: Pointers to left-most and right-most nodes

NEW: Pointer to newly created node

```
1. [ Create a new node ]
     NEW ←Create a new node
     If NEW = NULL then
          Write(”New node not created”)
          return
2. [ Set INFO of new node ]
     INFO(NEW) ←X
3. [ Is list empty? ]
     If L = NULL then
          L ←R ←NEW
          LPTR(NEW) ←RPTR(NEW) ←NULL
          return
4. [ Insert into non-empty list ]
     LPTR(NEW) ←R
     RPTR(NEW) ←NULL
     RPTR(R) ←NEW
     R ←NEW
     return
```

Doubly linked list as a queue (cont...)

Delete from left end

Algorithm: QREMOVE(L, R)

Assumptions: Args are passed by ref

Delete node from the left end of the queue

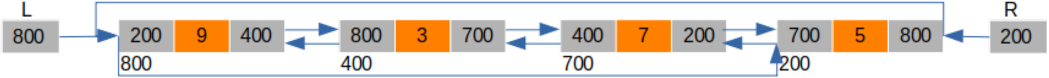
L, R: Pointers to left-most and right-most nodes

TEMP: Temporary variable to hold value 1. [Is list

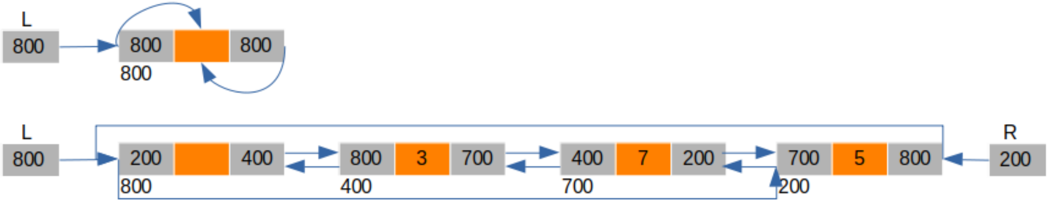
empty?]

```
    If L = NULL then
        Write("Queue is empty")
        return
2. [ Does queue contain only one node? ]
    If L = R then
        TEMP ←INFO(L)
        FREE(L)
        L ←R ←NULL
        return TEMP
3. [ Delete when queue has more than one nodes ]
    TEMP ←INFO(L)
    L ←RPTR(L)
    FREE(LPTR(L))
    LPTR(L) ←NULL
    return TEMP
```

Doubly linked circular list (without list head)



Doubly linked circular list (with list head)



More algorithms for practice

- Write an algorithm for insertion of node to the right of given node in doubly linked circular list (with list head)
- Write an algorithm for insertion of node to the right of given node in doubly linked circular list (without list head)
- Write an algorithm to print singly linked list in reverse
- Write an algorithm to delete a given node from doubly linked circular list (with list head)
- Write an algorithm to delete a given node from doubly linked circular list (without list head)