CS301
Data Structure and Algorithms
Lecture 5: Linked list Basics and Insert operations

Pandav Patel
Assistant Professor

Computer Engineering Department
Dharmsinh Desai University
Nadiad, Gujarat, India

Objective

- Understanding linked list and its operations
- Learn to insert new nodes into linked list

Linked list fundamentals

- What is a *node*?
  - What type of values can a node store?
    - can be one or more
    - can be primitive or non-primitive type
- How are nodes linked?
  - What is *self-referential structure/class*?

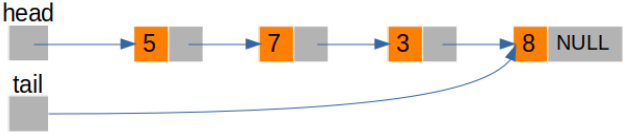
```
struct Node {
    int x, y;
    Complex c;
    struct Node *LINK;
};
```

- x, y and c are considered as INFO

Linked list fundamentals (CONT...)

- How to mark end of the list?
  - by storing NULL into LINK of last node
- What is *head* and *tail*?
  - **Pointers** to the *first node* and the *last node* in the list
  - NOTE: Not all implementations maintain tail pointer
- Let us consider simple node structure as follow

```
struct Node {
    int INFO;
    struct Node *LINK;
};
```



Ignite your curiosity

- Where are nodes located in memory? Is storage continuous?
  - How does non-contiguous storage impact *cache memory* and hence performance?
    - What is cache memory and locality of reference? Ans: here
    - Linked list and cache: here and here
- What if linked list has a *loop*?
  - Can we detect the loop? How? here, here and here
  - Can we find the node from where the loop starts? here

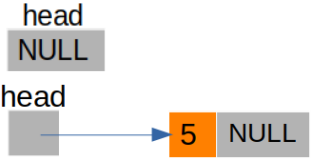
Operations on linked list

- *Insert* a node
- *Delete* a node
- *Search* for a value
- *Retrieve* value of a node at given position
- *Update* value of a node at given position
- *Traverse* a list (e.g. to print values of all nodes)
- *Copy* a list
- Find *Size* of the linked list
- *Join* two linked lists
- *Split* linked list into two
- *Sorting* of linked list

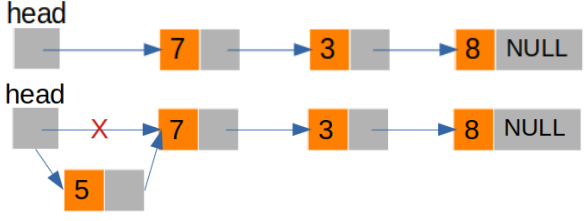
- We will look into these operations
- **Think of time and space complexity of the above operations**

Insert a node at the front

- Cases
  - Empty list



  - Non-empty list



Insert a node at the front (cont...)

- Steps
  1. Create a new node
  2. Check if new node is created properly
  3. Initialize INFO and LINK part of the new node
  4. Return address of the new node

- Clarification regarding algorithmic notation used in next slide
  - INFO(HEAD) in the algorithm is same as HEAD→INFO in C language
  - LINK(HEAD) in the algorithm is same as HEAD→LINK in C language
  - HEAD and NEW are pointers to node instances and are not node instances themselves

Algorithm: INSERT_AT_FRONT(X, HEAD)

Insert an element X at the front of the list
HEAD: Pointer to the first node of the linked list
NEW: Temporary node pointer
Algorithm should be called as
HEAD = INSERT_AT_FRONT(X, HEAD)

1. [Create a new node]
   NEW ←Create a new node
2. [Return if new node is not created]
   If NEW = NULL then
      Write("New node not created")
      return(HEAD)
3. [Initialize INFO and LINK of the new node]
   INFO(NEW) ←X
   LINK(NEW) ←HEAD
4. [Return the address of the new node to be inserted at front]
   return(NEW)

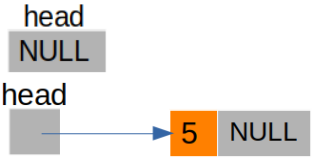
Insert a node at the front (cont...)

- What if calling algorithm calls INSERT_AT_FRONT as follow
  - INSERT_AT_FRONT(X, HEAD)
  - and not as follow
  - HEAD = INSERT_AT_FRONT(X, HEAD)
    - Ans: Calling algorithm maintains its own copy of HEAD (separate from HEAD in INSERT_AT_FRONT algorithm), it needs to be updated to point to NEW node (which will become first node in the list after insertion). And for that to happen it is important that calling algorithm calls INSERT_AT_FRONT algorithm as expected.

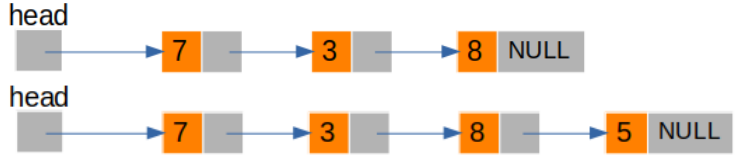
- Please note that the algorithm is returning the **address** of the first node.
  - NEW is a *pointer* to the node instance and not a node instance
- Also note that there is no separate logic for *empty* list and *non-empty* list.
  - Why?
    - Ans: because steps required are same in both the cases

Insert a node at the end

- Cases
  - Empty list



  - Non-empty list



Insert a node at the end (cont...)

- Steps
  1. Create a new node
  2. Check if new node is created properly
  3. Initialize INFO and LINK part of the new node
  4. Return address of new node if list is empty
  5. Traverse to the last node in the list
  6. Add new node behind the last node
  7. Return address of the first node

Algorithm: INSERT_AT_END(X, HEAD)

Insert an element X at the end of the list
HEAD: Pointer to the first node of the linked list
NEW, CURRENT: Temporary node pointers
Algorithm should be called as
HEAD = INSERT_AT_END(X, HEAD)

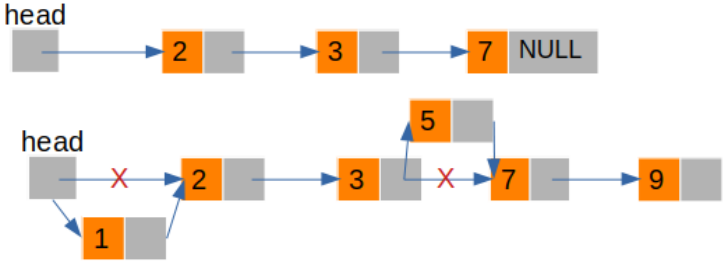
1. [Create a new node]
   NEW ←Create a new node
2. [Return if new node is not created]
   If NEW = NULL then
      Write("New node not created")
      return(HEAD)
3. [Initialize INFO and LINK of the new node]
   INFO(NEW) ←X
   LINK(NEW) ←NULL
4. [Is list empty?]
   If HEAD = NULL then
      return(NEW)
5. [Traverse to the last node in the list]
   CURRENT ←HEAD
   While LINK(CURRENT) $\neq$ NULL do
      CURRENT ←LINK(CURRENT)
6. [Set LINK of last node to newly created node]
   LINK(CURRENT) ←NEW
7. [Return the address of the first node in the list]
   return(HEAD)

Insert a node at the end (cont...)

- Why do we need to handle the empty list case separately?
  - Because steps required in case of empty list and non-empty list are very different
  - Also, if step 4 is not present in above algorithm, then in case of empty list, CURRENT will get NULL value at step 5.
  - Then in step 5, we are trying to access LINK(CURRENT). This will result in error if CURRENT is NULL

Insert a node in an ordered list

- cases
  - Empty list
  - Non-empty list
    - Insert at the front
    - Insert at the end
    - Insert in the middle



Insert a node in an ordered list (cont...)

- steps
  1. Create a new node
  2. Check if new node is created properly
  3. Initialize INFO part of the new node
  4. Set LINK of new node to NULL and return address of new node if list is empty
  5. If new node needs to be inserted at the front then set its LINK to point to first node in the list and return address of new node
  6. Search predecessor of the new node
  7. Set LINK of new node to point to its successor and LINK of predecessor to point to new node
  8. Return address of the first node in the list

Algorithm: INSERT_IN_ORDER(X, HEAD)
Insert an element X to the list in order
HEAD: Pointer to the first node of the linked list
NEW, CURRENT: Temporary node pointers
Algorithm should be called as
HEAD = INSERT_IN_ORDER(X, HEAD)

1. [Create a new node]
   NEW ←Create a new node
2. [Return if new node is not created]
   If NEW = NULL then
      Write("New node not created")
      return(HEAD)
3. [Initialize INFO of the new node]
   INFO(NEW) ←X
4. [Is list empty?]
   If HEAD = NULL then
      LINK(NEW) ←NULL
      return(NEW)
5. [Need to insert new node at front?]
   If INFO(NEW) ≤ INFO(HEAD) then
      LINK(NEW) ←HEAD
      return(NEW)
6. [Traverse to the predecessor of the new node]
   CURRENT ←HEAD
   While LINK(CURRENT) ≠ NULL and
   INFO(LINK(CURRENT)) ≤ INFO(NEW) do
      CURRENT ←LINK(CURRENT)
7. [Set LINK of new node and predecessor]
   LINK(NEW) ←LINK(CURRENT)
   LINK(CURRENT) ←NEW
8. [Return the address of the first node in the list]
   return(HEAD)

Insert a node in an ordered list (cont...)

- Why we did not need to handle insert at the end and insert in the middle cases separately?
  - While finding predecessor we have taken care of both cases in condition of the while loop
  - And once we have predecessor, steps required are same in both cases.
- What if we change $\leq$ to $<$ sign in both instances in the algorithm?
  - Algorithm will still work as expected