4. Linked Lists

Dr. Swati Agarwal

Agenda

1. Data Structures
2. Linked Lists
3. Basic Operations on LinkedList
   - Traversing a Linked List
   - Inserting into a Linked List
   - Deletion from a Linked List

Data Structures

Definition

A logical or mathematical model to group, store, and organize our data

Examples: Why? Use of organizing data in our real life.

- Sorted Dictionary
- City Map
- Transaction Statement

Same concept applies to the data used by computers as they deal with a wide variety and large amount of data.

Data Structures

1. Mathematical or Logical Model
    - Abstract view of data structures
    - High-level features and operations that define a data structures
    - **Example:** Abstract view of a TV, it can be turned on and off, it can receive signals, it can play the audio/video

Operations

- Store a given number of elements of any type
- Read elements by position
- Modify element at a position

2. Concrete implementation of the data structures.

Types of Data Structures

- **Linear Data Structures:** elements form a sequence or a linear list. The data is arranged in a linear fashion although the way they are stored in the memory need not to be sequential.
  1. Arrays
  2. Linked Lists
  3. Stacks (LIFO)
  4. Queues (FIFO)

- **Non-linear Data Structures:** data is not arranged in sequence. The ADT operations therefore not possible in a linear fashion
  1. Trees
  2. Graphs

Linked Lists

Linked Lists

A data structure in which the objects are organized in a linear order.

Linked Lists

A data structure in which the objects are organized in a linear order.

- Separate memory allocation request is made for each element or object

Linked Lists

A data structure in which the objects are organized in a linear order.

- Separate memory allocation request is made for each element or object

- Consecutive memory blocks may or may not be assigned

Linked Lists

A data structure in which the objects are organized in a linear order.

- Separate memory allocation request is made for each element or object
- Consecutive memory blocks may or may not be assigned
- Each element is divided into two memory blocks: data value and reference to the next element

Linked Lists

A data structure in which the objects are organized in a linear order.

- Separate memory allocation request is made for each element or object
- Consecutive memory blocks may or may not be assigned
- Each element is divided into two memory blocks: data value and reference to the next element
- Stores the additional information on each block to find the sequence of elements in the list

Linked Lists

A data structure in which the objects are organized in a linear order.

- Separate memory allocation request is made for each element or object
- Consecutive memory blocks may or may not be assigned
- Each element is divided into two memory blocks: data value and reference to the next element
- Stores the additional information on each block to find the sequence of elements in the list
- Address of the first node or head node gives the access of the complete list

Basic Operations on LinkedList

1. Traversing the list
2. Inserting an element (node) to the list
3. Deleting an element from the list

Traversing a Linked List

L represents the List, k represents the key to be traversed/searched in the list.
x represents the location /address of a node. *key* represents the value/data/key of a node.
next represents the pointer towards the next node.

function $\textsc{List-Search}(L, k)$
    $x \leftarrow head[L]$
    while $x \neq NIL$ AND $key[x] \neq k$
        $x \leftarrow next[x]$
    return x

Inserting into a Linked List

- **function List-Insert-Beginning**(L, x- the element to be inserted)

  $next[x] \leftarrow head[L]$

  $head[L] \leftarrow *x$

- **function List-Insert-End**(L, x)

  $p \leftarrow head[L]$

  while $next[p] \neq NIL$

  $\quad p \leftarrow next[p]$

  $next[p] \leftarrow *x$

  $next[x] \leftarrow NIL$

- **function List-Insert-Position**(L, x- element, k- position)

  while $p \neq k$

  $\quad p \leftarrow next[p]$

  $next[x] \leftarrow next[p]$

  $next[p] \leftarrow *x$

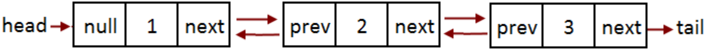
Deletion from a Linked List

p and q are two variables used to represent consecutive nodes. is p is the current node then q is the previous node traversed

- **function List-Delete-Beginning**(L)
  $head[L] \leftarrow next[head[L]]$

- **function List-Delete-End**(L)
  $p \leftarrow head[L]$
  while $next[p] \neq NIL$
  $\quad q \leftarrow p$
  $\quad p \leftarrow next[p]$
  $next[q] \leftarrow NIL$

- **function List-Delete-Position**(L, k)
  while $p \neq k$
  $\quad q \leftarrow p$
  $\quad p \leftarrow next[p]$
  $next[q] \leftarrow next[p]$



Doubly Linked List

Doubly Linked List

- **Advantages**
  1. Node will not only have the reference to the next node but also to the previous node.
  2. Given a node, we can navigate in both directions.
  3. A node in singly linked list cannot be removed unless we have pointer to the predecessor. But in doubly linked list, a node can be deleted even if we don't have previous node's address.

Doubly Linked List

- **Advantages**
  1. Node will not only have the reference to the next node but also to the previous node.
  2. Given a node, we can navigate in both directions.
  3. A node in singly linked list cannot be removed unless we have pointer to the predecessor. But in doubly linked list, a node can be deleted even if we don't have previous node's address.
- **Disadvantages**
  1. Each node requires an extra pointer, requiring more space.
  2. The insertion or deletion of a node takes a bit longer time (more pointer operations).

Insertion in Doubly Linked List

- **function DList-Insert-Beginning**(L, x)

  $next[x] \leftarrow head[L]$

  $prev[x] \leftarrow NIL$

  $prev[head[L]] \leftarrow x$

  $head[L] \leftarrow x$

- **function DList-Insert-End**(L, x)

  while $next[p] \neq NIL$

  $\quad p \leftarrow next[p]$

  $next[x] \leftarrow NIL$

  $prev[x] \leftarrow p$

  $next[p] \leftarrow x$

Insertion in Doubly Linked List

- **function DList-Insert-Position**(L, k)

  while $p \neq k$

  $\quad p \leftarrow next[p]$

  $next[k] \leftarrow next[p]$

  $prev[k] \leftarrow p$

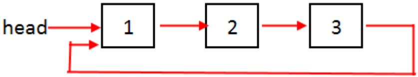
  $next[p] \leftarrow k$

  $prev[next[k]] \leftarrow k$

Deletion in Doubly Linked List

- **function DList-Delete-Beginning**(L)
  $head[L] \leftarrow next[head[L]]$
  $prev[head[L]] \leftarrow NIL$
- **function DList-Delete-End**(L)
  while $next[p] \neq NIL$
  $q \leftarrow p$
  $p \leftarrow next[p]$
  $next[q] \leftarrow NIL$

  since you have access to the prev pointer, you do not need to maintain the q node. this implementation is naive.
- **function DList-Delete-Position**(L, k)
  while $p \neq k$
  $q \leftarrow p$
  $p \leftarrow next[p]$
  $prev[next[k]] \leftarrow prev[k]$
  $next[q] \leftarrow next[k]$



Circular Linked List

Circular Linked List

- Unlike singly or doubly linked lists, circular linked lists do not have ends (No NULL value in the pointer of the last node).

- Last node of the circular linked list points to the head node (not head pointer).

Operations:

- Traversing the list
- Inserting a node
- Deleting a node

refer to the class notes for algorithms and complexities.