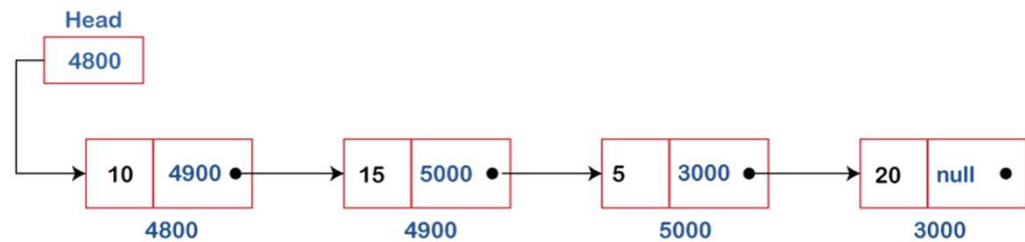# DATA STRUCTURES

WENYE LI

CUHK-SZ

OUTLINE

- Lists

- Implementation

- Examples

LIST

- Linked list: collection of nodes in which one node is connected to another node.

  - Like arrays, Linked List is a linear data structure.

  - Unlike arrays, linked list elements are not stored at a contiguous location; they are linked using pointers.

WHY LINKED LIST?

- Arrays stores linear data of similar types, with the following limitations.

  - The size of the arrays is fixed. We must know the upper limit on the number of elements in advance. The allocated memory is equal to the upper limit irrespective of the usage.

  - Inserting a new element in an array is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

- For example, in a system, if we maintain a sorted list of IDs in an array id[].

  - id[] = [1000, 1010, 1050, 2000, 2040].

  - And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

  - Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

ADVANTAGES

- Dynamic data structure:

  - The size of a linked list is not fixed as it can vary according to requirements.

- Insertion and Deletion:

  - Insertion and deletion in linked list are easier than array as the elements in an array are stored in a consecutive location. In contrast, in the case of a linked list, the elements are stored in a random location.

  - If we want to insert or delete the element in an array, then we need to shift the elements for creating the space.

  - In a linked list, we do not have to shift the elements. We just need to update the address of the pointer in the node.

- Memory efficient

  - Its memory consumption is efficient as the size of the linked list can grow or shrink according to our requirements.

DISADVANTAGES

- Memory usage

  - The node in a linked list occupies more memory than array as each node occupies two types of variables, i.e., one is a simple variable, and the other is a pointer variable that occupies 4 bytes in memory.

- Traversal

  - In an array, we can randomly access the element by index.

  - In a linked list, the traversal is not easy. If we want to access the element in a linked list, we cannot access the element randomly.

  - Example: If we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.

- Reverse traversing

  - In a linked list, backtracking or reverse traversing is difficult.

  - In a doubly linked list, it is easier but requires more memory to store the back pointer.

APPLICATIONS

```
Input:
    1st number = 5x² + 4x¹ + 2x⁰
    2nd number = -5x¹ - 5x⁰
Output:
    5x²-1x¹-3x⁰
Input:
    1st number = 5x³ + 4x² + 2x⁰
    2nd number = 5x^1 - 5x^0
Output:
    5x³ + 4x² + 5x¹ - 3x⁰
```

- Polynomial is a collection of terms in which each term contains coefficient and power.

- The coefficients and power of each term are stored as node and link pointer points to the next element in a linked list, so linked list can be used to create, delete and display the polynomial.
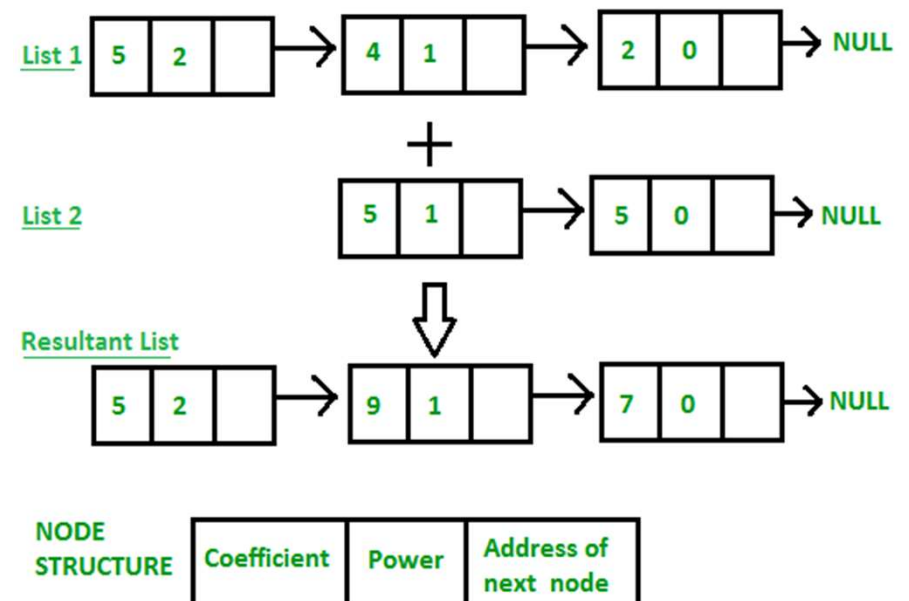
APPLICATIONS

- A sparse matrix.
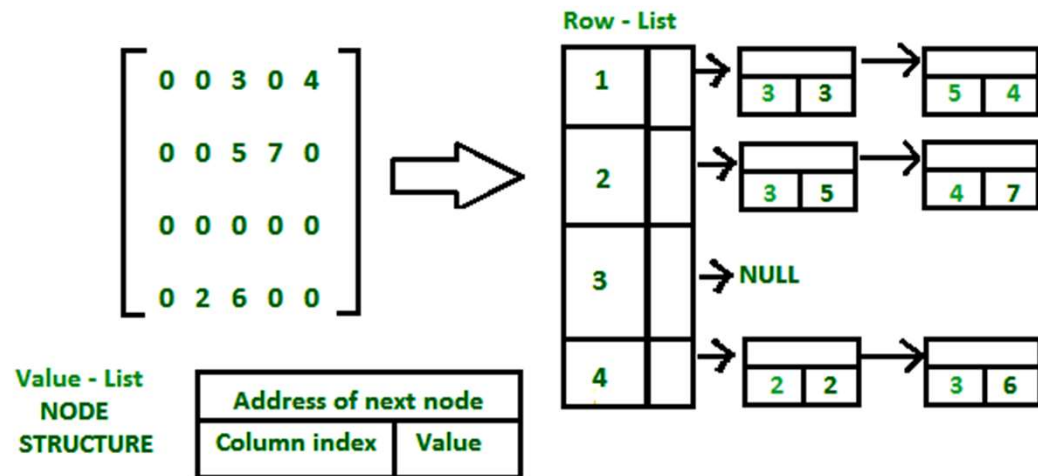
APPLICATIONS

- Various operations like student's details, employee's details or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.

- Stack, Queue, tree and various other data structures can be implemented using a linked list.

- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.

- To implement hashing, we require hash tables. The hash table contains entries that are implemented using linked list.

- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

```java
public class LinkedList
{
    Node head; // head of the list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;

        // Constructor to create a new node
        // Next is by default initialized
        // as null
        Node(int d)
        {
            data = d;
        }
    }
}
```

In Java, LinkedList can be represented as a class and a Node as a separate class.

The LinkedList class contains a reference of Node class type.

```java
// A simple Java program to introduce a linked list
class LinkedList {
    Node head; // head of list

    /* Linked list Node.  This inner class is made static so that
       main() can access it */
    static class Node {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        } // Constructor
    }

    /* method to create a simple linked list with 3 nodes*/
    public static void main(String[] args)
    {
        /* Start with the empty list. */
        LinkedList llist = new LinkedList();

        llist.head = new Node(1);
        Node second = new Node(2);
        Node third = new Node(3);

        /* Three nodes have been allocated dynamically.
           We have references to these three blocks as head,
           second and third

          llist.head          second                  third
             |                   |                       |
             |                   |                       |
         +----+------+       +----+------+       +----+------+
         | 1  | null |       | 2  | null |       | 3  | null |
         +----+------+       +----+------+       +----+------+ */

        llist.head.next = second; // Link first node with the second node

        /*  Now next of the first Node refers to the second.  So they
            both are linked.

          llist.head          second                  third
             |                   |                       |
             |                   |                       |
         +----+------+       +----+------+       +----+------+
         | 1  |  o-------->| 2  | null |       | 3  | null |
         +----+------+       +----+------+       +----+------+ */

        second.next = third; // Link second node with the third node

        /*  Now next of the second Node refers to third.  So all three
            nodes are linked.

          llist.head          second                  third
             |                   |                       |
             |                   |                       |
         +----+------+       +----+------+       +----+------+
         | 1  |  o-------->| 2  |  o-------->| 3  | null |
         +----+------+       +----+------+       +----+------+ */
    }
}
```

TYPES OF LINKED LISTS

- Singly Linked list

- Doubly Linked list

- Circular Linked list

- Doubly Circular Linked list

SINGLY LINKED LIST

- A singly linked list has two parts in a node

  - the data part

  - the address part, which contains the address of the next node, also known as a **pointer**.

- Only forward traversal is possible

  - We cannot traverse in the backward direction as it has only one link in the list.
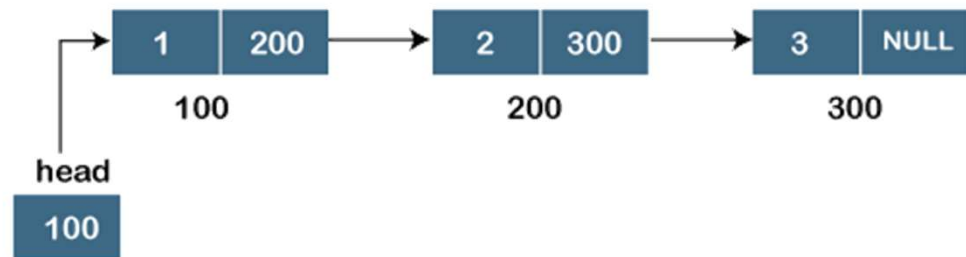
OPERATIONS ON SINGLY LINKED LIST

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

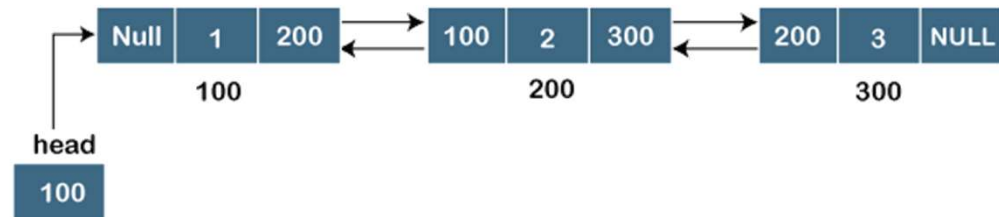| SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

OPERATIONS ON SINGLY LINKED LIST

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

DOUBLY LINKED LIST

- A doubly linked list has three parts in a node

  - a data part

  - a pointer to its previous node

  - a pointer to the next node

Head

1			

	Data	Prev	Next
1	13	-1	4
2			
3			
4	15	1	6
5			
6	19	4	8
7			
8	57	6	-1

Memory Representation of a Doubly linked list

OPERATIONS ON DOUBLY LINKED LIST

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

CIRCULAR LINKED LIST

- A circular linked list is a variation of a singly linked list.

- Difference between the *singly linked list* and a *circular linked list*

  - The last node does not point to any node in a singly linked list.

  - The last node connects to the first node in a circular linked list.

  - The circular linked list has no starting and ending node. We can traverse in any direction.

start

1

	Data	Next
1	13	4
2		
3		
4	15	6
5		
6	19	8
7		
8	57	1
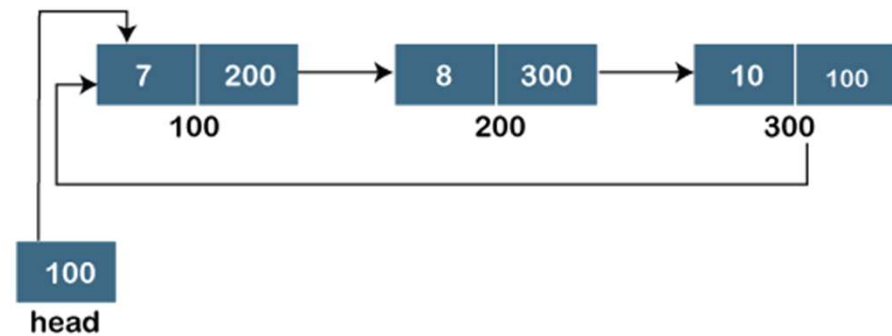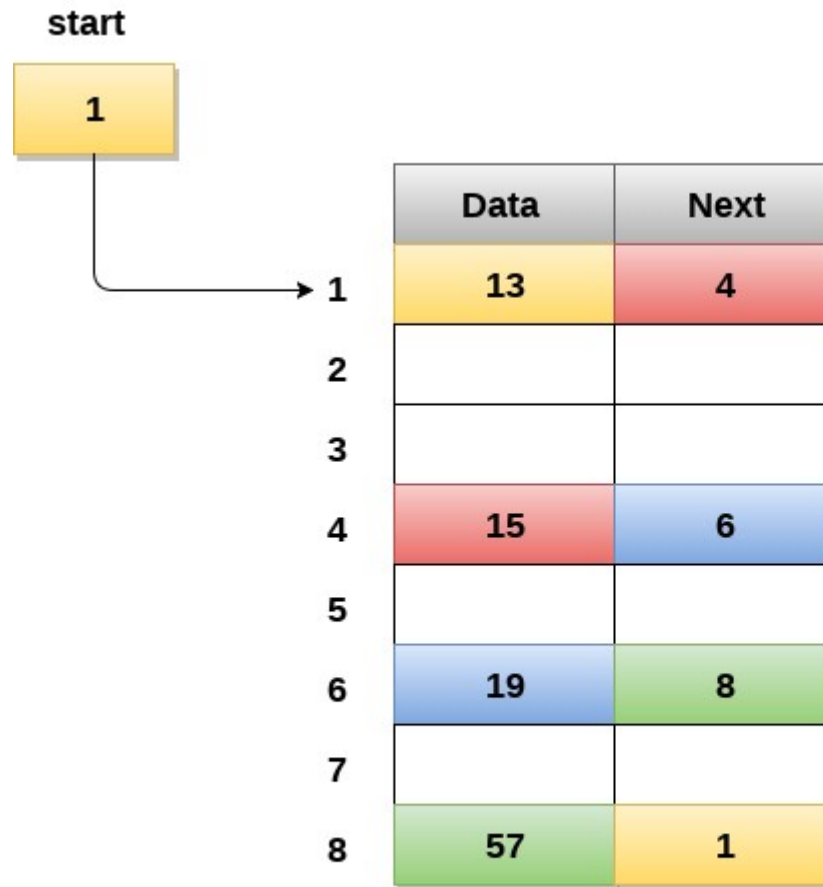
Memory Representation of a circular linked list

OPERATIONS ON CIRCULAR LINKED LIST

Insertion

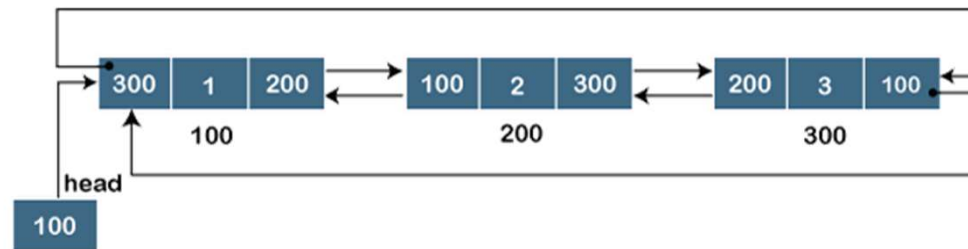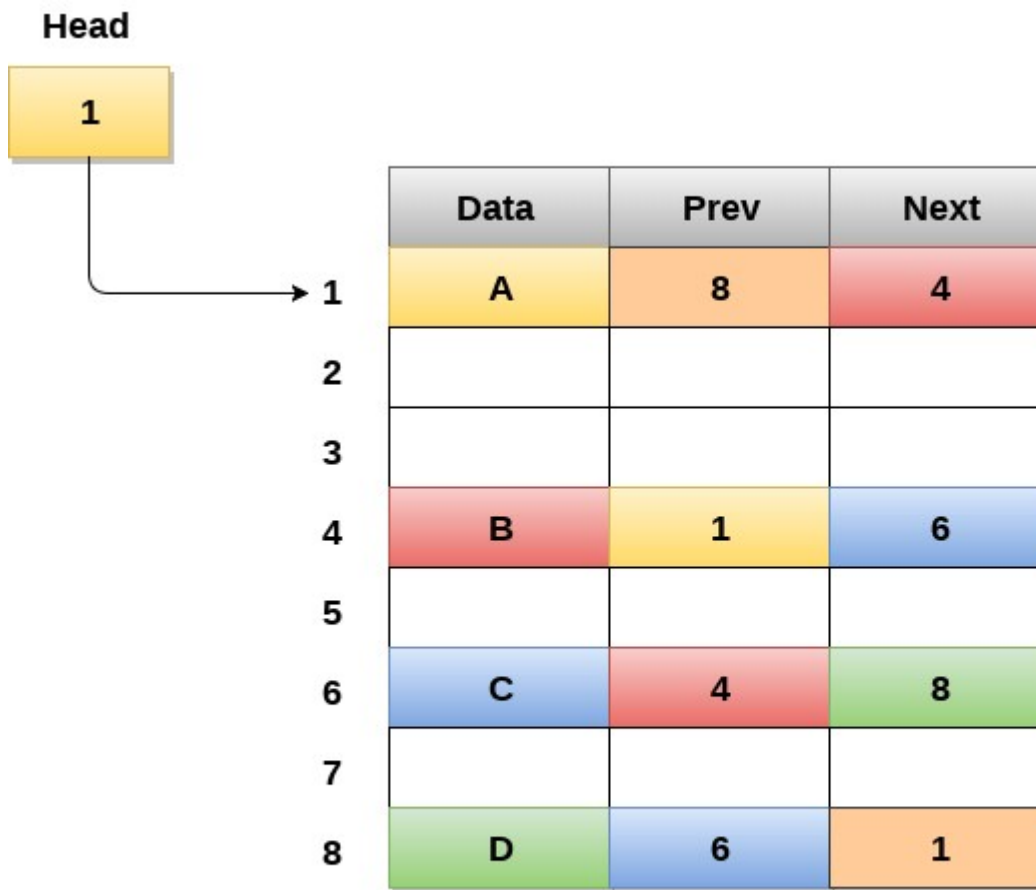| SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

DOUBLY CIRCULAR LINKED LIST

- The doubly circular linked list has the features of both the **circular linked list** and **doubly linked list**.

    - The last node is attached to the first node and creates a circle.

    - Each node holds the address of the previous node.

- A doubly circular linked list has three parts in a node

    - two address parts

    - one data part

    - So, its representation is similar to the doubly linked list.

Head

1		

	Data	Prev	Next
1	A	8	4
2			
3			
4	B	1	6
5			
6	C	4	8
7			
8	D	6	1

Memory Representation of a Circular Doubly linked list

OPERATIONS ON DOUBLY CIRCULAR LINKED LIST

SN	Operation	Description
1	Insertion at beginning	Adding a node in circular doubly linked list at the beginning.
2	Insertion at end	Adding a node in circular doubly linked list at the end.
3	Deletion at beginning	Removing a node in circular doubly linked list from beginning.
4	Deletion at end	Removing a node in circular doubly linked list at the end.

Traversing and searching in circular doubly linked list is similar to that in the circular singly linked list.

OUTLINE

- Lists
- Implementation
- Examples

OUTLINE

- Lists

- Implementation

- Examples