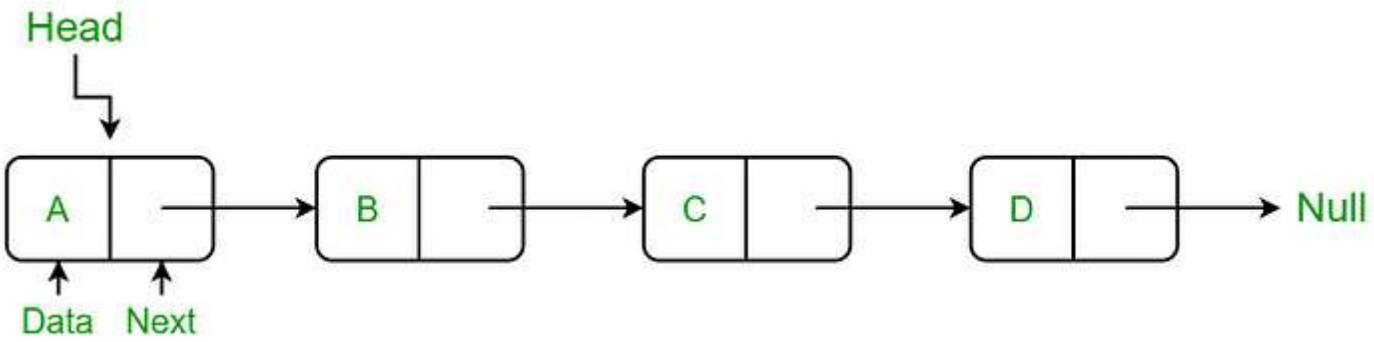
Linked Lists

What is a Linked List?

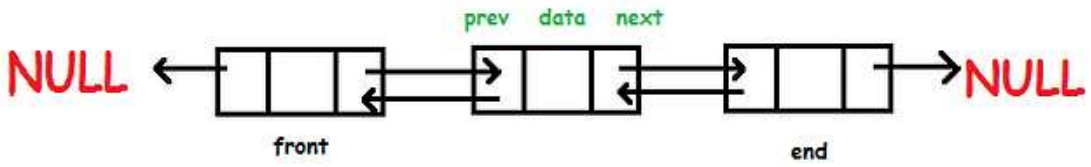
- A **linked list** is a linear data structure where elements (called nodes) are stored in a sequence.
- A linked list is a linear data structure whose elements are not stored in contiguous memory locations like arrays. This can be done because each element stored inside the linked list has a pointer to the next element.
- Unlike arrays, linked lists do not store elements in contiguous memory locations. Instead, each node contains:
  1. **Data**: The value or information stored.
  2. **Pointer/Reference**: A link to the next node in the sequence.
- The first node is called the **head**, and the last node typically points to **null** (indicating the end).



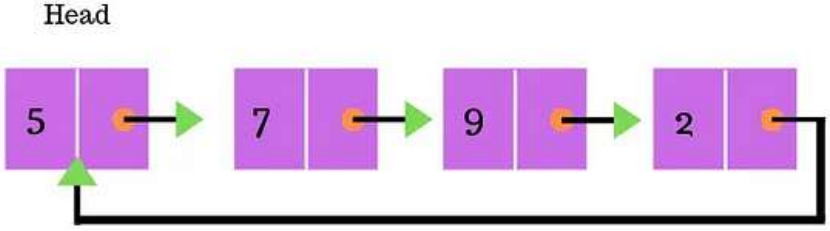
Types Of Linked Lists

- There are 4 different kinds of linked lists:
  - singly linked list
  - doubly linked list
  - circular list
  - doubly circular list.

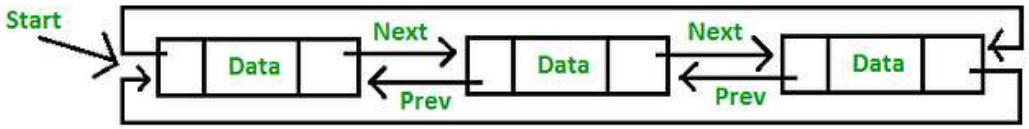
1. **Singly Linked List**
   - A singly linked list is a linked list where each node only points to one node and the tail node points to null. This was represented in the picture above.
2. **Doubly Linked List**
   - A node in a doubly linked list on the other hand has two pointers, one pointing to the next node and the other pointing to the previous node.
   - This allows you to traverse the linked list in both directions.



   - The head in a doubly linked list will point to both null and the next node. The tail in a doubly linked list will point to null and the previous node.
3. **Circular Linked List**
   - The Circular Linked list the next pointer of the last node points back to the first node and this results in forming a loop.
   - This type of linked list is simply a singly linked list where the tail, instead of pointing to null points back to the head node.



4. **Doubly Circular Linked List**
   - A doubly circular linked list is a doubly linked list whose head's previous node will be the tail instead of null. And the tail's next node will be the head instead of null.



Key Operations on Linked Lists

1. **Insertion**:
   - At the beginning: O(1) - Update head pointer.
   - At the end: O(n) for singly linked (traverse to end), O(1) for doubly linked with tail pointer.
   - At a specific position: O(n) - Traverse to the position.
2. **Deletion**:
   - At the beginning: O(1) - Update head pointer.
   - At the end or specific position: O(n) - Traverse to find the node.
   - Doubly linked lists make deletion slightly easier due to previous pointers.
3. **Traversal**:
   - O(n) - Visit each node sequentially.
4. **Search**:
   - O(n) - Linear search required as no random access.

Advantages Of Linked Lists

- An advantage of linked lists is they have dynamic size, meaning they can grow or shrink since memory allocation is done at runtime.
- This is compared to an array where (in some languages) you must declare the array with a predefined size, often larger than necessary to allow elements to be inserted or deleted easily.
- Another advantage is inserting and deleting elements of a linked list is easy. When adding or deleting elements in an array all the elements after the insertion/deletion point need to be shifted.
- This differs from linked lists where only the pointers around the inserted/deleted element need to be updated.
- Linked lists can also add elements indefinitely where arrays will need to resize.
- Lastly, linked lists are very efficient at inserting or deleting elements near the start of the list, something that arrays are not, especially when the list size becomes large.

Disadvantages Of Linked Lists

- Unlike arrays, linked lists do not have access to random elements within the list. In an array you can use the index to get any element you want instantly.
- However, if you want to receive the middle element of a linked list you must start at the head and keep moving deeper into the list until you get to the element you want.
- This makes finding an element in the linked list O(n) vs finding an element in an array (when knowing its index) O(1).
- Another disadvantage is linked lists create unnecessary storage since each node in the list must also store a reference to the next node.

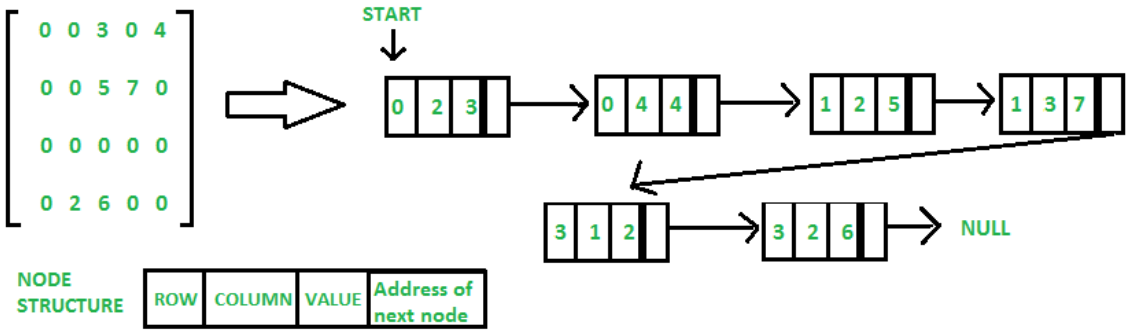
Applications of Linked Lists

- Applications of linked list contain
  1. Sparse Matrix Representation
  2. Polynomial Representation

3. Addition of polynomials
4. Multiplication of Two Polynomials

1) Sparse Matrix Representation

- A matrix is a two-dimensional data object made of M rows and N columns, therefore having total of M*N values.
- If most of the elements of the matrix have 0 value, then it is called a sparse matrix.
- Why to use **Sparse Matrix** instead of simple matrix ?
  i. **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
  ii.
  iii. **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..
- Unlike in standard Linked Lists in the Sparse Matrix Representation using linked list, each node has four fields. These four fields are defined as:
  i. **Row:** Index of row, where non-zero element is located
  ii. **Column:** Index of column, where non-zero element is located
  iii. **Value:** Value of the non zero element located at index – arr[i][j]
  iv. **Next node:** Address of the next node

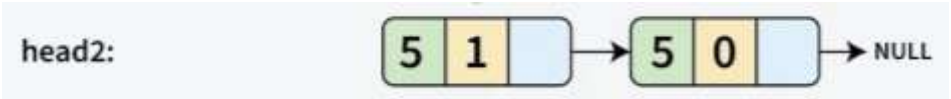


2) Polynomial Representation

- Polynomials (e.g., $3x^2 + 5x + 2$) can be represented using linked lists where each node stores a term's coefficient and exponent.
- For instance, the polynomial $4x^3 + 2x + 1$ would have nodes like [4,3] -> [2,1] -> [1,0], with pointers connecting them.
- This is more efficient than an array (which might store zero coefficients for missing terms) because only non-zero terms are stored.

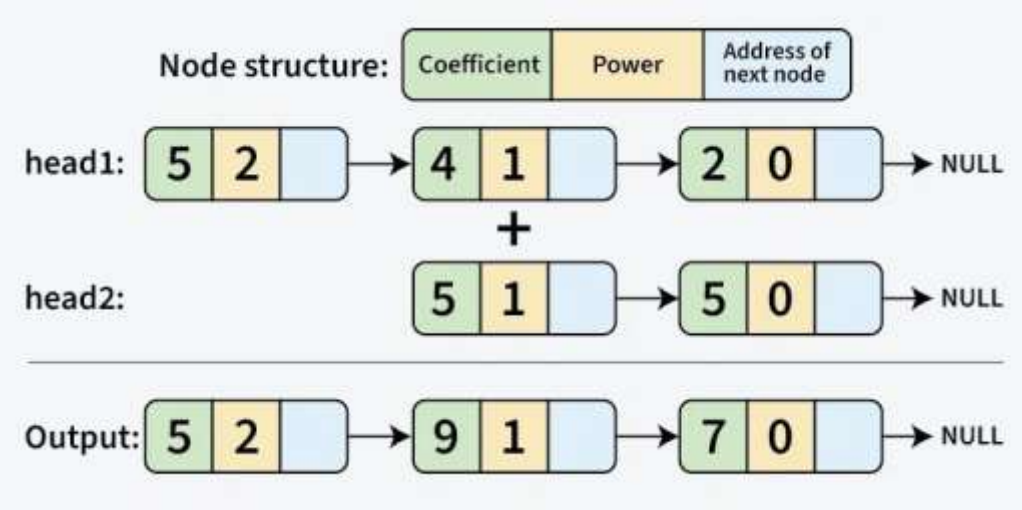
3) Addition of Polynomials

- Linked lists simplify adding two polynomials (e.g., $5x^2 + 4x + 2$ and $5x + 5$).
- Each polynomial is stored as a linked list of terms (coefficient, exponent).



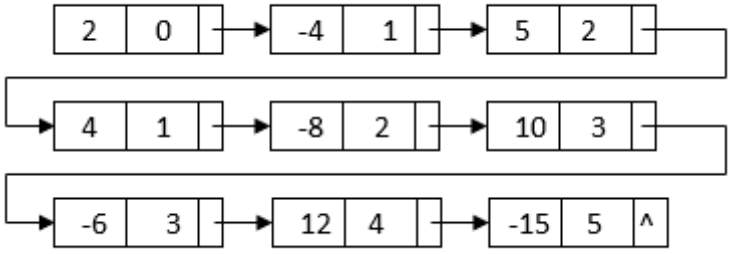
- To add them, traverse both lists simultaneously, comparing exponents: if they match, add the coefficients into a new node in the result list; if not, include the term with the higher exponent and move to the next node.

- Leftover terms from either list are appended to the result. This process is efficient (O(n+m) time, where n and m are the number of terms) and avoids unnecessary zero-term storage.

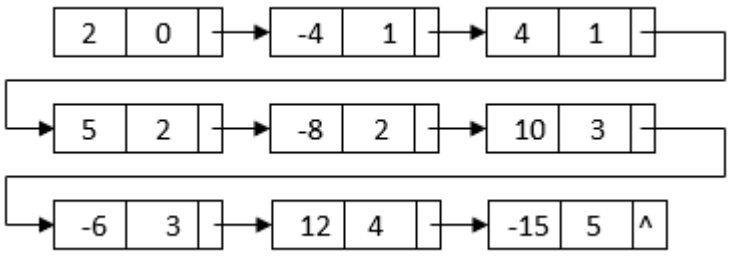


4) Multiplication of Two Polynomials

- Multiplying polynomials (e.g., 2x + 1 and x + 3) using linked lists involves generating all term combinations.
- Suppose the two polynomials have N and M terms. This process will create a polynomial with N*M terms.
- For example, after we multiply $\mathbf{5x^2-4x+2}$ and $\mathbf{-3x^3+2x+1}$, we can get a linked list:



- This linked list contains all terms we need to generate the final result. However, it is not sorted by the powers. Also, it contains duplicate nodes with like terms.
- To generate the final linked list, we can first **merge sort** the linked list based on each node's power.



- After the sorting, the like term nodes are grouped together. Then, we can merge each group of like terms and get the final multiplication result.