UNIVERSITI TEKNOLOGI MALAYSIA

DSPD1733: DATA STRUCTURES AND ALGORITHMS

Pusat Pengajian Diploma, SPACE UTM Kuala Lumpur

Jabatan Sains Komputer dan Perkhidmtan

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

CHAPTER 3:
LIST AND LINKED LIST

- Linear List Concepts
- Linear List Operations: Insertion, Deletion, Retrieval, Traversal,
- Linked Lists Concepts, Head Node and Data Node structure,
- Linked List Algorithms: Create List, Insert Node, Delete Node, Empty List, Traverse List,
- Complex Linked List Structures, Circularly-Linked Lists and Doubly-Linked Lists.

Linear List Concepts

A **linear list** is a fundamental data structure that organizes elements in a sequential manner.

Each element in the list is connected to its predecessor and successor, creating a straightforward, linear relationship.

This organization supports various operations and is crucial for understanding more complex data structures.

Key Characteristics of Linear Lists:

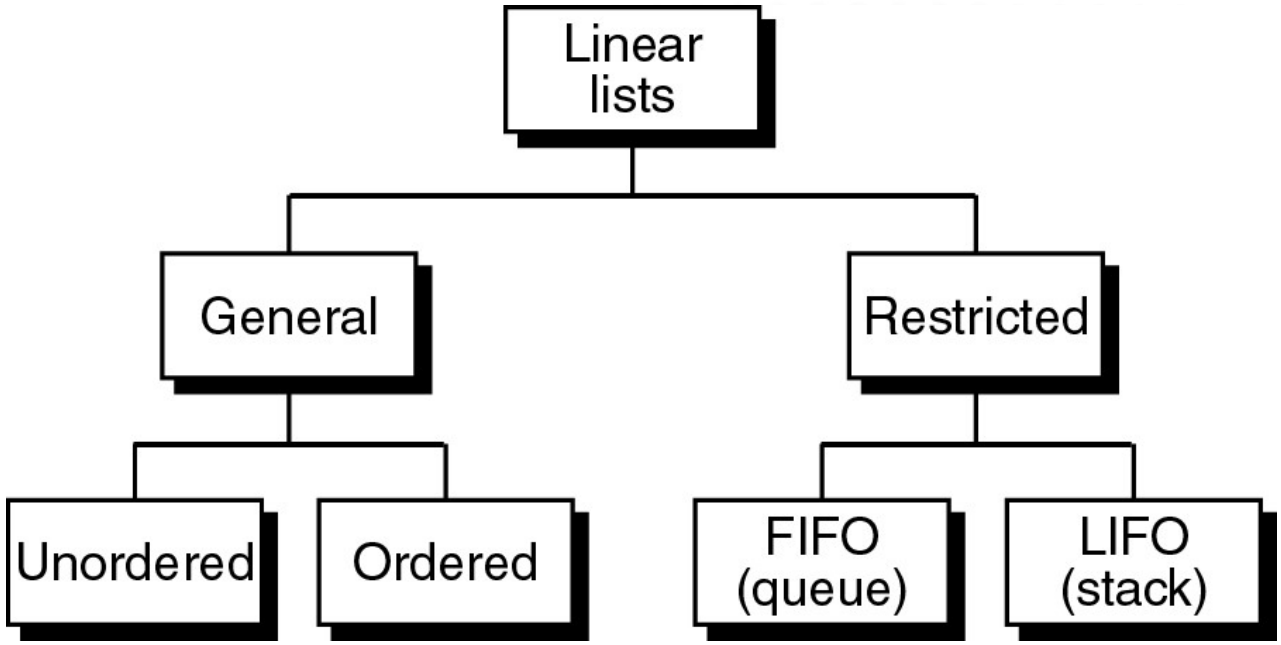
Ordered Sequence: Elements are stored in a specific order, allowing for easy navigation and access based on their position.

Fixed Size or Dynamic: Linear lists can be of fixed size (like arrays) or dynamic (like linked lists), where the size can change during runtime.

Easy Traversal: Linear lists allow for sequential access, meaning you can easily iterate over all elements.

Homogeneous or Heterogeneous: Elements can be of the same type (homogeneous) or different types (heterogeneous), depending on the specific implementation.

Type of Linear List



Linear List and Operations

A **linear list** is a data structure that organizes elements sequentially, allowing for various operations like insertion, deletion, retrieval, and traversal.

Linear List Operations:

Insertion: Adding an element at a specific position in the list.

Deletion: Removing an existing element from the list.

Retrieval: Accessing an element at a particular index.

Traversal: Visiting each element in the list to perform actions like displaying values or modifying them.

Linear List Operation : Insertion

Can be made

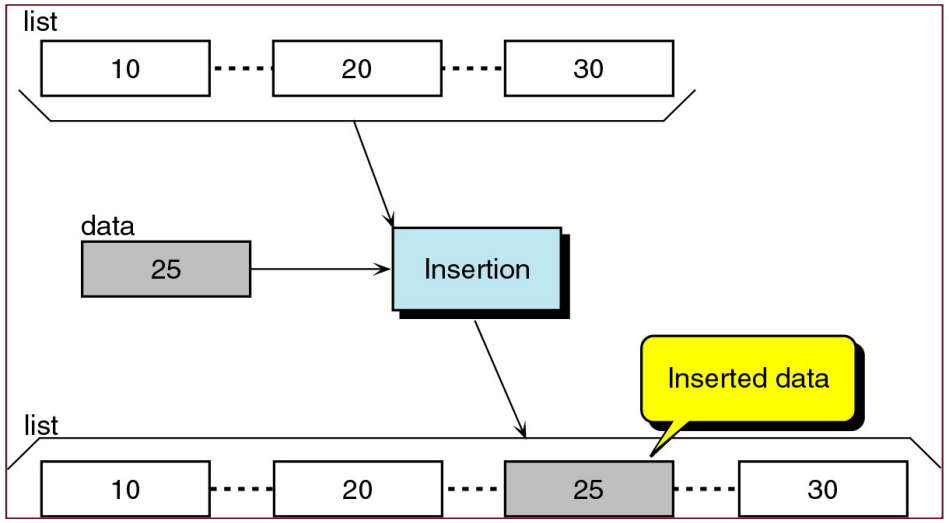
- at the beginning of the list
- in the middle of the list
- at the end of the list

No restriction on inserting data into a random list

Computer algorithm generally insert data at the end of the list

In **Ordered List**, data must be inserted in order to maintain it, use search algorithm.

Illustration of Ordered list insertion:



Linear List Operation : Deletion

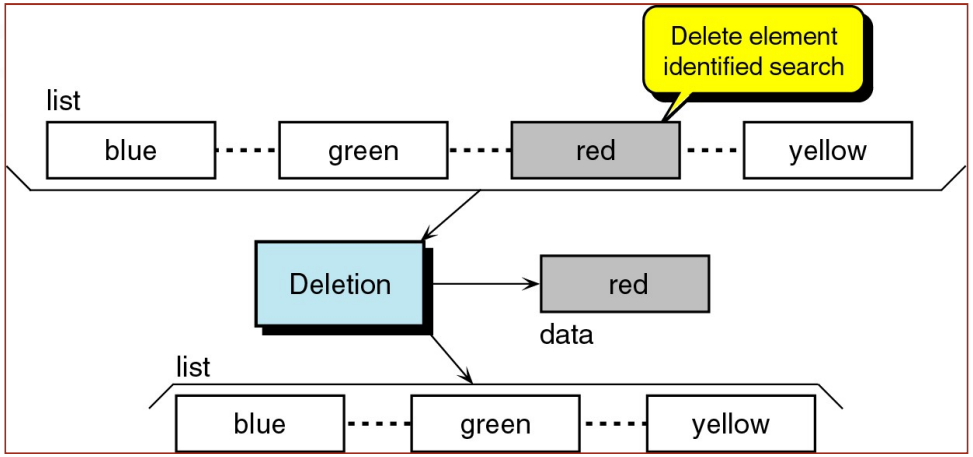
Deletion from a general list required that the list be searched to locate the data being deleted.

Any sequential search algorithm can be used to locate the data.

Once located, the data is removed from the list.

The data following the deleted item must be shifted to replace the empty element.

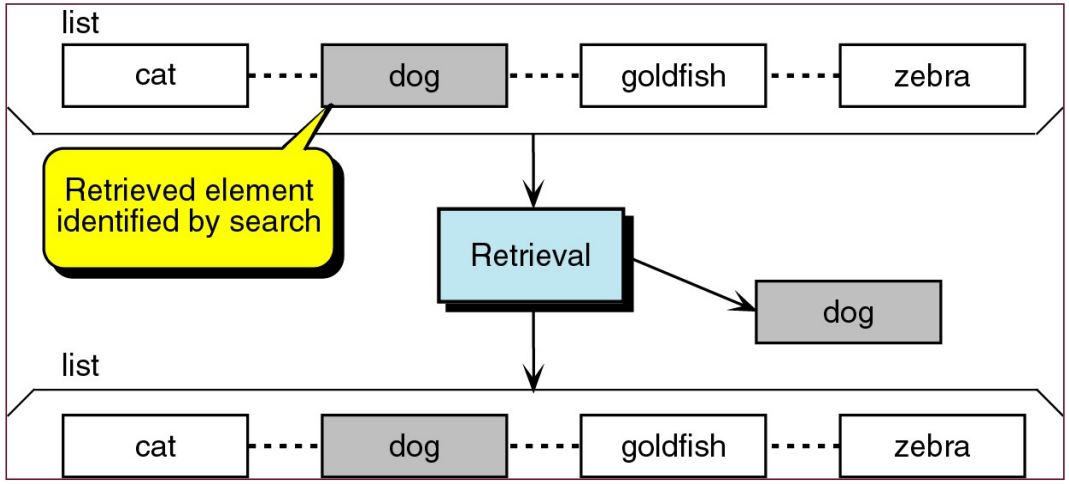
Illustration of ordered list deletion:



Linear List Operation : Retrieval

Required that data to be located in a list and presented to the calling module without changing the contents of the list.

Illustration of linear list retrieval:

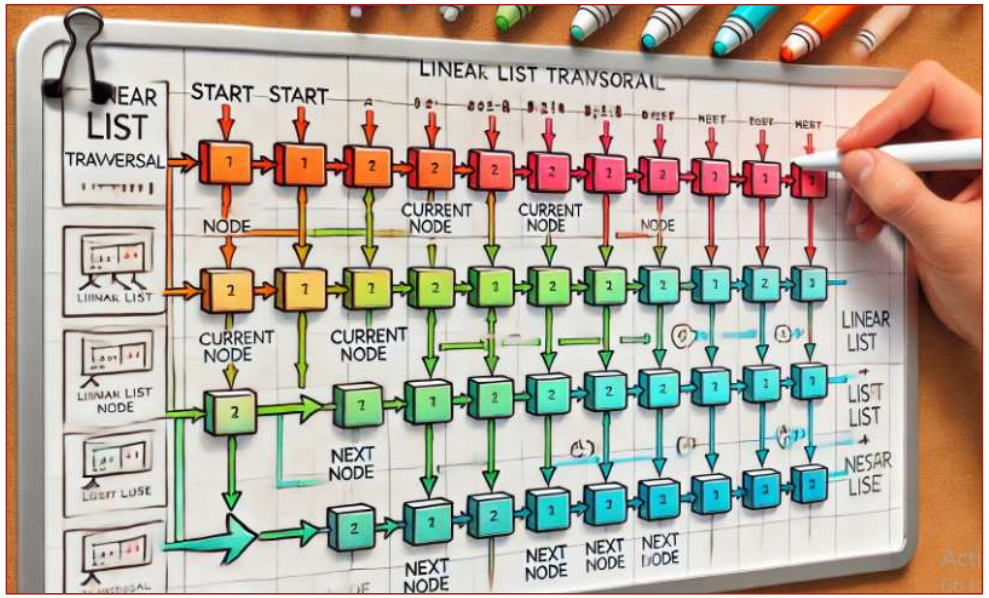


Linear List Operation : Traversal

Special case of retrieval in which all element are retrieved in sequence.

Requires a looping algorithm rather than a search.

illustration demonstrating linear list traversal:



Linear List implement using Array

Definition: An array is a collection of elements identified by index or key, stored in contiguous memory locations.

Characteristics:

Fixed Size: Once an array is created, its size cannot be changed during runtime. You must define the array size at the moment of creation.

Contiguous Memory Allocation: All elements are stored next to each other in memory, allowing for efficient access using indices.

Direct Access: Elements can be accessed in constant time (O(1)) using their index, making retrieval operations very fast.

Advantages:

Simplified syntax for accessing elements (via indices).

Better cache performance due to contiguous memory allocation.

Disadvantages:

Inflexibility in size, if more elements need to be added than the defined size, a new larger array must be created, and the elements copied over (which can be time-consuming).

Insertion and deletion operations can be costly (average time complexity (O(n))) because they require shifting elements.

Use Cases:

Frequently used in scenarios where the number of elements is known upfront and does not change, such as storing fixed collections of items.

Exercise 1: Linear List Using Array

(Marks for this exercise can be used for Assignment (Group) - 4M (OPTIONAL))

Write a program for storing and manage students data.

Problem Statement:

Design a data structure to manage a group of students data (maximum 5 students). Use concept linear list using array. The program should support the following operations:

Add a student that contains the following details: **Name, Matrix Number (unique ID) and Quiz Score**

Remove a student by its Matrix Number.

Search for student by its Matrix Number.

Display a student details information for one student that contain name, matrix number and quiz score.

Display all student that exist in the list by display all details data of each student.

Hints for Data Structure Design:

1.Classes or Objects:

- Create a Student class with attributes name, matrixNumber and quizScore. Include also function to display all the attributes.
- Create a list class to store a group of students and a list of operation that suitable for the linear list operations.

2.Storage:

- Use a **array** to store students.
- Use **list** to manipulate students data.

3.Operations:

- Add methods for adding, removing, searching and display students.

Linked List Concept

An ordered collection of data where element contains the location of the next element

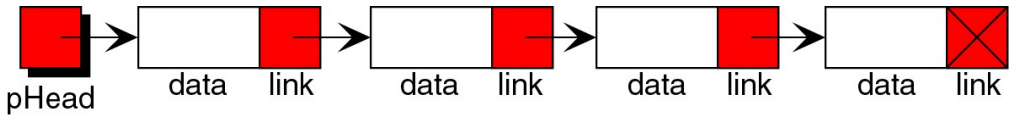
Each element contains two parts:

Data : holds the useful information (to be processed).

links : used to chain the data together.

A pointer variable identifies the first element in the list.

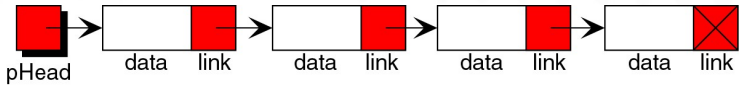
Simple linked list known as **singly linked list** because it contains only one link to single successor.



(a) A linked list with a head pointer: pHead

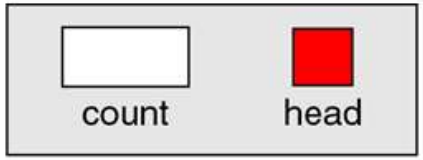
(b) An empty linked list

Linked List Concept (continue)



(a) A linked list with a head pointer: pHead

(b) An empty linked list



(a) Head structure

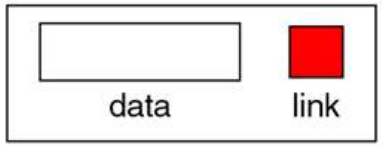
pHead (for pointer to the head of the list) that contain amount of elements connected.

The link in each element except the last points to its successor.

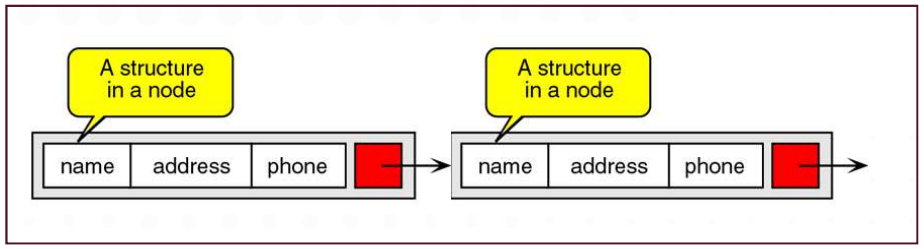
The link in the last element contains a null pointer, indicating the end of the list

A null head pointer is an empty list.

Linked List Node



(b) Data node structure



Elements in a linked list are called **nodes**.

A **node** in a **linked list** is a fundamental building block used to store data in the **linked list data structure**.

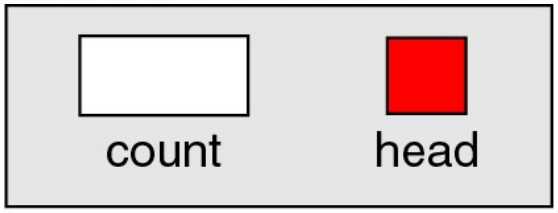
A node is a structure that has at least two main components:

- **Data (Value)** – The actual information or value that the node stores.
- **Pointer (Reference)** – Address of the **next node** in the sequence (or null in the case of the last node).

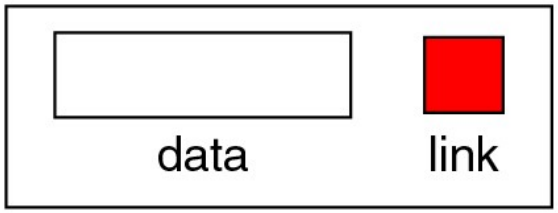
Nodes in a linked list are called **self-referential structures**.

In self-referential structure, each instance of the structure contain a pointer to a another instance of the same structural type.

Linked List Node Structure



(a) Head structure



(b) Data node structure

```
list
   count    <integer>
   head     <pointer>
end list


node
   data     <dataType>
   link     <pointer>
end node
```

When a node contains about a list, the data are known as **metadata**.

Linked List Algorithms

1. **Create List**: Initializes an empty linked list.
2. **Search List**: To find specific node in the list.
3. **Insert Node**: Adds a new node at a specified position—this could be at the beginning, end, or middle of the list.
4. **Delete Node**: Removes a node from a specified position within the list.
5. **Traverse List**: Iterates through the list to access or display data from each node.

Linked List: Create List

Purpose of creating a list in a linked list is to store and manage a **dynamic collection of data elements** in a **sequential manner**.

Unlike arrays, linked lists provide **flexibility** in memory allocation and are particularly useful for scenarios where the size of the data collection can change frequently.

Linked List Algorithms : Create List

Algorithm createList (ref List <metadata>)

Initialize metadata for alinked list.

Pre list is a metadata structure passed by reference

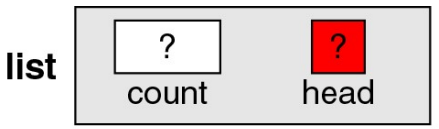
Post metadata initialized

1 list.head = null

2 list.count = 0

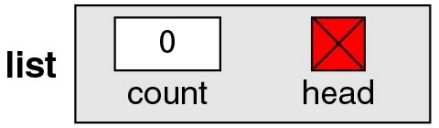
3 return

end createList



(a) Before create

list.head = null
list.count = 0



(b) After create

Linked List: Search List

Used by several algorithms to locate node in a list.

- **To insert node:**

  Need to know the logical predecessor to the new node.

- **To delete data:**

  Need to find the node to be deleted and identify its logical predecessor.

- **To retrieve data:**

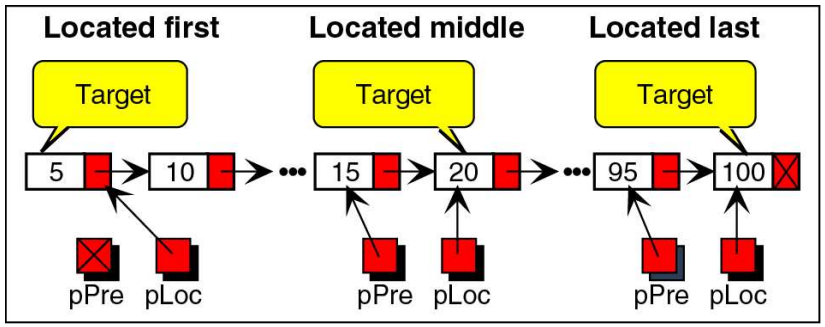
  Need to search the list and find the node.

Node stored in a linked list must use a sequential search because there is no physical relationship among the nodes.

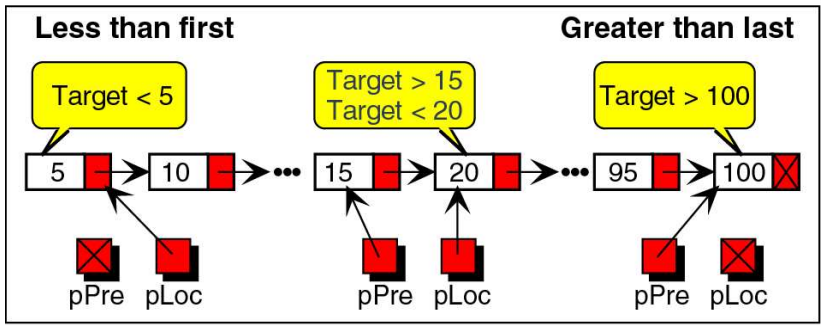
It will returns the location of an element when it is found and the address of the last element when it is not found.

Linked List : Search List (continue)

- Start at the beginning and search the list sequentially until the target value is no longer greater than the current node's key.
- At the end, the target value is either less than or equal to the current node's key while the predecessor is pointing to the node before the current node.



(a) Successful searches (return true)



(b) Unsuccessful searches (return false)

Search List : Algorithm

Algorithm searchList (val list <metadata>,
ref pPre <node pointer>,
ref pLoc <node pointer>,
val target <key type>)

Searches list and passes back address of node containing target and its logical predecessor.

Pre list is metadata structure to a valid list
pPre is pointer variable for predecessor
pLoc is pointer variable for current node
target is the key being search

Post pLoc points to the first node with equal / greeter key -or – null if target > key of last node
pPre points to largest node smaller than key –or- null if target < key of first node

Return true if found, false if not found

```
1    pPre = null
2    pLoc = list.head
3    loop (pLoc not null AND target >
     pLoc->data.key)
       1    pPre = pLoc
       2    pLoc = pLoc->link
4    end loop

     Set return value
1    if (pLoc null)
       1    found = false
2    else
       1    if target equal pLoc->data.key)
              1    Found = true
       2    Else
              1    Found = false
       3    end if
3    end if
4    return found

end searchList
```

Linked List: Insert Node

Add data to a linked list.

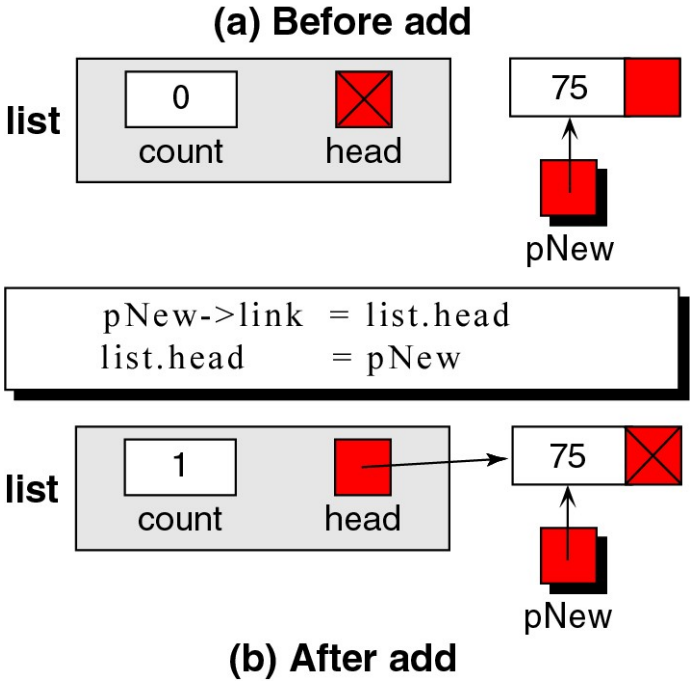
Need only its logical predecessor to insert a node into the list.

There three step to the insertion:

1. Allocate memory for the new node and insert data.
2. Point the memory node to its successor.
3. Point the new node's predecessor to the new node.

Insert Node : Into Empty

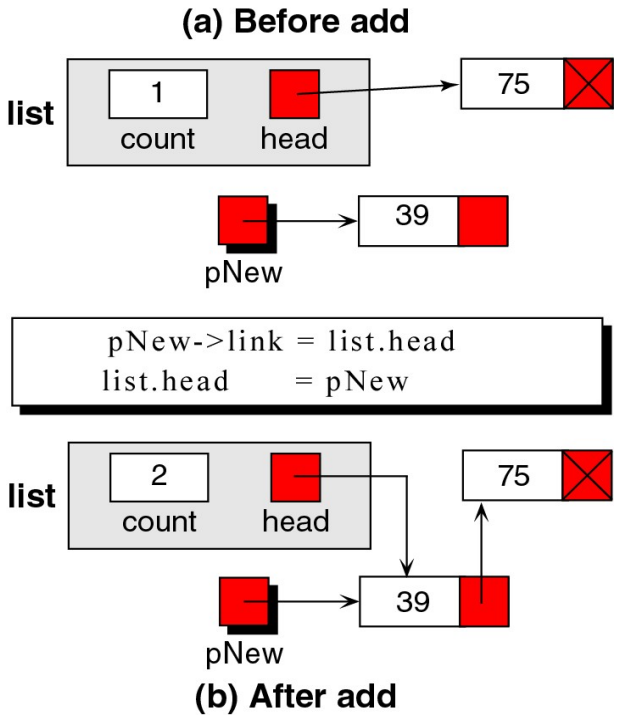
- When the head pointer is null then the list is empty.
- To add a node to an empty list:
  - Assign the list head pointer the address of the new node.
  - Make sure the link field is a null pointer (use the null pointer contained in the list head.




```
pNew->link  =  list.head      Set link to null pointer
list.head   =  pNew           Point list to first node
```

Insert Node : At Beginning

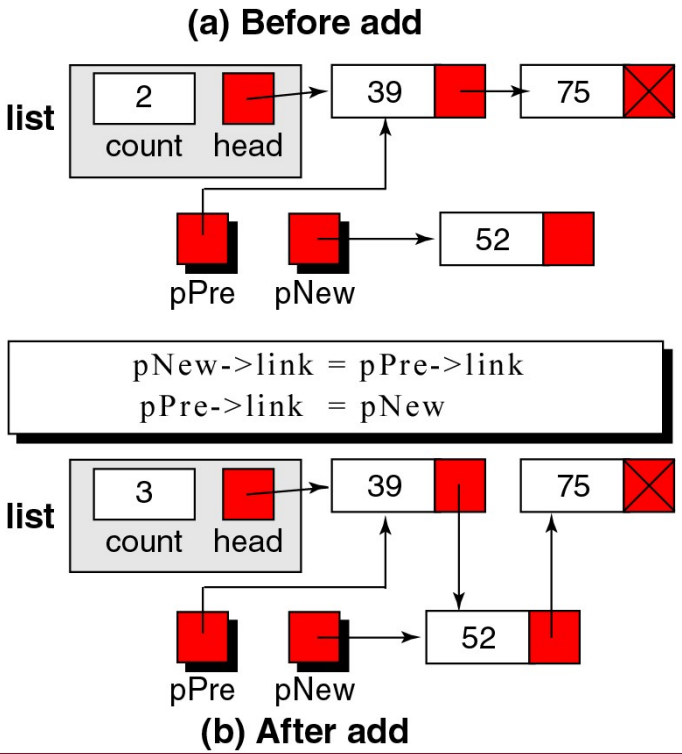
- Need to insert a node before the first node of the list.
- To insert a node at the beginning of the list:
  - Point the new node to the first node of the list. (The first node's address is stored in the head pointer)
  - Set the head pointer to point to the new first node.




```
pNew->link  =  list.head
list.head   =  pNew
```

Insert Node : In Middle

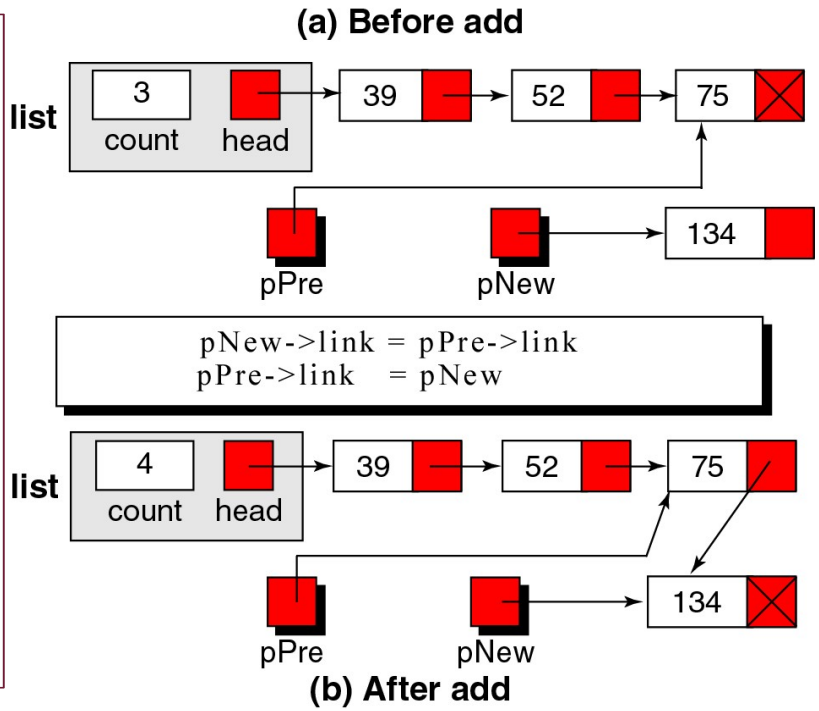
- Adding a node in the middle of the list, the predecessor contains an address.
- To insert a node between two nodes:
  - Point the new node to its successor.(The address of the new node's successor can be found in the predecessor's link field.
  - Point its predecessor to the new node.




```
pNew->link = pPre->link
pPre->link   = pNew
```

Insert Node : At End

- To insert a node at the end of the list:
  - Only need to point the predecessor to the new node.
  - there is no successor to point to.
  - Set the new node's link field to a null pointer.



pNew->link = null pointer

pPre->link = pNew

Linked List Algorithm : Insert Node

```
Algorithm  insertNode ( ref  list      <metadata>,
                        val  pPre     <node pointer>,
                        val  dataIn   <dataTypes>    )
Insert data into a new node in the linked list.
   Pre     list is metadata structure to a valid list
           pPre is pointer to data's logical predecessor
           dataIn contain data to be inserted
   Post    data have been inserted in sequence
   Return  true if succesful, false if memory overflow

1      allocate (pNew)
2      If (memory overflow)
   1       return false
3      end if
4      pNew->data = dataIn
5      If  (pPre null)
       Adding before first node or to empty
       list.
   1       pNew->link  =  list.head
   2       list.head     =  pNew
6      else
       Adding in the middle or at end.
   1       pNew->link = pPre->link
   2       pPre->link   = pNew
7      end  if
8      list.count = list.count + 1
9      return true
end sample
```

Linked List: Delete Node

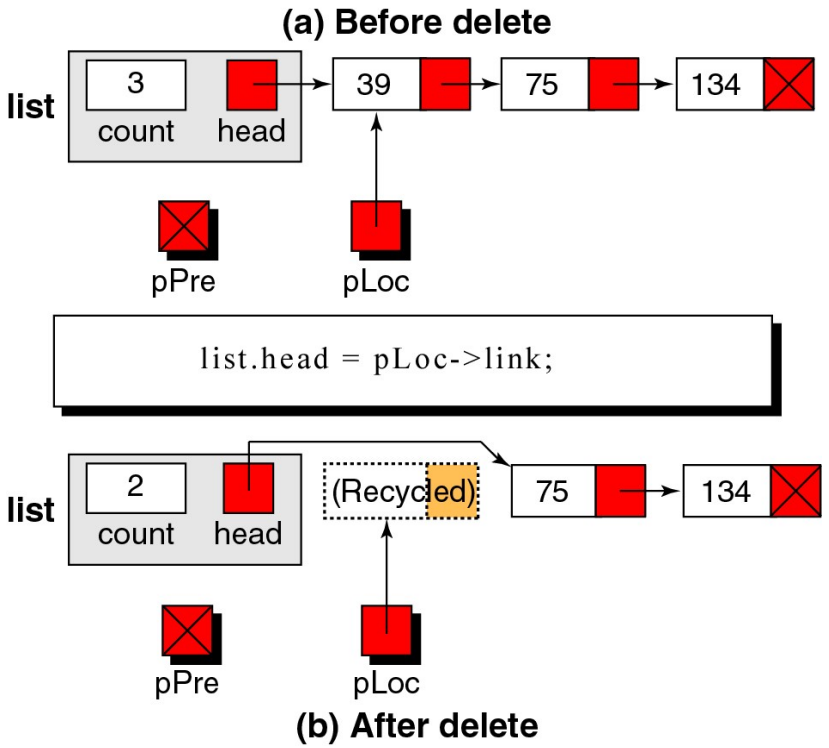
Logically: removes a node from the linked list by changing various link pointers and then physically deleting the node from dynamic memory.

There three step to the deletion:

1. Need to locate the node.
2. Get the predecessor address.
3. Change the its predecessor link field to point to the deleted node's successor.

Delete Node : Delete First Node

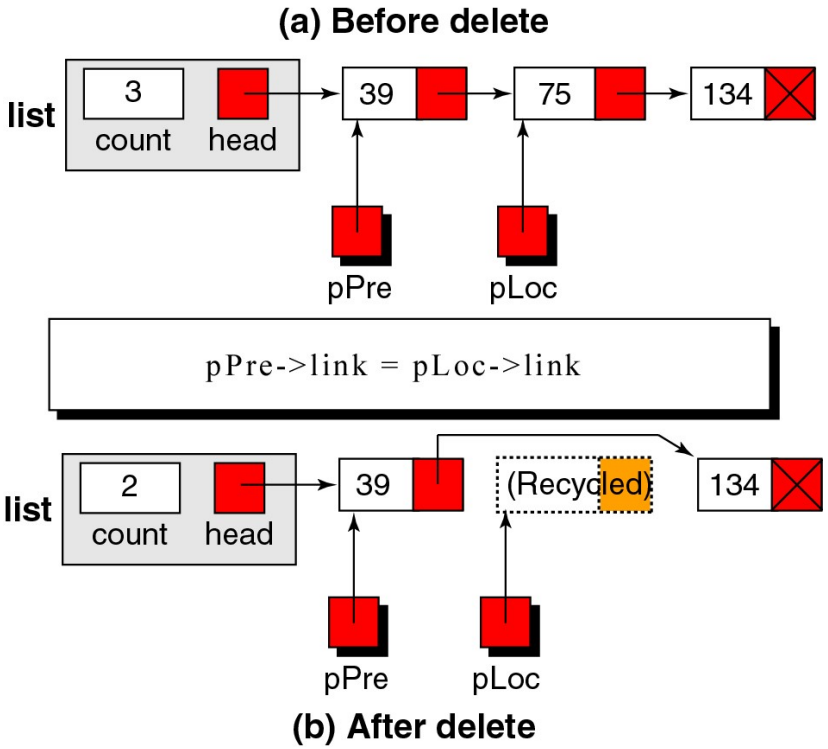
- When delete the first node, must reset the head pointer to point to the first node successor.
- If the predecessor is a null pointer, the first node is being deleted.




```
head  =  pLoc->link
recycle (pLoc)
```

Delete Node : General Delete Case

- Point the predecessor node to the successor of the node being deleted.




```
pPre->link  =  pLoc->link
recycle (pLoc)
```

Delete Node: Algorithms

Algorithm deleteNode(ref list <metadata>,
val pPre <node pointer>,
val pLoc <node pointer>,
ref dataOut <dataType>)

Delete data from a linked list and returns it ti calling module.

Pre list is metadata structure to a valid list
pPre is pointer to predecessor node
pLoc is pointer to node to be deleted
dataOut is variable to received deleted data

Post data have been deleted and returned to caller

```
1    dataOut = pLoc-> data
2    if (pPre null)

     Deleteing first node
   1    List.head =  pLoc->link
3       else

     Deleting other nodes
   1    pPre->link  = pLoc->link

4    end if
5    list.count = list.count -1
6    Recycle (pLoc)
7    return

end deleteNode
```

LABSKILL 1: Linked List (5M) - individually

Write a program for reusable code that can storing and manage a collection of data.

Problem Statement:

Design a data structure that can be reuse to create a linked list for storing a collect of value type char. The main program is to read a sentence of words. There will be three lists which hold a collection of vowel character and consonant character and special symbol extract from the sentence. The output shown as below:

Enter sentence: **Hello! Hey you, welcome to my world…**

Vowel List: e o u

Consonant List: H l y w c m t r d

Symbol List: ! , .

Hints for Data Structure Design:

1.Header Files:

- Define a DATA type that implement like char data type. Use typedef and define it in the header file (DATA.h)
- Define Node structure to be use with linked list structure. Use struct and define it in the header file (NODE.h)
- Create a list(linked list) class that have suitable attributes and operations to implement linked list which need to be define it in the header file (LIST.h)

2.Program File:

- Create main that can produce program such as the output shown.
- Must make use all the structure define in the header files.

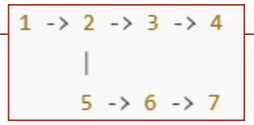
Complex Linked List Structure

A **complex linked list** is an advanced variation of a traditional linked list where each **node** contains **multiple pointers or references**, enabling it to represent more intricate relationships between data elements.

These structures are useful in scenarios requiring multi-level or cross-referencing relationships, such as graphs, trees, and certain cache implementations.

Types of Complex Linked Lists

-Multi-Level Linked List



Nodes contain a **pointer to the next node** and an additional pointer to a **sublist** (child list).

Used for representing **hierarchical structures** like **file systems** and **organizational charts**.

-Doubly Linked List with Random Pointers

Each node has:

- A **next pointer** to the next node.
- A **prev pointer** to the previous node.
- An **extra pointer (random)**, which can point to **any node** (even non-adjacent ones).

Used for **cache systems** like **LRU Cache** or **graph implementations**.

-Circular Linked List with Bidirectional Links

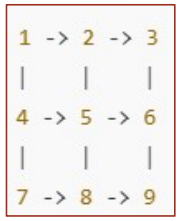
A **circular linked list** where the **last node points to the first node**, forming a **loop**.

Used in **buffered systems**, **round-robin scheduling**, and **multiplayer games**.

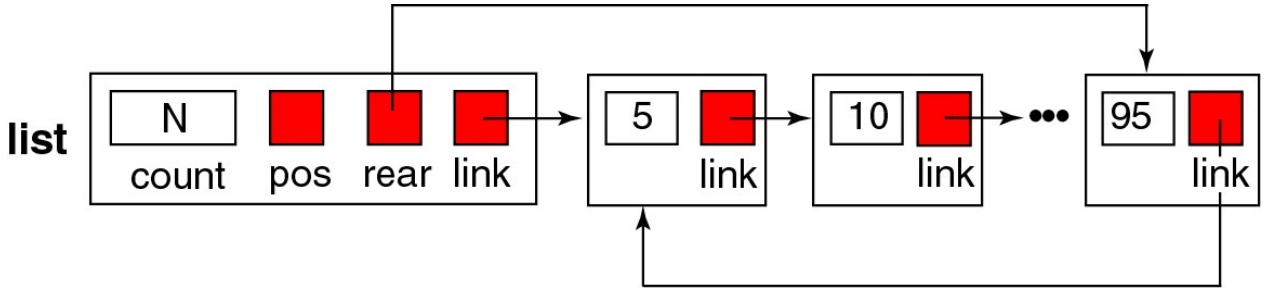
-Multidimensional Linked List

Nodes point to **multiple directions** (right, down, diagonal, etc.), forming a **grid-like structure**.

Useful for **2D matrix representation** and **pathfinding algorithms** in AI.

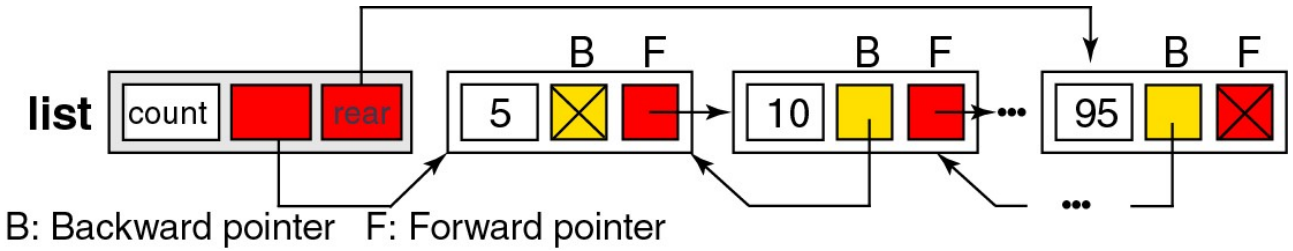


Circularly Linked List



- Last node's link point to the first node of the list.
- Insertion and deletion follow the same logic used in singly linked list.
- When inserting and deleting the last node, updating the rear pointer in the header, must point the link field to the first node.
- In Searching, it can directly access a node in the middle of the list.
  - In singly linked list: stopped when its hit the end of the list or when the target was less than the current node data.
  - In circular list: saving the starting node's address and stop when it have circled around it

Doubly Linked List



- Linked list structure that each node has a pointer to both : its successor and its predecessor.

•**Sequential Access**: Accessing an element requires traversal from the head of the list, resulting in time complexity of (O

Linear List implement using Linked List

Definition: A linked list is a collection of nodes, where each node contains data and a reference (or pointer) to the next node in the sequence. This structure allows for dynamic memory allocation.

Key Purposes of Linked Lists:

1. Dynamic Memory Allocation
2. Efficient Insertion and Deletion (without need to shifting elements)
3. Implementation of Abstract Data Types (ADTs)
4. Handling Large Data Sets
5. Flexibility in Memory Usage
6. Circular Navigation
7. Graph Representation
8. Polynomial Arithmetic
9. Undo and Redo Operations

Advantages:

•**Dynamic Size**: Can grow or shrink as needed during runtime.

•**Efficient Insertions/Deletions**: Operations can be done with constant time complexity (O(1)) if the position is known (e.g., at the beginning or end).

Disadvantages:

•**Memory Overhead**: Each node requires extra memory for pointers, which can be significant for large datasets.

•**Sequential Access**: Accessing an element requires traversal from the head of the list, resulting in time complexity of (O(n)).

Use Cases:

Useful in applications where frequent insertions and deletions occur, such as implementing stacks, queues, and adjacency lists in graphs.

CONCLUSION

- Understanding the types of linear lists arrays and linked lists is fundamental for effective data management in programming.
- Each type has its strengths and weaknesses, making it essential to choose the right one depending on the specific requirements of the application.
- Arrays offer fast access for fixed datasets, while linked lists provide flexibility for dynamic datasets.
- Their straightforward operations and structure pave the way for more complex data handling in algorithms.
- Understanding linear lists is crucial for building a solid foundation in computer science and software development.