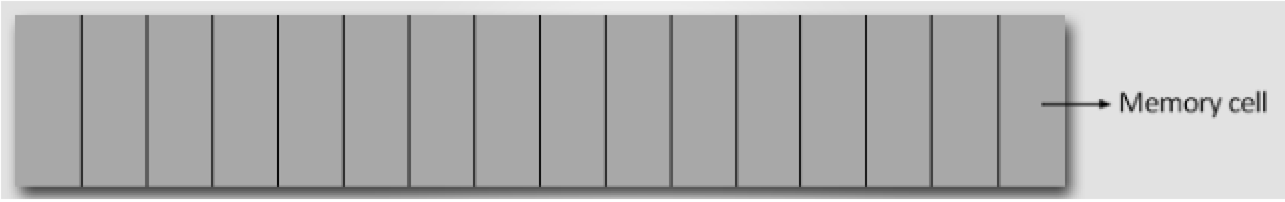
Data Structures CS-2001
Week # 4

Memory representation



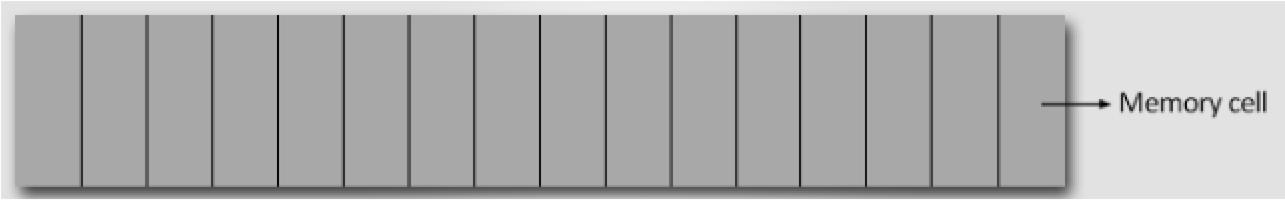
Introduction:

- A data structure is a logical representation of data and operation that can be performed on the data.
    1. Linear data structure
    2. Non linear data structure
- Linear data structure is an order of data elements. They are **arrays, stacks, queues,** and **linked lists**.

Linked list

- Linked list is a linear data structure. It contains nodes. Each node contains two parts,
- i.e. **DATA** part and **LINK** part.
  - The **data** contains **elements** and
  - **Link** contains **address of another node**.

Memory representation



Limitations Of Arrays

- Arrays are simple to understand and elements of an array are easily accessible
- But arrays have some limitations.
  - Arrays have a fixed dimension.
  - Once the size of an array is decided it can not be increased or decreased during execution.
  - Array elements are always stored in contiguous memory locations.
  - Operations like insertion or deletion of the array are pretty tedious.
- To over come this limitations we use linked list.

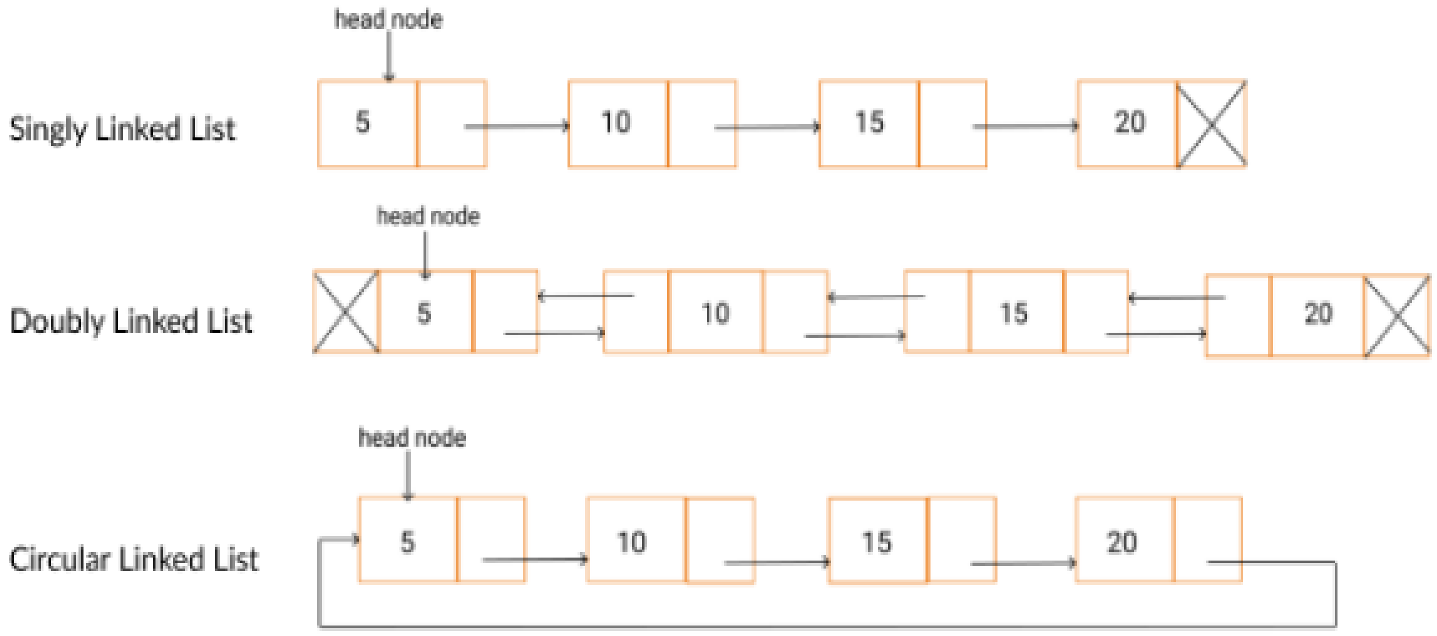
Array Vs. Linked List

Array	Linked List
Fixed size: Resizing is expensive	Fixed size: Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

Types Of Linked Lists

1. Single linked list
2. Double linked list
3. Circular linked list
4. Circular double linked list

Types Of Linked Lists



Single linked list

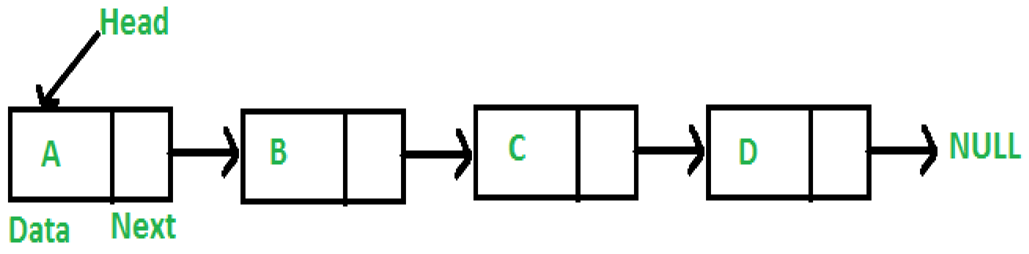
- A single linked list is one in which all nodes are linked together in some sequential manner.
- **Operations on Single Linked List**

  The following operations are performed on a Single Linked List

  - Insertion
  - Deletion
  - Display

Representation Of Linked List

- A linked list is represented by a pointer to the first node of the linked list.
- The first node is called the head. If the linked list is empty, then the value of the head is NULL.
- Each node in a list consists of at least two parts:
    1. Data
    2. Pointer (Or Reference) to the next noc



Representation Of Linked List

- In C, we can represent a node using structures. Below is an example of a linked list node with integer data.

```
struct Node {
    int data;
    struct Node* next;
};
```

- In Java or C# or C++, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

```
class Node {
public:
    int data;
    Node* next;
};
```

First Simple Linked List in C++

- Let us create a simple linked list with 3 nodes.

```
#include <iostream>
using namespace std;
class Node {
public:
    int data;
    Node* next;
};
// Program to create a simple linked
// list with 3 nodes
int main()
{
    // allocate 3 nodes in the heap
    Node head = new Node();
    Node second = new Node();
    Node third = new Node();

    head->data = 1; // assign data in first node
    head->next = second; // Link first node with
    // the second node
    second->data = 2;// assign data to second node
    second->next = third;// Link second node with the third node

    third->data = 3; // assign data to third node
    third->next = NULL;

    /*Note that only the head is sufficient to represent
    the whole list. We can traverse the complete
    list by following the next pointers. */
    return 0;
}
```

Linked List Traversal

- We can use the following steps to display the elements of a single linked list...
- **Step 1** - Check whether list is **Empty (head == NULL)**
- **Step 2** - If it is Empty then, display **'List is Empty!!!'** and terminate the function.
- **Step 3** - If it is **Not Empty** then, define a Node pointer 'temp' and initialize with head.
- **Step 4** - Keep displaying temp → data with an arrow (--->) until temp reaches to the last node
- **Step 5** - Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Linked List Traversal

```cpp
void printList(Node* n)
{
    while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
    }
}
```

```cpp
#include <iostream>
using namespace std;
class Node {
public:
    int data;
    Node* next;
};
void printList(Node* n)
    {
    while (n != NULL) {
        cout << n->data << " ";
        n = n->next;
        }
    }
// Program to create a simple linked
// list with 3 nodes
int main()
{
    // allocate 3 nodes in the heap
    Node    *head = new Node();
    Node    *second = new Node();
    Node    *third = new Node();

    head->data = 1; // assign data in first node
    head->next = second; // Link first node with
    // the second node
    second->data = 2;   // assign data to second node
    second->next = third; // Link second node with the third node
    third->data = 3; // assign data to third node
    third->next = NULL;
    /*
    Note that only the head is sufficient to represent
    the whole list. We can traverse the complete
    list by following the next pointers. */
    printList(head);
    return 0;
}
```

Find Length of a Linked List(**Iterative)**

- **Step 1** - Initialize count as 0
- **Step 2** - Initialize a node pointer, current = head.
- **Step 3** - Do following while current is not NULL

  a) current = current -> next

  b) count++;
- **Step 4** - Return count

```
int getCount(Node* head)
{
    int count = 0; // Initialize count
    Node* current = head; // Initialize current
    while (current != NULL)
    {
        count++;
        current = current->next;
    }
    return count;
}
```

Find Length of a Linked List(**Recursion)**

- **Step 1** - int getCount(head)
- **Step 2** - If head is NULL, return 0.
- **Step 3** - Else return 1 + getCount(head->next)

```
int getCount_r(Node* head)
{
    // Base case
    if (head == NULL)
        return 0;

    // count is 1 + count of remaining list
    return 1 + getCount_r(head->next);
}
```

Search an element in a Linked List **(Iterative)**

- **Step 1** - Initialize a node pointer, current = head.
- **Step 2** - Do following while current is not NULL

  a) current->key is equal to the key being searched return true.

  b) current = current->next
- **Step 3** - Return false

```
/* Checks whether the value x is present in linked list */
bool search(Node* head, int x)
{
    Node* current = head; // Initialize current
    while (current != NULL)
    {
        if (current->key == x)
            return true;
        current = current->next;
    }
    return false;
}
```

Search an element in a Linked List **(Recursive)**

- **Step 1** - If head is NULL, return false.
- **Step 2** - If head's key is same as x, return true;
- **Step 3** - Else return search(head->next, x)

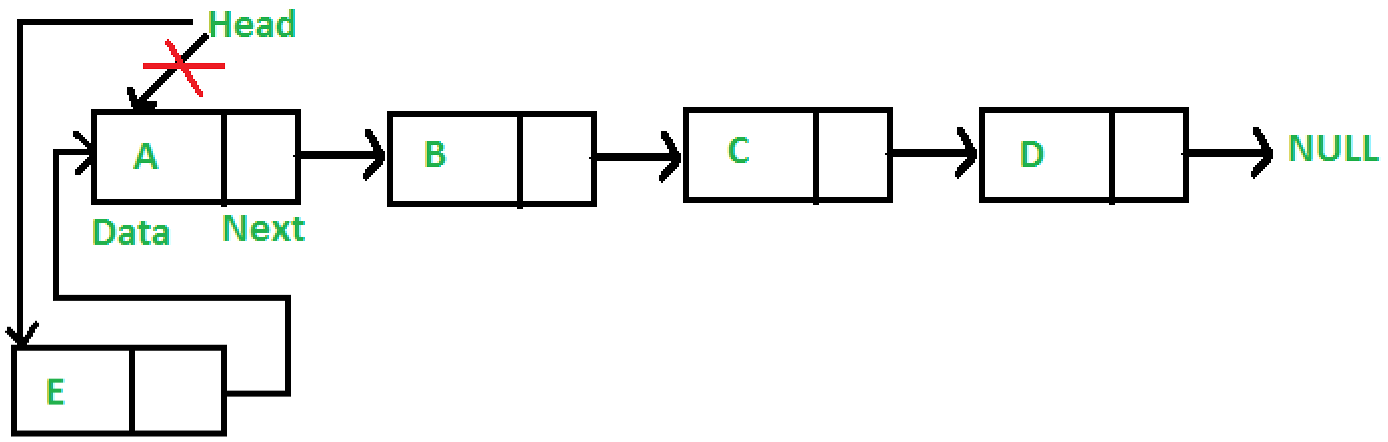
```
bool search(Node* head, int x)
{
    // Base case
    if (head == NULL)
        return false;

    // If key is present in current node, return true
    if (head->key == x)
        return true;

    // Recur for remaining list
    return search(head->next, x);
}
```

Linked List (Inserting a node)

- A node can be added in three ways
    1. At the front of the linked list
    2. After a given node.
    3. At the end of the linked list.

Add a node at the front:

- We can use the following steps to insert a new node at beginning of the single linked list...
  - **Step 1** - Create a **newNode** with given value.
  - **Step 2** - Check whether list is **Empty (head == NULL)**
  - **Step 3** - If it is Empty then, set **newNode→next** = NULL and **head = newNode**.
  - **Step 4** - If it is **Not Empty** then, set **newNode→next = head** and **head = newNode**.



Add a node at the front:

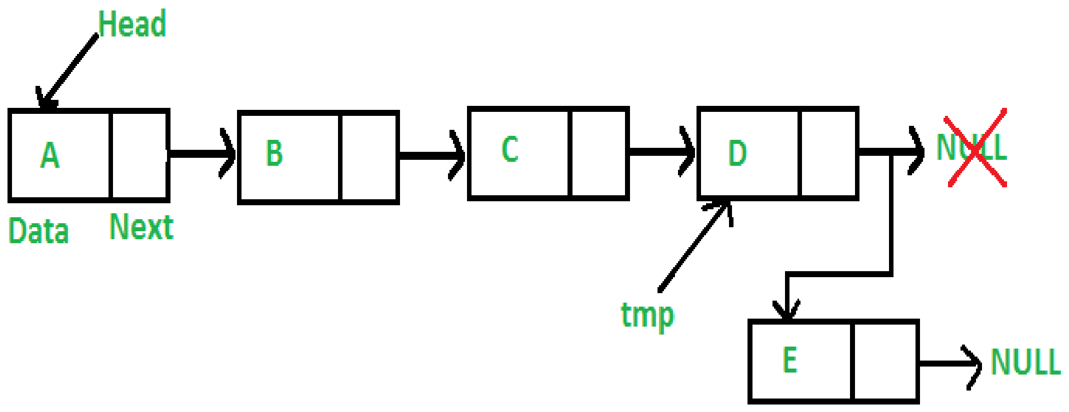
```
/* Given a reference (pointer to pointer)
to the head of a list and an int, inserts
a new node on the front of the list. */
void push(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref) = new_node;
}
```

Add a node at the end

- We can use the following steps to insert a new node at end of the single linked list...
  - **Step 1** - Create a **newNode** with given value and **newNode → next** as **NULL**.
  - **Step 2** - Check whether list is Empty **(head == NULL)**.
  - **Step 3** - If it is **Empty** then, set **head = newNode**.
  - **Step 4** - If it is **Not Empty** then, define a node pointer temp and initialize with head.
  - **Step 5** - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
  - **Step 6** - Set temp → next = newNode.

Add a node at the end

```
/* Given a reference (pointer to pointer) to the head
of a list and an int, appends a new node at the end */
void append(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

    Node *last = *head_ref; /* used in step 5*/

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. This new node is going to be
    the last node, so make next of
    it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty,
    then make the new node as head */
    if (*head_ref == NULL)
    {
        *head_ref = new_node;
        return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
    return;
}
```



Add a node after a given node:

- We can use the following steps to insert a new node after a node in the single linked list...
  - **Step 1** - Create a **newNode** with given value.
  - **Step 2** - Check whether list is **Empty (head == NULL)**
  - **Step 3** - If it is **Empty** then, set **newNode → next = NULL** and **head = newNode**.
  - **Step 4** - If it is **Not Empty** then, define a node pointer temp and initialize with head.
  - **Step 5** - Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).
  - **Step 6** - Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.
  - **Step 7** - Finally, Set **'newNode → next = temp → next'** and **'temp → next = newNode'**

Add a node after a given node:

```cpp
void insertAfter(Node* prevNode, int new_data)
{
    /* Input validation */
    if (prevNode == NULL) {
        cout<<"Error : Invalid node pointer !!!\n";
        return;
    }
    /* creates a new node */
    Node* newNode = new Node();
    newNode->data  = new_data;
    /* Set Next pointer of newNode to next pointer of nodePtr */
    newNode->next = prevNode->next;
    /* Set next pointer of prevNode to newNode */
    prevNode->next = newNode;
}
```