Chapter 5
Lists-PartI

Today

- Introduction to list
  - What is it?
  - Common operations
- List implementation
  - Array
  - Single linked list

The List ADT

- List is an ordered collection of items of some element type E.
  - Note that ordered doesn't mean that the objects are in ***sorted*** order, it just means that each object has a ***position*** in the List

$$E_1, E_2, E_3, \dots E_N$$

  - N: length of the list
  - E_1: first element
  - E_N: last element
  - E_i: element at position i
  - If N=0, then the list is empty
  - Linearly ordered
    - E_i precedes E_{i+1}
    - E_i follows E_{i-1}
- Lists are a basic example of containers, as they contain other values.
- If the same value occurs multiple times, each occurrence is considered a distinct item.

Common operations of the List ADT

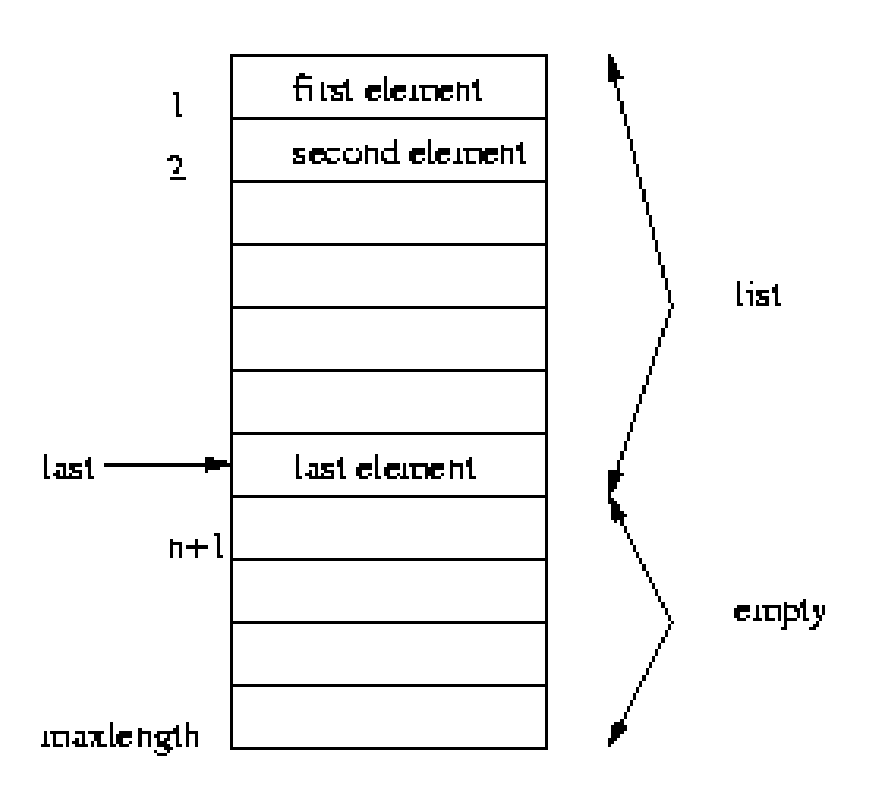
- **printList:** print the list
- **size**: return the number of items in the List
- **find**: locate the position of an object in a list
  - list: 34,12, 52, 16, 12
  - find(52) → 3
- **insert**: insert an object to a list
  - insert(x,4) → 34, 12, 52, x, 16, 12
- **remove**: delete an element from the list
  - remove(52) → 34, 12, x, 16, 12
- **findKth**: retrieve the element at a certain position K

Implementation of List ADT

- the list ADT can be represented using different data structures
  - E.g. arrays, LinkedList, double linked list, etc.
  - Each operation associated with the ADT is implemented by one or more subroutines(functions or methods)

Array Implementation List ADT

- Elements are stored in contiguous array positions



Array Implementation of List ADT

- Need to know the maximum number of elements in the list at the start of the program
  - Difficult
  - Wastes space if the guess is bad
- Adding/Deleting an element can take O(n) operations if the list has n elements.
  - As it requires shifting of elements
- Accessing/changing an element anywhere takes O(1) operations independent of n
  - Random access

Adding an element

- Normally first position (A[o]) stores the current size of the list
  - But you can choose another way to store size e.g on separate variable
- Actual number of elements currsize+ 1
- Adding at the beginning:
  - Move all elements one position up/behind
  - Add at position 1;
  - Increment the current size by 1

```
For (j = A[0]+1; j > 1; j--)
    A[j] = A[j-1];
A[1] = new element;
A[0]=A[0]+1;
```

Complexity: O(n)

Adding at the End

- Add the element at the end
- Increment current size by 1;

```
A[A[0]+1] = new element;
A[0]=A[0]+1;
```

- Complexity: O(1)

Adding at k^{th} position

- Basic Steps
  - Move all elements one position behind, k^{th} position onwards;
  - Add the element at the k^{th} position
  - Increment current size by 1;

```
For (j = A[0]+1; j > k; j--)
  A[j] = A[j-1];
A[k] = new element;
A[0]=A[0]+1;
```

- Complexity: O(n-k)

Deleting an Element at the Beginning

- Basic steps:
  - Move all elements one position ahead;
  - Decrement the current size by 1

```
    For (j = 1; j < A[0] ; j++)
     A[j] = A[j+1];
    A[0]=A[0]-1;
```

- Complexity: O(n)

Deleting an element at the End

- Basic steps:
  - Decrement current size by 1;

```
A[0]=A[0]-1;
```

- **Complexity**: O(1)

Deleting at the kth position

- Basic Steps
  - Move all elements down one position ahead, k+1th position onwards;
  - Decrement the current size by 1;

```
For (j = k; j < A[0]; j++)
    A[j] = A[j+1];
A[0]=A[0]-1;
```

- **Complexity**: O(n-k)

Accessing an Element at the k^{th} position

- Basic steps:
  - Retrieve the element at index position K

```
A[k];
```

- Changing element value at position k

```
A[k]= new value
```

- **Complexity** of both of the above operations : O(1) operation;

Linked list

Review on Structures

- Structures are aggregate data types have data members possibly from multiple different data types.
- Structure is defined using the **struct** keyword:

E.g.
```
struct Time {
  int hour;
  int minute;
  int second;
  };
```

- Structure variables are declared like variables of other types.
  - Syntax: <structure name> <variable name>;
  - Example:

```
Time timeObject;
Time *timeptr; //pointer of type Time;
```

Accessing Members of Structure Variables

- The Dot operator (.):to access data members of structure variables.
- The Arrow operator (->): to access data members of pointer variables pointing to a structure.
- E.g. Print member hour of time Object and timeptr.

```
cout<<timeObject.hour; or
cout<<timeptr->hour;
```

- Note :timeptr->hour is the same as (*timeptr) . hour
- The parentheses is required since (*) has lower precedence than (.)

Self-Referential Structures

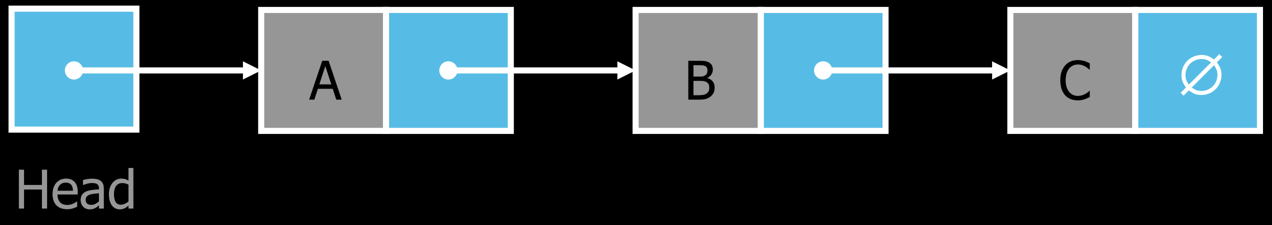
- Structures can hold pointers to instances of themselves.

```
struct List {
char name[10];
int count;
List *next; // this member makes it self referential
};
```

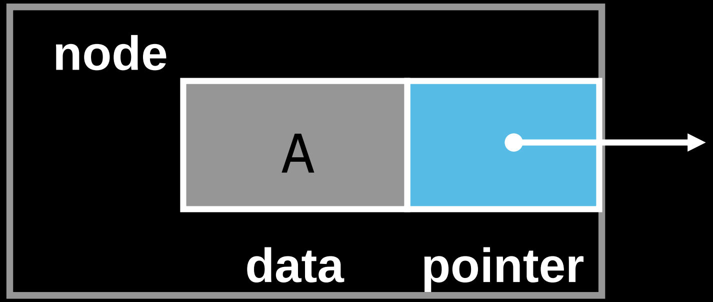
- However, structures cannot contain instances of themselves.

Linked list

Linked Lists implementation of List ADT



- A ***linked list*** is a series of connected ***nodes***
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- ***Head***: pointer to the first node
- The last node points to `NULL`



Where are linked lists used?

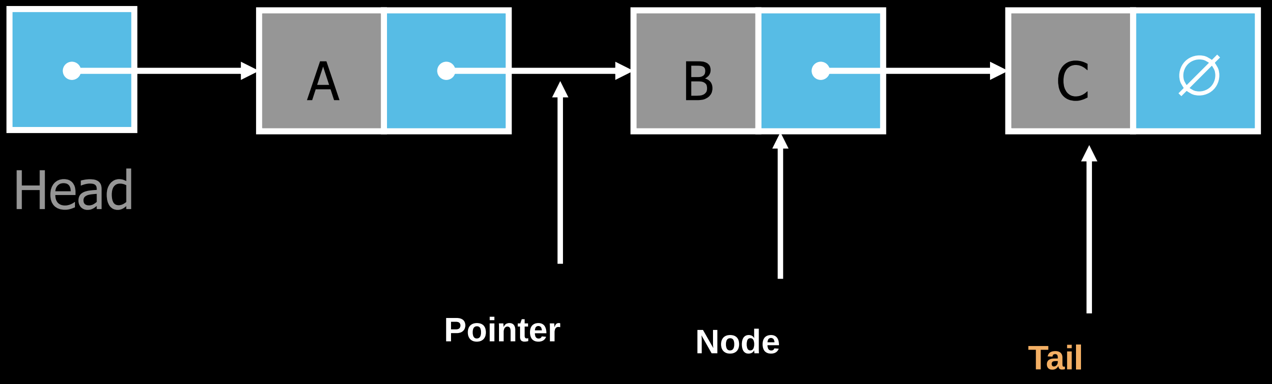
- Used in List, Queue & Stack implementations.
- Great for creating circular lists.
- Can easily model real world objects such as trains.
- Often used in the implementation of adjacency lists for graphs data structure.

Terminology

Head: The first node in a linked list

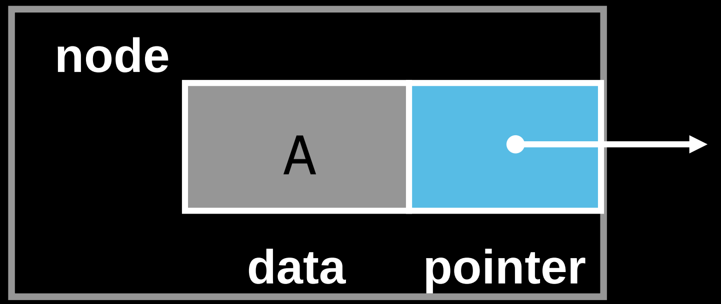
Tail: The last node in a linked list

Pointer: Reference to another node



Node: An object containing data and pointer(s)

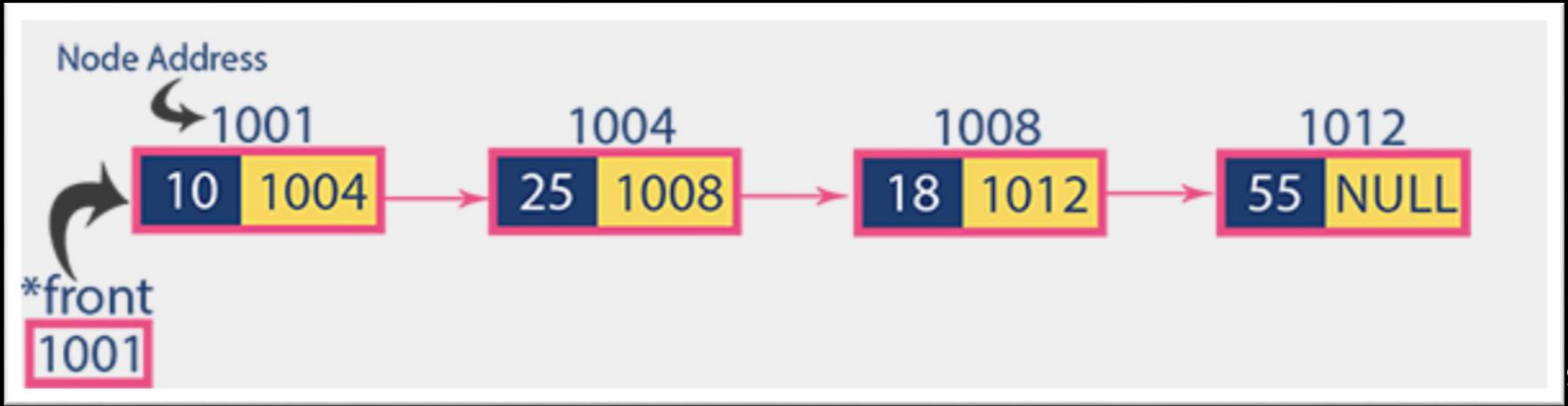
- nodes are usually represented as structs or classes when actually implemented



Singly Linked List

Singly linked lists

- Singly Linked Lists are sequence of nodes where each node stores data and a pointer only to the next node in the list
  - It does not store any pointer or reference to the **previous node**.
- The address of the first node is always stored in a reference node known as "**head**" (Some times it is also known as "**front**").
- Always next part (reference part) of the **last node** must be **NULL**.



Operations on Linked lists

- Inserting a node
  - At the beginning
  - At the end
  - At k^{th} position
- Removing Elements
  - From front
  - From end
  - From k^{th} position
- Traversing the list
- Finding item from the list

Empty list

- Before we implement actual operations, first we need to setup empty list.
- First perform the following steps before implementing actual operations.
  - **Step 1:** Define a **Node** structure with two members **data** and **next**
  - **Step 2:** Define a Node pointer '**head**' and set it to **NULL**.

```
struct Node
{
    int data;
    Node *next;
}*head =NULL;
int main()
{
    return 0;
}
```

Inserting At Beginning of the list

- 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**.

Example Implementation

```
int d;
Node *newNode=new Node;
cout<<"Enter the first node data:";
cin>>d;
newNode->data=d;
if(head==NULL)
{
    newNode->next=NULL;
    head=newNode;
}
else
{
    newNode->next=head;
    head=newNode;
}
```

Navigating through the list

- Necessary when you want to display, insert or delete a node from somewhere inside the list
- Since head pointer always have to point to the first node in the list, we should declare a temporary pointer for navigation

```
node *temp; //temp pointer is used to navigate
temp = head;
if (temp->nxt == NULL)
     cout<< "You are at the end of the list." << endl;
else
     temp = temp-> nxt;
```

- Moving the current pointer back one step is a little harder in singly linked list

Inserting At End of the list

- 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**.

SLL:
add node at the end of the list

```
int d;
Node *newNode=new Node;
cout<<"Enter the node data:";
cin>>d;
newNode->data=d;
newNode->next=NULL;
if(head==NULL)
{
    head=newNode;
}
else
{
    Node *temp=head;
    while(temp->next!=NULL)
    {
        temp=temp->next;
    }
    temp->next=newNode;
}
```

Inserting At Specific location in the list (After a 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**'

SLL:
Inserting a node at K^{th} position

```
int d;
Node *newNode=new Node;
cout<<"Enter the node data:";
cin>>d;
newNode->data=d;
if(head==NULL)
{
    newNode->next=NULL;
    head=newNode;
}
else
{
    Node *temp=head;
    while(temp->next!=NULL && temp->data!=10)
    {
        temp=temp->next;
    }
    if(temp->next==NULL)
    {
        cout<<"Given node is not found in the list!!! Insertion not possible!!!";
    }
    else
    {
        newNode->next=temp->next;
        temp->next=newNode;
        cout<<"New node added at the middle"<<endl;
    }
}
```

Displaying the list of nodes

- Basic steps

  1. Set a temporary pointer to point to the head
  2. If the pointer points to NULL, display the message "End of list‟ and stop.
  3. Otherwise, display the details of the node pointed to by the temp pointer.
  4. Make the temporary pointer point to the same thing as the nxt pointer of the node it is currently indicating.
  5. Jump back to step 2.

Displaying the list of nodes

```
temp = head;
do {
  if (temp = = NULL)
        cout<< "End of list" << endl;

  else {// Display details for what temp points to
        cout<< “Data : " << temp->d << endl;
        temp = temp-> nxt;
        }
} while (temp != NULL);
```

Deleting a node

- Basic steps
  - Find the desirable node (node to be deleted)
  - Set the pointer of the predecessor of the found node to the successor of the found node
  - Release the memory occupied by the found node

- When it comes to delete nodes, we have three choices:
  - Delete a node from the start of the list,
  - Delete one from the end of the list, or
  - Delete at the kth position

Deleting from Beginning of the list

- We can use the following steps to delete a node from beginning of the single linked list...
  - **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
  - **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
  - **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
  - **Step 4:** Check whether list is having only one node (**temp** → **next** == **NULL**)
  - **Step 5:** If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
  - **Step 6:** If it is **FALSE** then set **head** = **temp** → **next**, and delete **temp**.

SLL: Deleting from Beginning of the list

```
if(head==NULL)
{
    cout<<"List is Empty!!! Deletion is not possible";
}
else
{
    Node *temp=head;
    if(temp->next==NULL)
    {
        head=NULL;
    }
    else
    {
        head=temp->next;
    }
    delete temp;
    cout<<"Node is deleted"<<endl;
}
```

Deleting from End of the list

- Steps to delete a node from end of the single linked list...
  - **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
  - **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
  - **Step 3:** If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialize **'temp1'** with **head**.
  - **Step 4:** Check whether list has only one Node (**temp1** → **next** == **NULL**)
  - **Step 5:** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
  - **Step 6:** If it is **FALSE**. Then, set **'temp2 = temp1 '** and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1** → **next** == **NULL**)
  - **Step 7:** Finally, Set **temp2** → **next** = **NULL** and delete **temp1**.

SLL: Deleting the last Node

```
if(head==NULL)
{
    cout<<"List is Empty!!! Deletion is not possible";
}
else
{
    Node *temp1=head;
    Node *temp2;
    if(temp1->next==NULL)
    {
        head=NULL;
    }
    else
    {
        while(temp1->next!=NULL)
        {
            temp2=temp1;
            temp1=temp1->next;
        }
        temp2->next=NULL;
    }
    delete temp1;
    cout<<"Node is deleted at the end"<<endl;
}
```

Deleting a Specific Node from the list

- Steps to delete a specific node from the single linked list...
  - **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
  - **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
  - **Step 3:** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
  - **Step 4:** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
  - **Step 5:** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
  - **Step 6:** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Cont…

- **Step 7:** If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8:** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9:** If **temp1** is the first node then move the **head** to the next node (**head = head** → **next**) and delete **temp1**.
- **Step 10:** If **temp1** is not first node then check whether it is last node in the list (**temp1** → **next == NULL**).
- **Step 11:** If **temp1** is last node then set **temp2** → **next** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 12:** If **temp1** is not first node and not last node then set **temp2** → **next** = **temp1** → **next** and delete **temp1** (**free(temp1)**).

SLL:
Add node at specific location (page1)

```
if(head==NULL)
{
    cout<<"List is Empty!!! Deletion is not possible"<<endl;
}
else
{
    Node *temp1=head;
    Node *temp2;
    if(temp1->next==NULL){
        head=NULL;
    }
    else
    {
        while(temp1->next!=NULL && temp1->data!=20)
        {
            temp2=temp1;
            temp1=temp1->next;
        }
        if(temp1->next==NULL && temp1->data!=20)
        {
            cout<<"Given node not found in the list! Deletion not possible!!!"<<endl;
        }
        else
        {
            if(head->next==NULL)
            {
                head=NULL;
                delete temp1;
            }
```

SLL:
Delete node at specific location (page 2)

```
                if(head->next==NULL)
                {
                    head=NULL;
                    delete temp1;
                }
                else
                {
                    if(temp1==head)
                    {
                        head=head->next;
                        delete temp1;
                    }
                    else if(temp1->next==NULL)
                    {
                        temp2->next=NULL;
                        delete temp1;
                    }
                    else
                    {
                        temp2->next=temp1->next;
                        delete temp1;
                    }

                }
            }
        }
```

Array Vs. Linked list

Array

- Physically Contiguous
- Fixed Length
- Access Elements by Index
- Insertion/Removal is Costly

Linked Lists

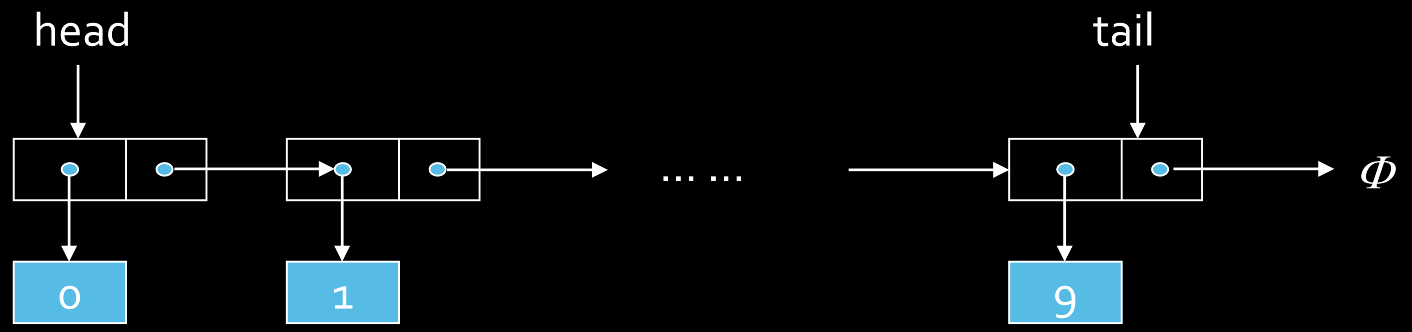
- Logically Contiguous Only
- Changeable Length
- Access Elements by Traversal
- Insertion/Removal is Efficient
- Uses more storage than array with same number of items

Review: Basic code for linked list implementation

- **Create a new node**
  - `Structure_Name *new_node= new Structure_Name;`
  - `Student *S1= new Student;`
- **Getting data from the list**
  - `S1->age; or S1->name;`
- **Checking whether the list empty or not**
  - `If(head==NULL) else not empty`
- **Finding the last node**
  - `temp=head;`
  - `while(temp->next!=NULL)`
    - `temp=temp->next;`

Single Linked List: Exercises

Write a C++ program to create a linked list as shown below.



The list represented by the above diagram has got a tail pointer that points to the last node of the list in addition to the head pointer. The tail pointer makes adding new node to the end of the list simple i.e O(1) which is O(n) with out it.
Initially, both head and tail point to NULL.
When the 1st node added, both head and tail point to the same node.

Next Time !!

Doubly Linked List