

# Programming and Algorithms

COMP1038.PGA

## **Week 8 – Lecture 1, 2: Singly Linked List**

Dr. Pushpendu Kar

# Overview

- Linked list
  - Introduction
  - Creation
  - Insertion
  - Deletion
  - Printing
  - Searching
  - Application



# Introduction

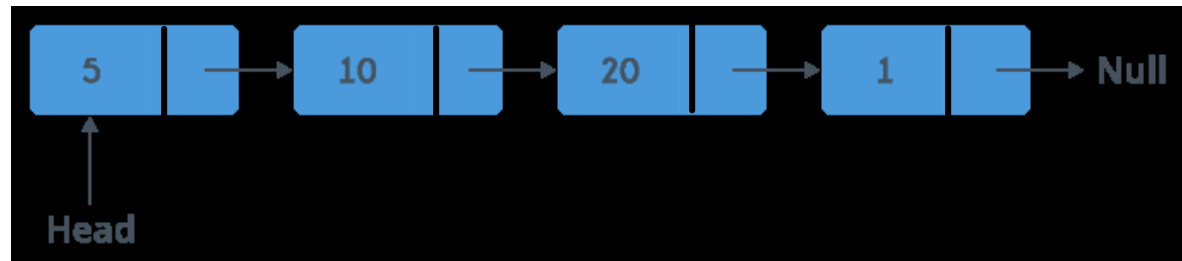
- Lists are linear data structures which store elements of the list one after another.
- Unlike arrays, we can add/remove elements without having to re-create the entire data structure.
- Insert/remove elements from anywhere in the list.
- Access elements anywhere in the list, but slower than arrays.



# Introduction cont...

- Linked list is a list of elements that are connected to each other by pointers.

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



# Comparison with Array

- Collection of items stored at continuous memory location.

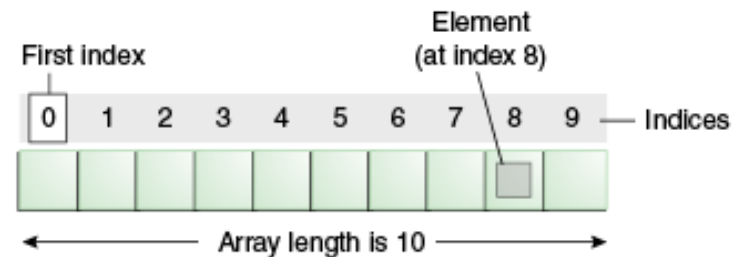
```
#include <stdio.h>

int main()
{
    int arr[3] = {1, 2, 3};

    int i = 0;
    for(i = 0; i < 3; i++)
    {
        printf("#%d: %d\n", i, arr[i]);
    }

    // random access
    printf("\n\nRandom access #%d: %d\n", 2, arr[2]);

    return 0;
}
```



Source: <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html>

```
[z2017233@CSLinux Desktop]$ gcc c_test.c -o c_test
[z2017233@CSLinux Desktop]$ ./c_test
#0: 1
#1: 2
#2: 3

Random access #2: 3
```

# Comparison with Array

## cont...

- Linked list:
  - Can be of any size as long as memory permits
  - Insertion and deletion of data is easier
- Array:
  - Size is fixed i.e. if array is created, cannot change size again during execution of program.
  - Insertion and deletion of data is difficult



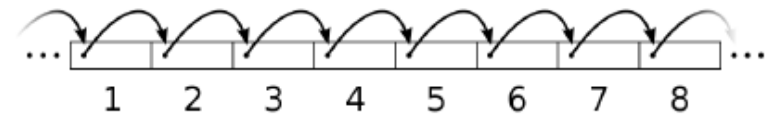
# Comparison with Array

## cont...

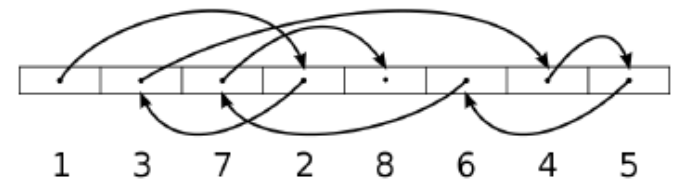
- Linked list: sequential access

```
// This function prints contents of linked list starting from head
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}
```

### Sequential access



### Random access



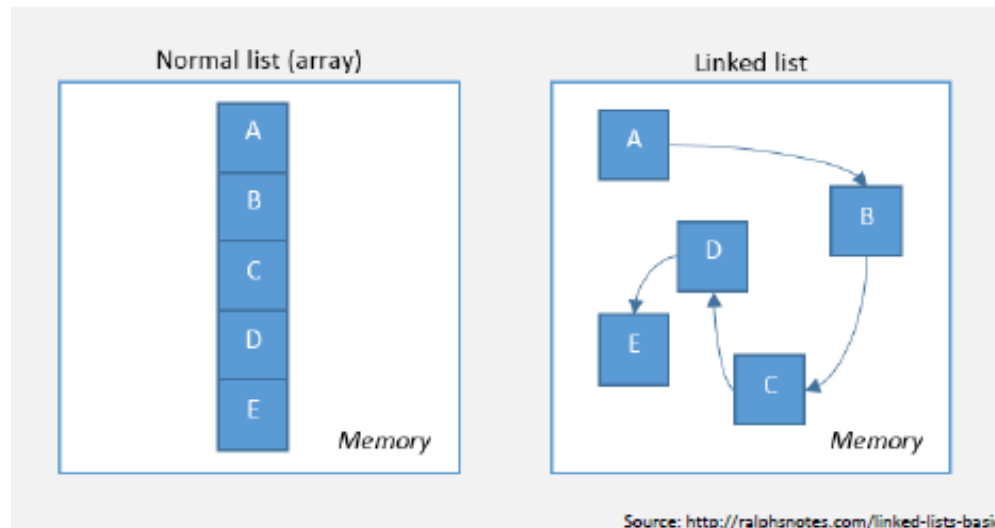
Source: [https://en.wikipedia.org/wiki/Random\\_access](https://en.wikipedia.org/wiki/Random_access)

- Array: random access

```
// random access
printf("\n\nRandom access #d: %d\n", 2, arr[2]);
```

# Comparison with Array cont...

- Memory allocation:

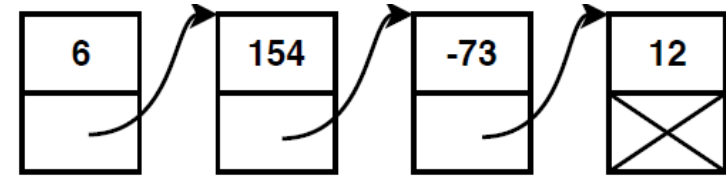




# Types

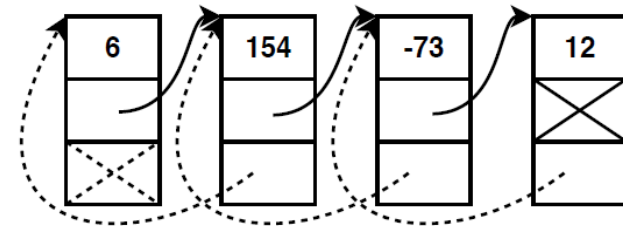
## ■ Singly Linked list

- A pointer to the next element of the list.
- Indicate end of list with NULL in the last pointer.
- Can only navigate the list in one direction.
- Accessing previous element requires traversing from the start of the list again.



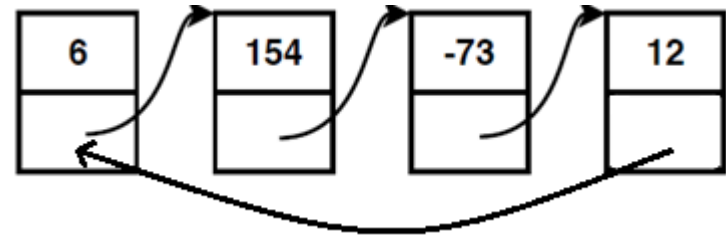
## ■ Doubly Linked list

- Pointer to next element as with singly-linked list.
- Pointer to previous element as well.
- Can access previous element just by using previous pointer.
- More efficient navigation but more complex algorithms and larger storage requirements



## ■ Circular Linked list

- A pointer to the next element of the list.
- End node of the list points to the first node of the list
- Can only navigate the list in one direction.



# Creation

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

```
int main()
{
    struct Node *list =
        malloc(sizeof(struct Node));
    ...
}
```



# Insertion

## ■ Insertion at the beginning

*/\* 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 insertBeginning(struct Node** head_ref, int new_data)
{
```

*/\* 1. allocate node \*/*

```
struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
```

*/\* 2. put in the data \*/*

```
new_node->data = new_data;
```

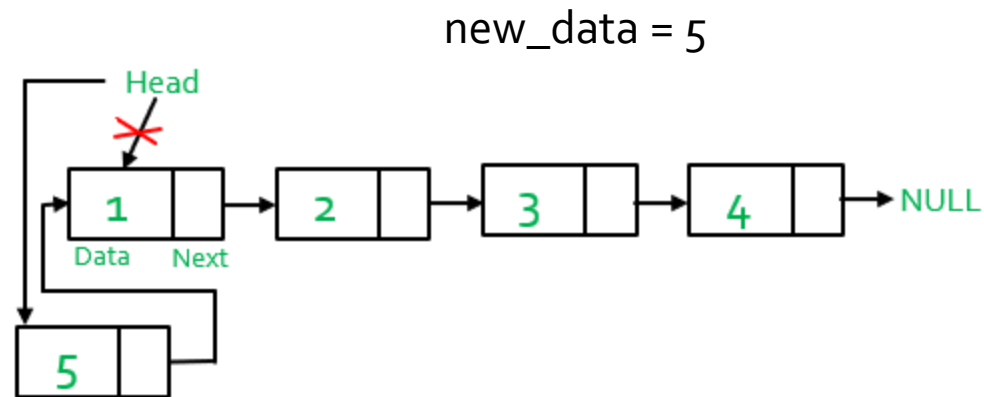
*/\* 3. Make next of new node as head \*/*

```
new_node->next = (*head_ref);
```

*/\* 4. move the head to point to the new node \*/*

```
(*head_ref) = new_node;
```

```
}
```



# Insertion cont...

## ■ Insertion 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(struct Node** head_ref, int new_data)
```

```
{
```

```
    /* 1. allocate node */
```

```
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
```

```
    struct Node *tmp = *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 (tmp->next != NULL)
```

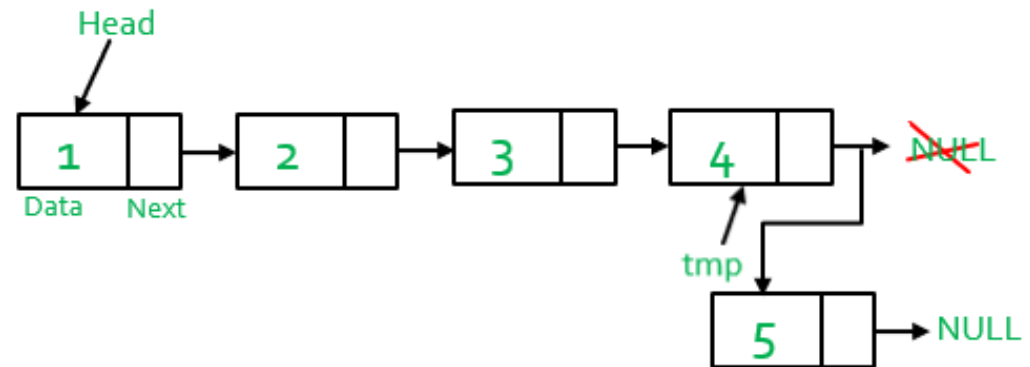
```
        tmp = tmp->next;
```

```
    /* 6. Change the next of last node */
```

```
    tmp->next = new_node;
```

```
    return;
```

```
}
```



# Insertion cont...

## ■ Add a node after a given key:

*/\* Given a key, insert a new node after the key \*/*

```
void insertAfter(struct Node **head_ref, int new_data, int insert_after)
```

```
{
```

```
    struct Node *tmp = *head_ref;
```

*/\*1. Search for the key to be inserted after \*/*

```
    while (tmp != NULL && tmp->data != insert_after)
```

```
        tmp = tmp->next;
```

*/\*2. If key was not present in linked list \*/*

```
    if (tmp == NULL)
```

```
        return;
```

*/\*3. allocate node \*/*

```
    struct Node *new_node = (struct Node *) malloc(sizeof(node));
```

*/\*4. put in the data \*/*

```
    new_node->data = new_data;
```

*/\*5. Make next of new node as next of prev\_node \*/*

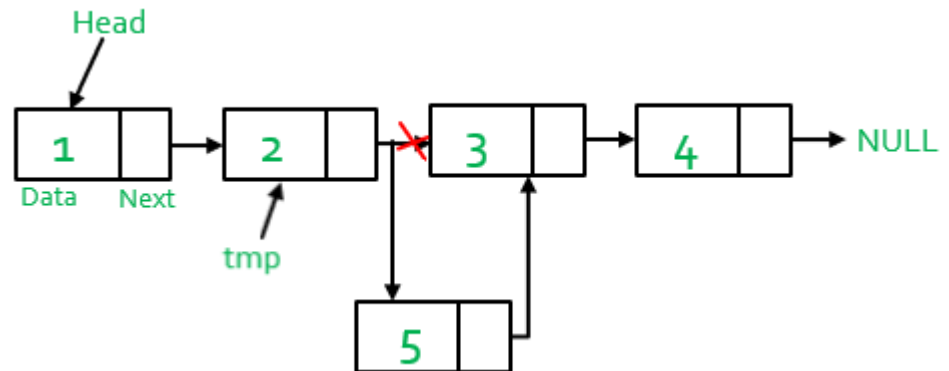
```
    new_node->next = tmp->next;
```

*/\*6. move the next of prev\_node as new\_node \*/*

```
    tmp->next = new_node;
```

```
}
```

new\_data = 5, insert\_after = 2

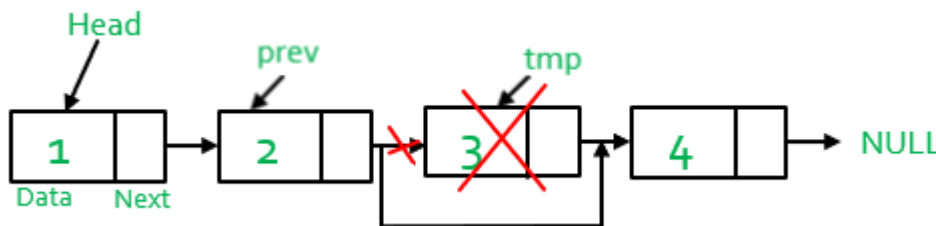


# Deletion

## ■ Delete a node with given key

- 1) Find previous node of the node to be deleted.
- 2) Change the next of previous node.
- 3) Free memory for the node to be deleted.

Key = 3



*/\* Given a reference (pointer to pointer) to the head of a list and a key, deletes the first occurrence of key in linked list \*/*

`void deleteKey(struct Node **head_ref, int key)`

`{`

*/\*1. Store head node \*/*

`struct Node* tmp = *head_ref, *prev;`

*/\*2 If head node itself holds the key to be deleted \*/*

`if (tmp != NULL && tmp->data == key)`

`{`

`*head_ref = tmp->next; // Changed head`

`free(tmp); // free old head`

`return;`

`}`

*/\*3 Search for the key to be deleted, keep track of the previous node as we need to change 'prev->next' \*/*

`while (tmp != NULL && tmp->data != key)`

`{`

`prev = tmp;`

`tmp = tmp->next;`

`}`

*/\*4 If key was not present in linked list \*/*

`if (tmp == NULL) return;`

*/\*5 Unlink the node from linked list \*/*

`prev->next = tmp->next;`

`free(tmp); // Free memory`

`}`

# Deletion cont...

## ■ Delete a Linked List node at a given index

*/\* Given a reference (pointer to pointer) to the head of a list and a index, deletes the node at the given index \*/*

```
void deleteAtIndex(struct Node **head_ref, int index)
```

```
{ /*1 If linked list is empty */
```

```
  if (*head_ref == NULL)
```

```
    return;
```

```
  /*2 Store head node */
```

```
  struct Node* tmp = *head_ref;
```

```
  /*3 If head needs to be removed */
```

```
  if (index == 0)
```

```
  { *head_ref = tmp->next; // Change head
```

```
    free(tmp); // free old head
```

```
    return; }
```

```
  /*4 Find previous node of the node to be deleted */
```

```
  for (int i=0; tmp!=NULL && i<index-1; i++)
```

```
    tmp = tmp->next;
```

```
  /*5 If position is more than number of nodes */
```

```
  if (tmp == NULL || tmp->next == NULL)
```

```
    return;
```

```
  /*6 Node tmp->next is the node to be deleted Store pointer to the next of node to be deleted */
```

```
  struct Node *next = tmp->next->next;
```

```
  /*7 Unlink the node from linked list
```

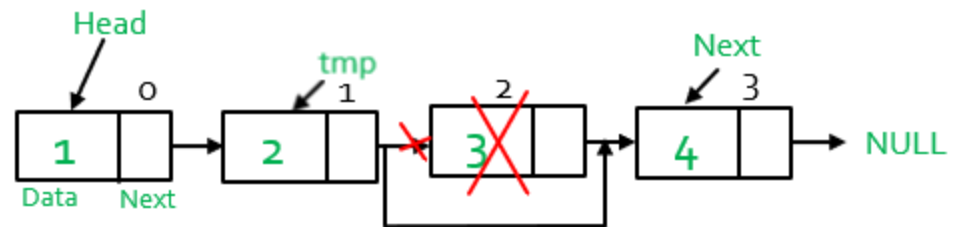
```
    free(tmp->next); // Free memory
```

```
  /*8 Unlink the deleted node from list */
```

```
  tmp->next = next;
```

```
}
```

index = 2

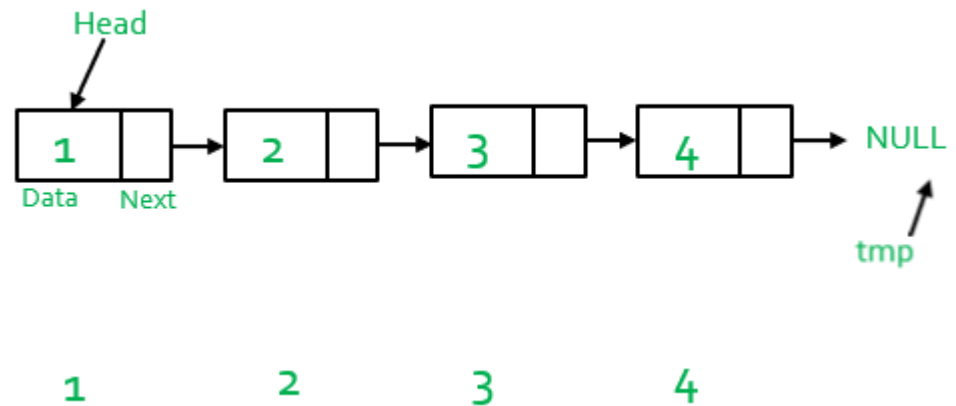


# Printing

*/\* This function prints contents of linked list starting from the given node \*/*

```
void printList(struct Node **head_ref)
```

```
{  
    struct Node *tmp = *head_ref;  
    if(tmp==NULL)  
    {  
        printf("empty");  
        return;  
    }  
    while (tmp != NULL)  
    {  
        printf("%d ",tmp->data);  
        tmp = tmp->next;  
    }  
}
```

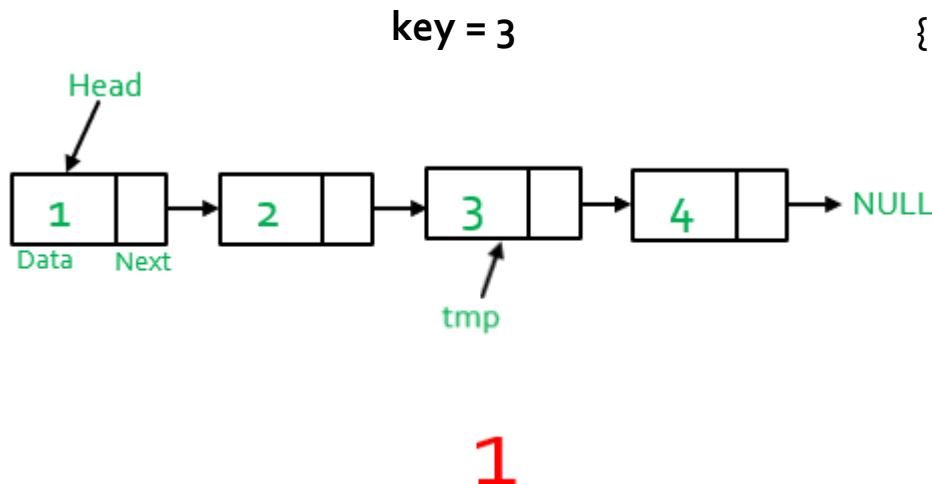




# Searching an item

## Steps:

- 1) Initialize a node pointer, tmp = head.
- 2) Do following while tmp is not NULL
  - a) tmp ->data is equal to the key being searched, return 1.
  - b) tmp = tmp->next
- 3) Return 0



*/\* Checks whether the key is present in linked list \*/*

```
int search(struct Node** head_ref, int key)
{ struct Node* tmp = *head_ref; // Initialize tmp
  while (tmp != NULL)
  {
    if (tmp ->data == key)
      return 1;
    tmp = tmp ->next;
  }
  return 0;
```

# Linked list: application

- **Applications of linked list in computer science:**

- Implementation of stacks and queues
- Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
- Dynamic memory allocation : We use linked list of free blocks.
- Maintaining directory of names
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of linked list representing sparse matrices

- **Applications of linked list in real world:**

- Image viewer – Previous and next images are linked, hence can be accessed by next and previous button.
- Previous and next page in web browser – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.



# The End

