CS 2124: DATA STRUCTURES
Spring 2024

Fifth Lecture

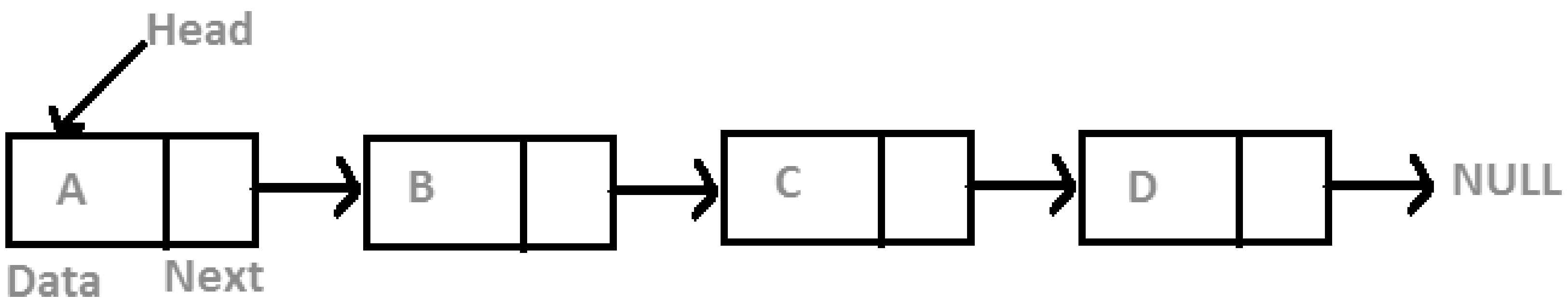
Topics: Queues and **Linked Lists**

Topics

- LinkedList
- Making a LinkedList
- LinkedList (Types)
- LinkedList (Operations)
    - Traversing
    - Insertion
    - Deletion
    - Searching
- Pointer to Pointer (Example)
- LinkedList (Complete Code)
    - Advantages and disadvantages
    - Applications
    - LinkedList (Memory)

LinkedList

- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below images:

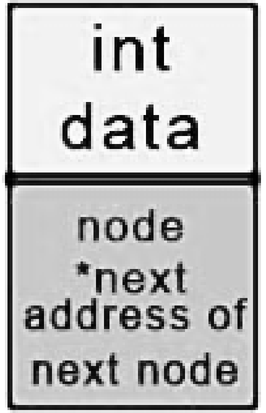


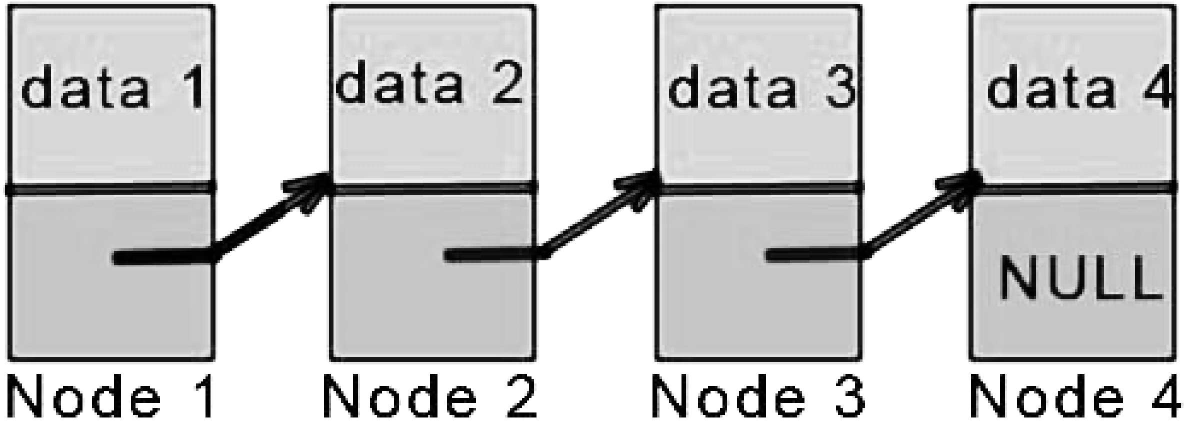
- **Node Structure:** A node in a linked list typically consists of two components:
  - **Data:** It holds the actual value or data associated with the node.
  - **Next Pointer:** It stores the memory address (reference) of the next node in the sequence.
- **Head and Tail:** The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

Making a LinkedList

- A linked list is made up of many nodes which are connected in nature. Every node is mainly divided into two parts, one part holds the data and the other part is connected to a different node
- In C, we achieve this functionality by using **structures and pointers**. Each structure represents a node having some data and also a pointer to another structure of the same kind. This pointer holds the address of the next node and creates the link between two nodes. So, the structure is something like:

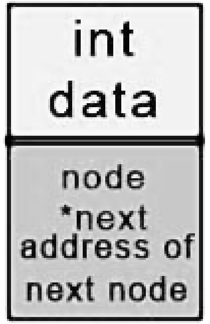
```c
#include <stdio.h>
#include <stdlib.h>

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



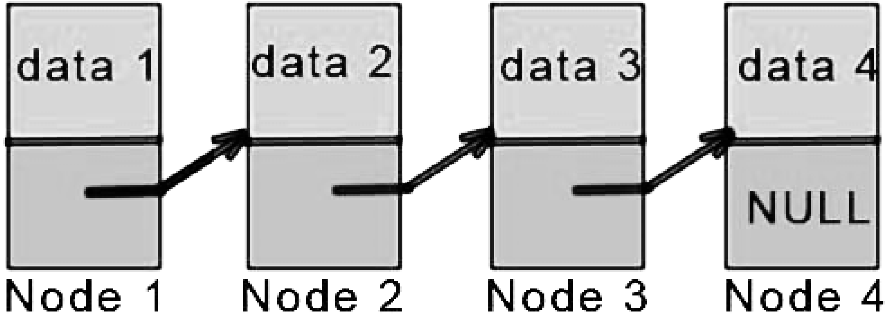


LinkedList

```c
#include <stdio.h>
#include <stdlib.h>
// Creating a node
struct node {
  int value;
  struct node *next;
};
```



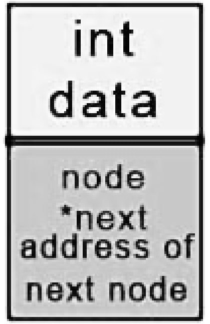

```c
int main() {
  // Initialize nodes
  struct node *head;
  struct node *one = NULL;
  struct node *two = NULL;
  struct node *three = NULL;
  struct node *four = NULL;
  // Allocate memory
  one = malloc(sizeof(struct node));
  two = malloc(sizeof(struct node));
  three = malloc(sizeof(struct node));
  four = malloc(sizeof(struct node));
  // Connect nodes
  one->next = two;
  two->next = three;
  three->next = four;
  four->next = NULL;
  // printing node-value
  printf("<Name, abc123, SP24>\n");
  head = one;
  printLinkedlist(head);
}
```




```c
// print the linked list value & address
void printLinkedlist(struct node *p) {
  while (p != NULL) {
    printf("Value: %d , Add: %p \n ", p->value, &p->value);
    p = p->next;
  } }
```

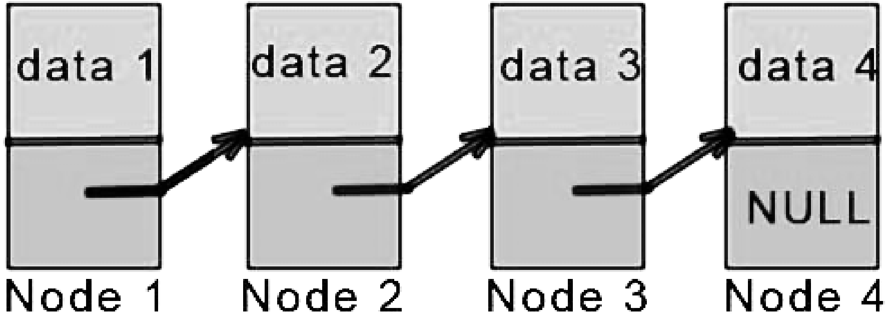
LinkedList

```c
#include <stdio.h>
#include <stdlib.h>
// Creating a node
struct node {
   int value;
   struct node *next;
};
```



- What if we do not create a Head node?

```c
int main() {
   // Initialize nodes
   struct node *one = NULL;
   struct node *two = NULL;
   struct node *three = NULL;
   struct node *four = NULL;
   // Allocate memory
   one = malloc(sizeof(struct node));
   two = malloc(sizeof(struct node));
   three = malloc(sizeof(struct node));
   four = malloc(sizeof(struct node));
   // Connect nodes
   one->next = two;
   two->next = three;
   three->next = four;
   four->next = NULL;
   // printing node-value
   printf("<Name, abc123, SP24>\n");
   printLinkedlist(one);
}
```




```c
// print the linked list value & address
void printLinkedlist(struct node *p) {
  while (p != NULL) {
    printf("Value: %d , Add: %p \n ", p->value, &p->value);
    p = p->next;
  } }
```

LinkedList

```c
#include <stdio.h>
#include <stdlib.h>
// Creating a node
struct node {
  int value;
  struct node *next;
};
```

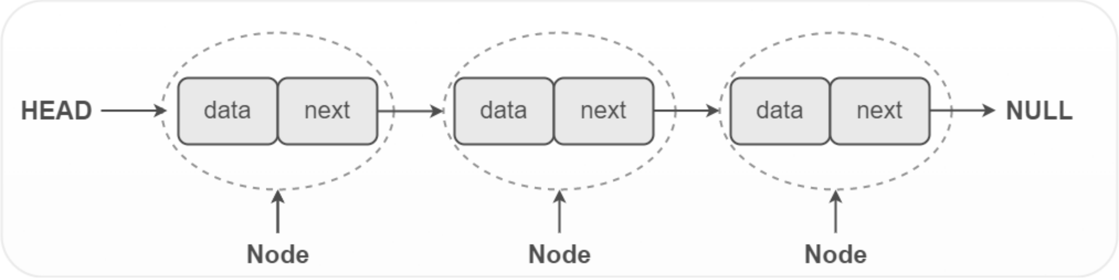
```c
// print the linked list value & address
void printLinkedlist(struct node *p) {
  while (p != NULL) {
    printf("Value: %d , Add: %p \n ", p->value, &p->value);
    p = p->next;
  } }
```

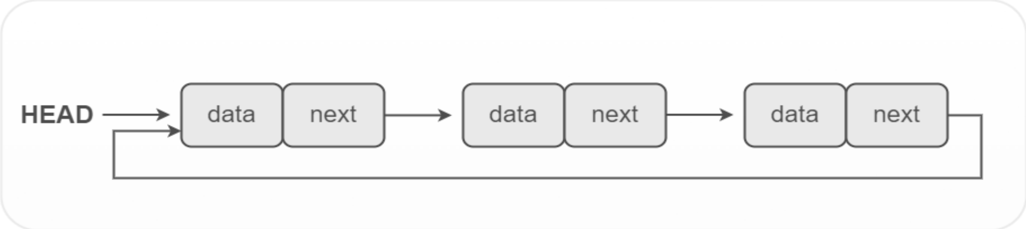
```c
int main() {
 // Initialize nodes
  struct node *head;
  struct node *one = NULL;
  struct node *two = NULL;
  struct node *three = NULL;
  struct node *four = NULL;
 // Allocate memory
  one = malloc(sizeof(struct node));
  two = malloc(sizeof(struct node));
  three = malloc(sizeof(struct node));
  four = malloc(sizeof(struct node));
// Assign value values
  one->value = 2;
  two->value = 0;
  three->value = 2;
  four->value = 4;
 // Connect nodes
  one->next = two;
  two->next = three;
  three->next = four;
  four->next = NULL;
 // printing node-value
  printf("<Name, abc123, SP24>\n");
  head = one;
  printLinkedlist(head);
}
```

1. Will there be an error
2. Will compile but will not output values or addresses
3. Only warning but will compile without any output
4. Will run without warning and will display output

LinkedList (Types)

1. **Single-linked list:** Traversal of items can be done in the forward direction only due to the linking of every node to its next node.
   - Operations: Insertion (start, end or specific location of list), Deletion (start, end or specific node), search, and display
2. **Double linked list:** Traversal of items can be done in both forward and backward directions as every node contains an additional prev pointer that points to the previous node.
   - Operations: Insertion (start, end, after or before a node or specific location of list), Deletion (start, end or specific node), and display.
3. **Circular linked list:** A circular linked list is a type of linked list in which the first and the last nodes are also connected to each other to form a circle, there is no NULL at the end.
   - Operations: Insertion (empty list, start, end, or between nodes of list), Deletion (start, end or specific node), and display.





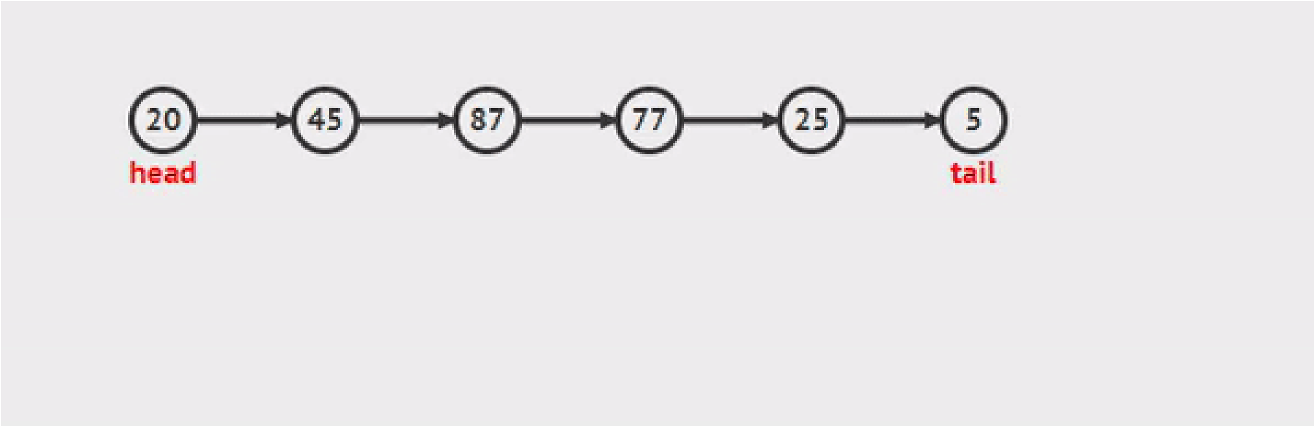
LinkedList

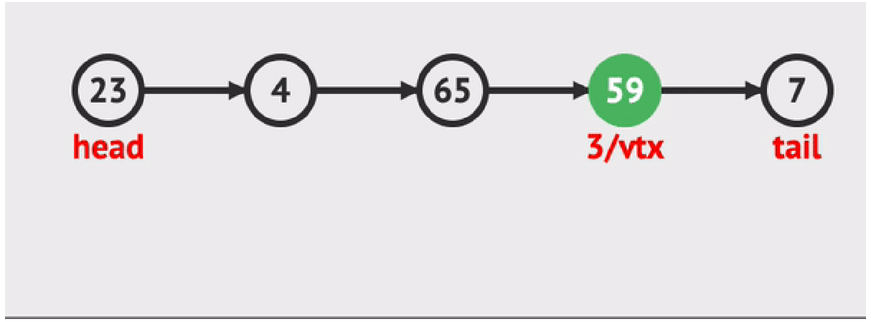
- It is basically chains of nodes, each node contains information such as data and a pointer to the next node in the chain.
- In the linked list there is a head pointer, which points to the first element of the linked list, and if the list is empty then it simply points to null or nothing.

Linkedlist	Array
Not stored in a Contiguous location	Stored in Contiguous location
Dynamic size	Fixed size
Memory allocation on run time	Memory allocation on compile time
Use more memory then array	Use less memory then LinkedList
Access require traversing	Easy access to elements
Insertion & deletion fast	Insertion & deletion slow

LinkedList (Operations)

- Traversing: To traverse all nodes one by one.
- Insertion: To insert new nodes at specific positions.
- Deletion: To delete nodes from specific positions.
- Searching: To search for an element from the linked list.





LinkedList (Operations - Traversing)

- We can use the NULL (pointer) to identify that we have reached the end of the LinkedList.

Code from slide 8

```c
// print the linked list value & address
void printLinkedlist(struct node *p) {
  while (p != NULL) {
    printf("Value: %d , Add: %p \n ", p->value, &p->value);
    p = p->next;
    printf("Next Add: %p \n ", p);
  } }
```

```c
int main() {
  // Initialize nodes
  struct node *head;
  struct node *one = NULL;
  struct node *two = NULL;
  struct node *three = NULL;
  struct node *four = NULL;
  // Allocate memory
  one = malloc(sizeof(struct node));
  two = malloc(sizeof(struct node));
  three = malloc(sizeof(struct node));
  four = malloc(sizeof(struct node));
  // Connect nodes
  one->next = two;
  two->next = three;
  three->next = four;
  four->next = NULL;
  // printing node-value
  printf("<Name, abc123, SP24>\n");
  head = one;
  printLinkedlist(head);
}
```

Not equal (!=)

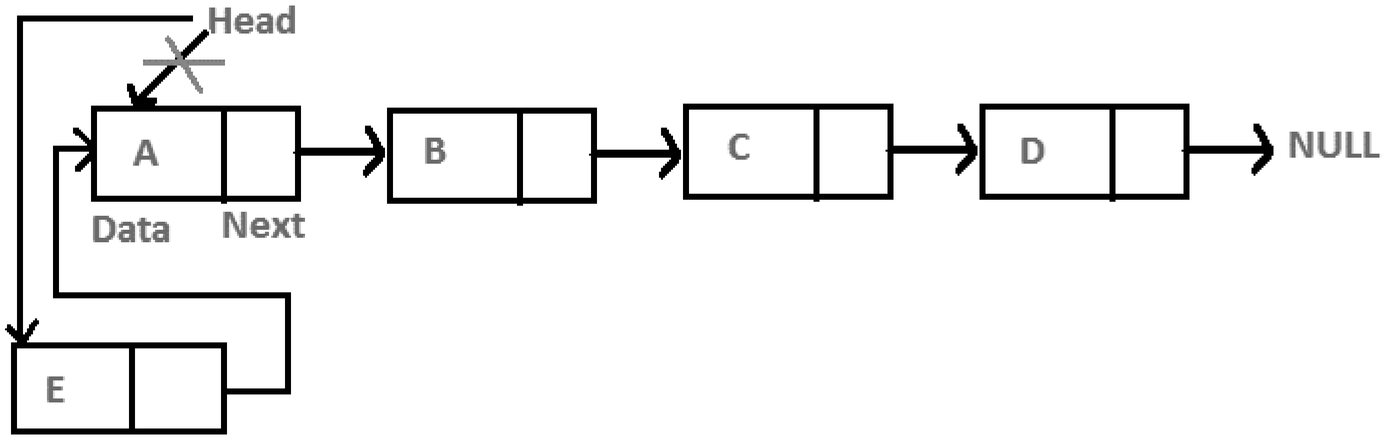
LinkedList (Operations - **Insert at the beginning & End**)

- **Insert at the beginning**
  1. Allocate memory for new node
  2. Store data
  3. Change next of new node to point to head
  4. Change head to point to recently created node

The function that adds at the front of the list is push().

The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node

```c
int main()
{
// Start with the empty list
struct Node* head = NULL;
// Insert 2->NULL
append(&head, 2);
// Insert 0->2->NULL
push(&head, 0);
printf("Created Linked list:\n ");
printList(head);
}
```



LinkedList (Operations - **Insert at the beginning & End**)

```c
#include <stdio.h>
#include <stdlib.h>
// A linked list node
struct Node
{
int data;
struct Node *next;
};
// 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(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;
}
```

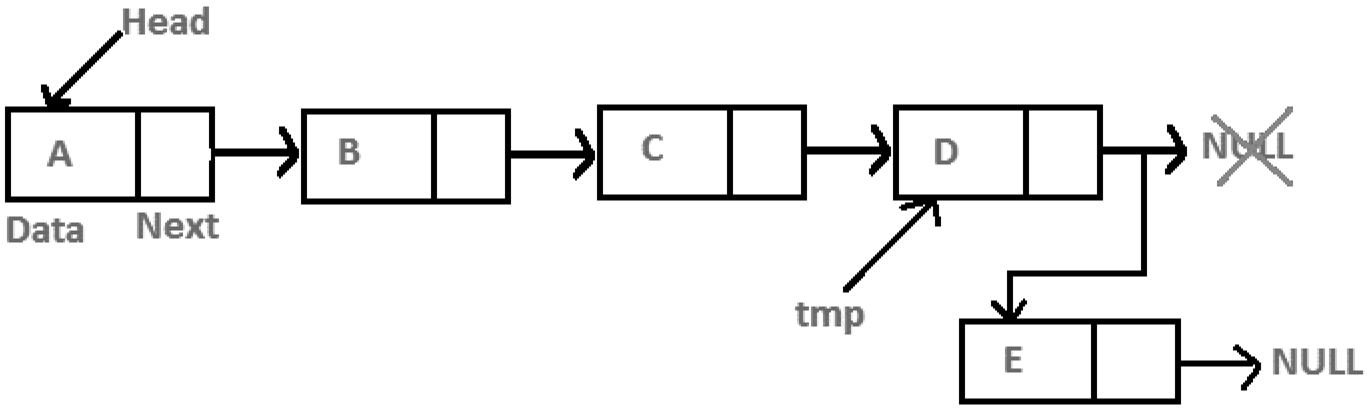
```c
int main()
{
// Start with the empty list
struct Node* head = NULL;
// Insert 2->NULL
append(&head, 2);
// Insert 0->2->NULL
push(&head, 0);
printf("Created Linked list:\n ");
printList(head);
}
```

LinkedList (Operations - **Insert at the beginning & End**)

- **Insert at the End**
    1. Allocate memory for new node
    2. Store data
    3. Traverse to last node
    4. Change next of last node to recently created node

Since a Linked List is typically represented by the head of it, we have to traverse the list till the end and then change the next to last node to a new node.

```
int main()
{
// Start with the empty list
struct Node* head = NULL;
// Insert 2->NULL
append(&head, 2);
// Insert 0->2->NULL
push(&head, 0);
// Insert 0->2->3->NULL
append(&head, 3);
printf("Created Linked list:\n ");
printList(head);
}
```



LinkedList (Operations - Insert at the beginning & End)

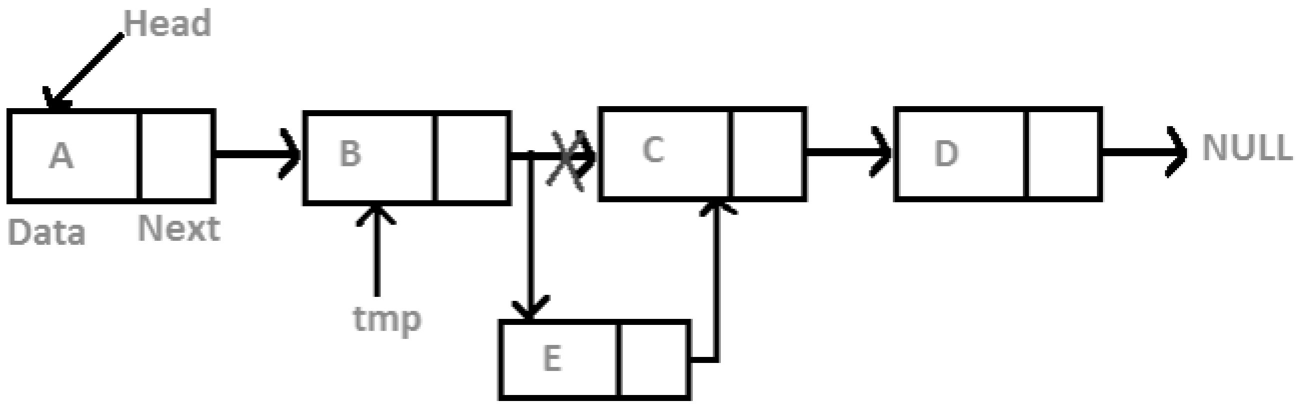
```c
// Given a reference (pointer to pointer) to
// the head of a list and an int, appends a
// new node at the end */
void append(struct Node** head_ref,
            int new_data)
{
    // 1. Allocate node
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));
    // Used in step 5
    struct Node *last = *head_ref;
    // 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;
}
```

```c
int main()
{
// Start with the empty list
struct Node* head = NULL;
// Insert 2->NULL
append(&head, 2);
// Insert 0->2->NULL
push(&head, 0);
// Insert 0->2->3->NULL
append(&head, 3);
printf("Created Linked list:\n ");
printList(head);
}
```

LinkedList (Operations - **Insert at the Middle**)

- **Insert at the Middle**
  1. Allocate memory and store data for new node
  2. Traverse to node just before the required position of new node
  3. Change next pointers to include new node in between

```c
// Driver code
int main()
{
// Start with the empty list
struct Node* head = NULL;
// Insert 2->NULL
append(&head, 2);
// Insert 0->2->NULL
push(&head, 0);
// Insert 0->1->2->NULL
insertAfter(head, 1);
printf("Created Linked list:\n ");
printList(head);
}
```



LinkedList (Operations - **Insert at the Middle**)

```c
// Given a node prev_node, insert a
// new node after the given prev_node
void insertAfter(struct Node* prev_node,int new_data)
{
    // 1. Check if the given prev_node is NULL
    if (prev_node == NULL)
    {
    printf("the given previous node cannot be NULL");
    return;
    }
    // 2. Allocate new node
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));
    // 3. Put in the data
    new_node->data = new_data;
    // 4. Make next of new node as next of prev_node
    new_node->next = prev_node->next;
    // 5. Move the next of prev_node as new_node
    prev_node->next = new_node;
}
```

```c
// Driver code
int main()
{
// Start with the empty list
struct Node* head = NULL;
// Insert 2->NULL
append(&head, 2);
// Insert 0->2->NULL
push(&head, 0);
// Insert 0->1->2->NULL
insertAfter(head, 1);
printf("Created Linked list:\n ");
printList(head);
}
```

LinkedList (Operations - Deletion)

1. Delete from Beginning:

- Point head to the next node i.e. second node
- temp = head
- head = head->next
- Make sure to free unused memory
- free(temp); or delete temp;

```
int main()
{
    Node* list = malloc(sizeof(Node));
    list->next = NULL;
    //Create node 0->Null
    Push(&list, 15);
    //Create node 15->0->Null
    Push(&list, 25);
    //Create node 25->15->0->Null
    printList(list);
    deleteN(&list, 1);
    // Delete first node of the L.L 15->0->Null
    printf("After Deletion:\n");
    printList(list);
}
```

LinkedList (Operations - Deletion)

2. Delete from End:

- Point head to the previous element i.e. last second element
- Change next pointer to null
- struct node *end = head;
- struct node *prev = NULL;
- while(end->next)
- {
- prev = end;
- end = end->next;
- }
- prev->next = NULL;
-
- Make sure to free unused memory
- free(end); or delete end;

```c
int main()
{
    Node* list = malloc(sizeof(Node));
    list->next = NULL;
    //Create node 0->Null
    Push(&list, 15);
    //Create node 15->0->Null
    Push(&list, 25);
    //Create node 25->15->0->Null
    printList(list);
    deleteN(&list, 3);
    // Delete Last node of the L.L 25->15->Null
    printf("After Deletion:\n");
    printList(list);
}
```

LinkedList (Operations - Deletion)

3. **Delete from any location:**

- Keeps track of pointer before node to delete and pointer to node to delete
- temp = head;
- prev = head;
- for(int i = 0; i < position; i++)
- {
- if(i == 0 && position == 1)
- head = head->next;
- free(temp)
- else
- {
- if (i == position - 1 && temp)
- {
- prev->next = temp->next;
- free(temp);
- }
- else
- {
- prev = temp;
- if(prev == NULL) // position was greater than number of nodes in the list
- break;
- temp = temp->next;
- }   }   }

```c
int main()
{
    Node* list = malloc(sizeof(Node));
    list->next = NULL;
    //Create node 0->Null
    Push(&list, 15);
    //Create node 15->0->Null
    Push(&list, 25);
    //Create node 25->15->0->Null
    printList(list);
    deleteN(&list, 2);
    // Delete a node of the L.L 25->0->Null
    printf("After Deletion:\n");
    printList(list);
}
```

LinkedList (Operations - Deletion)

Remaining code for slides 19, 20 & 21

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int number;
    struct Node* next;
} Node;
void Push(Node** head, int A)
{
    Node* n = malloc(sizeof(Node));
    n->number = A;
    n->next = *head;
    *head = n;
}
```

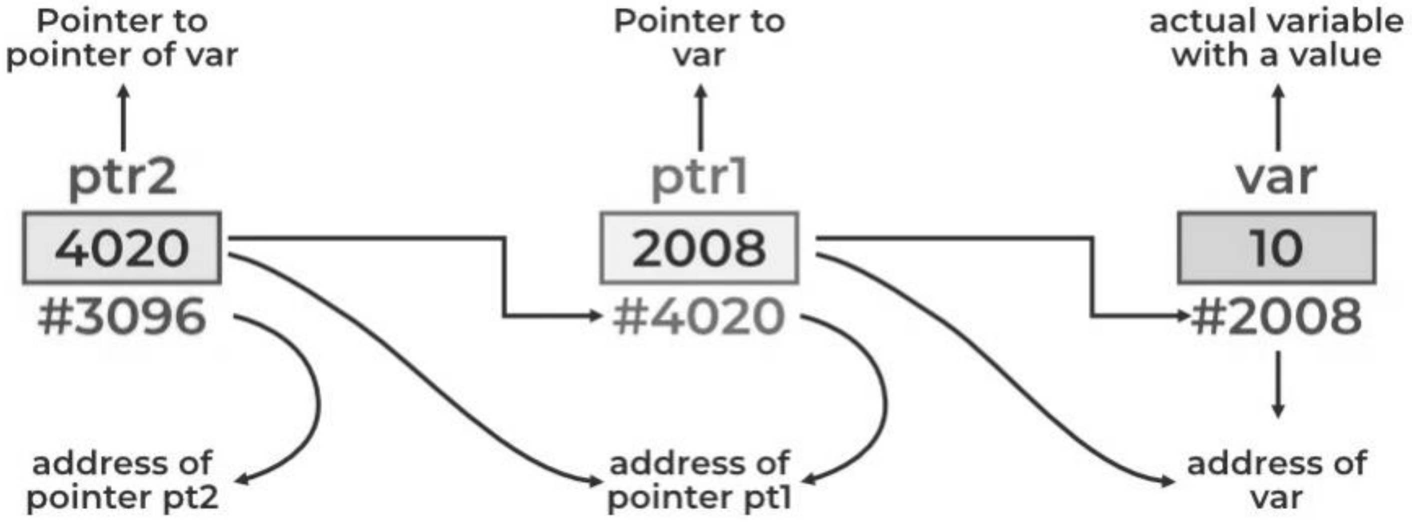
```c
void deleteN(Node** head, int position)
{
    Node* temp;
    Node* prev;
    temp = *head;
    prev = *head;
    for (int i = 0; i < position; i++) {
        if (i == 0 && position == 1) {
            *head = (*head)->next;
            free(temp);
        }
        else {
            if (i == position - 1 && temp) {
                prev->next = temp->next;
                free(temp);
            }
            else {
                prev = temp;
                // Position was greater than
                // number of nodes in the list
                if (prev == NULL)
                    break;
                temp = temp->next;
            }       }   }   }
```

```c
void printList(Node* head)
{
    while (head) {
        printf("Data:%d [%p]->%p\n", head->number, head,
            head->next);
        head = head->next;
    }
    printf("\n");
}
```

LinkedList (Operations - Searching)

- You can search an element on a linked list using a loop. We are finding item on a linked list:

1. Make head as the current node.
2. Run a loop until the current node is NULL because the last element points to NULL.
3. In each iteration, check if the key of the node is equal to item. If it the key matches the item, return true otherwise return false.



Pointer to Pointer (Example)

```c
// C program to demonstrate pointer to pointer
#include <stdio.h>
int main()
{
    int var = 1;
    // pointer for var
    int* ptr2;
    // double pointer for ptr2
    int** ptr1;
    // storing address of var in ptr2
    ptr2 = &var;
    // Storing address of ptr2 in ptr1
    ptr1 = &ptr2;
    // Displaying value of var using
    // both single and double pointers
    printf("Value of var = %d\n", var);
    printf("Value of var using single pointer = %d\n", *ptr2);
    printf("Value of var using double pointer = %d\n", **ptr1);
    printf("Add of var = %p\n", &var);
    printf("Add of var using single pointer = %p\n", &*ptr2);
    printf("Add of var using double pointer = %p\n", &**ptr1);
}
```

Pointer to Pointer (Example)

```c
// C program to demonstrate pointer to pointer
#include <stdio.h>
int main()
{
    int var = 1;
    // pointer for var
    int* ptr2;
    // double pointer for ptr2
    int** ptr1;
    // storing address of var in ptr2
    ptr2 = &var;
    // Storing address of ptr2 in ptr1
    ptr1 = &ptr2;
    // Displaying value of var using
    // both single and double pointers
    printf("Value of var = %d\n", var);
    printf("Value of var using single pointer = %d\n", *ptr2);
    printf("Value of var using double pointer = %d\n", **ptr1);
    printf("Add of var = %p\n", &var);
    printf("Add of var using single pointer = %p\n", &*ptr2);
    printf("Add of var using double pointer = %p\n", &**ptr1);
    printf("Add of single pointer = %p\n", &ptr2);
    printf("Add of double pointer = %p\n", &ptr1);
}
```

Additional Source: [Link]()

LinkedList (Complete Code)

Insert at Beginning & End, Deletion & Searching based on Key value

```c
// Linked list operations in C
#include <stdio.h>
#include <stdlib.h>
// Create a node
struct Node {
  int data;
  struct Node* next;
};
// Insert at the beginning
void insertAtBeginning(struct Node** head_ref, int new_data) {
  // Allocate memory to a node
struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  // insert the data
new_node->data = new_data;
new_node->next = (*head_ref);
  // Move head to new node
  (*head_ref) = new_node;
}
// Insert a node after a node
void insertAfter(struct Node* prev_node, int new_data) {
  if (prev_node == NULL) {
  printf("the given previous node cannot be NULL");
  return;
  }
  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  new_node->data = new_data;
  new_node->next = prev_node->next;
  prev_node->next = new_node;
}
```

```c
// Insert the the end
void insertAtEnd(struct Node** head_ref, int new_data) {
  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  struct Node* last = *head_ref; /* used in step 5*/
  new_node->data = new_data;
  new_node->next = NULL;
  if (*head_ref == NULL) {
  *head_ref = new_node;
  return;
  }
  while (last->next != NULL) last = last->next;
  last->next = new_node;
  return;
}
// Delete a node
void deleteNode(struct Node** head_ref, int key) {
  struct Node *temp = *head_ref, *prev;

  if (temp != NULL && temp->data == key) {
  *head_ref = temp->next;
  free(temp);
  return;
  }
  // Find the key to be deleted
  while (temp != NULL && temp->data != key) {
  prev = temp;
  temp = temp->next;
  }
```

LinkedList (Complete Code)

Insert at Beginning & End, Deletion & Searching based on Key value

```
    // If the key is not present
    if (temp == NULL) return;
    // Remove the node
    prev->next = temp->next;
    free(temp);
}
// Search a node
int searchNode(struct Node** head_ref, int key) {
    struct Node* current = *head_ref;

    while (current != NULL) {
    if (current->data == key) return 1;
    current = current->next;
    }
    return 0;
}
// Print the linked list
void printList(struct Node* node) {
    while (node != NULL) {
    printf("\n %d ", node->data);
    node = node->next;
    }
}
```

```
// Driver program
int main() {
    struct Node* head = NULL;
    insertAtEnd(&head, 1);
    insertAtBeginning(&head, 2);
    insertAtBeginning(&head, 3);
    insertAtEnd(&head, 4);
    insertAfter(head->next, 5);
    printf("Linked list: ");
    printList(head);
    printf("\nAfter deleting an element (3): ");
    deleteNode(&head, 3);
    printList(head);
    printf("\nSearch element (3): ");
    int item_to_find = 3;
    if (searchNode(&head, item_to_find)) {
    printf("\n%d is found", item_to_find);
    } else {
    printf("\n%d is not found", item_to_find);
    }
    printList(head);
}
```

LinkedList (Complete Code)

Insert at Beginning & End, Deletion & Searching based on Key value

```
// Driver program
int main() {
  struct Node* head = NULL;
  insertAtEnd(&head, 1);
  insertAtBeginning(&head, 2);
  insertAtBeginning(&head, 3);
  insertAtEnd(&head, 4);
  insertAfter(head->next, 5);
  printf("Linked list: ");
  printList(head);
  printf("\nAfter deleting an element (3): ");
  deleteNode(&head, 3);
  printList(head);
  printf("\nSearch element (3): ");
  int item_to_find = 3;
  if (searchNode(&head, item_to_find)) {
  printf("\n%d is found", item_to_find);
  } else {
  printf("\n%d is not found", item_to_find);
  }
  printList(head);
}
```

LinkedList (Complete Code)

Insert at Beginning & End, Deletion & Searching based on Key value

```
81  // Driver program
82  int main() {
83    struct Node* head = NULL;
84    insertAtEnd(&head, 1);
85    insertAtBeginning(&head, 2);
86    insertAtBeginning(&head, 3);
87    insertAtEnd(&head, 4);
88    insertAfter(head, 5);
89    printf("Linked list: ");
90    printList(head);
91    printf("\nAfter deleting an element (4): ");
92    deleteNode(&head, 4);
93    printList(head);
94    printf("\nSearch element (4): ");
95    int item_to_find = 4;
96    if (searchNode(&head, item_to_find)) {
97    printf("\n%d is found", item_to_find);
98    } else {
99    printf("\n%d is not found", item_to_find);
100   }
101   printList(head);
102 }
```

Changes in :

- Line 88
- Line 92
- Line 95

LinkedList (Complete Code)

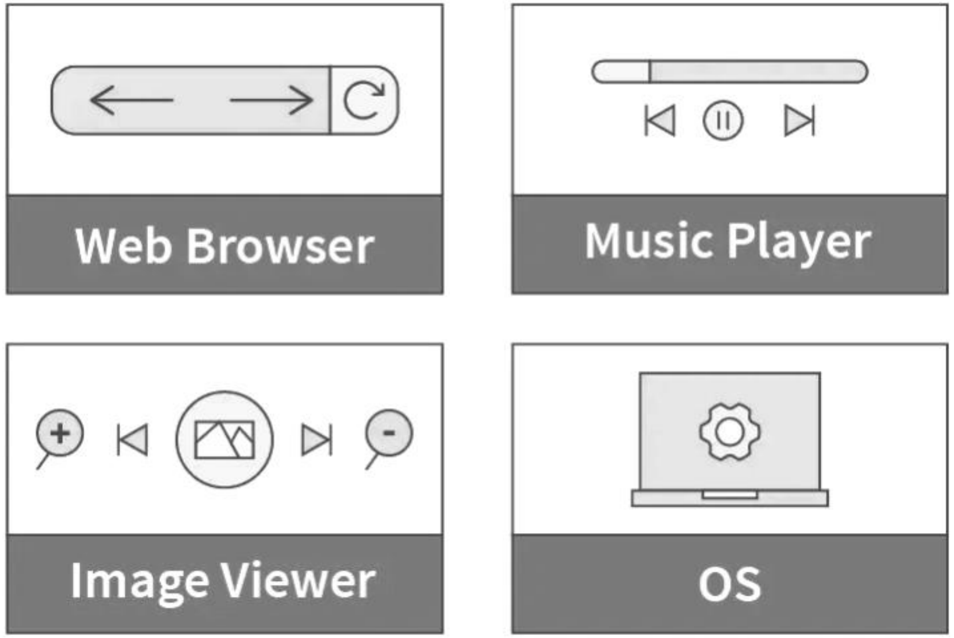
Insert at Beginning & End, Deletion & Searching based on Key value

```
// Driver program
int main() {
  struct Node* head = NULL;
  insertAtEnd(&head, 1);
  insertAtBeginning(&head, 2);
  insertAtBeginning(&head, 3);
  insertAtEnd(&head, 4);
  insertAfter(head, 5);
  printf("Linked list: ");
  printList(head);
  printf("\nAfter deleting an element (4): ");
  deleteNode(&head, 4);
  printList(head);
  printf("\nSearch element (4): ");
  int item_to_find = 4;
  if (searchNode(&head, item_to_find)) {
  printf("\n%d is found", item_to_find);
  } else {
  printf("\n%d is not found", item_to_find);
  }
  printList(head);
}
```

LinkedList

- Advantages:
  - **Dynamic nature:** Linked lists are used for dynamic memory allocation.
  - **Memory efficient:** Memory consumption of a linked list is efficient as its size can grow or shrink dynamically according to our requirements, which means effective memory utilization hence, no memory wastage.
  - **Ease of Insertion and Deletion:** Insertion and deletion of nodes are easily implemented in a linked list at any position.
  - **Implementation:** For the implementation of stacks and queues and for the representation of trees and graphs.
  - The linked list can be expanded in constant time.
- Disadvantages:
  - **Memory usage:** The use of pointers is more in linked lists hence, complex and requires more memory.
  - **Accessing a node:** Random access is not possible due to dynamic memory allocation.
  - **Search operation costly:** Searching for an element is costly and requires O(n) time complexity.
  - **Traversing in reverse order:** Traversing is more time-consuming and reverse traversing is not possible in singly linked lists.

LinkedList (Applications)



- Linear data structures such as stack, queue, and non-linear data structures such as hash maps, and graphs can be implemented using linked lists.
- In web browsers and editors, doubly linked lists can be used to build a forwards and backward navigation button.
- A circular doubly linked list can also be used for implementing data structures like Fibonacci heaps.
- Switching between two applications is carried out by using “alt+tab” in windows and “cmd+tab” in mac book. It requires the functionality of a circular linked list.

LinkedList (Memory)

- Should we use **malloc** or **calloc**?
- What's is the difference between them?
- Malloc is used for **dynamic memory allocation** and is useful when you don't know the amount of memory needed during compile time. Allocating memory allows objects to exist beyond the scope of the current block.
- The name "calloc" stands for **contiguous allocation**. The malloc() function allocates memory and leaves the memory uninitialized, whereas the calloc() function allocates memory and initializes all bits to zero.

LinkedList (Memory)

- Malloc is used for dynamic memory allocation and is useful when you don't know the amount of memory needed during compile time.
- Linked List relies heavily on the malloc() function to allocate some memory for new nodes dynamically.
- The reason why we need to use the temporary variable is that we don't want to change the address stored in the head pointer.
    - **malloc p** = malloc(n) - **allocates n bytes of *heap memory**; the memory contents remain uninitialized.
    - **calloc p** = calloc(count, size) allocates count*size bytes of heap memory and initializes it all to zero; this call is appropriate when you want to allocate an array of count items, each of size bytes.

*Heaps are memory areas allocated to each program. Memory allocated to heaps can be dynamically allocated, unlike memory allocated to stacks. As a result, the heap segment can be requested and released whenever the program needs it.