Linked list Node Creation:

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

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

void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}

int main()
{
    struct Node *head;
    struct Node *second;
    struct Node *third;
    struct Node *fourth;

    // Allocate memory for nodes in the linked list in Heap
    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));
    fourth = (struct Node *)malloc(sizeof(struct Node));

    // Link first and second nodes
    head->data = 7;
    head->next = second;

    // Link second and third nodes
    second->data = 11;
    second->next = third;

    // Link third and fourth nodes
    third->data = 41;
    third->next = fourth;
```

```
    // Terminate the list at the third node
    fourth->data = 66;
    fourth->next = NULL;

    linkedListTraversal(head);
    return 0;
}
```

Operations on linked list

the cases:

1. Inserting at the beginning -> Time complexity: O(1)
2. Inserting in between -> Time complexity: O(n)
3. Inserting at the end -> Time complexity: O(n)
4. Inserting after a given Node -> Time complexity: O(1)

Insertion at the beginning:

1. Create a struct Node* function *insertAtFirst* which will return the pointer to the new head.
2. We'll pass the current head pointer and the data to insert at the beginning, in the function.
3. Create a new struct Node* pointer *ptr*, and assign it a new memory location in the heap.
4. Assign head to the next member of the ptr structure using ptr-> next = head, and the given data to its data member.
5. Return this pointer *ptr.*

// Insert at first

```
struct Node * insertAtFirst(struct Node *head, int data){
        struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
        ptr->data = data;
        ptr->next = head;
        head = ptr;
        return ptr;
}
```

Insertion in index:

1. Create a struct Node* function *insertAtIndex* which will return the pointer to the head.
2. We'll pass the current head pointer and the data to insert and the index where it will get inserted, in the function.
3. Create a new struct Node* pointer *ptr*, and assign it a new memory location in the heap.

4. Create a new struct Node* pointer pointing to *head*, and run a loop until this pointer reaches the index, where we are inserting a new node.
5. Assign p->next to the next member of the ptr structure using ptr-> next = p->next, and the given data to its data member.
6. Break the connection between p and p->next by assigning p->next the new pointer. That is, p->next = ptr.
7. Return head.

// Insert at index

```
struct Node * insertAtIndex(struct Node *head, int data, int index){
        struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
        struct Node * p = head;
        int i = 0;

        while (i!=index-1)
        {
        p = p->next;
        i++;
        }
        ptr->data = data;
        ptr->next = p->next;
        p->next = ptr;
        return head;
}
```

Insertion at the end:

1. Inserting at the end is very similar to inserting at any index. The difference holds in the limit of the while loop. Here we run a loop until the pointer reaches the end and points to NULL.
2. Assign NULL to the next member of the new ptr structure using ptr-> next = NULL, and the given data to its data member.
3. Break the connection between p and NULL by assigning p->next the new pointer. That is, p->next = ptr.
4. Return head.

// Insert at last

```
struct Node * insertAtEnd(struct Node *head, int data){
        struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
        ptr->data = data;
        struct Node * p = head;
```

```c
        while(p->next!=NULL){
        p = p->next;
        }
        p->next = ptr;
        ptr->next = NULL;
        return head;
}
```

Insertion after a mentioned node:

1. Here, we already have a struct Node* pointer to insert the new node just next to it.
2. Create a struct Node* function *insertAfterNode* which will return the pointer to the head.
3. Pass into this function, the head node, the previous node, and the data.
4. Create a new struct Node* pointer *ptr*, and assign it a new memory location in the heap.
5. Since we already have a struct Node* *prevNode* given as a parameter, use it as p we had in the previous functions.
6. Assign prevNode->next to the next member of the ptr structure using ptr-> next = prevNode->next, and the given data to its data member.
7. Break the connection between prevNode and prevNode->next by assigning prevNode->next the new pointer. That is, prevNode->next = ptr.
8. Return head.

// Insert after node

```c
struct Node * insertAfterNode(struct Node *head, struct Node *prevNode, int data){
        struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
        ptr->data = data;

        ptr->next = prevNode->next;
        prevNode->next = ptr;
        return head;
}
```

Program:

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

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

```
// Printing all nodes
void linkedListTraversal(struct Node *ptr)
{
	while (ptr != NULL)
	{
	printf("Element: %d\n", ptr->data);
	ptr = ptr->next;
	}
}

// Insert at first
struct Node * insertAtFirst(struct Node *head, int data){
	struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
	ptr->data = data;
	ptr->next = head;
	return ptr;
}

// Insert at index
struct Node * insertAtIndex(struct Node *head, int data, int index){
	struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
	struct Node * p = head;
	int i = 0;

	while (i!=index-1)
	{
	p = p->next;
	i++;
	}
	ptr->data = data;
	ptr->next = p->next;
	p->next = ptr;
	return head;
}

// Insert at last
struct Node * insertAtEnd(struct Node *head, int data){
	struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
	ptr->data = data;
	struct Node * p = head;

	while(p->next!=NULL){
	p = p->next;
	}
```

```c
	p->next = ptr;
	ptr->next = NULL;
	return head;
}

// Insert after node
struct Node * insertAfterNode(struct Node *head, struct Node *prevNode, int data){
	struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
	ptr->data = data;

	ptr->next = prevNode->next;
	prevNode->next = ptr;


	return head;
}


int main(){
	struct Node *head;
	struct Node *second;
	struct Node *third;
	struct Node *fourth;

	// Allocate memory for nodes in the linked list in Heap
	head = (struct Node *)malloc(sizeof(struct Node));
	second = (struct Node *)malloc(sizeof(struct Node));
	third = (struct Node *)malloc(sizeof(struct Node));
	fourth = (struct Node *)malloc(sizeof(struct Node));

	// Link first and second nodes
	head->data = 2;
	head->next = second;

	// Link second and third nodes
	second->data = 3;
	second->next = third;

	// Link third and fourth nodes
	third->data = 4;
	third->next = fourth;

	// Terminate the list at the third node
	fourth->data = 5;
```

```
    fourth->next = NULL;

    printf("Linked list after creating\n");
    linkedListTraversal(head);
    // head = insertAtFirst(head, 1);
    head = insertAtIndex(head, 17, 3);
    // head = insertAtEnd(head, 90);
    // head = insertAfterNode(head, second, 54);
    printf("\nLinked list after inserting\n");
    linkedListTraversal(head);
    return 0;
}
```

all the cases:

1. Deleting the first node -> Time complexity: O(1)
2. Deleting a node in between -> Time complexity: O(n)
3. Deleting the last node -> Time complexity: O(n)
4. Deleting the element with a given value from the linked list -> Time complexity: O(n)

Deleting the first node :

1. Create a struct Node* function *deleteFirst* which will return the pointer to the new head after deleting the current head.
2. We'll pass the current head pointer in the function.
3. Create a new struct Node* pointer *ptr*, and make it point to the current head.
4. Assign head to the next member of the list, by head = head->next, because this is going to be the new head of the linked list.
5. Free the pointer *ptr.* And return the head.

```
// Case 1: Deleting the first element from the linked list
struct Node * deleteFirst(struct Node * head){
    struct Node * ptr = head;
    head = head->next;
    free(ptr);
    return head;
}
```

Deleting a node in between:

1. Create a struct Node* function *deleteAtIndex* which will return the pointer to the head.
2. In the function, we'll pass the current head pointer and the index where the node is to be deleted.

3. Create a new struct Node* pointer *p* pointing to *head*.
4. Create a new struct Node* pointer q pointing to *head->next*, and run a loop until this pointer reaches the index, from where we are deleting the node.
5. Assign q->next to the next member of the p structure using p-> next = q->next.
6. Free the pointer q, because it has zero connections with the list now.
7. Return head.

```
// Case 2: Deleting the element at a given index from the linked list
struct Node * deleteAtIndex(struct Node * head, int index){
        struct Node *p = head;
        struct Node *q = head->next;
        for (int i = 0; i < index-1; i++)
        {
        p = p->next;
        q = q->next;
        }

        p->next = q->next;
        free(q);
        return head;
}
```

Deleting the last node :

1. Deleting the last node is quite similar to deleting from any other index. The difference holds in the limit of the while loop. Here we run a loop until the pointer reaches the end and points to NULL.
2. Assign NULL to the next member of the p structure using p-> next = NULL.
3. Break the connection between q and NULL by freeing the ptr q.
4. Return head.

```
// Case 3: Deleting the last element
struct Node * deleteAtLast(struct Node * head){
        struct Node *p = head;
        struct Node *q = head->next;
        while(q->next !=NULL)
        {
        p = p->next;
        q = q->next;
        }

        p->next = NULL;
        free(q);
```

```
    return head;
}
```

Deleting the element with a given value from the linked list :

1. Here, we already have a value that needs to be deleted from the list. The main thing is that we'll delete only the first occurrence.
2. Create a struct Node* function *deleteByValue* which will return the pointer to the head.
3. Pass into this function the head node, the value which needs to be deleted.
4. Create a new struct Node* pointer *p* pointing to the head.
5. Create another struct Node* pointer q pointing to the next of head.
6. Run a while loop until the pointer q encounters the given value or the list finishes.
7. If it encounters the value, delete that node by making p point the next node, skipping the node q. And free q from memory.
8. And if the list just finishes, it means there was no such value in the list. Continue without doing anything.
9. Return head.

// Case 4: Deleting the element with a given value from the linked list

```
struct Node * deleteByValue(struct Node * head, int value){
    struct Node *p = head;
    struct Node *q = head->next;
    while(q->data!=value && q->next!= NULL)
    {
    p = p->next;
    q = q->next;
    }

    if(q->data == value){
    p->next = q->next;
    free(q);
    }
    return head;
}
```

Program:

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

```
#include <stdlib.h>

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

void linkedListTraversal(struct Node *ptr)
{
	while (ptr != NULL)
	{
	printf("Element: %d\n", ptr->data);
	ptr = ptr->next;
	}
}

// Case 1: Deleting the first element from the linked list
struct Node * deleteFirst(struct Node * head){
	struct Node * ptr = head;
	head = head->next;
	free(ptr);
	return head;
}

// Case 2: Deleting the element at a given index from the linked list
struct Node * deleteAtIndex(struct Node * head, int index){
	struct Node *p = head;
	struct Node *q = head->next;
	for (int i = 0; i < index-1; i++)
	{
	p = p->next;
	q = q->next;
	}

	p->next = q->next;
	free(q);
	return head;
}

// Case 3: Deleting the last element
struct Node * deleteAtLast(struct Node * head){
	struct Node *p = head;
	struct Node *q = head->next;
```

```
        while(q->next !=NULL)
        {
        p = p->next;
        q = q->next;
        }

        p->next = NULL;
        free(q);
        return head;
}


// Case 4: Deleting the element with a given value from the linked list
struct Node * deleteByValue(struct Node * head, int value){
    struct Node *p = head;
    struct Node *q = head->next;
    while(q->data!=value && q->next!= NULL)
    {
        p = p->next;
        q = q->next;
    }

    if(q->data == value){
        p->next = q->next;
        free(q);
    }
    return head;
}

int main()
{
        struct Node *head;
        struct Node *second;
        struct Node *third;
        struct Node *fourth;

        // Allocate memory for nodes in the linked list in Heap
        head = (struct Node *)malloc(sizeof(struct Node));
        second = (struct Node *)malloc(sizeof(struct Node));
        third = (struct Node *)malloc(sizeof(struct Node));
        fourth = (struct Node *)malloc(sizeof(struct Node));

        // Link first and second nodes
        head->data = 2;
```

```
    head->next = second;

    // Link second and third nodes
    second->data = 3;
    second->next = third;

    // Link third and fourth nodes
    third->data = 4;
    third->next = fourth;

    // Terminate the list at the third node
    fourth->data = 5;
    fourth->next = NULL;

    printf("Linked list before deletion\n");
    linkedListTraversal(head);

    // head = deleteFirst(head);
    // For deleting first element of the linked list
    // head = deleteAtIndex(head, 2);
    // head = deleteAtLast(head);
    // head = deleteByValue(head,4);
    head = deleteAtLast(head);
    printf("Linked list after deletion\n");
    linkedListTraversal(head);

    return 0;
}
```