Singly Linked Lists

Introduction

Singly linked lists are a type of a linked list where each node points to the next node in the sequence. It does not have any pointer that points to the previous node. That means we can traverse the list only in forward direction. Figure 1 shows an example of a singly linked list with 4 nodes.

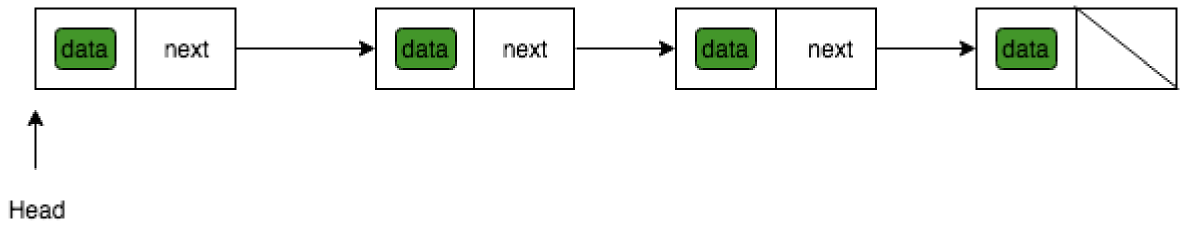

Fig 1: An example of a singly linked list

The first item in the list is pointed by a pointer called head. Sometimes we use another pointer called tail that points to the last item in the list. I am using only head in this tutorial to make it simple.

Operations on a singly linked list

Insert item at the head

Inserting an item at the head of the list requires 3 steps.

1. Create a new node. Insert the item in the data field of the node.
2. Set the new node’s next pointer to the node current head is pointing to.
3. Make the head pointer point to the newly added node.

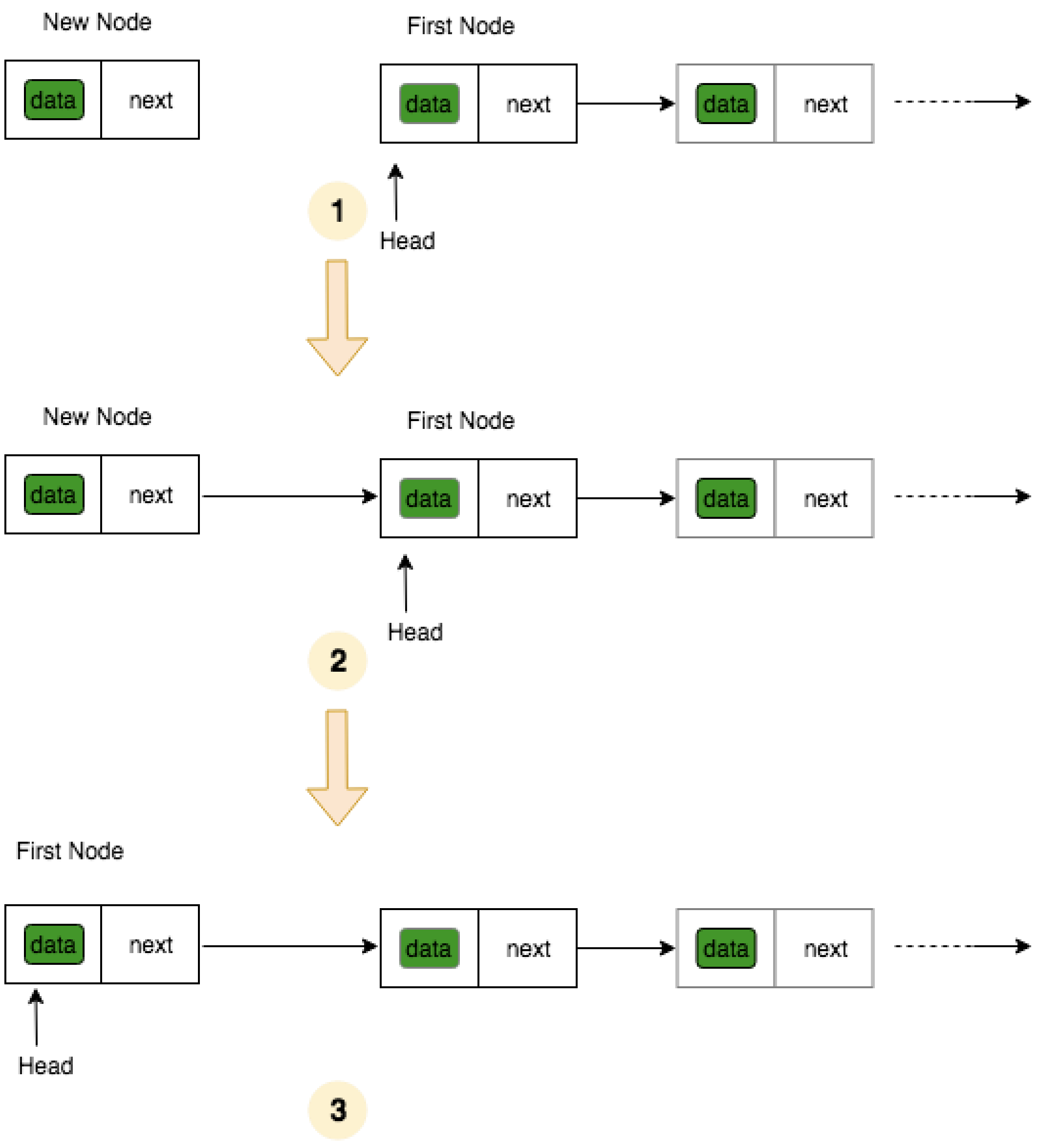

Fig 2: Insertion at the head of the list

Insert an item at the end

To insert an item at the end of the list, use following steps.

1. Seek through the list until the final node is reached.
2. Create a new node using the item to be inserted.
3. Set the last node’s next pointer to the newly created node.
4. Set the next pointer of the new node to null.

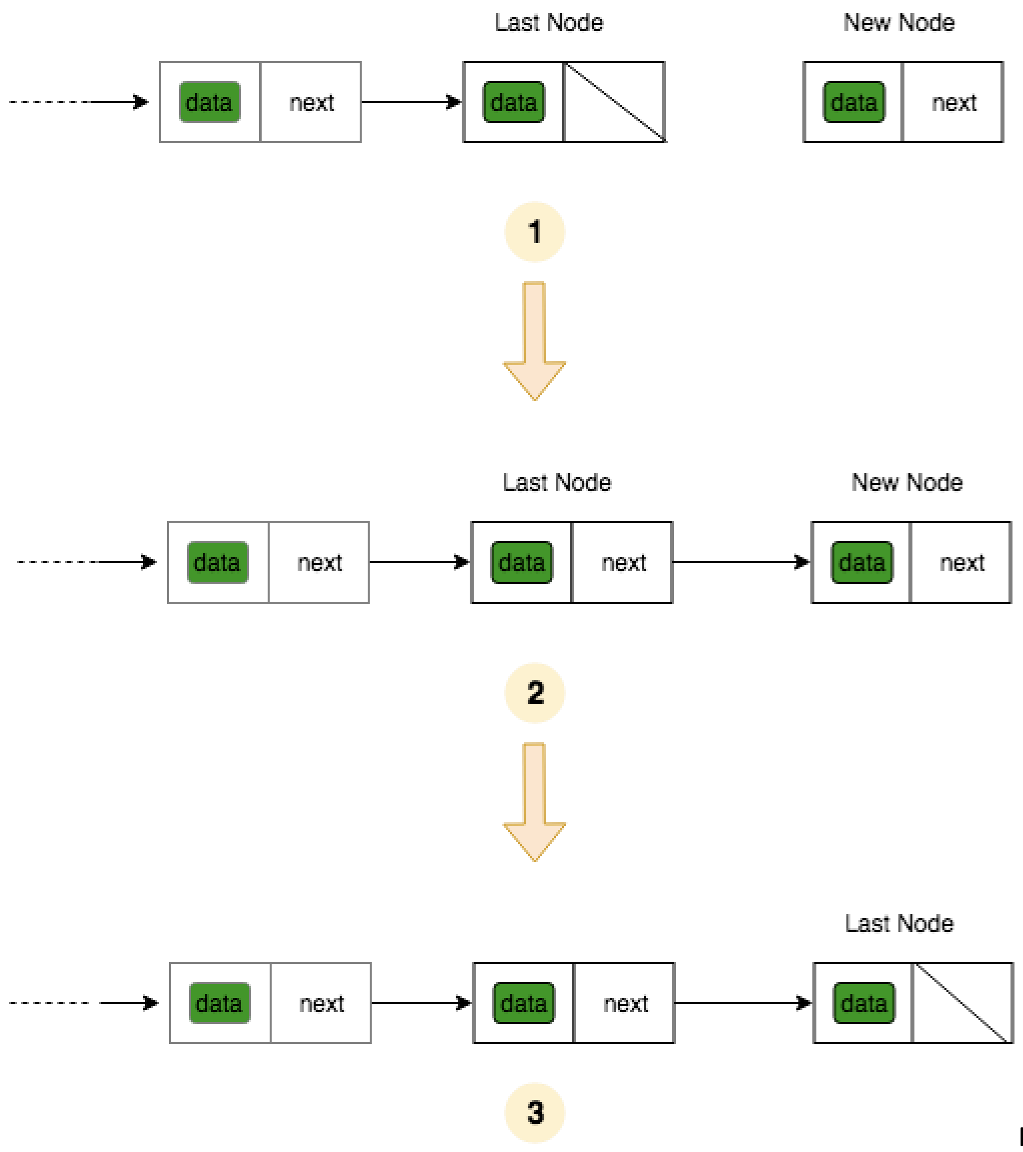

Fig 3: Insertion at the end of the list (Steps 1 and 2 are merged together in step 1)

Insert item after another item

To insert an item anywhere between the first and the last node, use the following steps.

1. Seek through the list until the desired node N (after which you want to insert the new node) is found.
2. Create a new node using the item to be inserted.
3. Set the new node's next pointer to the node N.
4. Set N's next pointer to the newly created node.

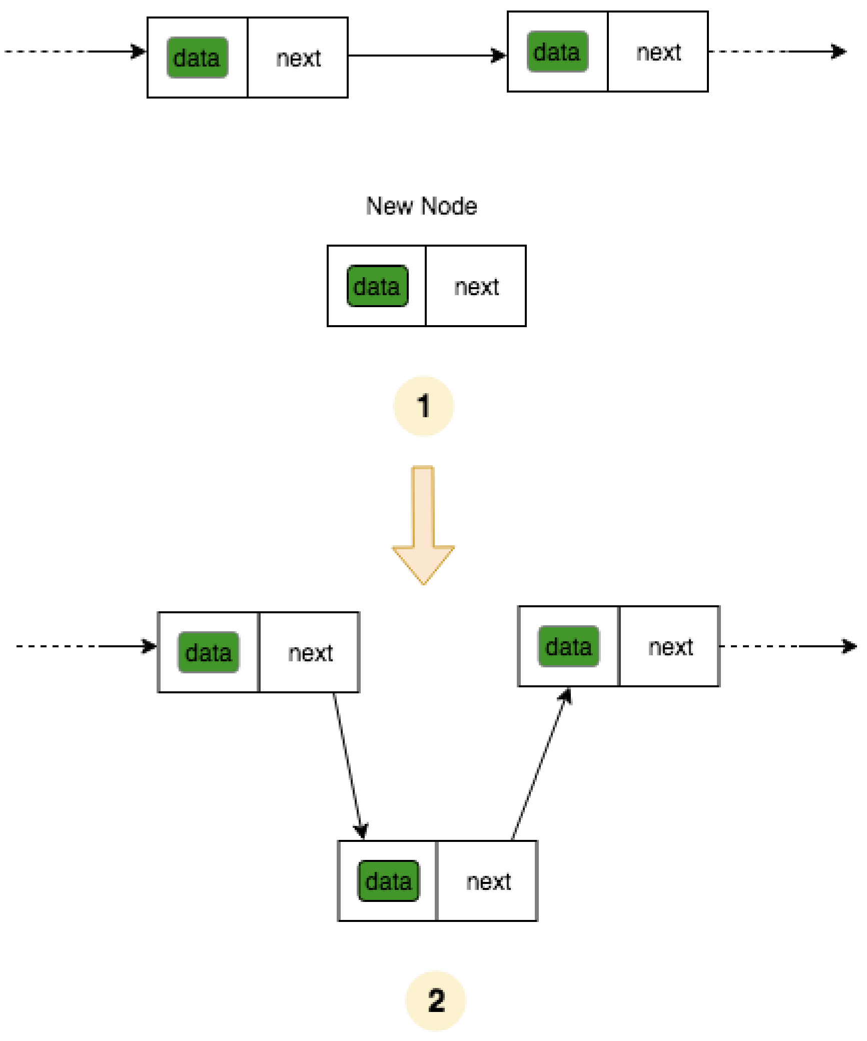

Fig 4: Insertion at the middle (Steps 1, 2 and 3, 4 are merged together)

Searching an item in the list

Searching for an item in the list requires the following step.

1. Start with the head.
2. If the data matches, your search is complete.
3. If the data does not match, go to the next node and repeat step 2.
4. If the next of the item is null, we reach the end of the list and data is not found in the list.

Delete item at the head

For all the delete operations, please keep in mind that, the object needs to be deleted from the heap memory. Languages like Java, Python have Garbage Collector that takes care of this but in C/C++ you need to delete the objects yourself

Use the following steps to delete the item at the head of the linked list.

1. Set the next pointer of the first node to null.
2. Move the current head of the list to the second node.

3. Delete the first node.

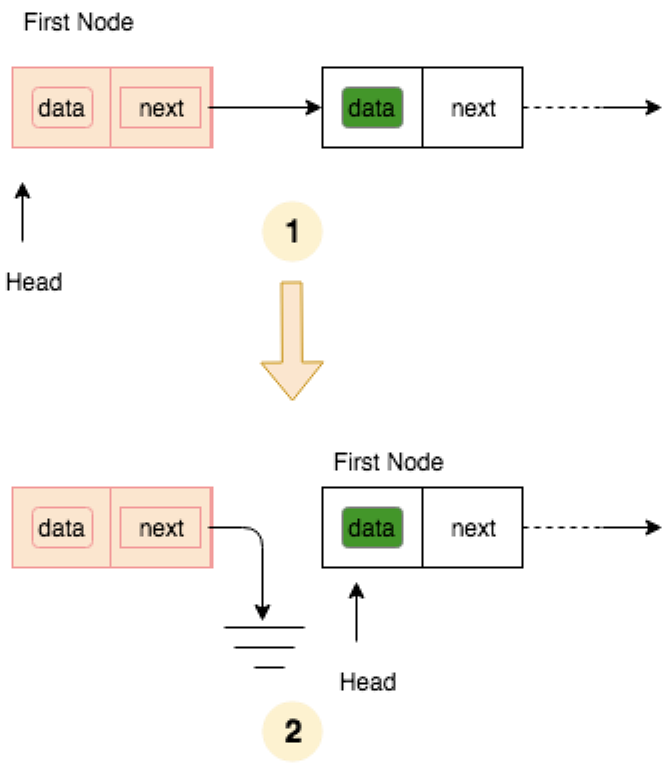

Fig 5: Deletion at the head of the list

Delete item at the end

Use the following steps to delete the item at the end of the list.

1. Seek through the list until you get to the last node (You might need to keep track of the previous node as well).
2. Set the previous node’s next to null.
3. Delete the last node.

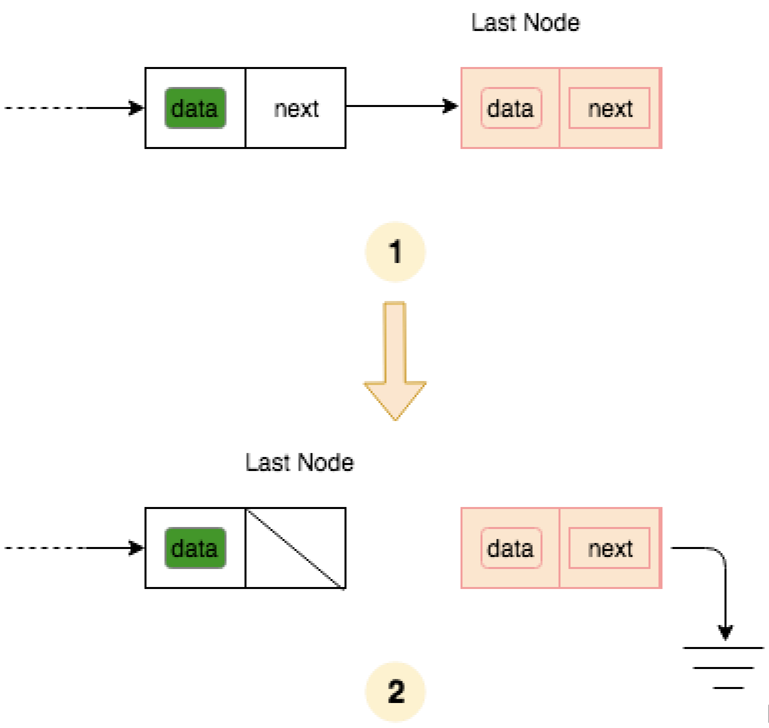

Fig 6: Deletion at the end of the list

Delete item anywhere in the list

1. Search the list until you find the item you are looking for (You might need to keep track of the previous node as well). Let's call this node N.
2. Set N's previous node's next to the N's next node's next.
3. Delete N.

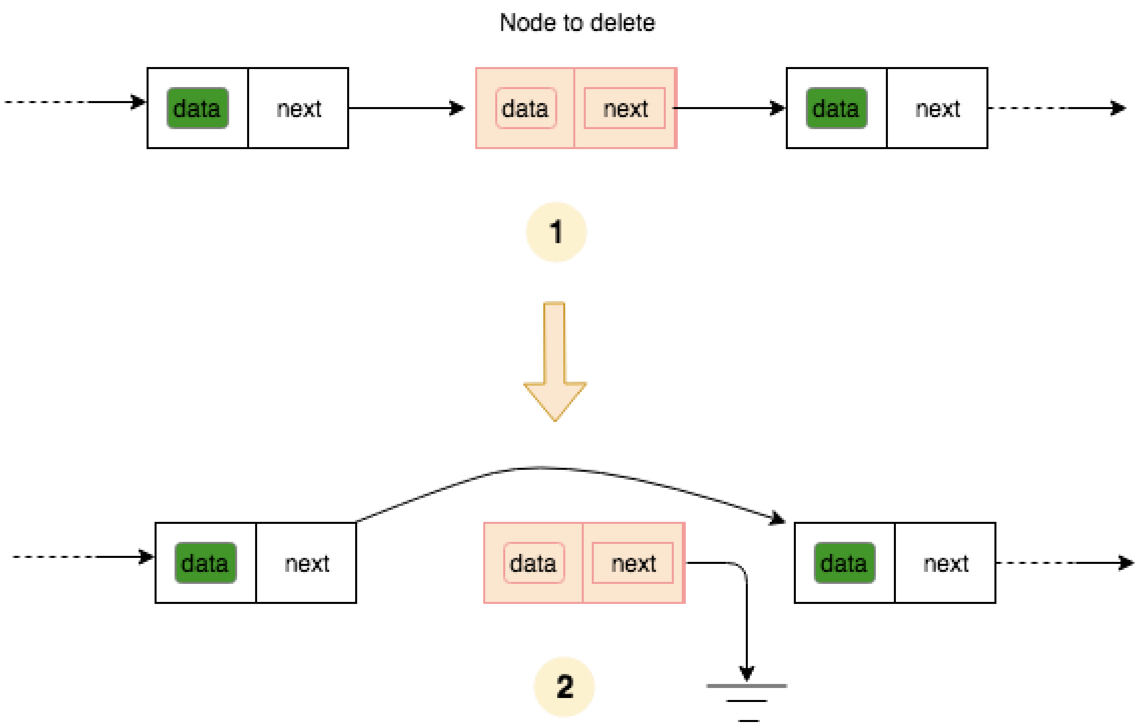

Fig 7: Deletion of node anywhere in the list

Complexities

The complexities of the operations discussed above are summarized in the table below.

Operation	**Complexity**
Insert at the head	O(1)
Insert at the end	O(n)
Insert at the middle	O(n)
Delete at the head	O(1)
Delete at the end	O(n)
Delete at the middle	O(n)
Search the list	O(n)

The complexities given above are for the linked list that has only head pointer. If we have tail pointer then inserting at the end takes only a constant time.

Doubly Linked Lists (With Code in C, C++, Java, and Python)

Introduction

A doubly linked list is a linear data structure where each node has a link to the next node as well as to the previous node. Each component of a doubly linked list has three components.

1. prev: It is a pointer that points to the previous node in the list.
2. data: It holds the actual data.
3. next: It is a pointer that points to the next node in the list.

Since each node has pointers in both the direction, doubly linked list can be traversed in both forward and backward directions. Figure 1 shows an example of a doubly linked list containing 2 items.

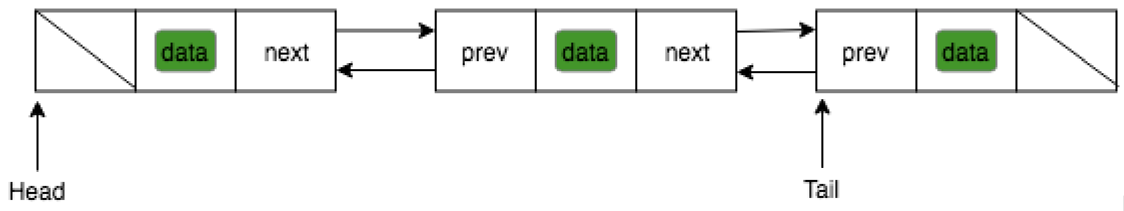

Fig 1: An example of a doubly linked list

The first node is pointed by a pointer called head and the last node is pointed by a pointer called tail. The first node does not have a previous pointer and the last node does not have the next pointer.

Operations on a doubly linked list

Insert at the head

1. Create a new node with the item to be inserted. Initially set both prev and next pointer to null.
2. Set next pointer of the new node to head.
3. Set the prev pointer of head to the new node.
4. Set head pointer to the new node.

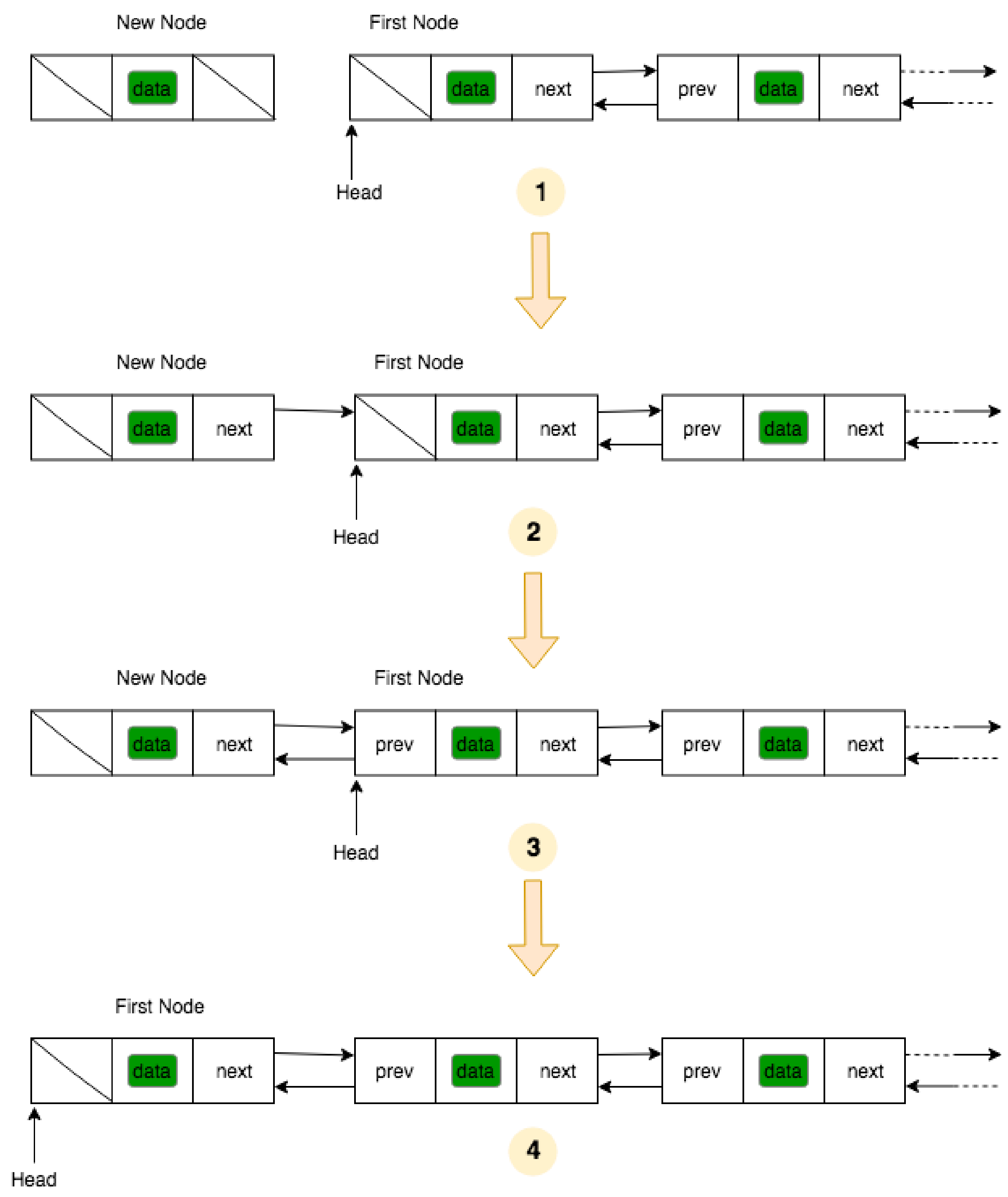

Fig 2: Insertion at the head

Insert at the tail

1. Create a new node with the item to be inserted. Initially set both prev and next pointer to null.
2. Set next pointer of tail to the new node.
3. Set prev pointer of the new node to tail.
4. Set tail pointer to the new node.

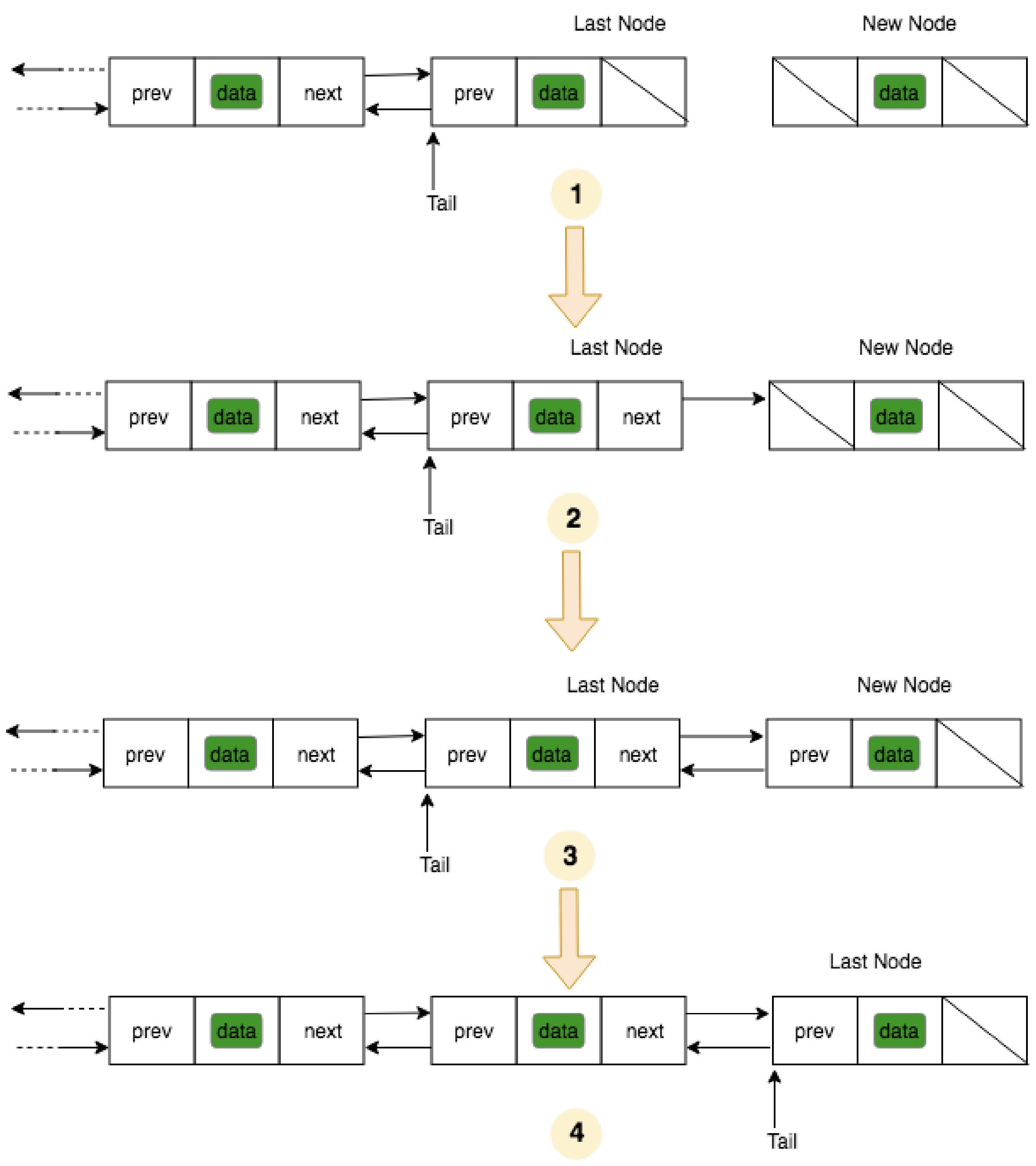

Fig 3: Insertion at the tail

Insert at the middle

1. Create a new node with the item to be inserted. Initially set both prev and next pointer to null.
2. Seek through the list until you find a node after which you want to insert the new node. Call this node D and node next to it N.
3. Set D’s next pointer to the new node and new node’s prev pointer to D.
4. Set the new node’s next pointer to N and N’s prev pointer to the new node.

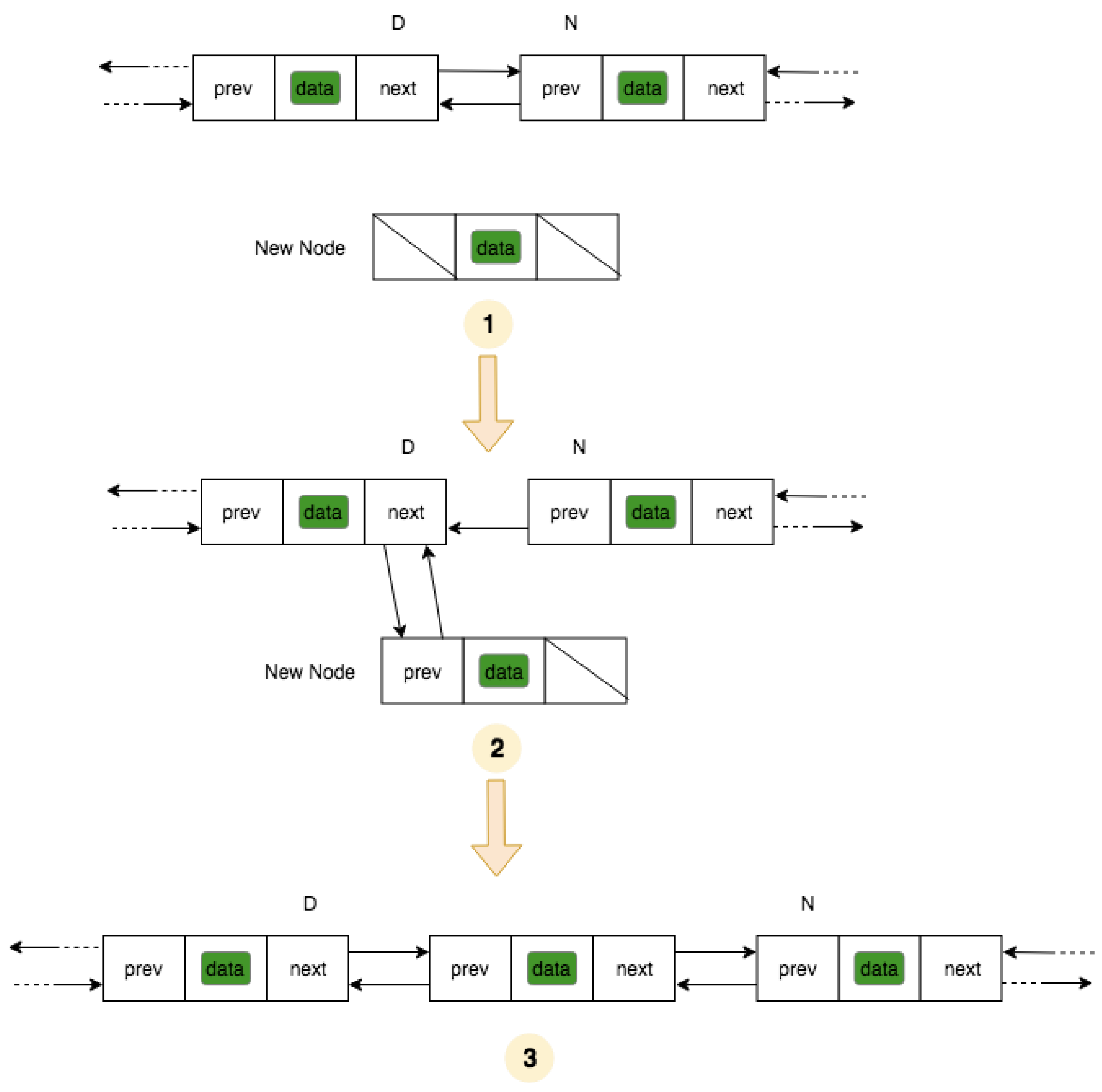

Fig 4: Insertion at the middle (Steps 1 and 2 are combined into 1)

Find the item in the list

1. Start with head or tail. Call it curr.
2. If curr has the item you are looking for, return the item.
3. Proceeds to the next node. If you started with head use the next pointer to go to the next item otherwise use prev pointer.
4. Repeat step 2 and 3 until you get null pointer. In case you get to the null pointer, you have come to the end and item is not on the list.

Remove the item at the head

(Note: for all the delete operations, the node needs to be destroyed in order to free the memory. Some languages like Java, Python, etc has a garbage collector that handles this automatically while other languages like C, C++ does not have such a mechanism and programmer has to destroy it explicitly).

1. Find the next node of the head and call it N.
2. Set the prev pointer of N to null.
3. Set both the pointers of the head to null.
4. Destroy the node pointed by head and set the head pointer to N.

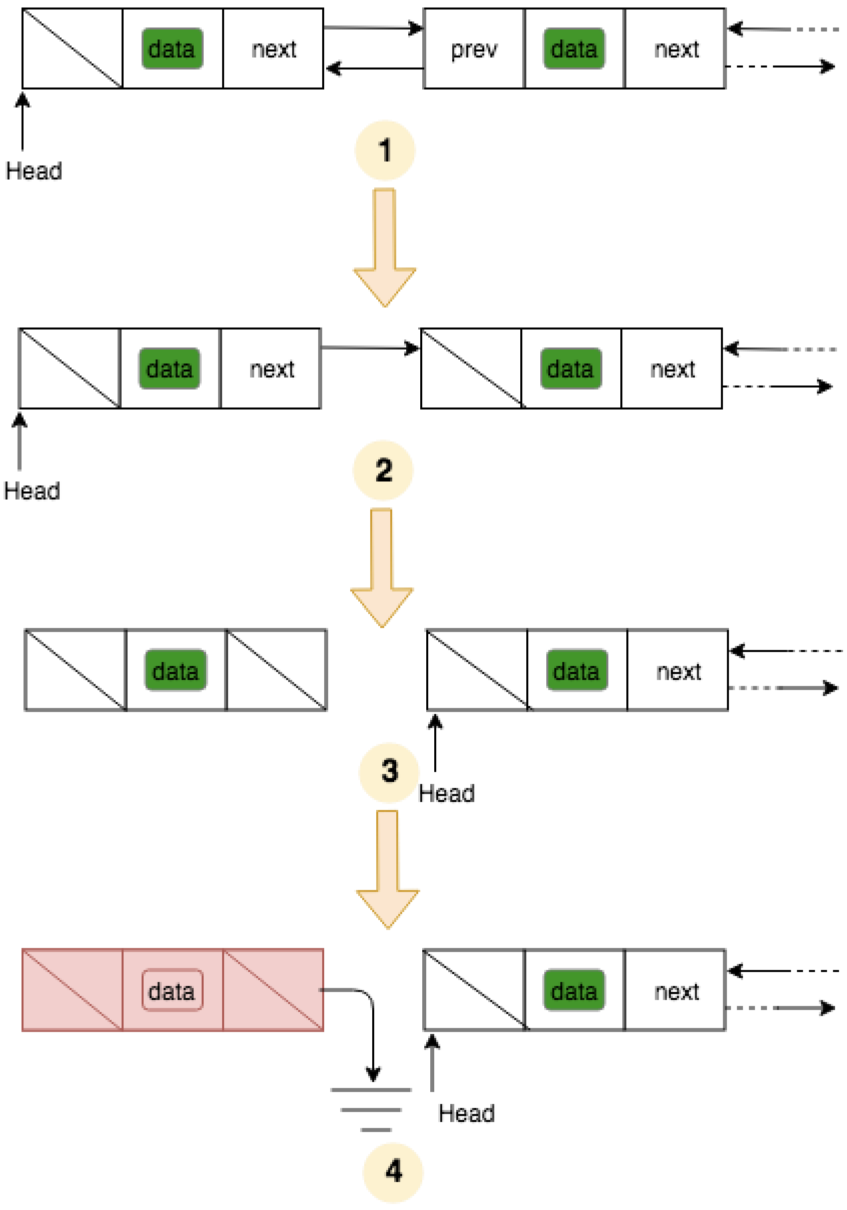

Fig 5: Deletion at the head

Remove the item at the tail

1. Find the previous node of the tail and call it N.
2. Set the next pointer of N to null.
3. Set both the pointers of the tail to null.

4. Destroy the node pointed by tail and set the tail pointer to N.

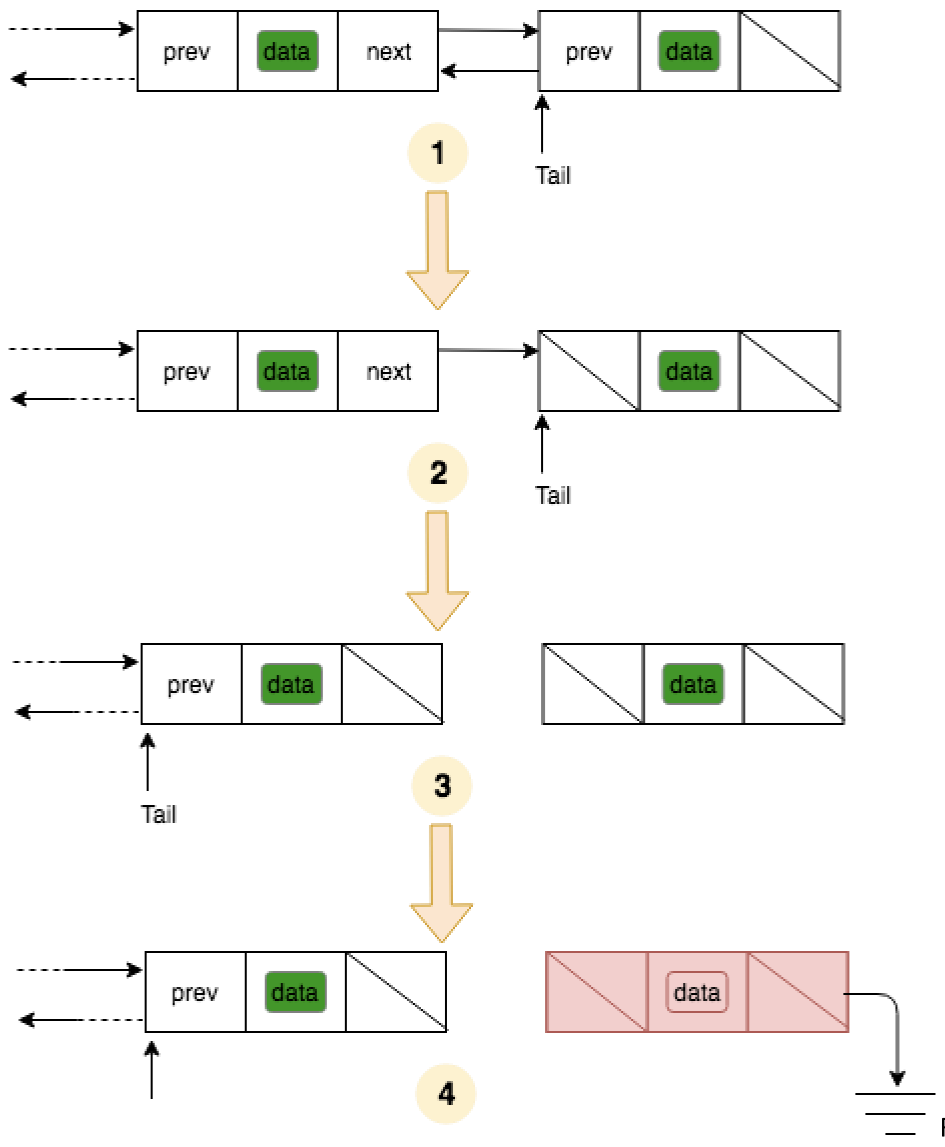

Fig 6: Deletion at the tail

Remove item anywhere in the list

1. Seek through the list until you found the desired node. Call it D.
2. Set D's previous node's next pointer to D's next node and set D's next node's prev pointer to D's previous node.
3. Set both the pointers of D to null.
4. Destroy the node pointed by D.

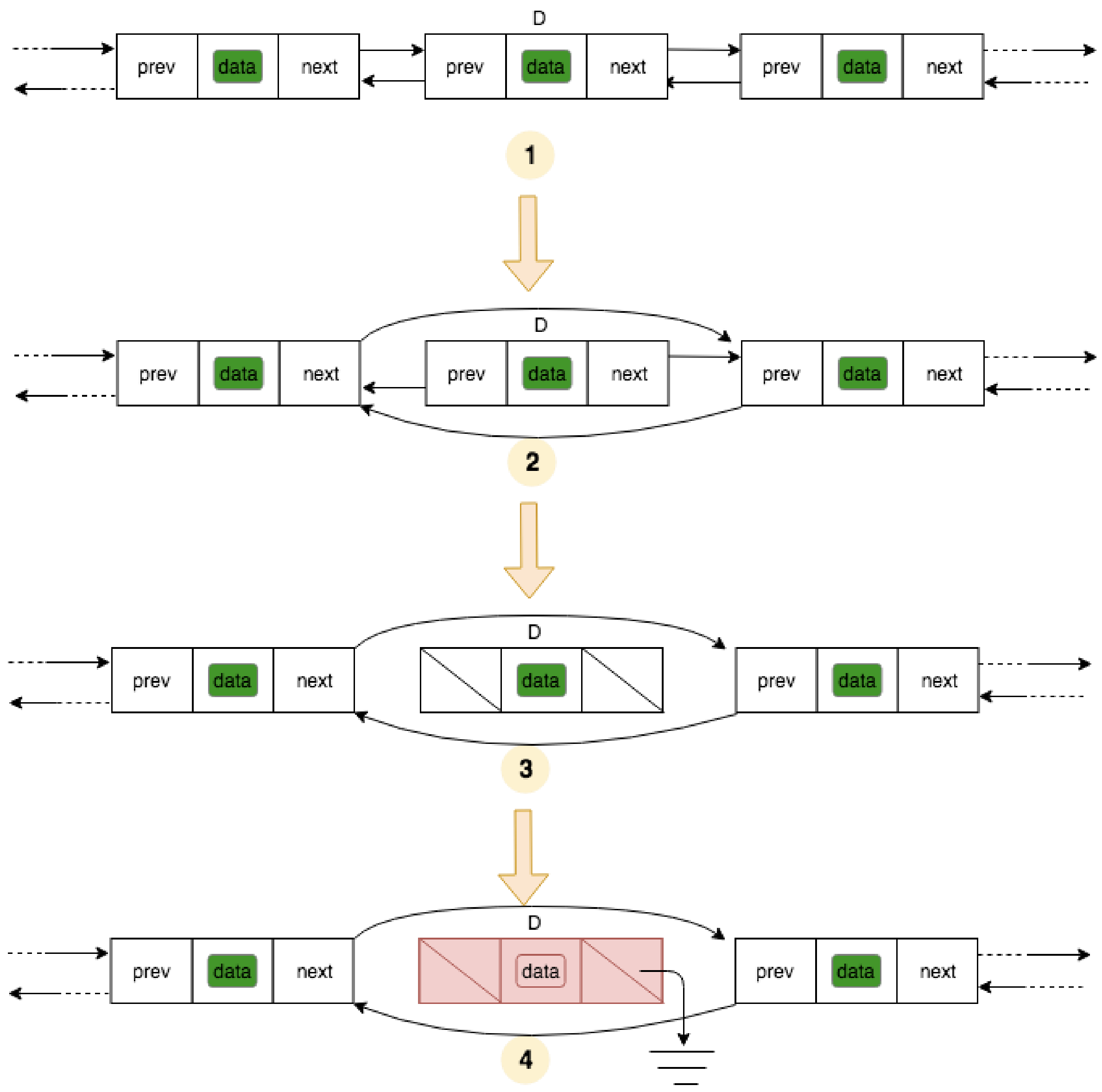

Fig 7: Deletion at the middle

Complexities

The table below shows the operation and corresponding complexity of all the operations discussed above.

Operation	**Complexity**
Insert at the head	O(1)
Insert at the tail	O(1)
Insert at the middle	O(n)

Operation	Complexity
Delete at the head	O(1)
Delete at the tail	O(1)
Delete at the middle	O(n)
Search the list	O(n)

Linked List Operations: Traverse, Insert and Delete

There are various linked list operations that allow us to perform different actions on linked lists. For example, the insertion operation adds a new element to the linked list.

Here's a list of basic linked list operations that we will cover in this article.

- Traversal - access each element of the linked list
- Insertion - adds a new element to the linked list
- Deletion - removes the existing elements
- Search - find a node in the linked list
- Sort - sort the nodes of the linked list

Before you learn about linked list operations in detail, make sure to know about Linked List first.

Things to Remember about Linked List

- `head` points to the first node of the linked list
- `next` pointer of the last node is `NULL`, so if the next current node is `NULL`, we have reached the end of the linked list.

In all of the examples, we will assume that the linked list has three nodes `1 --->2 --->3` with node structure as below:

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

Traverse a Linked List

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When `temp` is `NULL`, we know that we have reached the end of the linked list so we get out of the while loop.

```
struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL) {
  printf("%d --->",temp->data);
  temp = temp->next;
}
```

The output of this program will be:

```
List elements are -
1 --->2 --->3 --->
```

Insert Elements to a Linked List

You can add elements to either the beginning, middle or end of the linked list.

1. Insert at the beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

```c
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
```

2. Insert at the End

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

```c
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;

struct node *temp = head;
while(temp->next != NULL){
  temp = temp->next;
}

temp->next = newNode;
```

3. Insert at the Middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

```c
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;

struct node *temp = head;
```

```
for(int i=2; i < position; i++) {
  if(temp->next != NULL) {
    temp = temp->next;
  }
}
newNode->next = temp->next;
temp->next = newNode;
```

Delete from a Linked List

You can delete either from the beginning, end or from a particular position.

1. Delete from beginning

- Point head to the second node

```
head = head->next;
```

2. Delete from end

- Traverse to second last element
- Change its next pointer to null

```
struct node* temp = head;
while(temp->next->next!=NULL){
  temp = temp->next;
}
temp->next = NULL;
```

3. Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++) {
  if(temp->next!=NULL) {
    temp = temp->next;
  }
}

temp->next = temp->next->next;
```

Search an Element on a Linked List

You can search an element on a linked list using a loop using the following steps. We are finding `item` on a linked list.

- Make `head` as the `current` node.
- Run a loop until the `current` node is `NULL` because the last element points to `NULL`.
- 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`.

```
// Search a node
bool searchNode(struct Node** head_ref, int key) {
  struct Node* current = *head_ref;

  while (current != NULL) {
    if (current->data == key) return true;
      current = current->next;
  }
  return false;
}
```

Sort Elements of a Linked List

We will use a simple sorting algorithm, Bubble Sort, to sort the elements of a linked list in ascending order below.

1. Make the `head` as the `current` node and create another node `index` for later use.
2. If `head` is null, return.
3. Else, run a loop till the last node (i.e. `NULL`).
4. In each iteration, follow the following step 5-6.
5. Store the next node of `current` in `index`.
6. Check if the data of the current node is greater than the next node. If it is greater, swap `current` and `index`.

Check the article on bubble sort for better understanding of its working.

```
// Sort the linked list
void sortLinkedList(struct Node** head_ref) {
  struct Node *current = *head_ref, *index = NULL;
  int temp;

  if (head_ref == NULL) {
    return;
  } else {
    while (current != NULL) {
      // index points to the node next to current
      index = current->next;

        while (index != NULL) {
        if (current->data > index->data) {
          temp = current->data;
          current->data = index->data;
          index->data = temp;
           }
           index = index->next;
        }
        current = current->next;
    }
  }
}
```