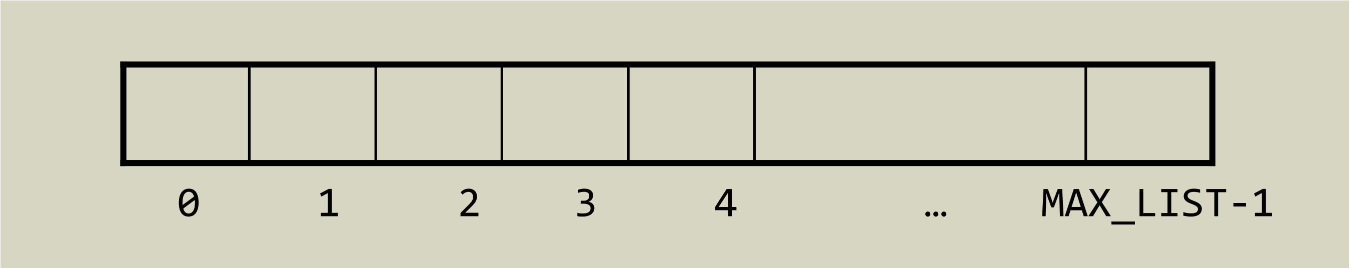
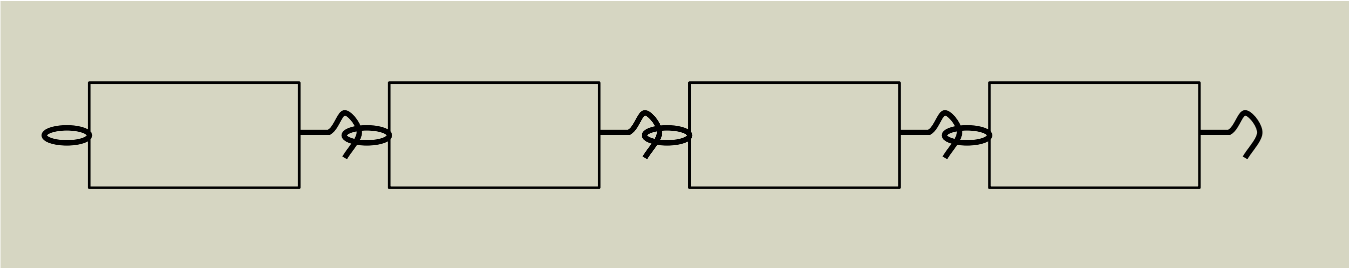
General List

What is List?

- Definition
  - A linear collection of storing elements of the same types.
  - May have duplicate elements.
  - Implemented as either an **array** or a **linked list**.





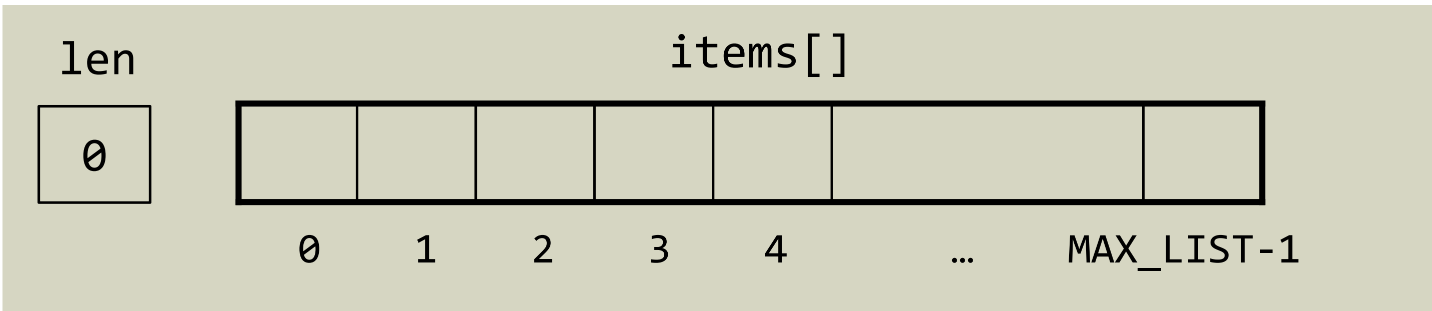
What is List?

- Operations
  - **InitList**: Make a list empty.
  - **IsEmpty**: Check whether the list is empty.
  - **IsFull**: Check whether the list is full.
  - **Insertion**: Insert an item at the specific position.
    - **InsertFirst**: Insert an item at the first position.
    - **InsertLast**: Insert an item at the last position.
    - **InsertMiddle**: Insert an item at the k-th position.
  - **Deletion**: Remove an item at the specific position.
    - **DeleteFirst**: Remove an item at the first position.
    - **DeleteLast**: Remove an item at the last position.
    - **DeleteMiddle**: Remove an item at the k-th position.

  - **Retrieval**: Read or replace an item at the k-th position.
  - **Traversal**: Read each item in a list in sequence.

ArrayList Implementation

- ArrayList representation




```c
#define MAX_LIST 100

typedef enum { false, true } bool;
typedef int Data;

typedef struct {
    Data items[MAX_LIST];
    int len;
} ArrayList;
```

ArrayList Implementation

- Operations (OR Interface)

```
// Make a list empty.
void InitList(ArrayList* plist);
// Check whether the list is empty.
bool IsEmpty(ArrayList* plist);
// Check whether the list is full.
bool IsFull(ArrayList* plist);
// Insert an item at the k-th position.
void InsertMiddle(ArrayList* plist, int pos, Data item);
// Remove an item at the k-th position.
void RemoveMiddle(ArrayList* plist, int pos);
// Read an item at the k-th position.
Data ReadItem(ArrayList* plist, int pos);
// Print each item in a list in sequence.
void PrintList(ArrayList* plist);
```

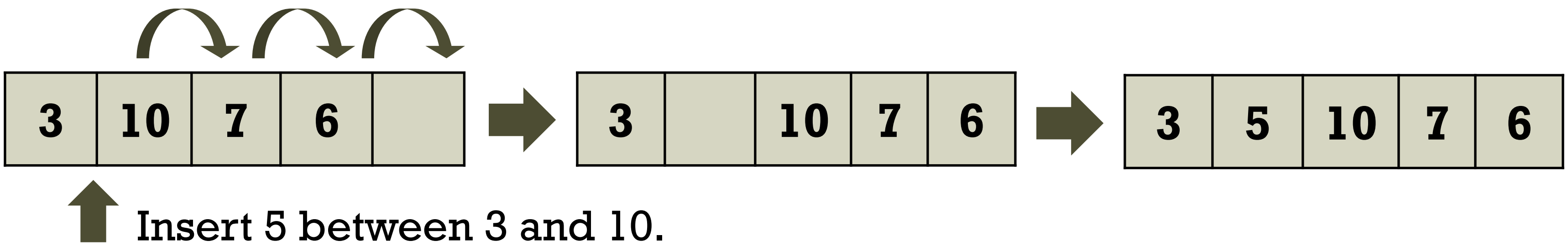
Initializing ArrayList

- InitList, IsEmpty, and IsFull operations

```
// Make a list empty.
void InitList(ArrayList* plist)
{
    plist->len = 0;
}

// Check whether the list is empty.
bool IsEmpty(ArrayList* plist)
{
    return plist->len == 0;
}

// Check whether the list is full.
bool IsFull(ArrayList* plist)
{
    return plist->len == MAX_LIST;
}
```

ArrayList Implementation

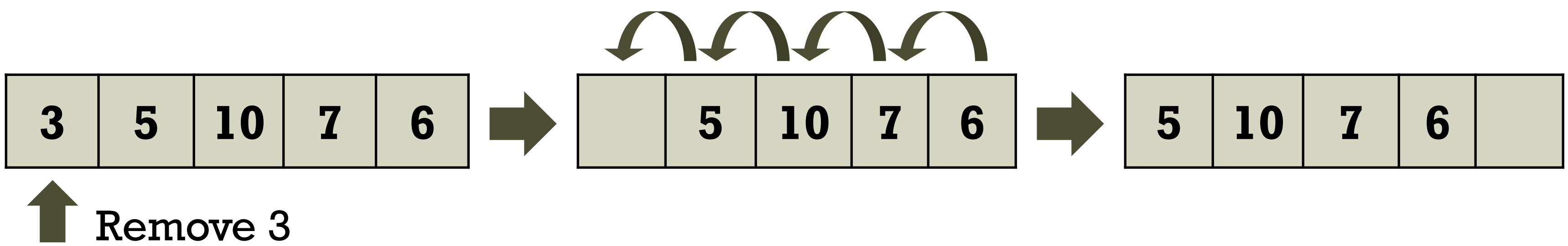
- InsertMiddle operation
  - Shifting right from the k-th position to the last position




```c
// Insert an item at the k-th position.
void InsertMiddle(ArrayList* plist, int pos, Data item)
{
    if (IsFull(plist) || pos < 0 || pos > plist->len)
        exit(1);

    for (int i = plist->len - 1; i >= pos; i--)
        plist->items[i + 1] = plist->items[i];
    plist->items[pos] = item;
    plist->len++;
}
```

ArrayList Implementation

- RemoveMiddle operation
  - Shifting left from the k-th position to the last position




```c
// Remove an item at the k-th position.
void RemoveMiddle(ArrayList* plist, int pos)
{
    if (IsEmpty(plist) || pos < 0 || pos >= plist->len)
        exit(1);

    for (int i = pos; i < plist->len; i++)
        plist->items[i] = plist->items[i + 1];
    plist->len--;
}
```

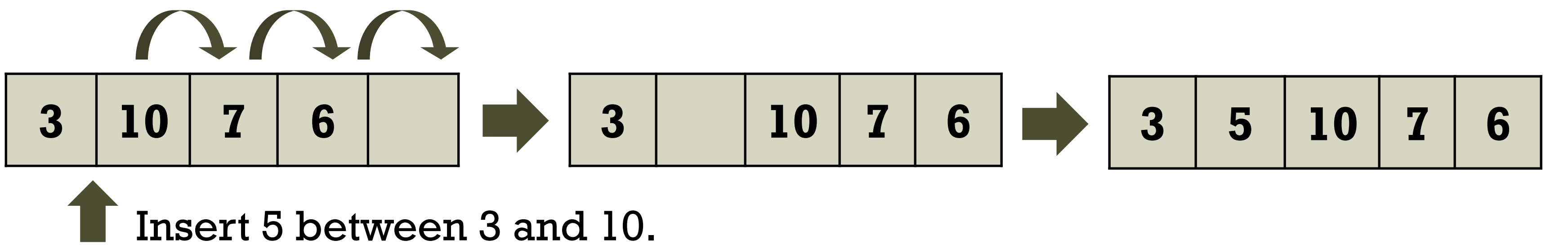
ArrayList Implementation

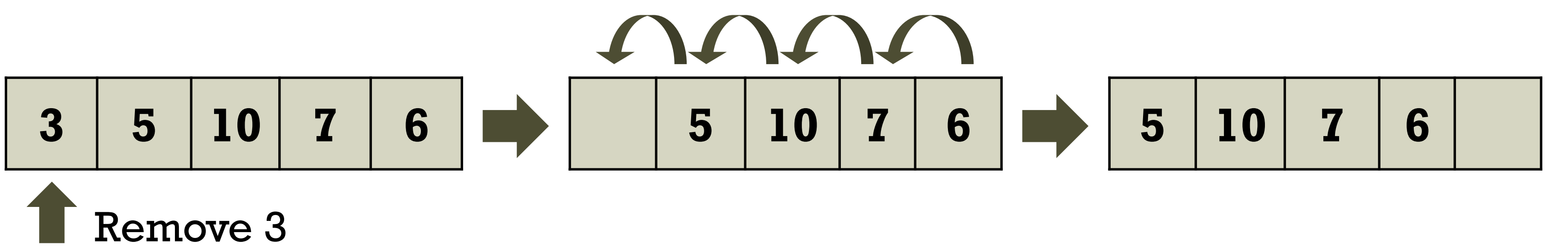
- ReadItem and PrintList operations

```
// Read an item at the k-th position.
Data ReadItem(ArrayList* plist, int pos)
{
    if (IsEmpty(plist) || pos < 0 || pos >= plist->len)
        exit(1);

    return plist->items[pos];
}

// Print each item in a list in sequence.
void PrintList(ArrayList* plist)
{
    for (int i = 0; i < plist->len; i++)
        printf("%d\n", plist->items[i]);
}
```

Weaknesses of ArrayList

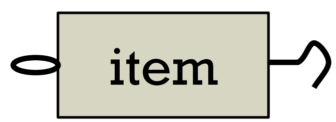
- Pre-defined size
  - The maximum size should be predictable.
- Insertion & deletion are time-consuming.
  - Require O(n) time complexity.



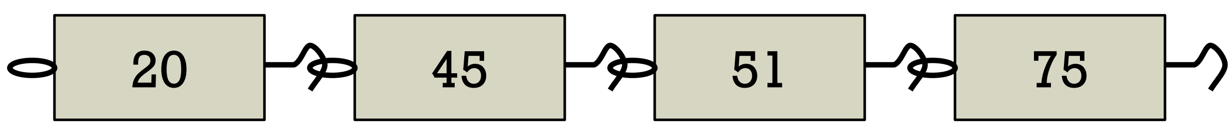


What is Linked List?

- Definition
  - A linear collection of elements, called **nodes**, each pointing to the next node
    - Variable size, easy to change the size while running
    - Easy insertions and deletions
  - Each node consists of an item with link (hook).

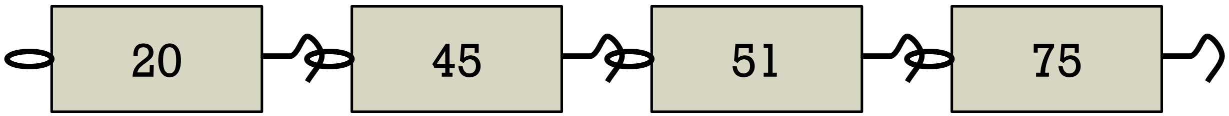


- Example

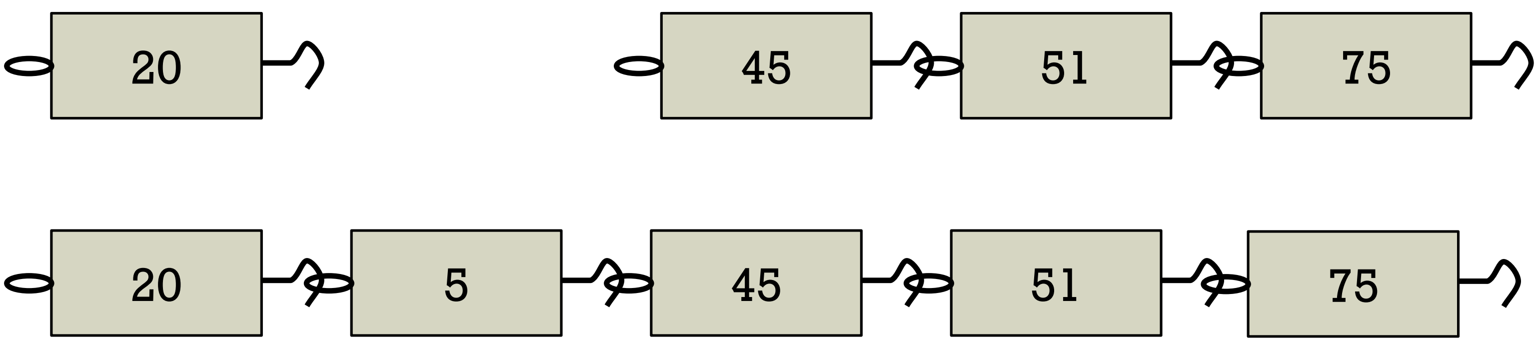


Inserting an Element

- How to insert an element in a linked list?

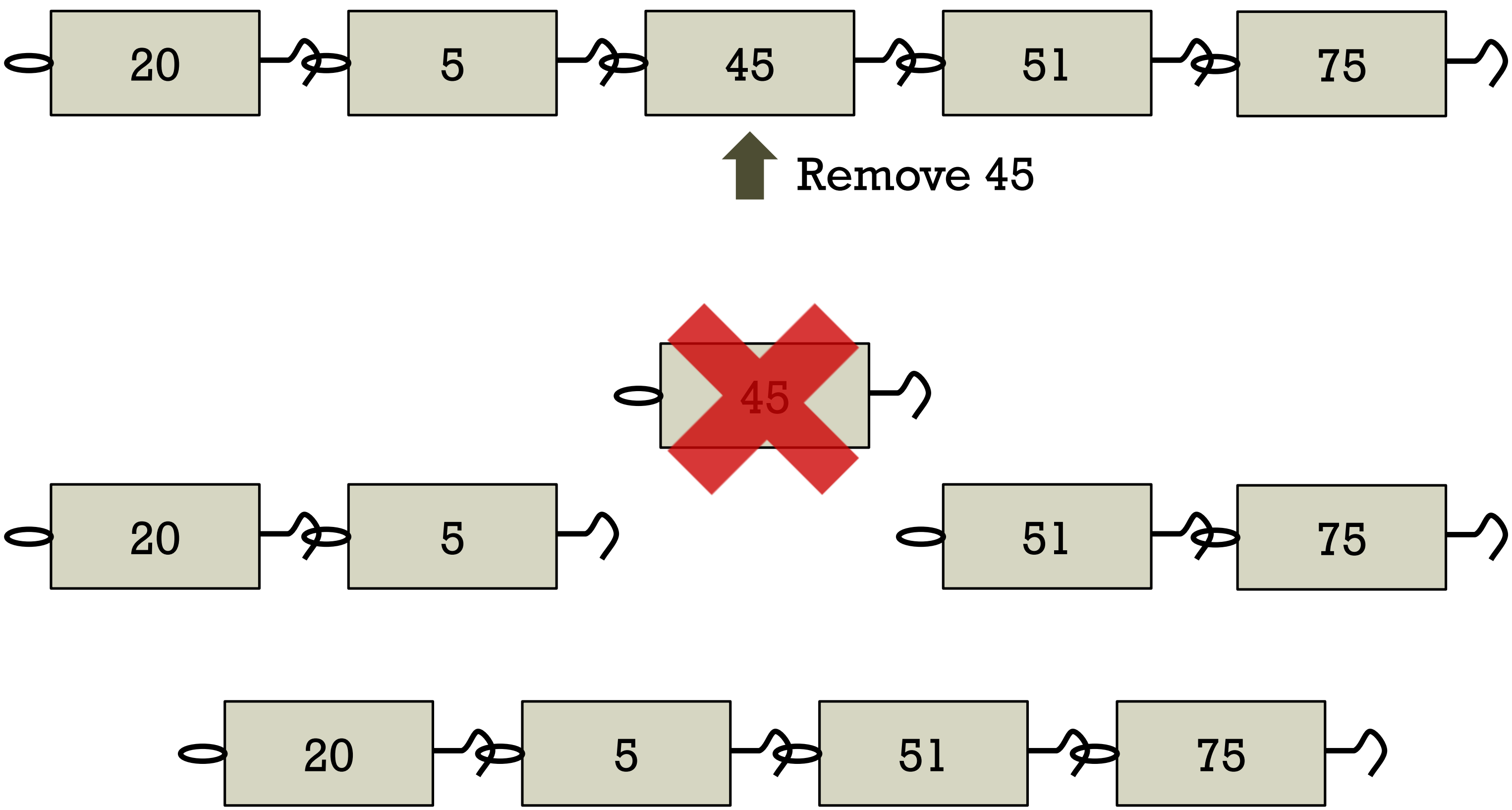


Insert 5 between 20 and 45.



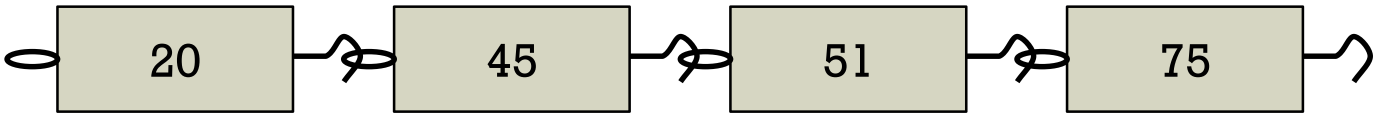
Deleting an Element

- How to delete an element in a linked list?

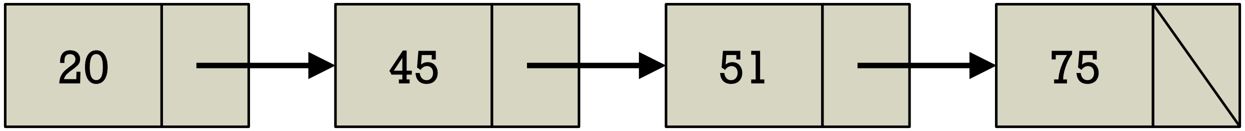


List Representation

- Conceptual representation



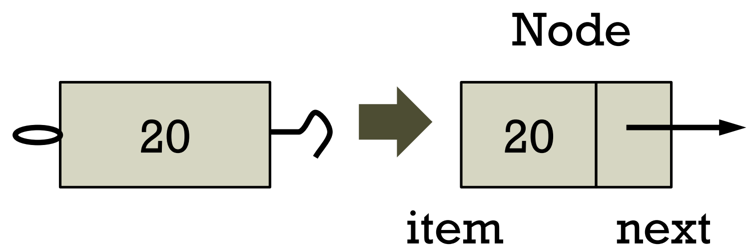
- Arrow and box and representation
  - Box: an item value
  - Arrow: a pointer to the next box
    - If the arrow dose not refer to other nodes, it is a **NULL pointer**.

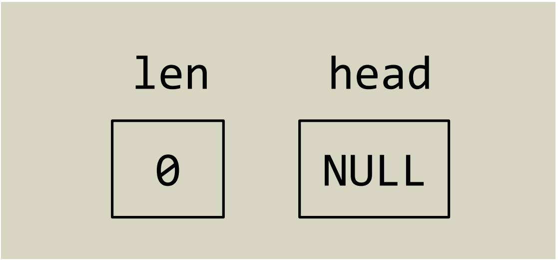


Linked List Implementation

- Representation
  - A node consists of an item and a next pointer.
    - item: a value, next: a pointer to the next node
  - A linked list consist of a head node and the length of items.

```c
typedef enum { false, true } bool;
typedef int Data;

typedef struct _Node
{
    Data item;
    struct _Node* next;
} Node;

typedef struct
{
    Node* head;
    int len;
} LinkedList;
```





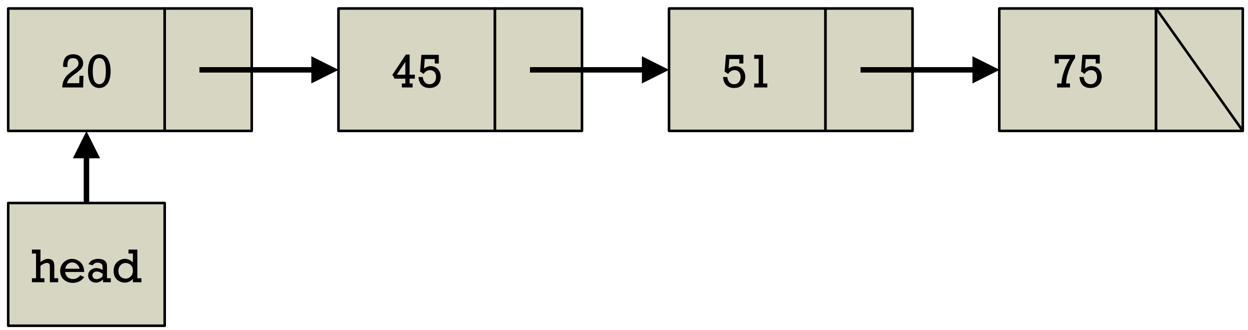
Linked List Implementation

- Operations

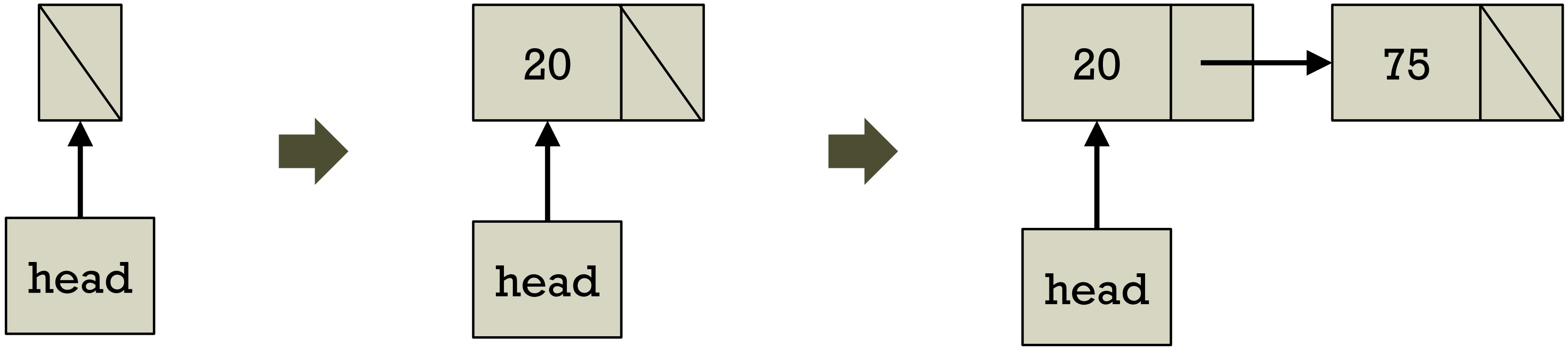
```
// Make a list empty.
void InitList(LinkedList* plist);
// Check whether the list is empty.
bool IsEmpty(LinkedList* plist);
// Insert an item at the k-th position.
void InsertMiddle(LinkedList* plist, int pos, Data item);
// Remove an item at the k-th position.
void RemoveMiddle(LinkedList* plist, int pos);
// Read an item at the k-th position.
Data ReadItem(LinkedList* plist, int pos);
// Print each item in a list in sequence.
void PrintList(LinkedList* plist);
// Remove all nodes in a list in sequence.
void ClearList(LinkedList* plist);
```

Initializing Linked List

- The "head" pointer is necessary.
  - A pointer variable pointing to the first member



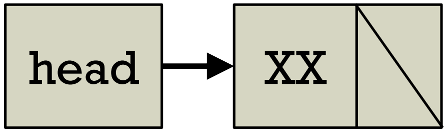
- Why?
  - It is easier to manage nodes for insertions and deletions.



Initializing Linked List

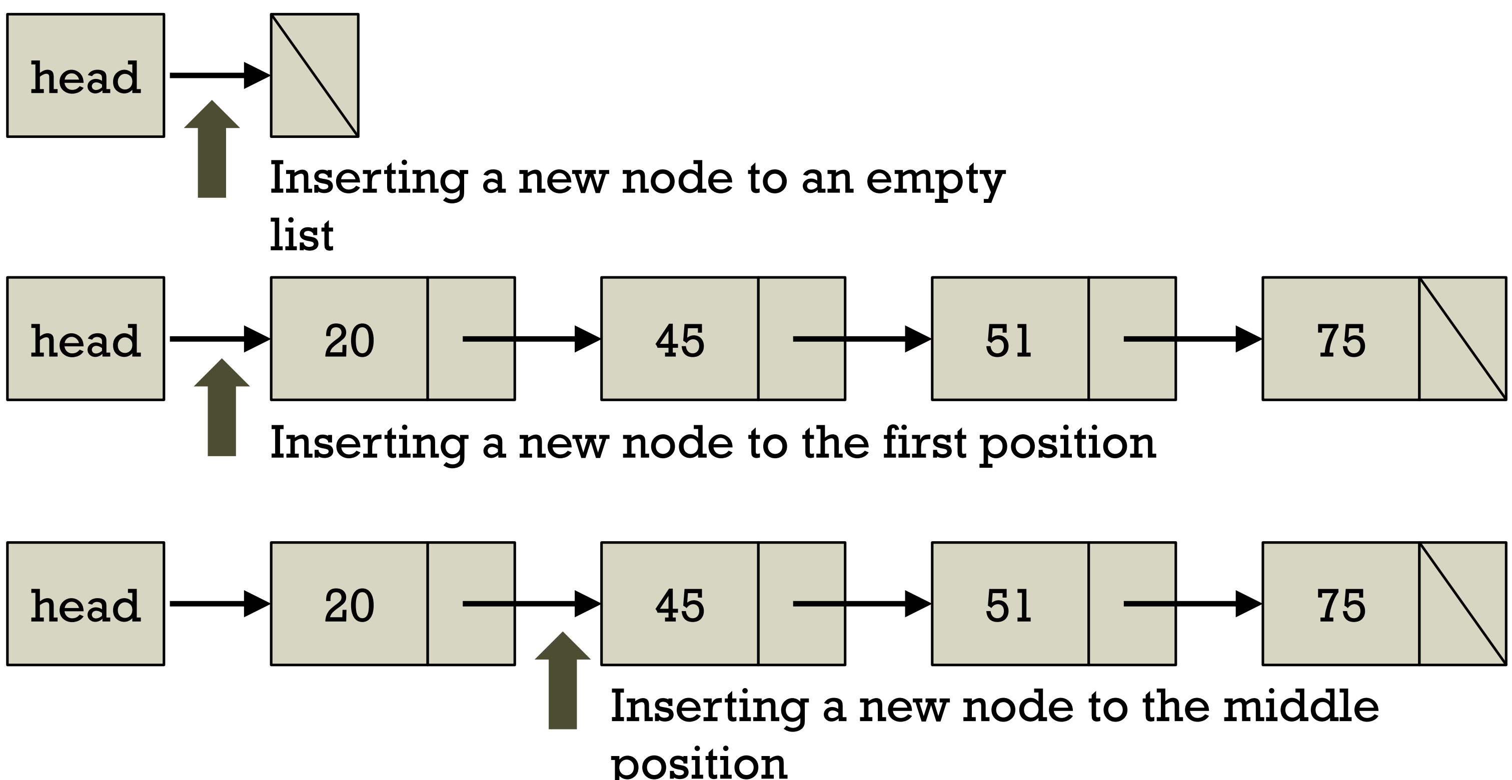
- InitList operation
  - When initializing a list, it first creates a **dummy node**.
  - The dummy node makes insertions and deletions much easier.
    - Particularly useful for **inserting and deleting the first node**.

```c
// Make a list empty.
void InitList(LinkedList* plist)
{
    // Create a dummy node;
    plist->head = (Node *)malloc(sizeof(Node));
    plist->head->next = NULL;
    plist->len = 0;
}

// Check whether the list is empty.
bool IsEmpty(LinkedList* plist)
{
    return plist->len == 0;
}
```



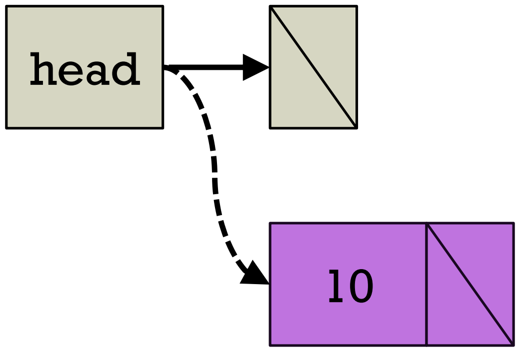
Inserting Nodes in Linked List

- Three cases for insertions
  - Inserting an node to an empty list
  - Inserting the node to the first position, i.e., k = 0
  - Inserting the node to the k-th position, i.e., 0 < k <= len

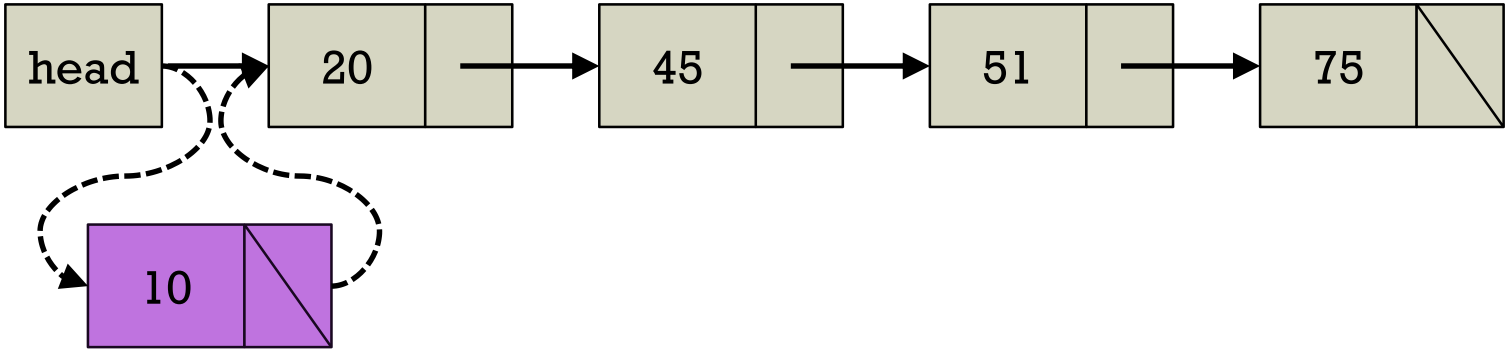


Inserting Nodes in Linked List

- Inserting an node to an empty list
  - Creating a new node.
  - Link the head pointer to the new node.

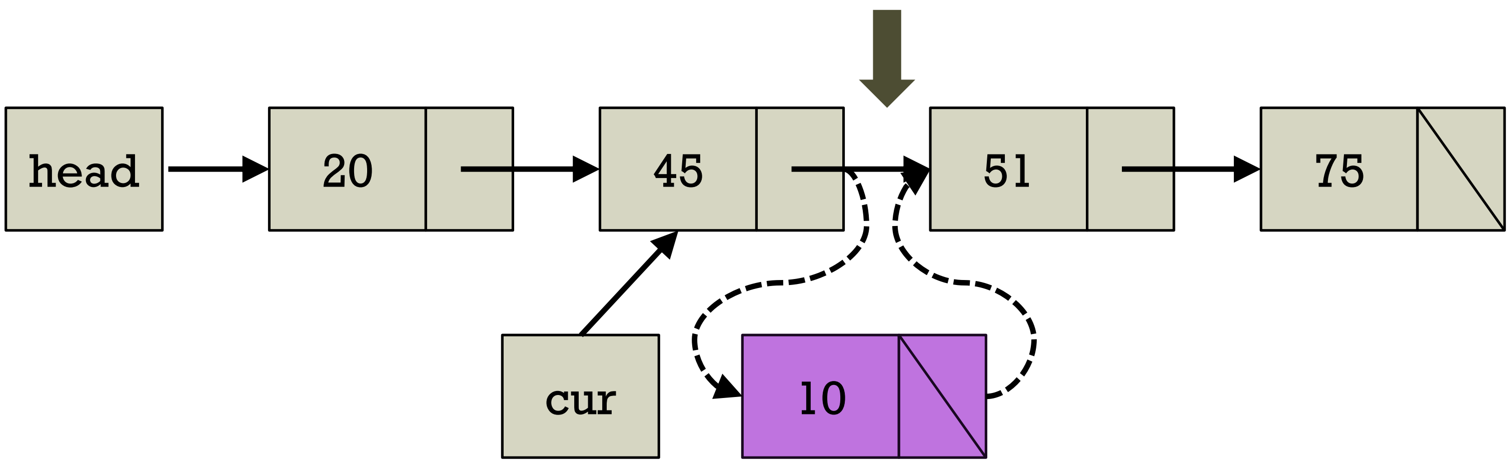


- Inserting the node to the first position
  - Create a new node.
  - Link the new node to the first node.
  - Link the head pointer to the new node.



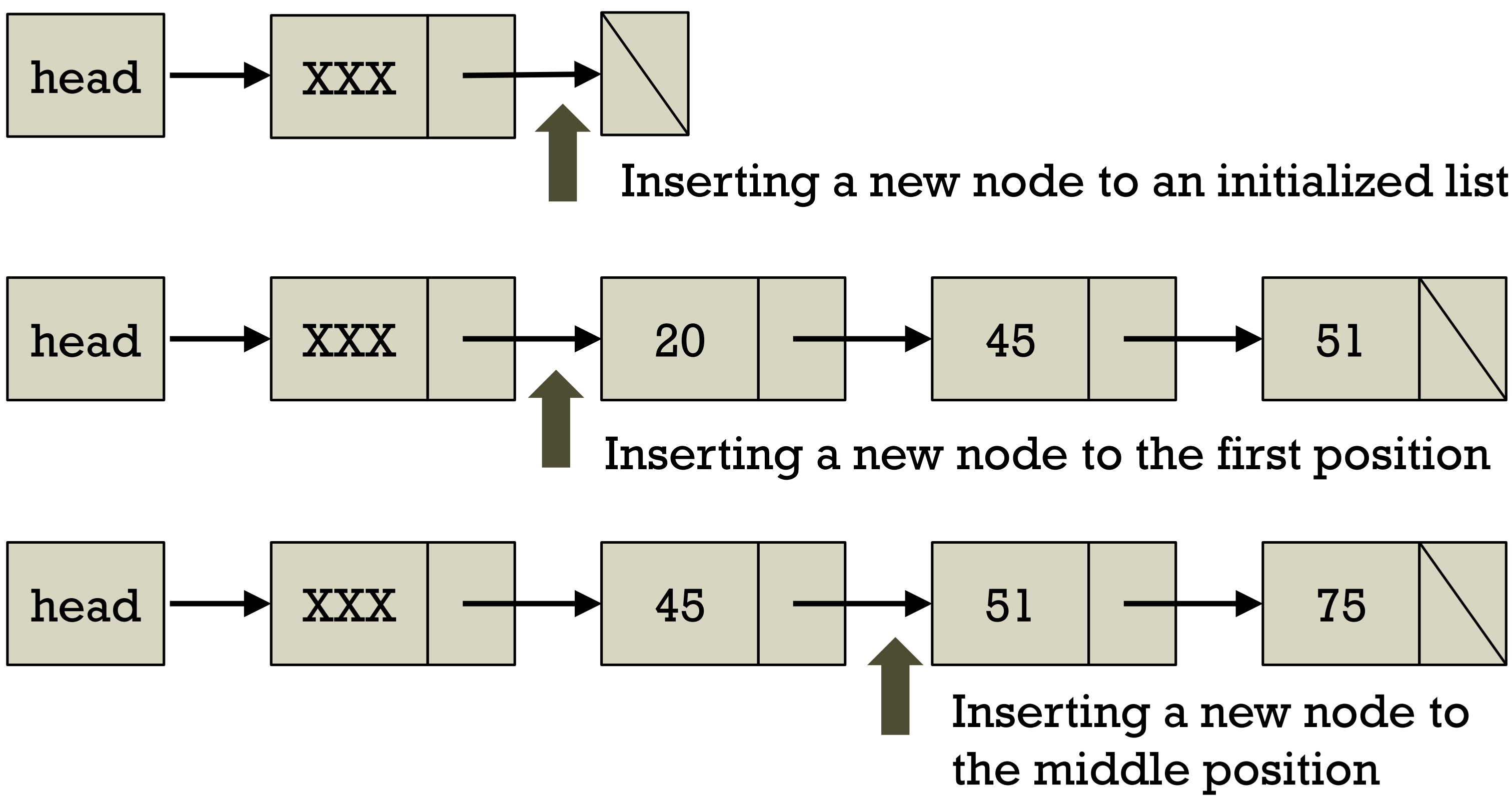
Inserting Nodes in Linked List

- Inserting the node to the k-th position
  - Create a new node.
  - Move the current pointer to the (k-1)-th position.
  - Link the new node to the k-th node.
  - Link the (k-1)-th node to the new node.



Inserting Nodes in Linked List

- Using the dummy node, the three cases for insertions can be addressed with one case.
  - Move the current pointer to the (k-1)-th position, and link the new node in between the (k-1)-th node and the k-th node.



Inserting Nodes in Linked List

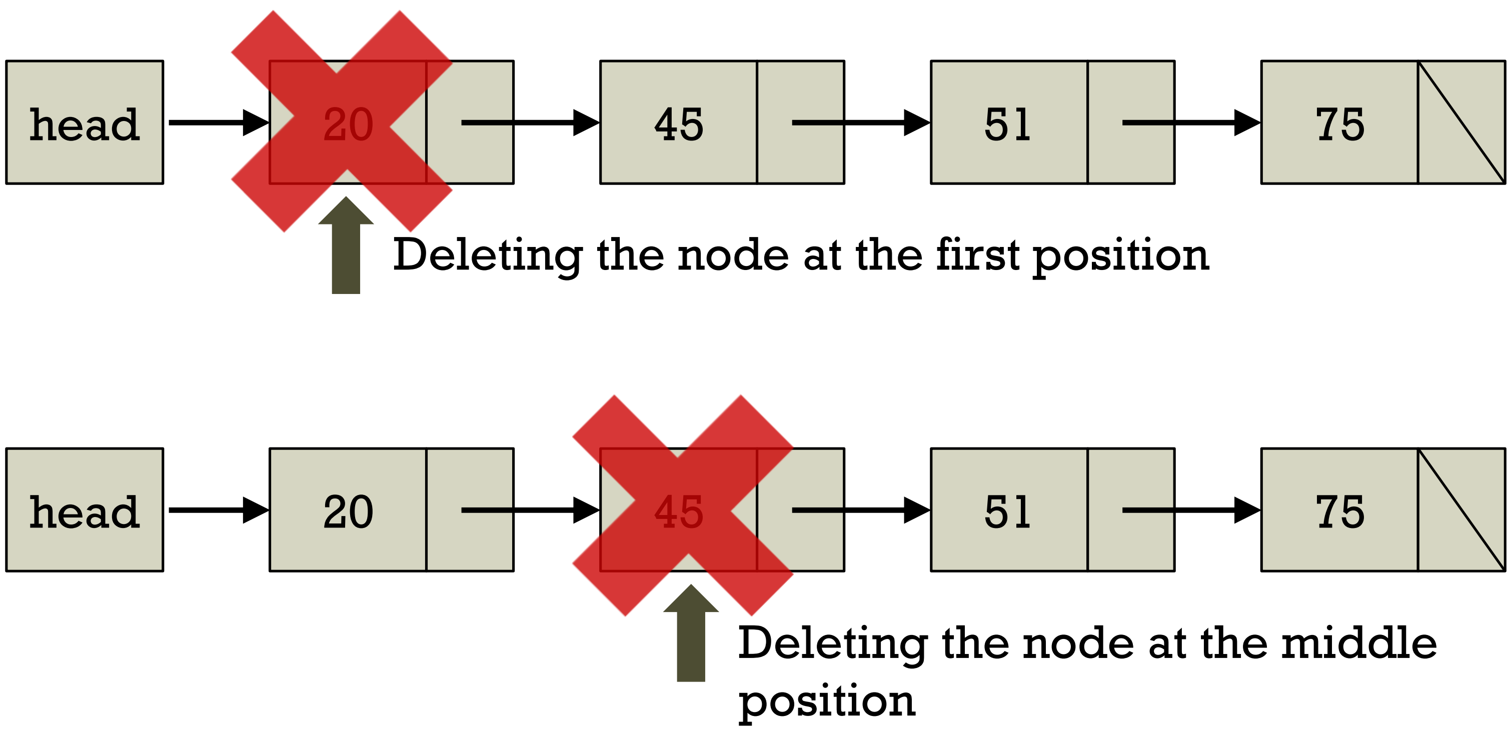
- InsertMiddle operation

```
// Insert an item at the k-th position.
void InsertMiddle(LinkedList* plist, int pos, Data item)
{
    Node* cur, *newNode;
    if (pos < 0 || pos > plist->len)
        exit(1);
    // Create a new node.
    newNode = (Node *)malloc(sizeof(Node));
    newNode->item = item;
    newNode->next = NULL;
    // Move the cur pointer to the (k-1)-th position.
    cur = plist->head;
    for (int i = 0; i < pos; i++)
        cur = cur->next;
    // Insert the new node to the k-th position.
    newNode->next = cur->next;
    cur->next = newNode;
    plist->len++;
}
```

what if there is no space?

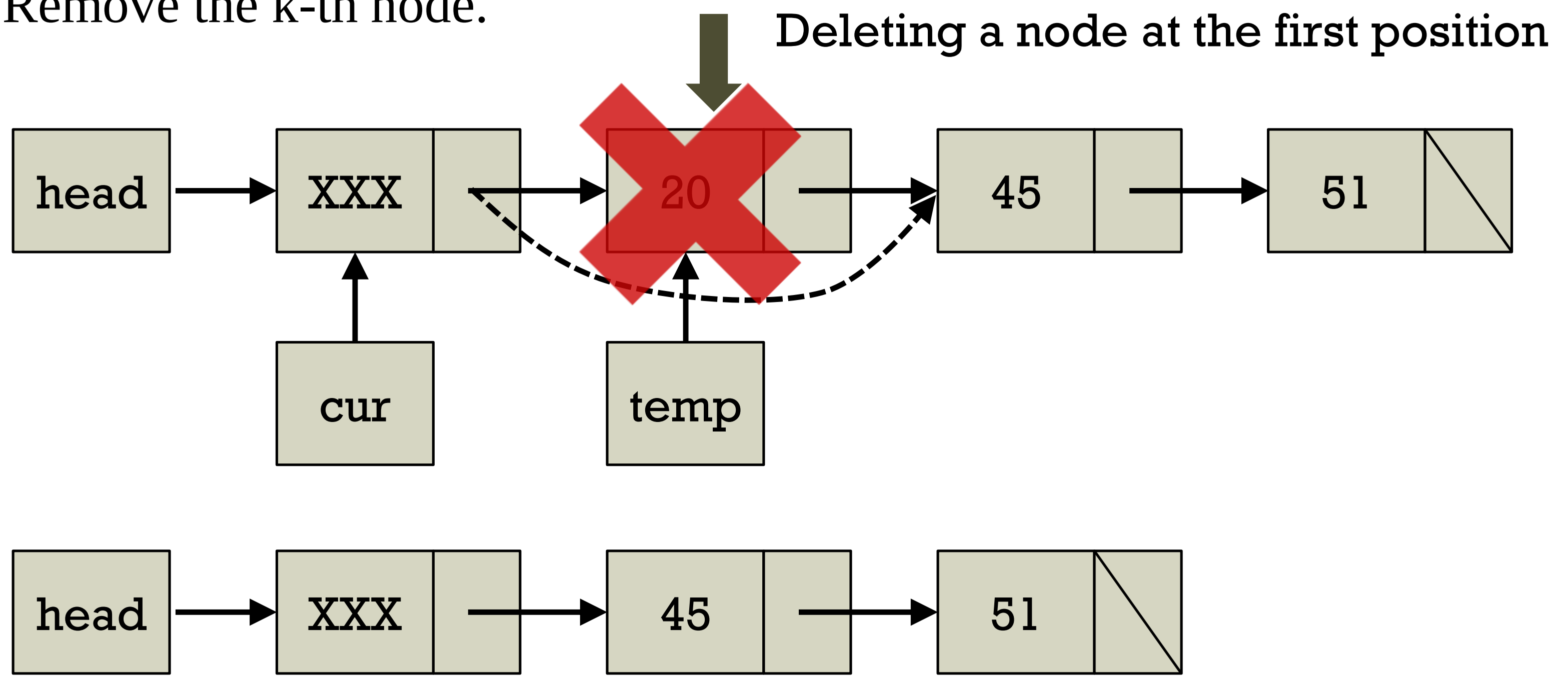
Deleting Nodes in Liked List

- Two cases for deletions
  - Deleting the node at the first position, i.e., k = 0
  - Deleting the node at the k-th position, i.e., 0 < k <= len



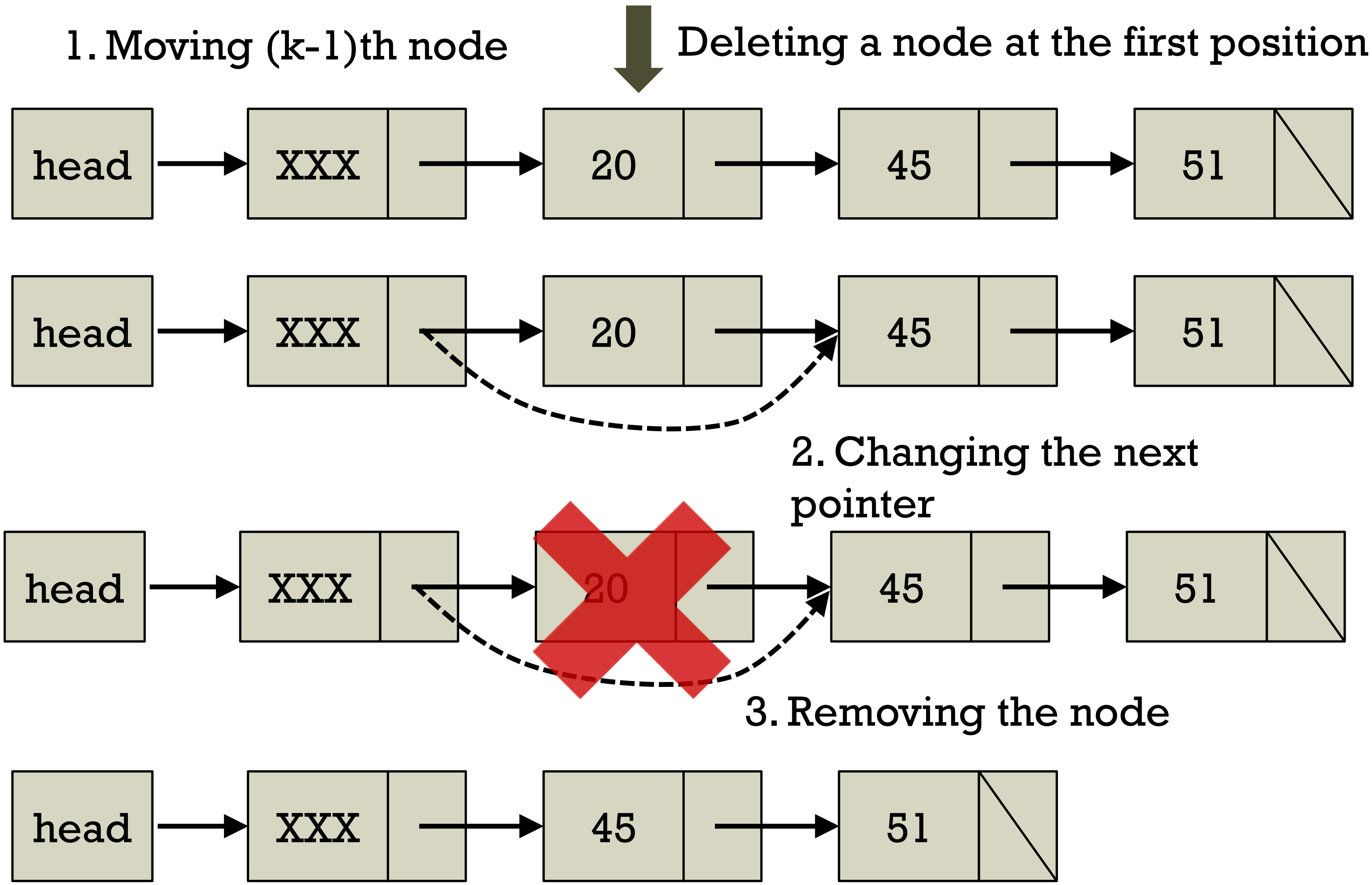
Deleting Nodes in Liked List

- Using the dummy node, the two cases for deletions can be addressed with one case.
  - Move the current pointer to the (k-1)-th position.
  - Refer to the k-th node.
  - Link the (k-1)-th node to (k+1)-th node
  - Remove the k-th node.



Deleting Nodes in Liked List

- How the deletion operation work?



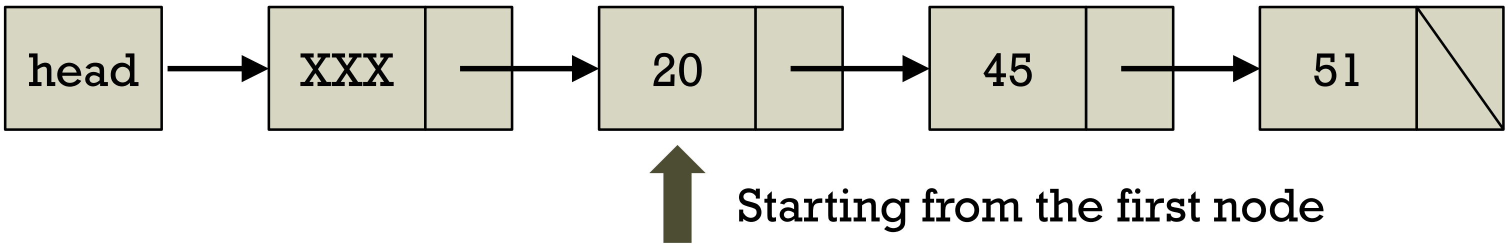
Deleting Nodes in Liked List

- RemoveMiddle operation

```c
// Remove an item at the k-th position.
void RemoveMiddle(LinkedList* plist, int pos)
{
    Node* cur, * temp;
    if (IsEmpty(plist) || pos < 0 || pos >= plist->len)
        exit(1);

    // Move the cur pointer to the (k-1)-th position.
    cur = plist->head;
    for (int i = 0; i < pos; i++)
        cur = cur->next;

    // Remove the node to the k-th position.
    temp = cur->next;
    cur->next = cur->next->next;
    plist->len--;
    free(temp);
}
```

Reading Nodes in Linked List

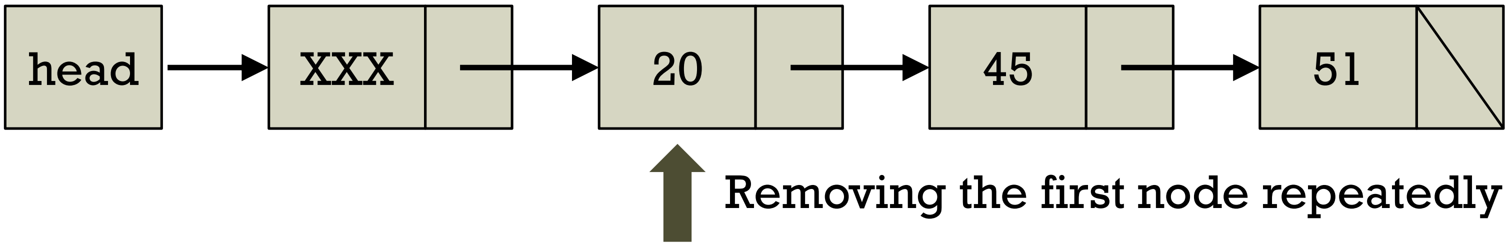
- ReadItem operation




```
// Read an item at the k-th position.
Data ReadItem(LinkedList* plist, int pos)
{
    Node* cur;
    if (IsEmpty(plist) || pos < 0 || pos >= plist->len)
        exit(1);

    // Move the cur pointer to the k-th position.
    cur = plist->head->next;
    for (int i = 0; i < pos; i++)
        cur = cur->next;

    return cur->item;
}
```

Clearing All Nodes in Linked List

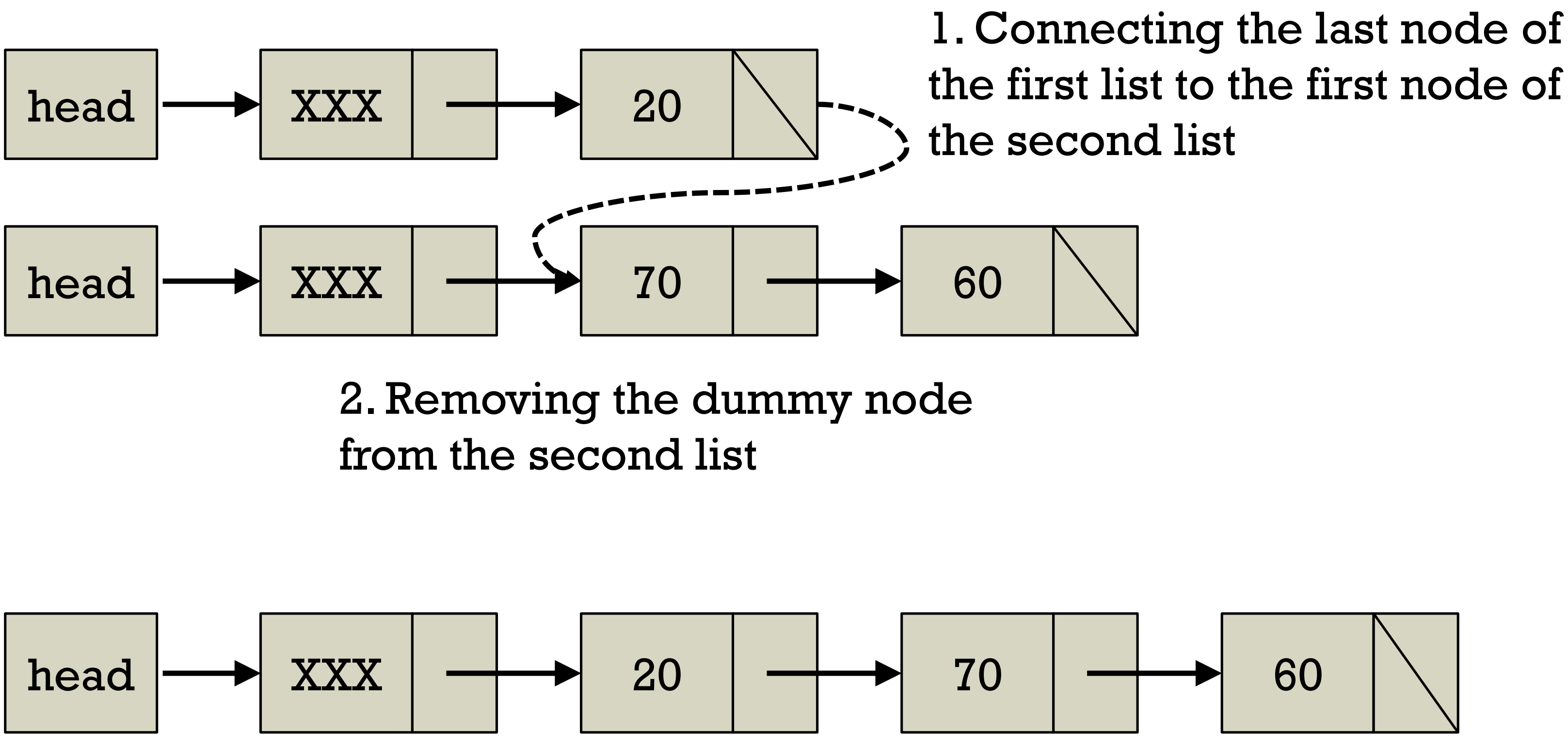
- PrintList and ClearList operations




```
// Print each item in a list in sequence.
void PrintList(LinkedList* plist)
{
    for (Node* cur = plist->head->next; cur != NULL; cur = cur->next)
        printf("%d\n", cur->item);
}

// Remove all nodes in a list in sequence.
void ClearList(LinkedList* plist)
{
    while (plist->head->next != NULL)
        RemoveFirst(plist);
    free(plist->head);
}
```

Advanced: Merging Two Lists

- Concatenating two linked lists



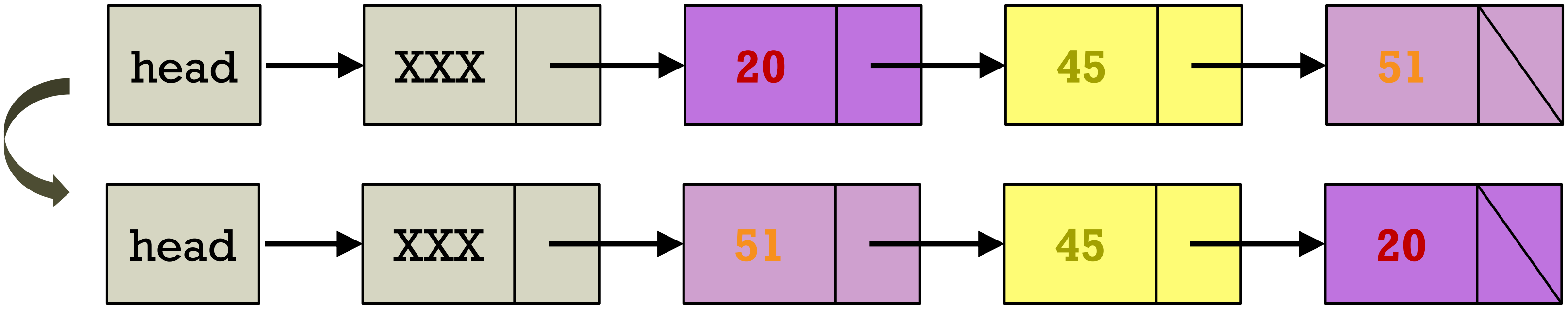
Advanced: Merging Two Lists

- Concatenating two linked lists

```
LinkedList* Concatenate(LinkedList* plist1, LinkedList* plist2)
{
    if (plist1->head->next == NULL) return plist2;
    else if (plist2->head->next == NULL) return plist1;
    else {
        // Move the current pointer to the last position.
        Node* cur = plist1->head->next;
        while (cur->next != NULL)
            cur = cur->next;
        // Link the current pointer to the second list.
        cur->next = plist2->head->next;
        // Remove the dummy node from the second list.
        free(plist2->head);
        return plist1;
    }
}
```

Advanced: Making Reverse List

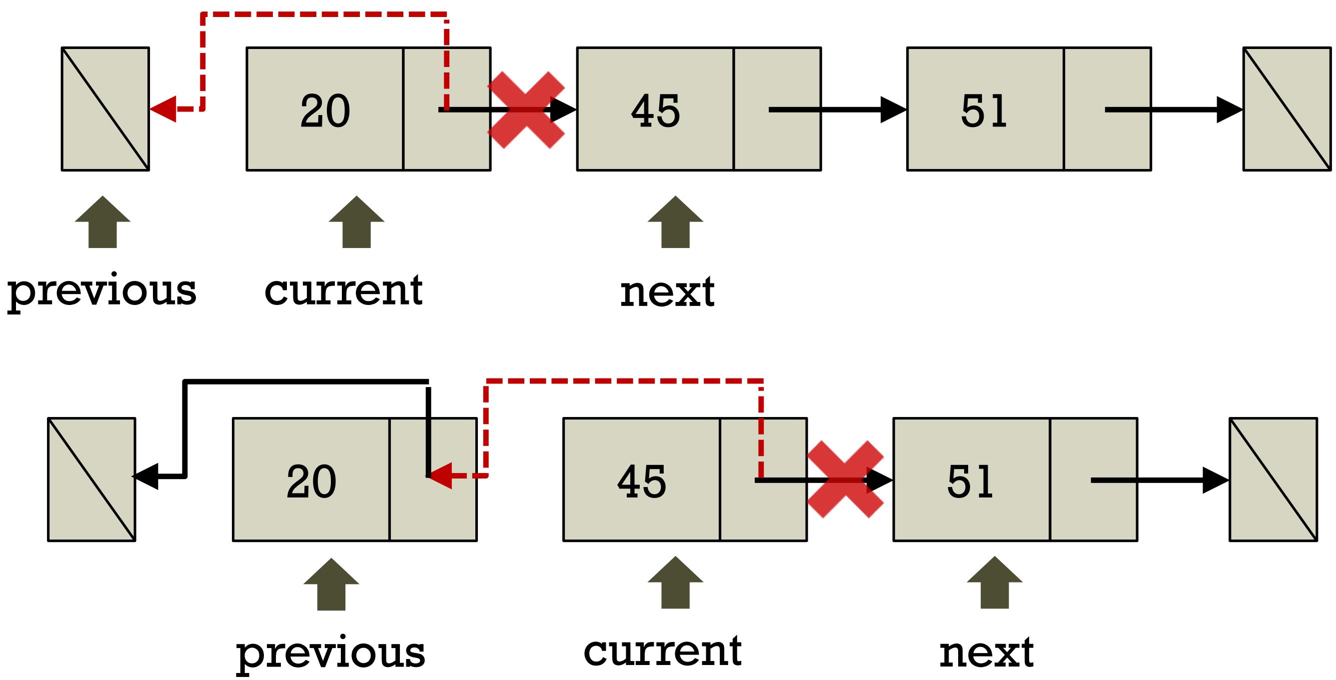
- Description
  - Converting the nodes of lists in a reverse sequence



- How can we do this?

Advanced: Making Reverse List

- How to convert nodes in the list
  - Set previous, current, and next nodes.
  - Link the current pointer to the previous node.
  - Repeat until the next node is NULL.



Advanced: Making Reverse List

- Making nodes in a reverse sequence

```
// Make the list in reverse sequence.
void Reverse(LinkedList* plist)
{
    Node*prev = NULL, *cur = NULL;
    Node *next = plist->head->next;
    // Repeat the next node is NULL.
    while (next != NULL)
    {
        // Set the previous, current, and next nodes.
        prev = cur;
        cur = next;
        next = next->next;
        // Change the link of the current node.
        cur->next = prev;
    }
    // Connect the dummy node to the current node.
    plist->head->next = cur;
}
```