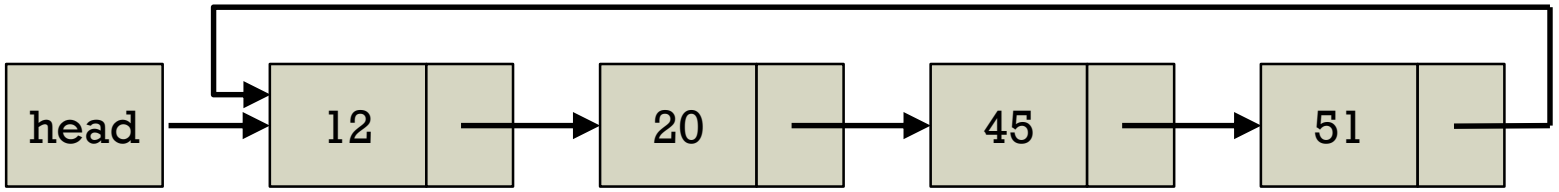
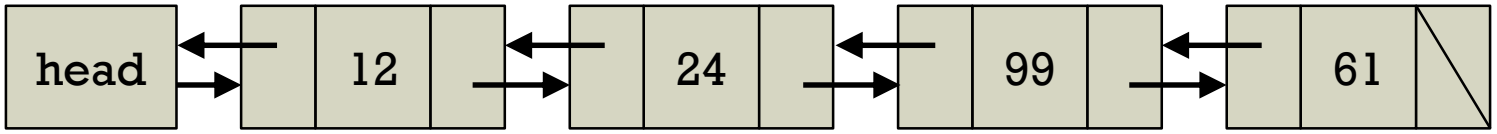
Variations of Linked List and Its Applications

Variations of Linked List

- Circular Linked List
  - The last node is linked to the first node.



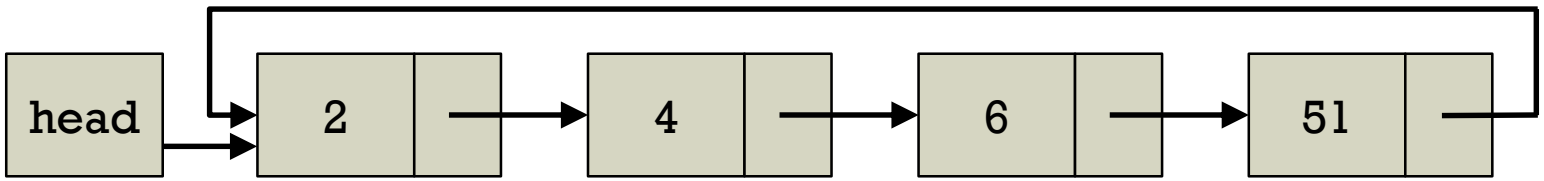
- Double Linked List
  - Each node is linked to the previous and next nodes.



Circular Linked List

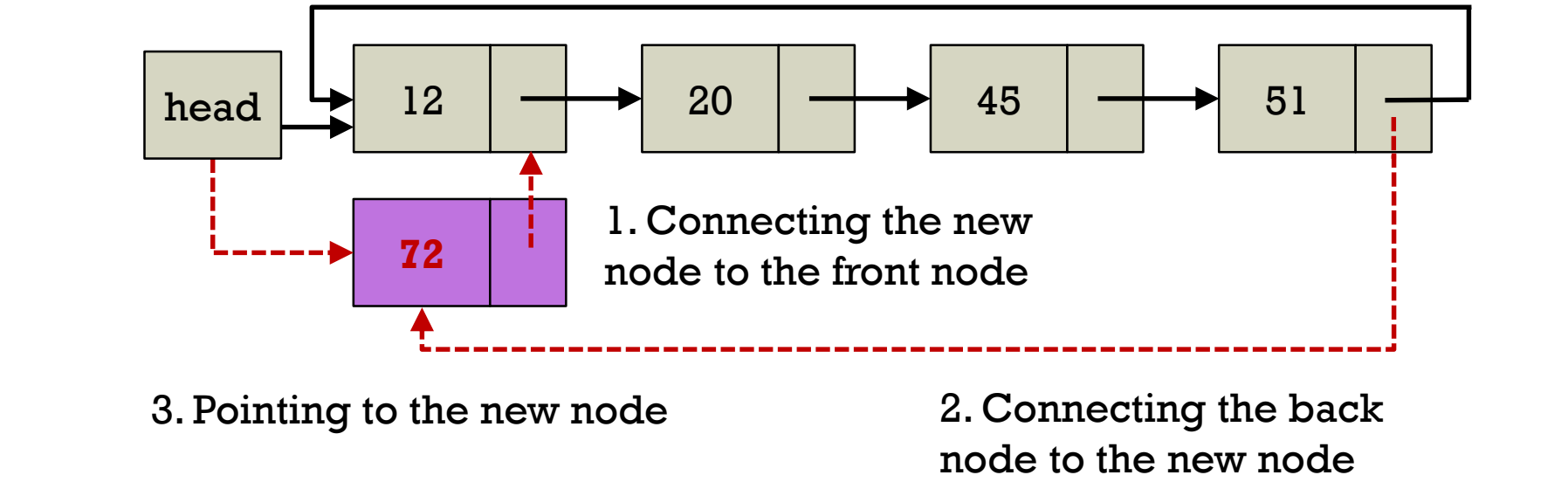
- Definition
  - All nodes are continuously linked in a circle.

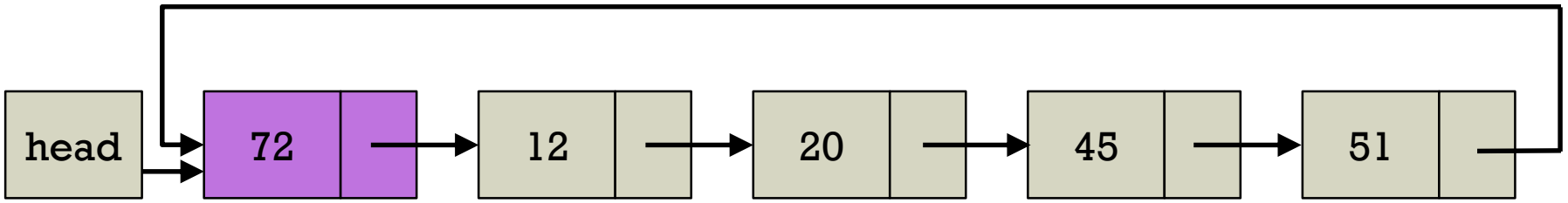
- Advantages
  - Useful for implementation of queue
  - Fast insertions for the front and the back positions
  - Any node can be a head point.



Insertion in Circular Linked List

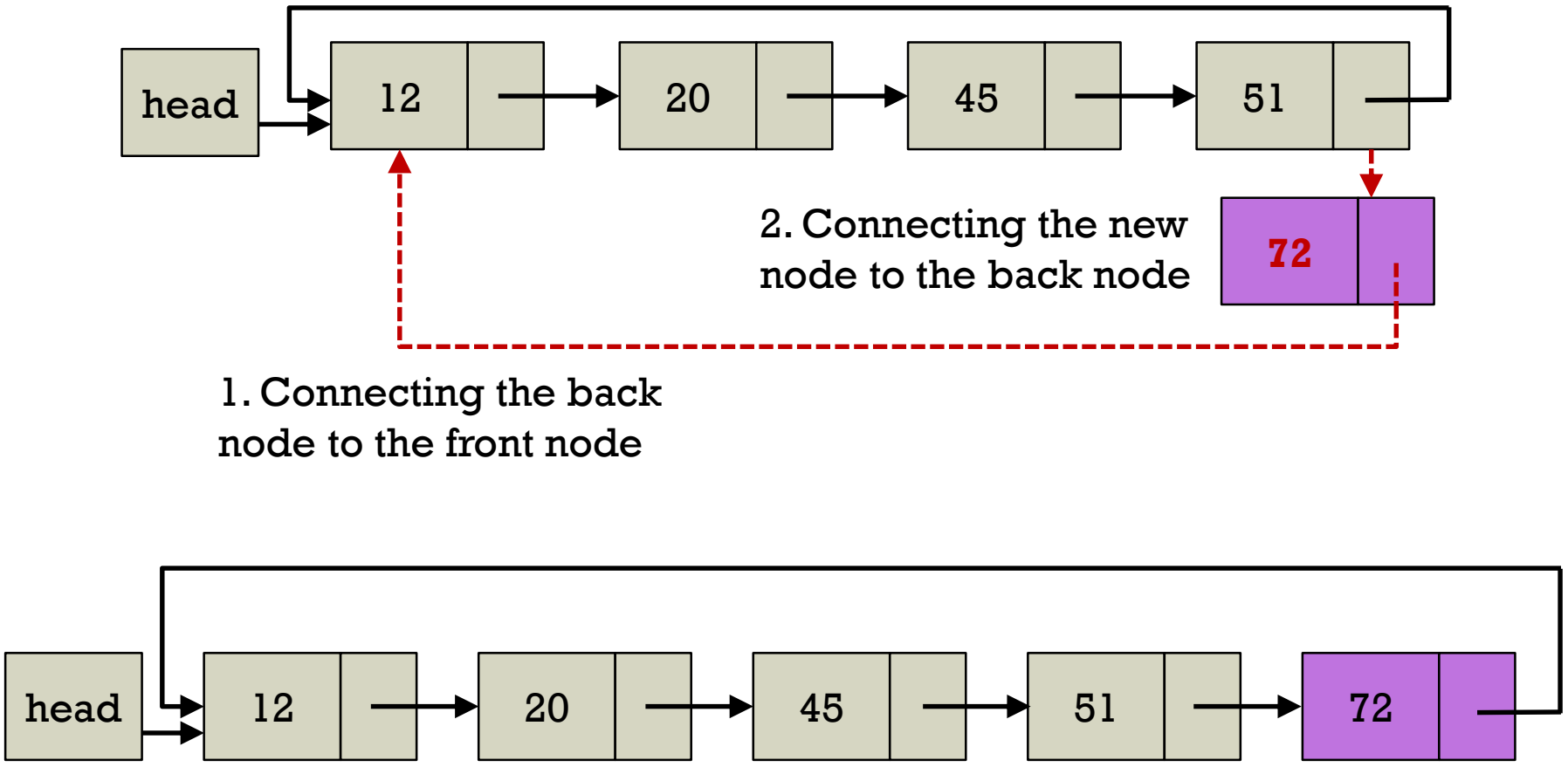
- Inserting an item to the front



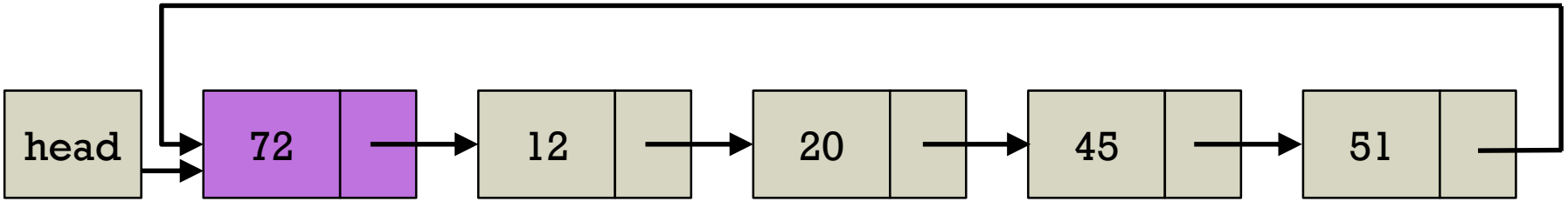


Insertion in Circular Linked List

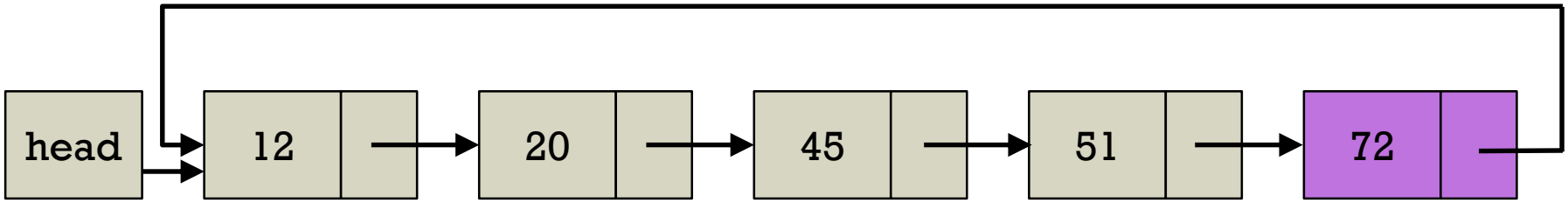
- Inserting an item to the back



Insertion at the head

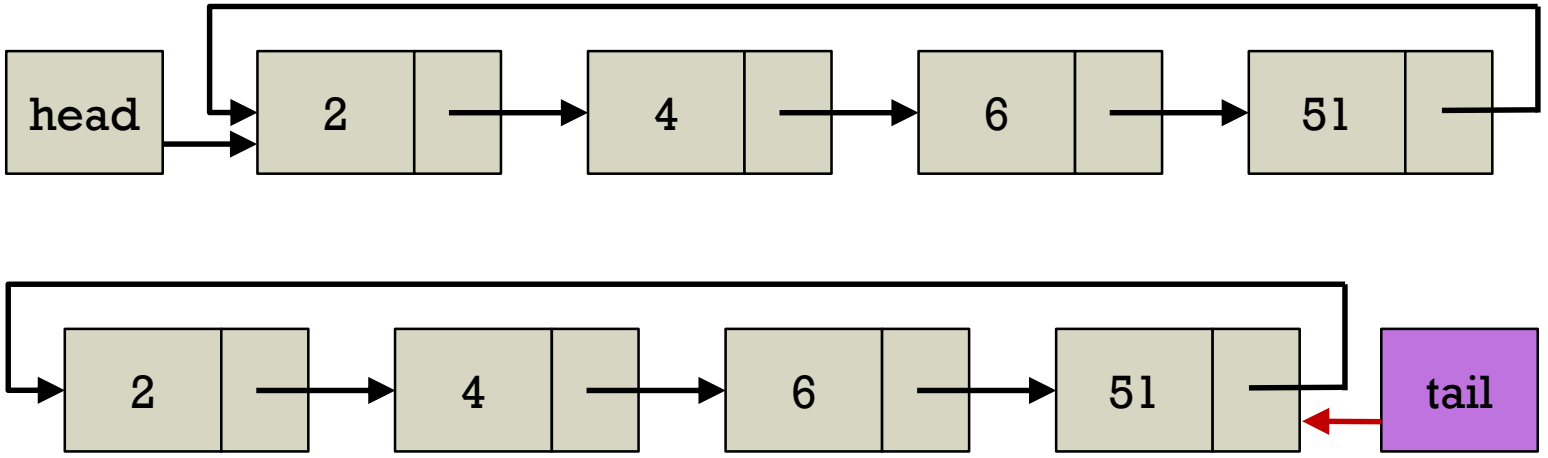


Insertion at the tail



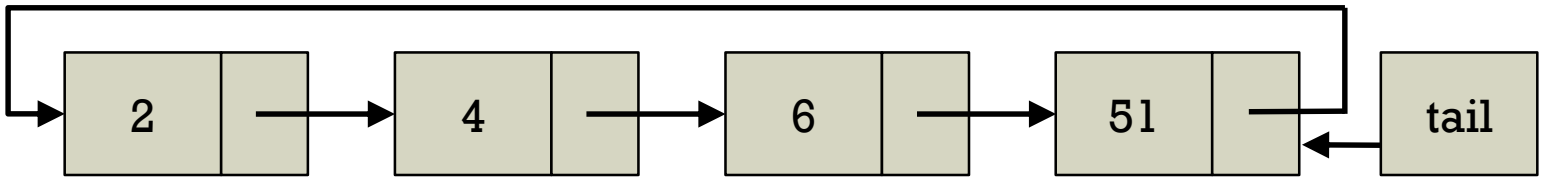
Advantage of Circular Linked List

- Modification for improving implementation
  - If we point to the starting position and we want to add/remove an item to the front, we need to go through the entire list.
  - Because it can traverse at any node, we can modify it to the circular list that points to the last position.



Are they same?

Advantage of Circular Linked List

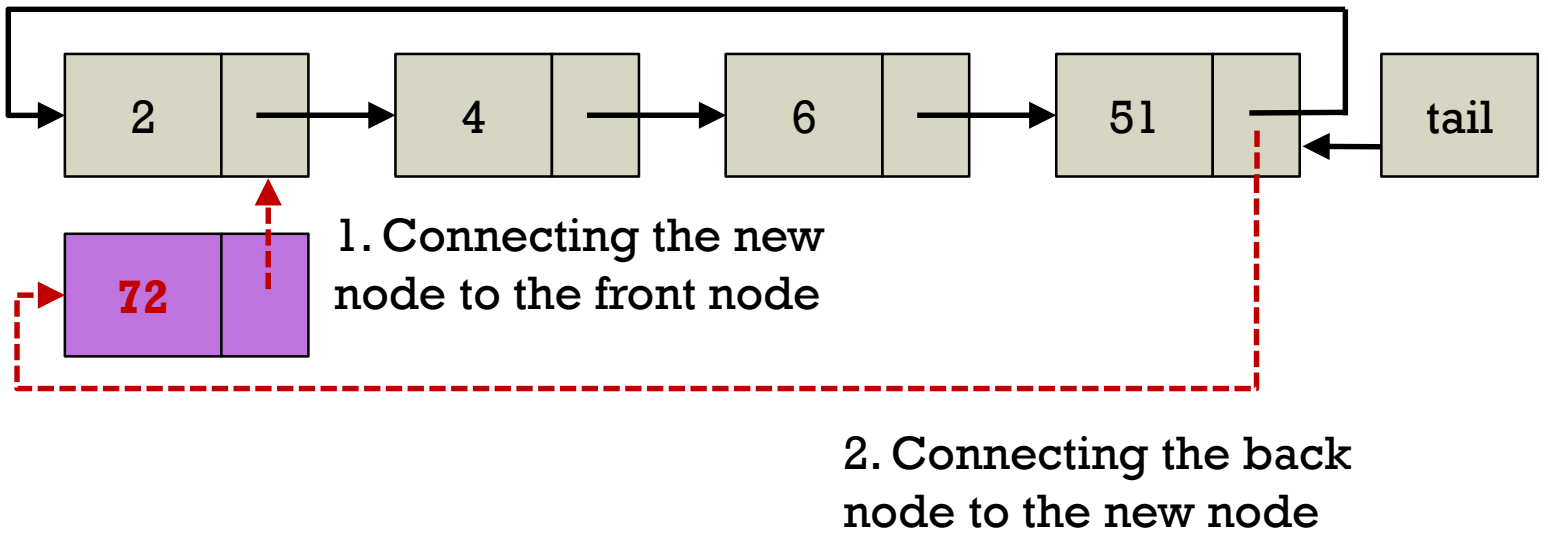


- Q: What is the pointer that refers to the tail?
- A: tail
- Q: What is the pointer that refers to the front?
- A: tail->next

- The tail pointer can be used as the head pointer instead!

Insertion with Tail Pointer

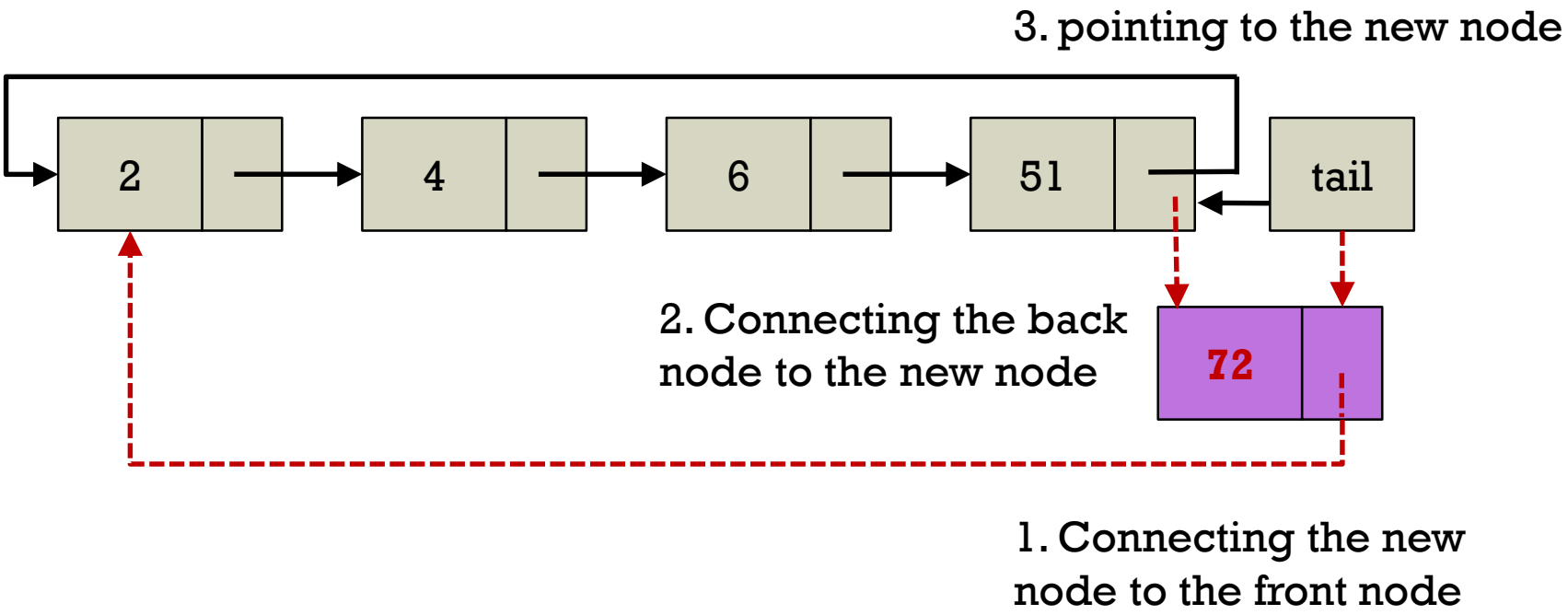
- Inserting an item to the front



- How about removing an item from the front?

Insertion with Tail Pointer

- Inserting an item to the back



- How about removing an item from the back?

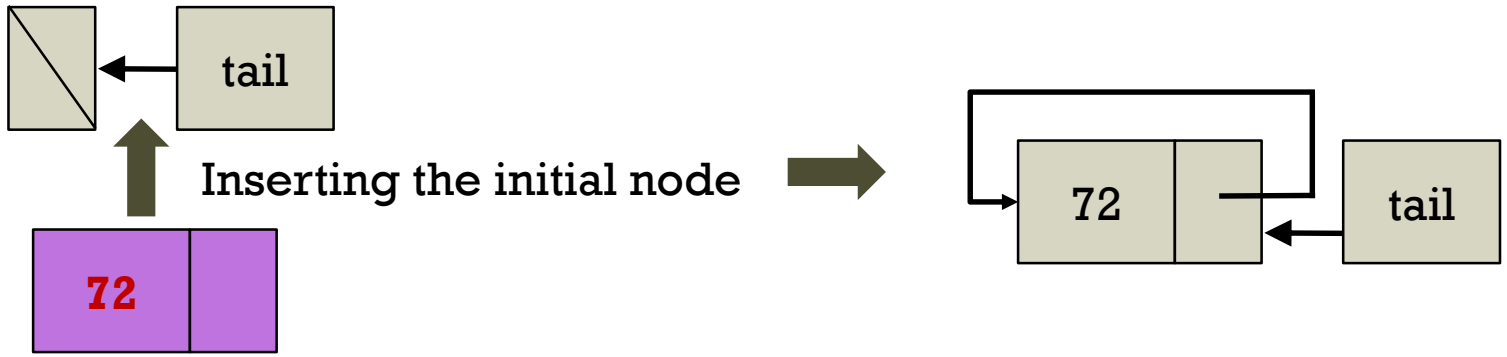
Circular List Implementation

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

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

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

typedef struct
{
     Node* tail;
     int len;
} CircularList;
```

Inserting an Initial Node

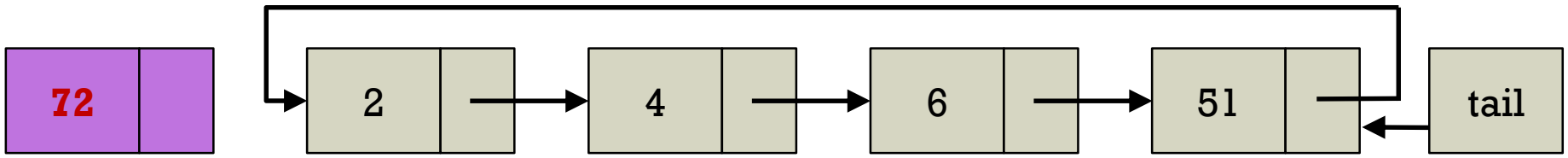
- A special case: inserting the initial node




```
void InsertInitItem(CircularList* plist, Data item)
{
     // Create a new node.
     Node* newNode = (Node *)malloc(sizeof(Node));
     newNode->item = item;
     newNode->next = newNode;

     plist->tail = newNode;
     plist->len++;
}
```

Inserting a Node to the Front

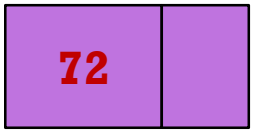
- How to insert a node to the front?

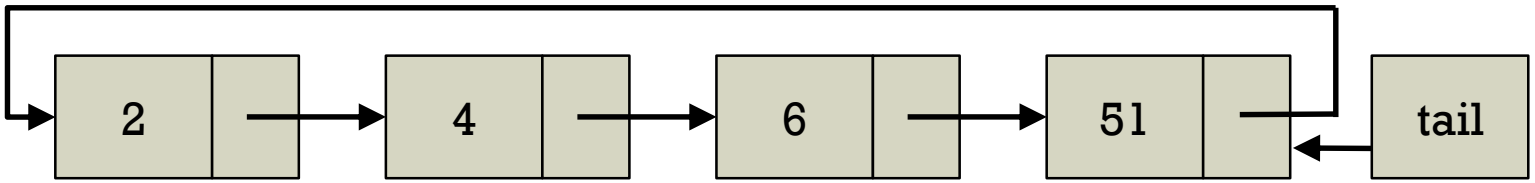



```
void InsertFirst(CircularList* plist, Data item)
{
     if (plist->len == 0)
          InsertInitItem(plist, item);
     else {
          Node* newNode = (Node *)malloc(sizeof(Node));
          newNode->item = item;
          newNode->next = plist->tail->next;

          // Connect the back node to the new node.
          plist->tail->next = newNode;
          plist->len++;
     }
}
```

Inserting a Node to the Back

- How to insert a node to the back?

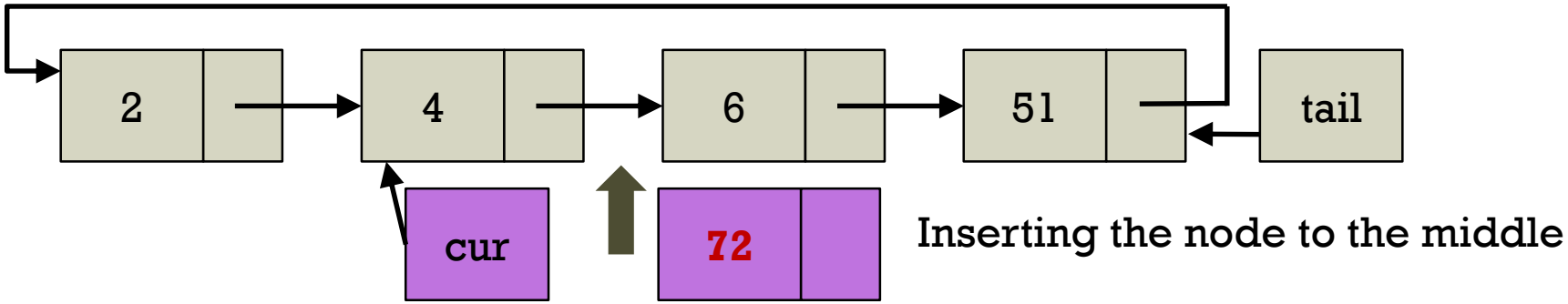





```
void InsertLast(CircularList* plist, Data item)
{
     if (plist->len == 0)
          InsertInitItem(plist, item);
     else {
          Node* newNode = (Node *)malloc(sizeof(Node));
          newNode->item = item;
          newNode->next = plist->tail->next;

          // Connect the back node to the new node.
          plist->tail->next = newNode;
          plist->tail = newNode;
          plist->len++;
     }
}
```

Inserting a Node to the Middle

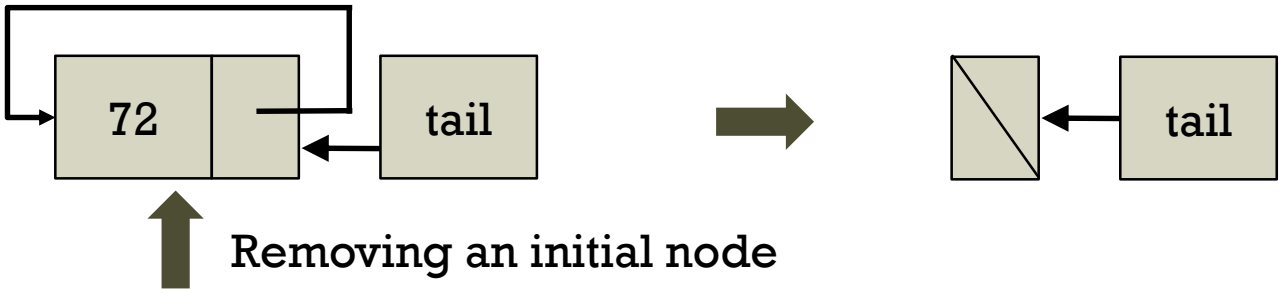
- How to insert a node to the middle?




```
void InsertMiddle(CircularList* plist, int pos, Data item)
{
      if (plist->len == 0)
            InsertInitItem(plist, item);
      else {
            Node* cur, *newNode;
            cur = plist->tail;
            for (int i = 0; i < pos; i++) // moving (k-1)-th position
                  cur = cur->next;

            newNode = (Node *)malloc(sizeof(Node));
            newNode->item = item;
            newNode->next = cur->next;
            cur->next = newNode; // linking the current node to the new node
            plist->len++;
      }
}
```

Removing an Initial Node

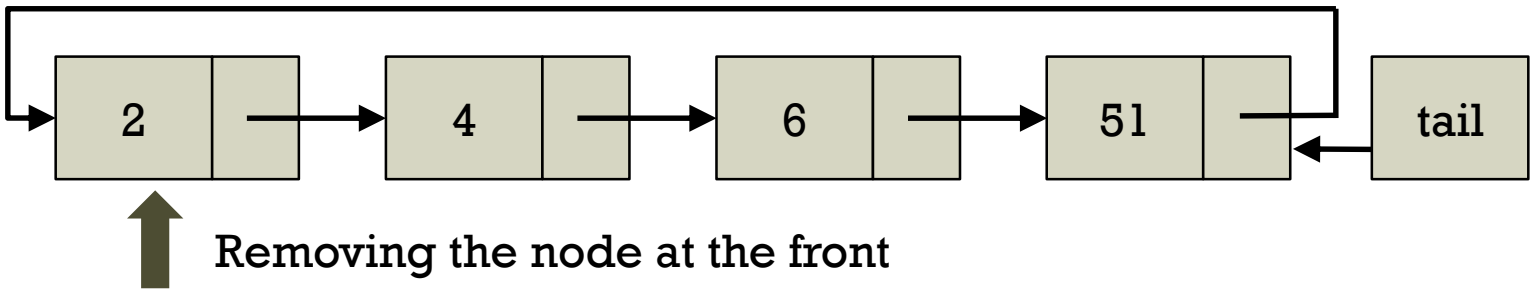
- A special case: removing the initial node




```
void RemoveInitItem(CircularList* plist)
{
     if (IsEmpty(plist)) exit(1);

     if (plist->len == 1) {
          free(plist->tail);
          plist->len--;
          plist->tail = NULL;
     }
}
```

Removing a Node from the Front

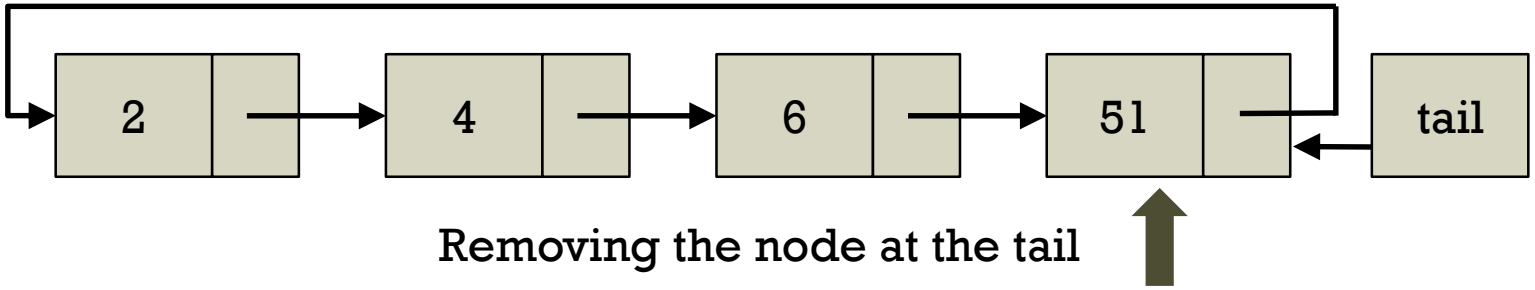
- How to remove a node at the front?




```
void RemoveFirst(CircularList* plist)
{
     if (plist->len == 1)
          RemoveInitItem(plist);
     else {
          Node* temp = plist->tail->next;
          plist->tail->next = temp->next;

          free(temp);
          plist->len--;
     }
}
```

Removing a Node from the Back

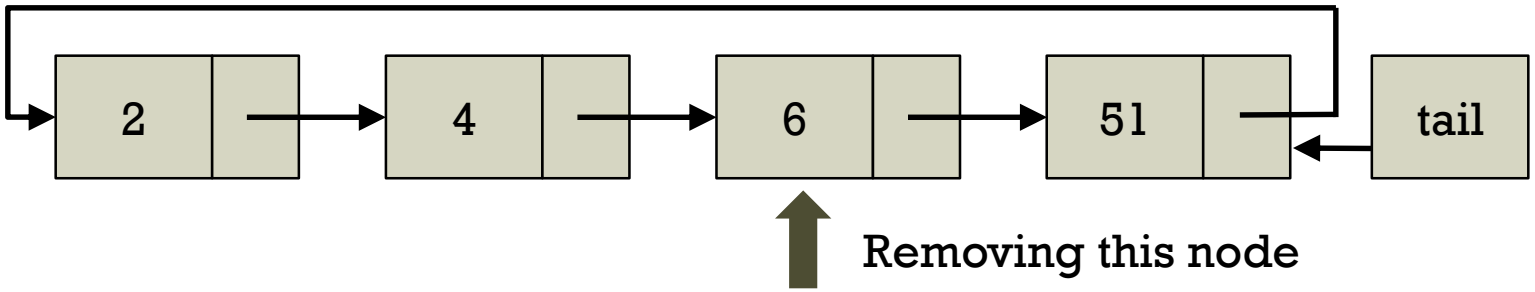
- How to remove a node at the back?




```
void RemoveLast(CircularList* plist)
{
     if (plist->len == 1)
          RemoveInitItem(plist);
     else {
          Node *cur, *temp;
          cur = plist->tail;
          for (int i = 0; i < plist->len - 1; i++)
               cur = cur->next;
          temp = cur->next;
          cur->next = temp->next;

          free(temp);
          plist->tail = cur;
          plist->len--;
}
```

Removing a Node from the Middle

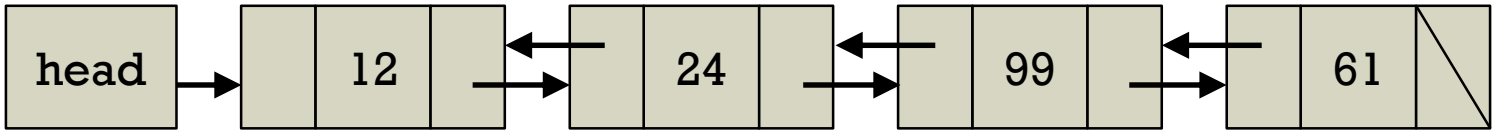
- How to remove a node to the middle?




```
void RemoveMiddle(CircularList* plist, int pos)
{
     if (plist->len == 1)
          RemoveInitItem(plist);
     else {
          Node *cur, *temp;
          cur = plist->tail;
          for (int i = 0; i < pos; i++)
               cur = cur->next;
          temp = cur->next;
          cur->next = temp->next;

          free(temp);
          plist->len--;
     }
}
```

Double Linked List

- Definition
  - Each node is linked to the **previous** and to the **next** node.
- Advantages
  - Can traverse in both directions from the front and from the end.
  - Easy to reverse the linked list
- Disadvantages
  - Compared to the single linked list, it requires more space per node because one extra field is needed.



Linked List Implementation

- Representation
  - A node consists of an item, a previous, and a next pointer.
    - item: a value
    - prev: a pointer to the previous node
    - next: a pointer to the next node

```
typedef enum { false, true } bool;
typedef int Data;
typedef struct _Node
{
    Data item;
    struct _Node* prev;
    struct _Node* next;
} Node;

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

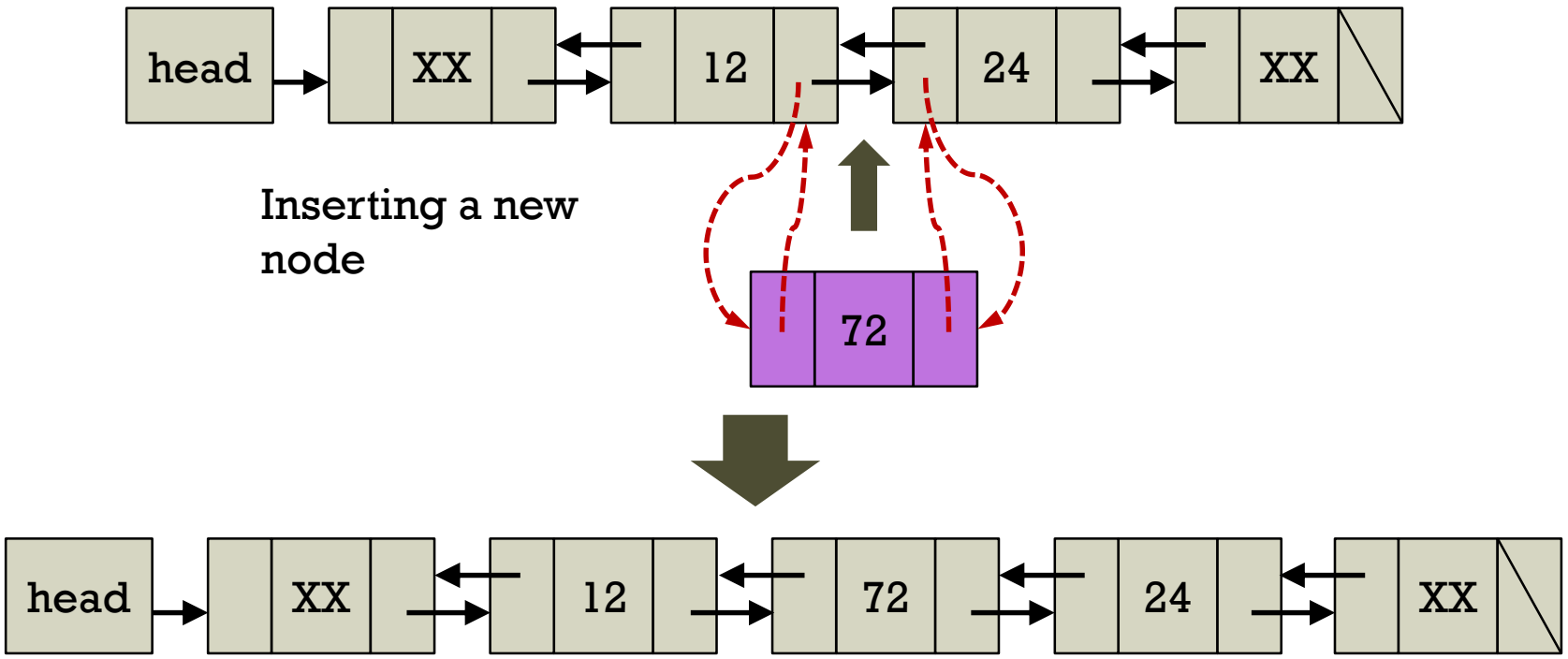
Initializing a Double Linked List

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

```
void InitList(DoubleLinkedList* plist)
{
     Node* dummy1, * dummy2;
     dummy1 = (Node*)malloc(sizeof(Node));
     dummy2 = (Node*)malloc(sizeof(Node));

     dummy1->prev = NULL;
     dummy1->next = dummy2;
     dummy2->prev = dummy1;
     dummy2->next = NULL;

     plist->head = dummy1;
     plist->len = 0;
}
```

Inserting Nodes in Linked List

- How to insert an node in the middle?
  - Step 1: Create a new node.
  - Step 2: For the new node, link it to adjacent nodes.
  - Step 3: Update the connection for adjacent nodes.



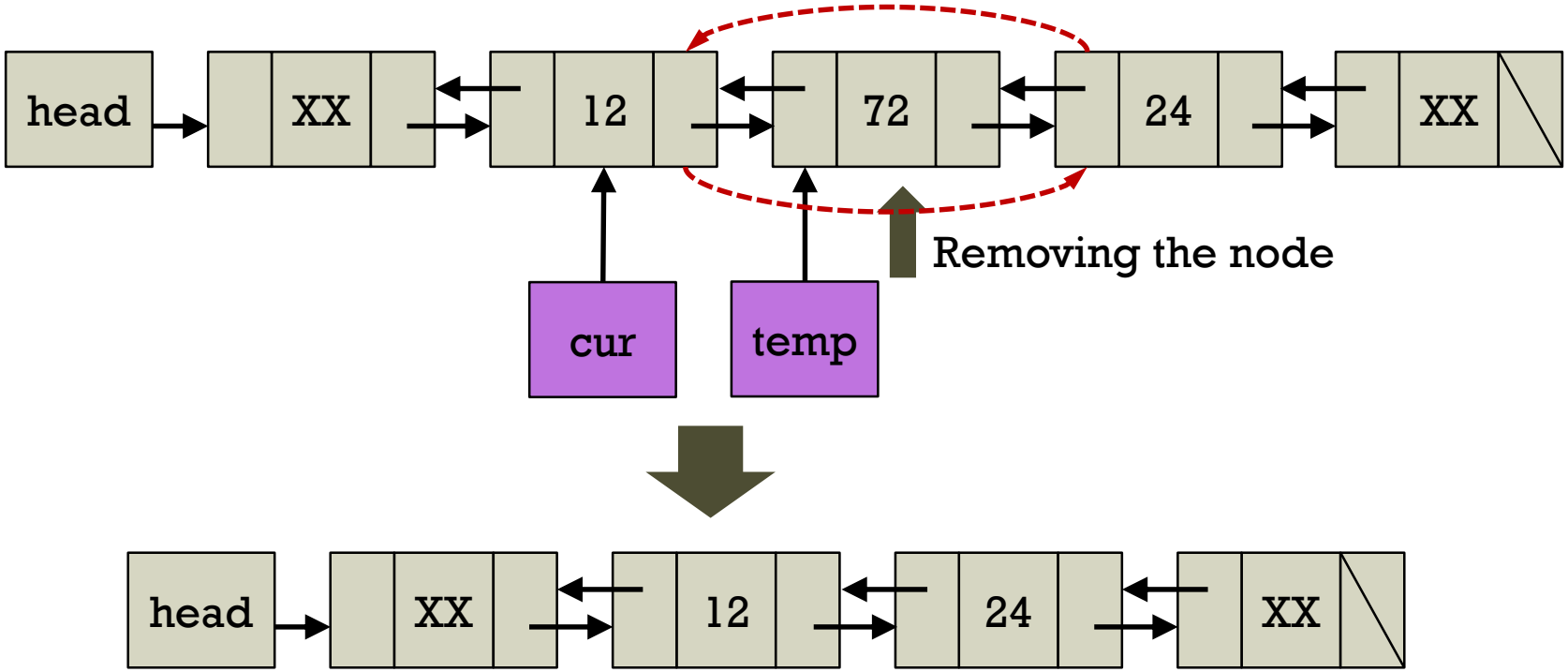
Inserting Nodes in Linked List

- InsertMiddle operation

```
void InsertMiddle(DoubleLinkedList* plist, int pos, Data item)
{
     Node* cur, *newNode;

     // Create a new node.
     newNode = (Node *)malloc(sizeof(Node));
     newNode->item = item;
     newNode->prev = NULL;
     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->prev = cur;
     newNode->next = cur->next;
     cur->next->prev = newNode;
     cur->next = newNode;
     plist->len++;
}
```

Removing Nodes in Linked List

- How to remove an node in the middle?
  - Step 1: Move the current pointer to the (k-1)-th position.
  - Step 2: Refer to the k-th node.
  - Step 3: Link the (k-1)-th node to (k+1)-th node
  - Step 4: Remove the k-th node.



Removing Nodes in Liked List

- RemoveMiddle operation

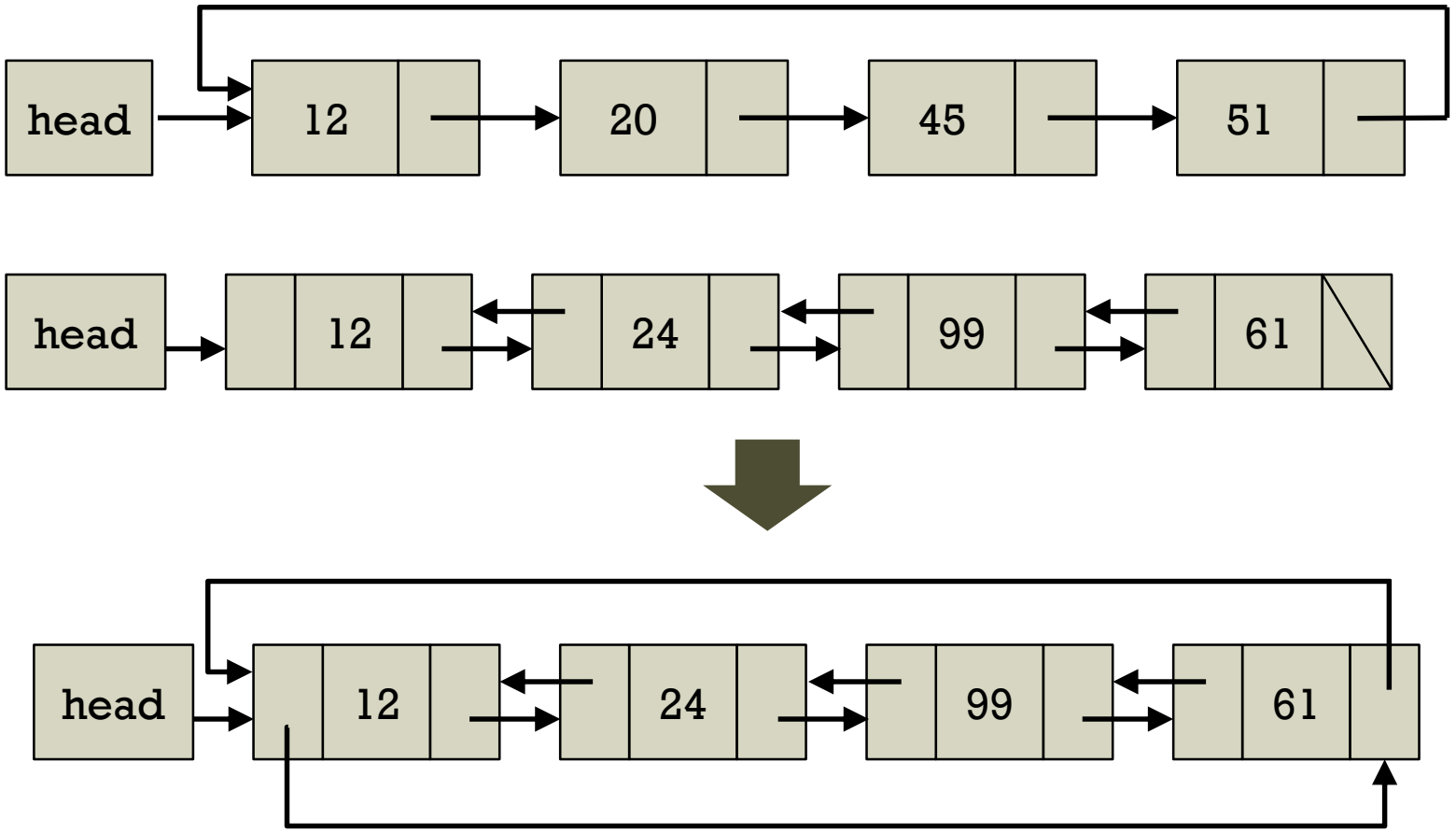
```
void RemoveMiddle(DoubleLinkedList* plist, int pos)
{
     Node* cur, *temp;
     if (IsEmpty(plist) || pos < 0 || pos >= plist->len)
          exit(1);

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

     // Connect adjacent nodes to remove the k-th node.
     temp = cur->next;
     temp->next->prev = cur;
     cur->next = temp->next;

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

Applications of Linked List

- Implementation for dynamic stack
- Implementation for dynamic queue
- Implementation for polynomials
  - It is effective for implementing sparse polynomials

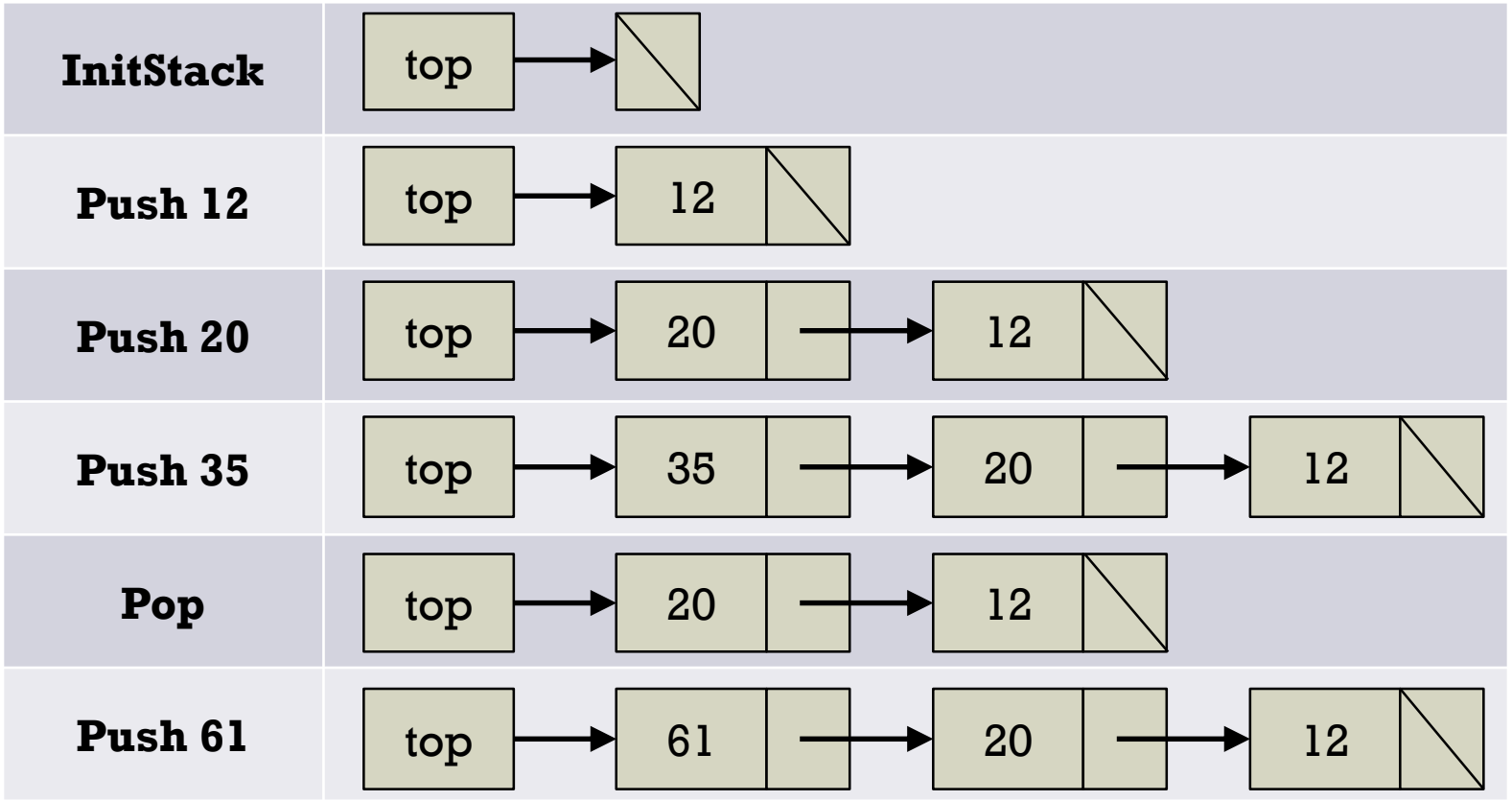
Linked List Used in API

- Circular double linked list
  - Circular linked list + double linked list



How Dynamic Stack Works?

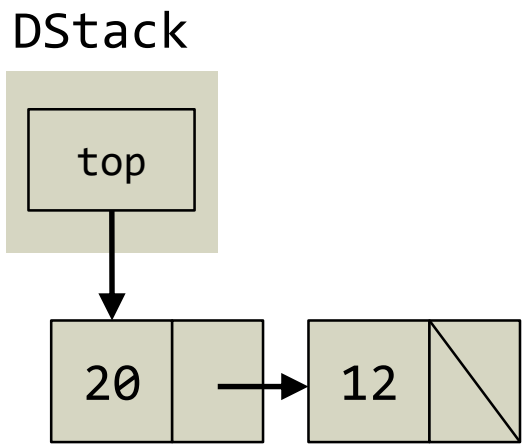
- How to represent stack by linked list
  - Good memory usage without pre-defined size



Dynamic Stack

- Representation
  - A node consists of an item and a next pointer.
    - item: a value, next: a pointer to the next node
  - A stack consists of a top pointer.

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

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

typedef struct
{
    Node* top;
} DStack;
```



Dynamic Stack

- Operations
  - The **IsFull** operation does not need to implement.

```
// Make stack empty.
void InitStack(DStack *pstack);
// check whether stack is empty.
bool IsEmpty(DStack *pstack);

// Read the item at the top.
Data Peek(DStack *pstack);
// Insert an item at the top.
void Push(DStack *pstack, Data item);
// Remove the item at the top.
void Pop(DStack *pstack);
```

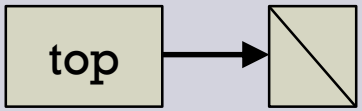
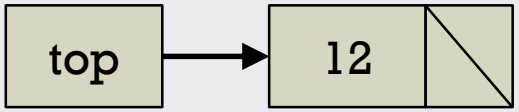
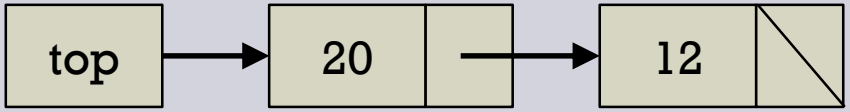
Implementing Dynamic Stack

- InitStack, IsEmpty, and Peek operations

```
// Make stack empty.
void InitStack(DStack *pstack)
{
    pstack->top = NULL;
}

// check whether stack is empty.
bool IsEmpty(DStack *pstack)
{
    return pstack->top == NULL;
}

// Read the item at the top.
Data Peek(DStack *pstack)
{
    if (IsEmpty(pstack))
        exit(1);
    return pstack->top->item;
}
```

Implementing Dynamic Stack

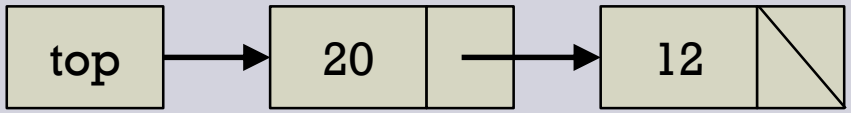
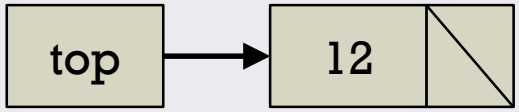
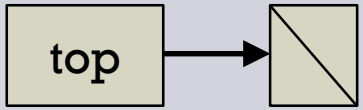
- Push operation

InitStack	
Push 12	
Push 20	

```
void Push(DStack *pstack, Data item)
{
    Node* newNode = (Node *)malloc(sizeof(Node));
    newNode->item = item;
    newNode->next = pstack->top;
    pstack->top = newNode;
}
```

Implementing Dynamic Stack

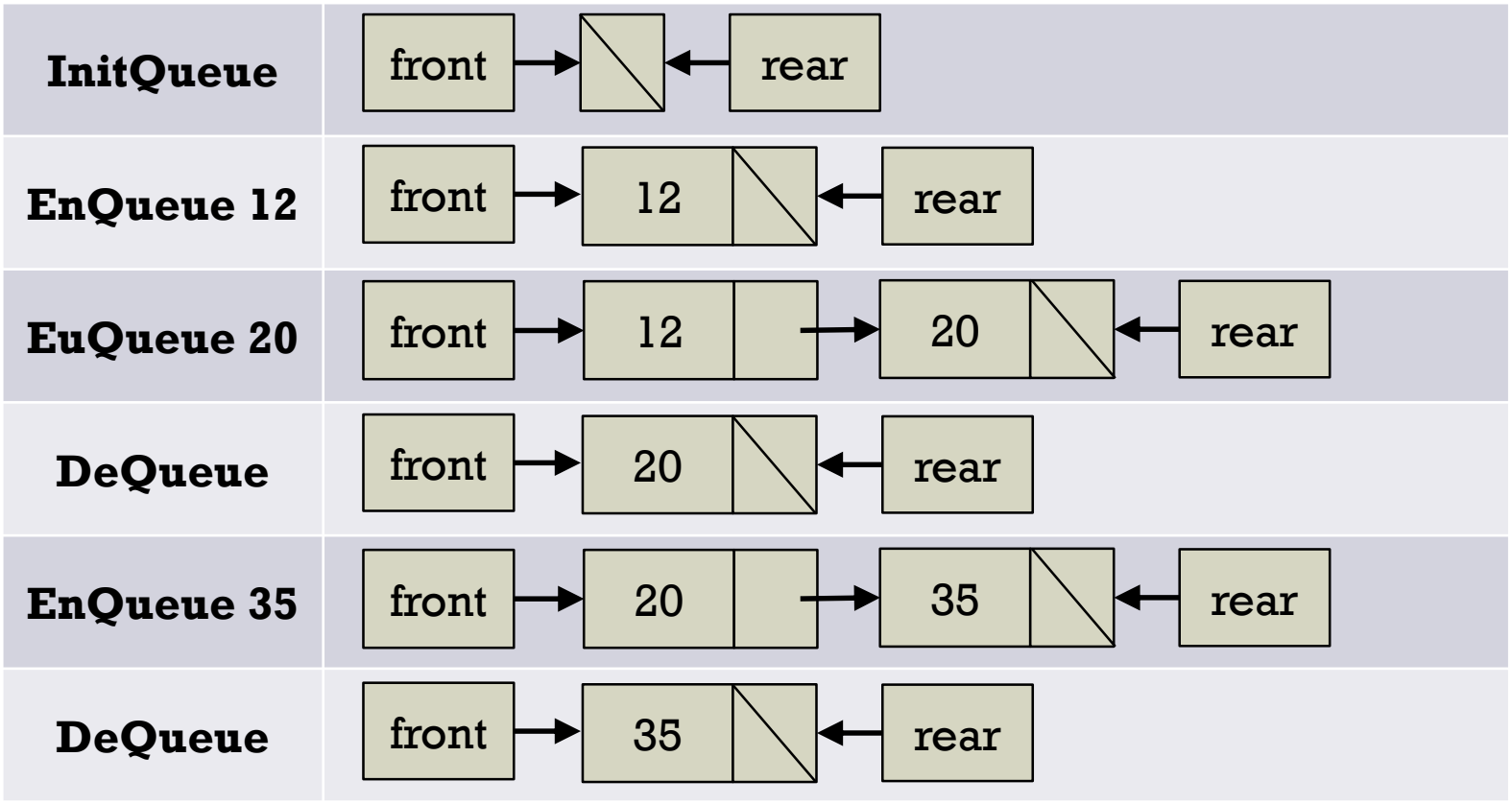
- Pop operation

	
Pop	
Pop	

```
// Remove the item at the top.
void Pop(DStack *pstack)
{
    Node* temp;
    if (IsEmpty(pstack)) exit(1);

    temp = pstack->top;
    pstack->top = pstack->top->next;
    free(temp);
}
```

How Dynamic Queue Works?

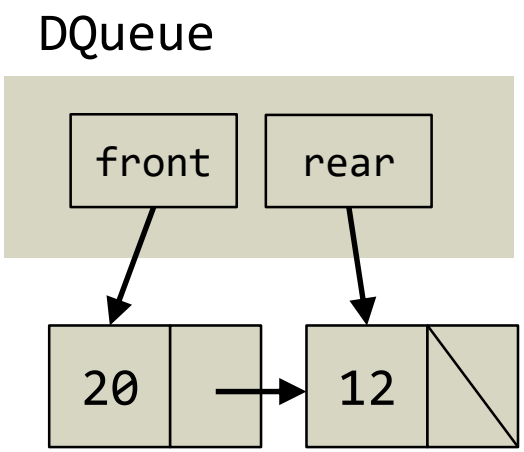
- How to represent queue by linked list
  - Good memory usage without pre-defined size



Dynamic Queue

- Representation
  - A node consists of an item and a next pointer.
    - item: a value, next: a pointer to the next node
  - A queue consist of a front pointer and a rear pointer.

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

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

typedef struct
{
    Node* front;
    Node* rear;
} DQueue;
```



Dynamic Queue

- Operations
  - The **IsFull** operation does not need to implement.

```
// Make a queue empty.
void InitQueue(DQueue *pqueue);
// Check whether a queue is empty.
bool IsEmpty(DQueue *pqueue);

// Read the item at the front.
Data Peek(DQueue *pqueue);
// Insert an item at the rear.
void EnQueue(DQueue *pqueue, Data item);
// Delete an item at the front.
void DeQueue(DQueue *pqueue);
```

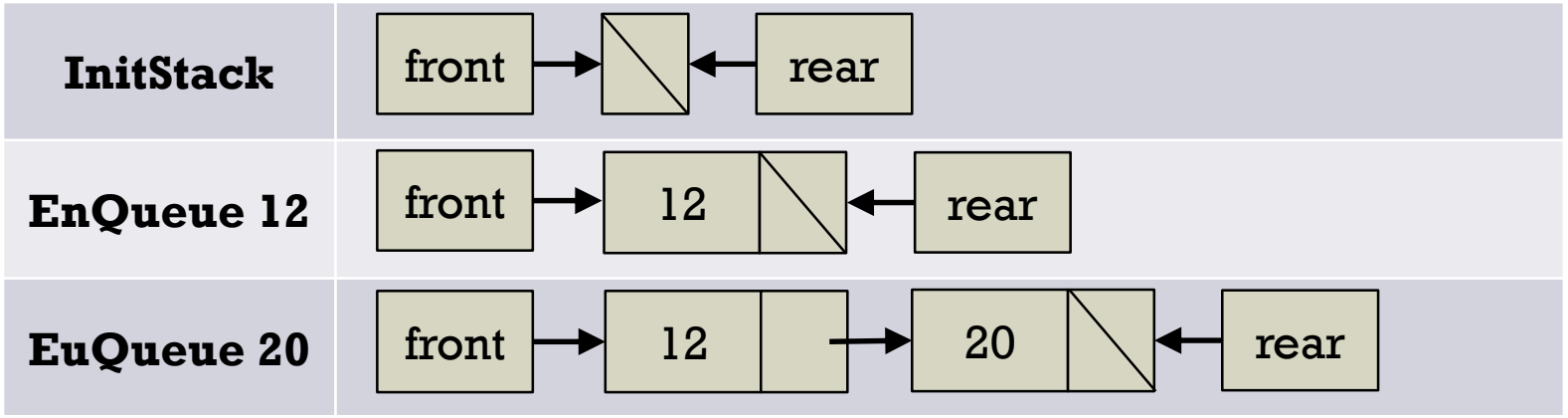
Implementing Dynamic Queue

- InitQueue, IsEmpty, and Peek operations

```
// Make a queue empty.
void InitQueue(DQueue *pqueue)
{
    pqueue->front = pqueue->rear = NULL;
}
// Check whether a queue is empty.
bool IsEmpty(DQueue *pqueue)
{
    return pqueue->front == NULL;
}

// Read the item at the front.
Data Peek(DQueue *pqueue)
{
    if (IsEmpty(pqueue)) exit(1);

    return pqueue->front->item;
}
```

Implementing Dynamic Queue

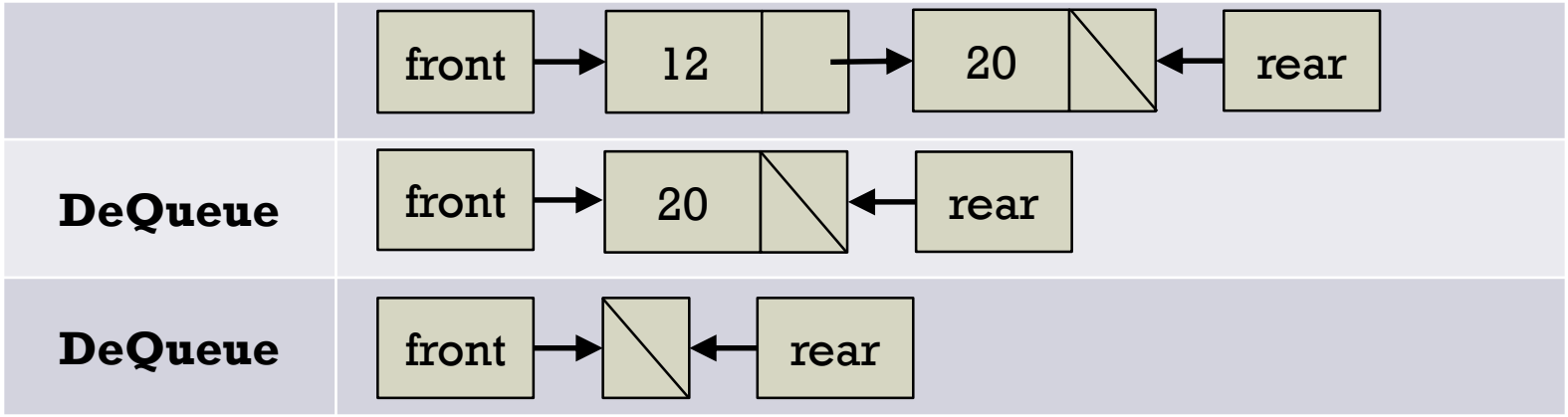
- EnQueue operation




```
void EnQueue(DQueue *pqueue, Data item)
{
     Node* newNode = (Node *)malloc(sizeof(Node));
     newNode->item = item;

     if (IsEmpty(pqueue))
          pqueue->front = pqueue->rear = newNode;
     else {
          pqueue->rear->next = newNode;
          pqueue->rear = newNode;
     }
}
```

Implementing Dynamic Queue

- DeQueue operation




```
void DeQueue(DQueue *pqueue)
{
     Node* temp;
     if (IsEmpty(pqueue)) exit(1);

     temp = pqueue->front;
     if (temp->next == NULL)
          pqueue->front = pqueue->rear = NULL;
     else
          pqueue->front = temp->next;
     free(temp);
}
```

Dynamic Stack and Queue

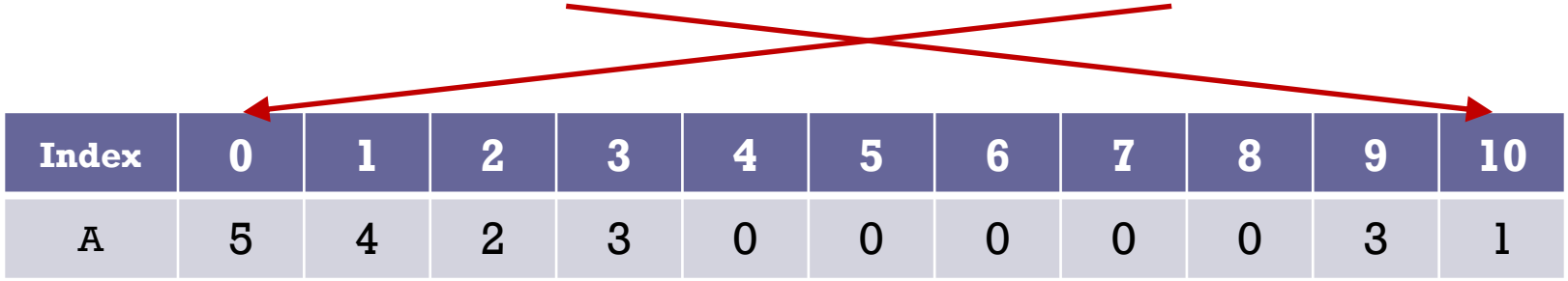
- Representing stack and queue by linked lists
  - No data movement is necessary: O(1)
  - The **IsFull** operation is unnecessary
  - Pre-defined size is unnecessary.

- What is advantage of implementing stack and queue by arrays?

Polynomial using Array

- Representing polynomial as an array

$$p(x) = x^{10} + 3x^9 + 3x^3 + 2x^2 + 4x + 5$$



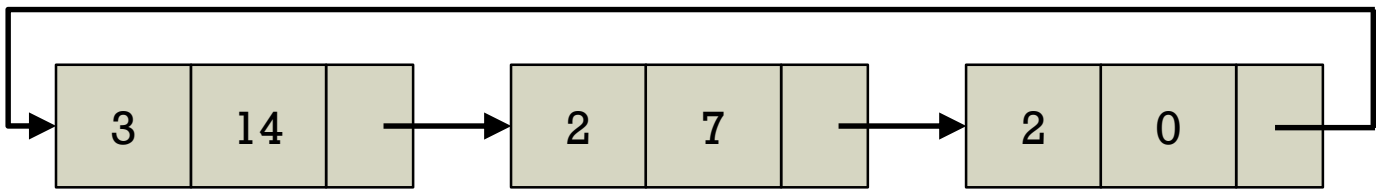
Index	0	1	2	3	4	5	6	7	8	9	10
A	5	4	2	3	0	0	0	0	0	3	1

- Advantage
  - Easy to implement
- Disadvantage
  - It requires a lot of memory space if the polynomial is sparse.

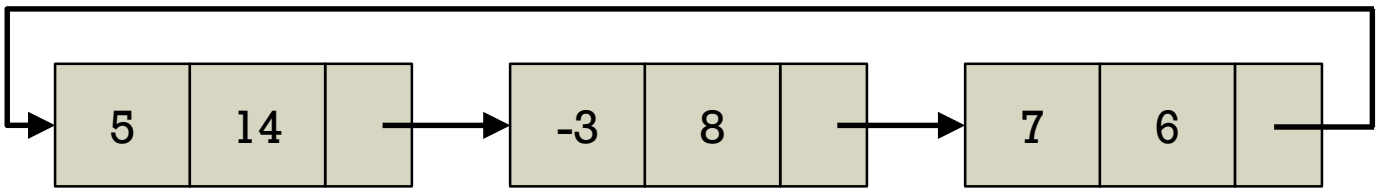
Polynomial using Linked List

- Representing polynomials as a single linked list
  - Two values: coefficient and exponent
  - Pointer: point to a next node

$a = 3x^{14} + 2x^{7} + 2$



$b = 5x^{14} - 3x^{8} + 7x^{6}$



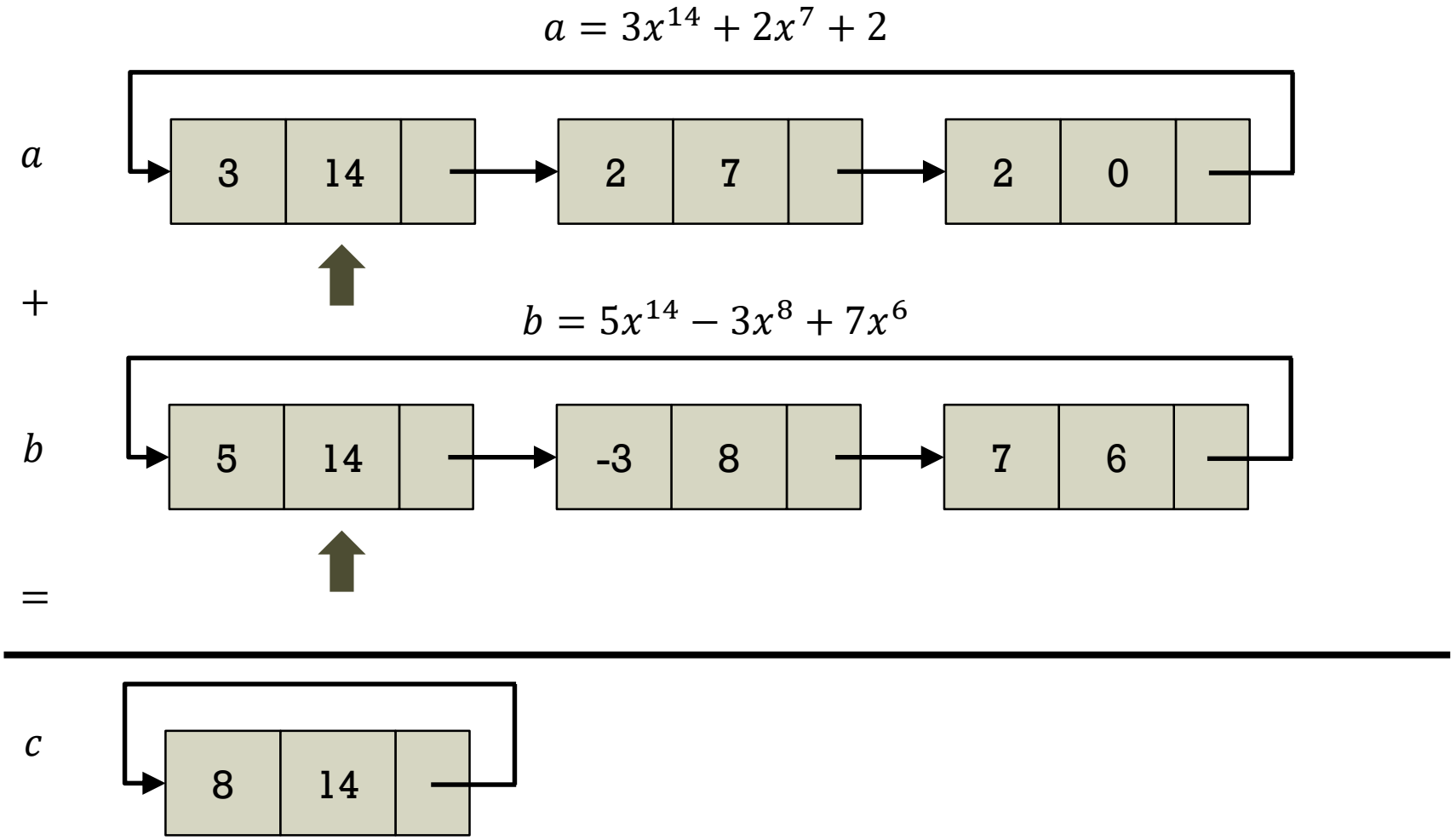
Polynomial using Linked List

- 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.

```
typedef struct _Node
{
     int coef;
     int expon;
     struct _Node* next;
} Node;

typedef struct
{
     Node* tail;
     int len;
} Polynomial;
```

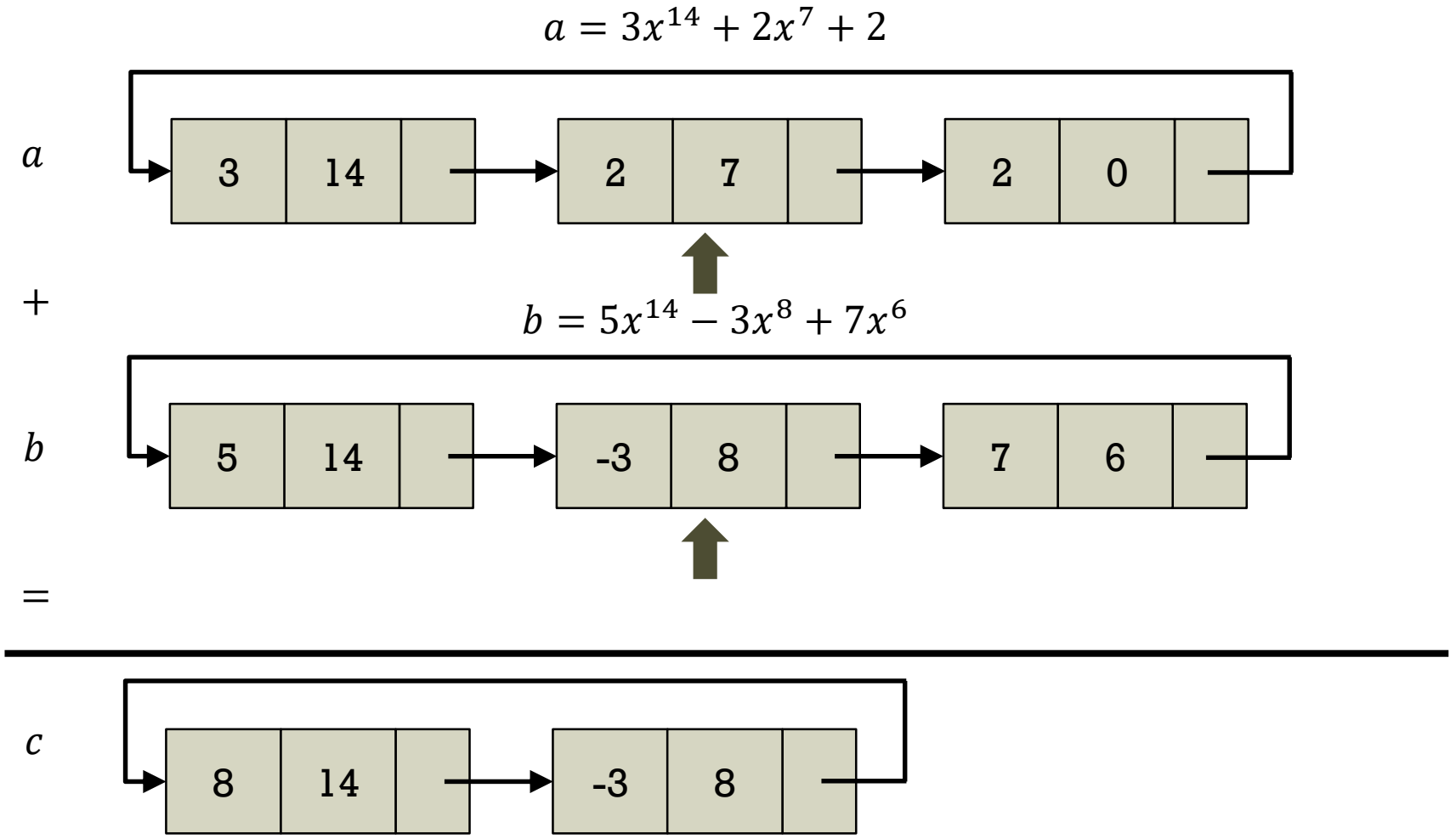
Adding Two Polynomials

- Comparing two exponents



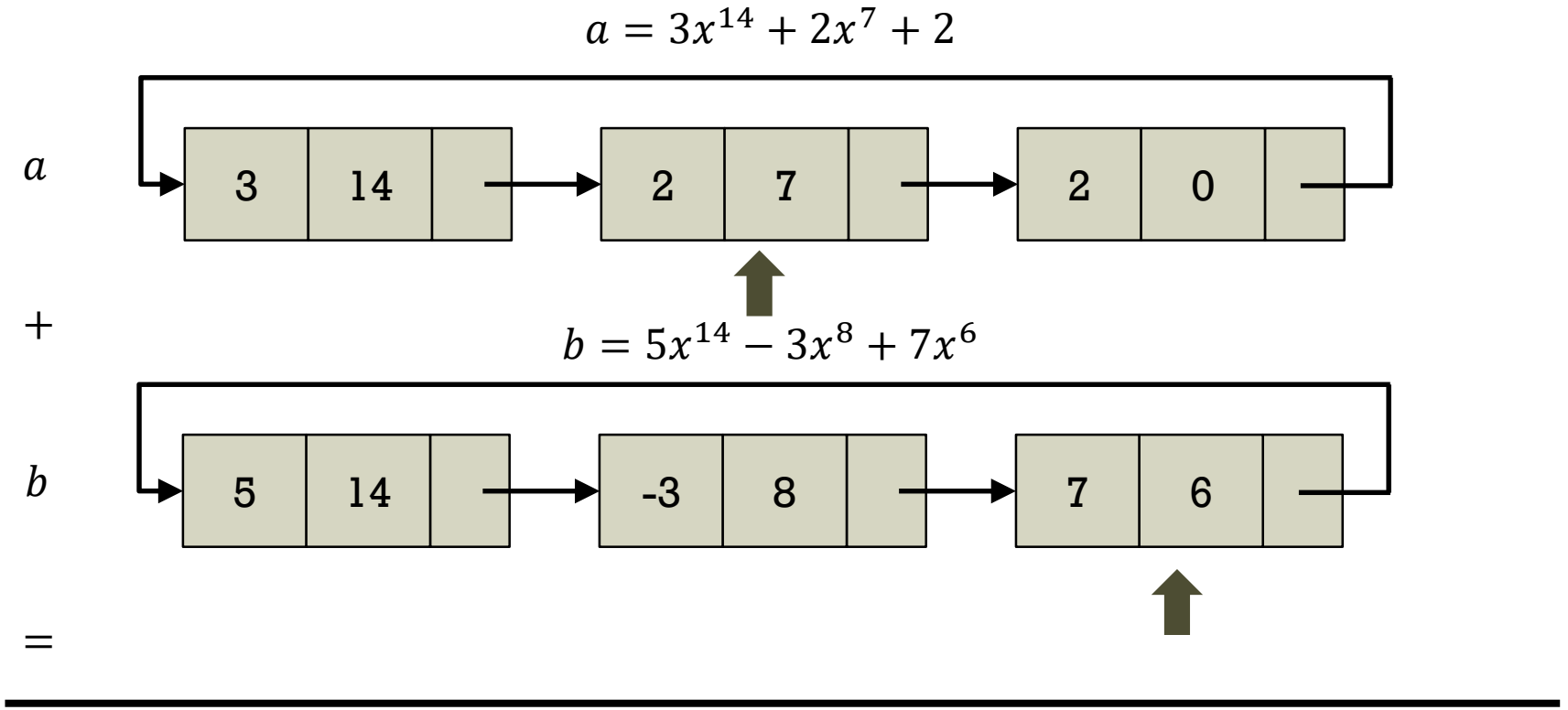
Adding Two Polynomials

- Comparing two exponents



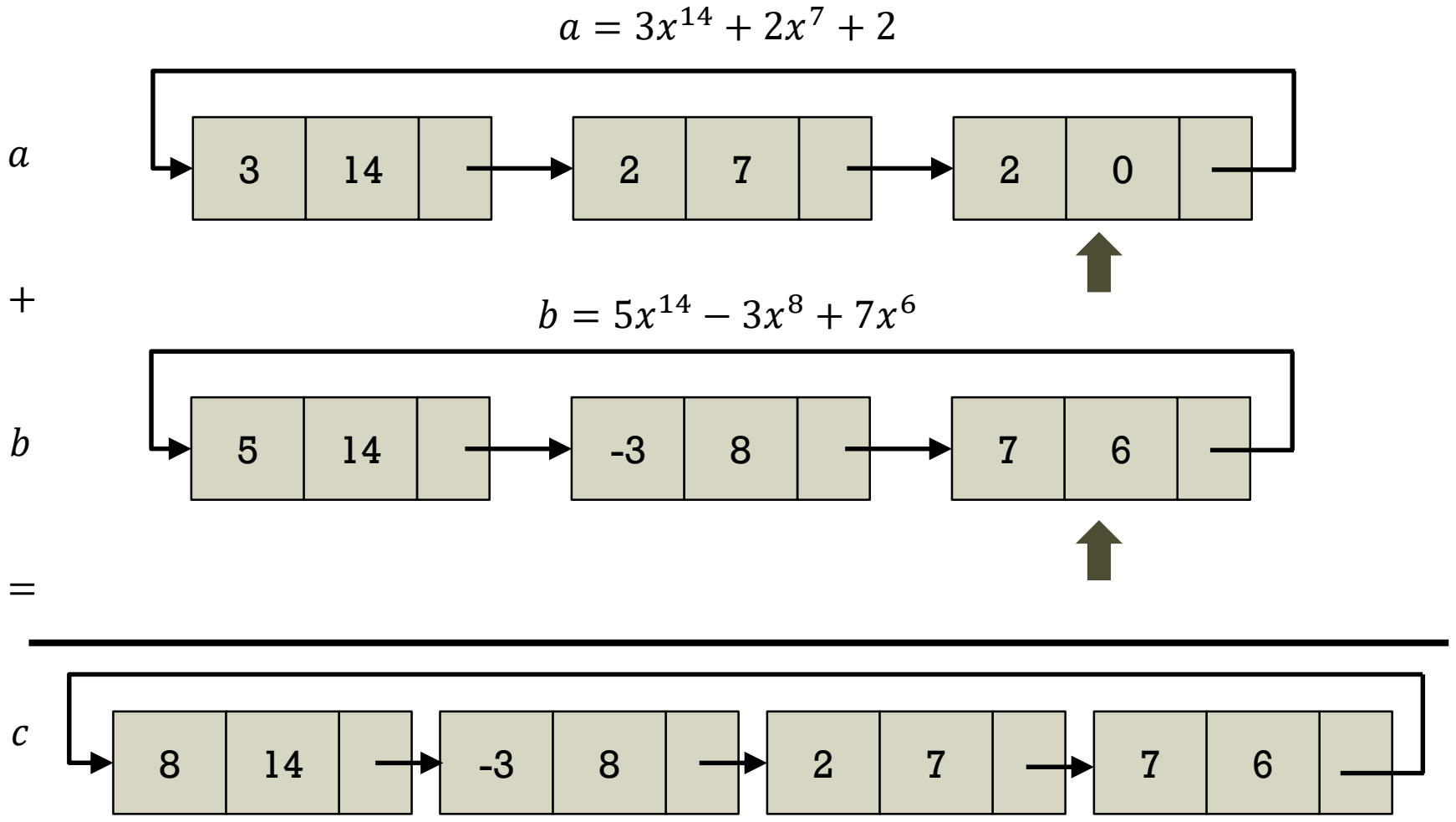
Adding Two Polynomials

- Comparing two exponents



Adding Two Polynomials

- Comparing two exponents



Discussion on Polynomials

- How to implement more operations?
  - Printing polynomials
  - Subtracting two polynomials
  - Multiplying two polynomials

- What is the time complexity of the operations?