Module 2: LINKED LISTS

LINKED LIST

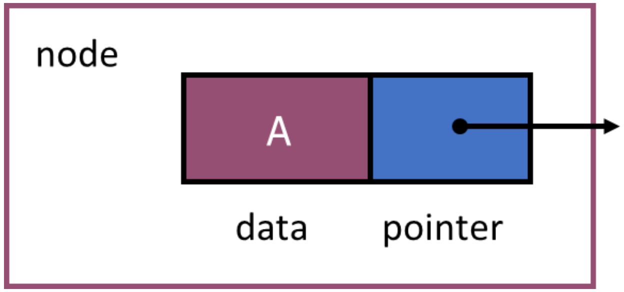
A Linked List is a linear data structure in which elements (called nodes) are connected using pointers.

Structure of a Node

Each node consists of:

- **Data** – stores the actual value.
- **Pointer (next)** – stores the address of the **next node**.

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

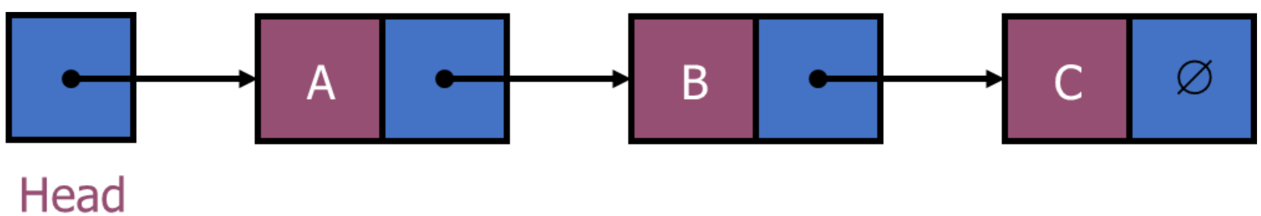


ARRAY VS LINKED LIST

Feature	Array	Linked List
Memory Allocation	Fixed (at compile time)	Dynamic (at runtime using malloc)
Access Time	Fast (direct indexing)	Slow (traverse from head)
Insertion/Deletion	Costly (need shifting)	Easy (adjust pointers)
Memory Utilization	May waste space	Uses exact required memory
Data Storage	Contiguous memory	Non-contiguous (scattered in memory)
Size Flexibility	Fixed size	Dynamic size

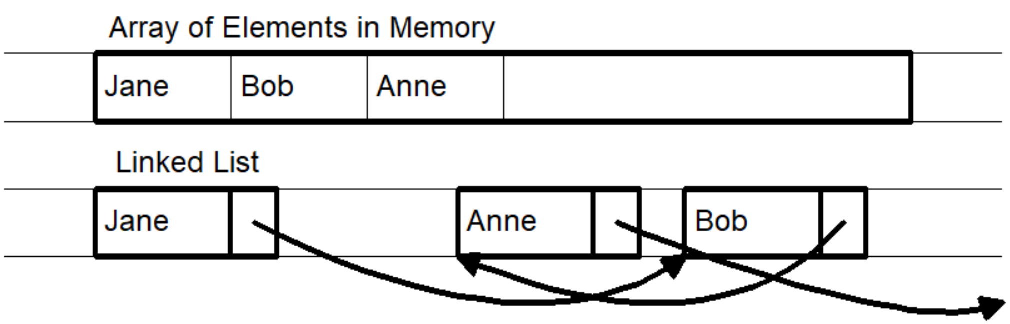
LINKED LISTS REPRESENTATION

- A *linked list* is a series of connected *nodes*
- Each node contains at least
- A piece of data (any type)
- Pointer to the next node in the list
  - *Head*: pointer to the first node
  - The last node points to NULL



DYNAMICALLY ALLOCATING ELEMENTS

- Allocate elements one at a time as needed, and have each element keep track of the next using a pointer.
- The resulting structure is called a **linked list**.



PROS & CONS OF LINKED LIST

Advantages:

- Dynamic memory allocation.
- Easy insertion/deletion (no shifting like arrays).

Disadvantages:

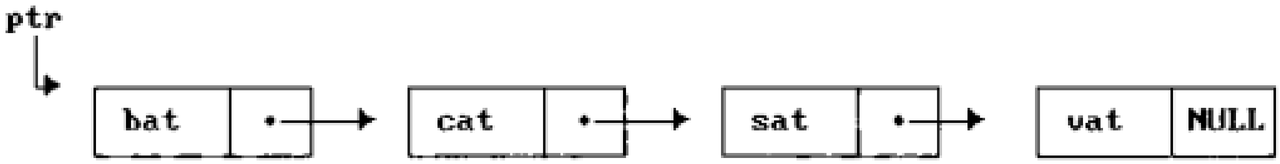
- Sequential access only (no random indexing).
- Extra memory for pointers.

LINKED LIST TYPES

- Singly Linked List
- Doubly Linked List
- Circular Linked List

SINGLY LINKED LISTS (SLL)

1. Each node has only one link part.
2. Each link part contains the address of the next node in the list.
3. The link part of last node contains NULL indicating end of list.
4. The nodes are pointing in only one direction as shown below.



NODE CREATION FOR SLL

Creating the structure of a node in C program using pointers

- **struct node {**
- **int data;**
- **struct node *next;**
- **};**
- **struct node *head, *newNode;**

A node is dynamically allocated

- **newNode = (struct node *)malloc(sizeof(struct node));**

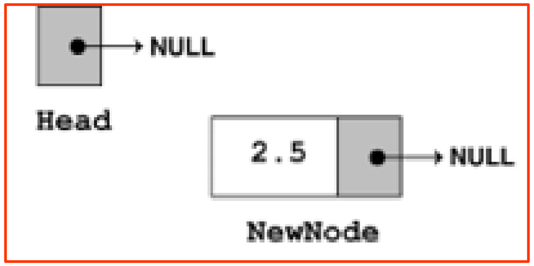
OPERATIONS ON SINGLY LINKED LIST

Operation	Description
Insertion at Beginning	Add a new node at the front. Update head to new node.
Insertion at End	Traverse to the last node and add the new node.
Insertion After a Node	Insert a new node after a specified node.
Deletion at Beginning	Remove the first node. Update head to next node.
Deletion at End	Traverse to second-last node and remove the last node.
Deletion of Specific Node	Find the node before the one to delete, then update the pointer.
Searching an Element	Traverse the list to find a node with a given value.
Reversing the List	Reverse the direction of next pointers. Update head to the last node.

CREATING AND INSERTING A NODE

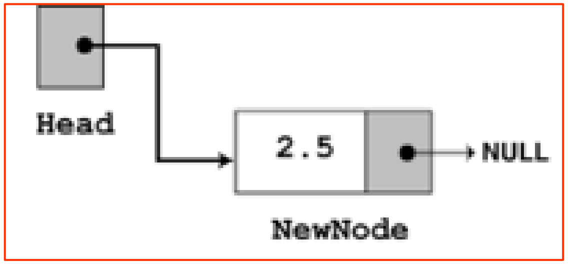
Creating a new node

- struct Node *newNode;
- newNode = (struct Node*)malloc(sizeof(struct Node));
- newNode->data = 2.5;



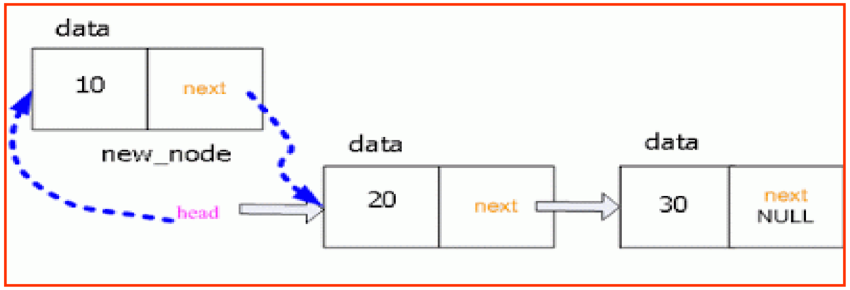
Case 1: If the list is empty

- head=newNode;
- head->next = NULL;



Case 2: If the list already has elements

- newNode->next = head;
- head=newNode;



INSERT AT THE BEGINNING

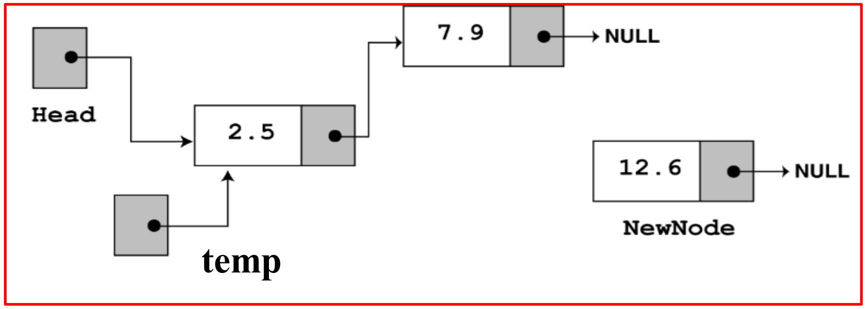
- void beginsert() {
- struct node *newNode;
- int item;
- newNode = (struct node *) malloc(sizeof(struct node));
- if (newNode == NULL) {
- printf("\nOVERFLOW"); }
- else {
- printf("\nEnter value: ");
- scanf("%d", &item);
- newNode->data = item;
- newNode->next = head;
- head = newNode;
- printf("\nNode inserted");
- } }

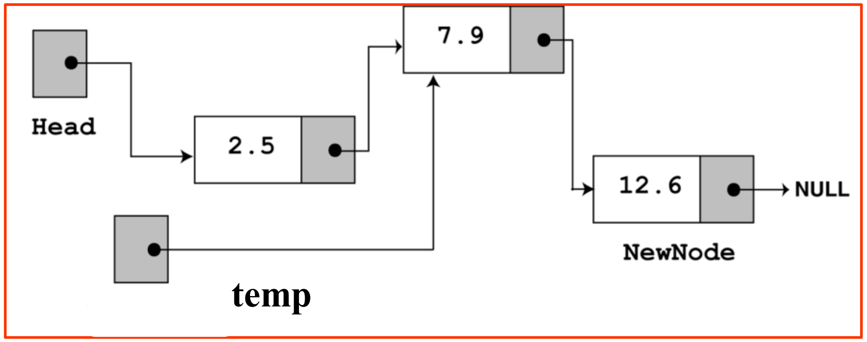
INSERT AT THE END

- void lastinsert() {
- struct node *newNode, *temp;
- int item;
- newNode = (struct node *) malloc(sizeof(struct node));
- if (newNode == NULL) {
- printf("\nOVERFLOW"); }
- else {
- printf("\nEnter value: ");
- scanf("%d", &item);
- newNode->data = item;
- newNode->next = NULL;

INSERT AT THE END

```
if (head == NULL) {
    head = newNode;
    printf("\nNode inserted");
} else {
    temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    printf("\nNode inserted");
}
  }
}
```



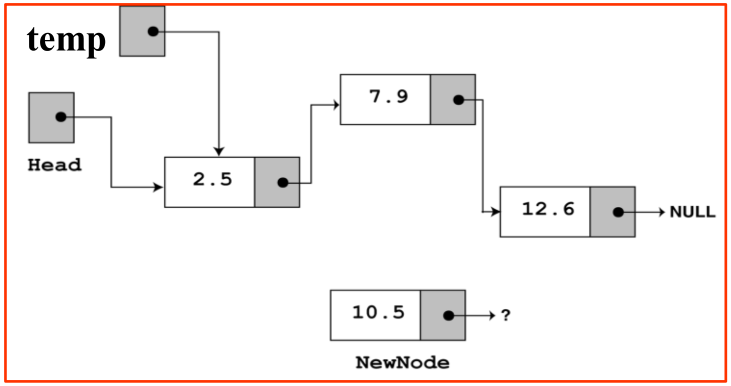


INSERT A NEW NODE IN THE MIDDLE

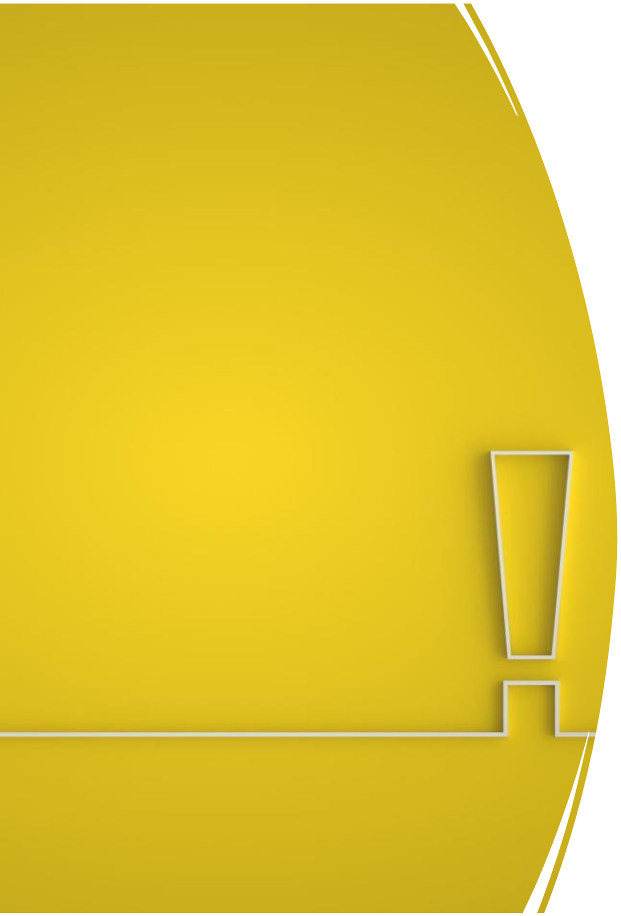
```
• void middleinsert() {
•    int i, loc, item;
•    struct node *newNode, *temp;
•    newNode = (struct node *) malloc(sizeof(struct node));
•    if (newNode == NULL) {
•       printf("\nOVERFLOW");
•    } else {
•       printf("\nEnter element value: ");
•       scanf("%d", &item);
•       newNode->data = item;
•       printf("\nEnter the location after which you want to insert: ");
•       scanf("%d", &loc);
```

INSERT A NEW NODE IN THE MIDDLE

- temp = head;
- for (i = 0; i < loc; i++) {
-         if (temp == NULL) {
-             printf("Can't insert — location exceeds list size");
-             return; }
-         temp = temp->next;
-     }
-     newNode->next = temp->next;
-     temp->next = newNode;
-     printf("\nNode inserted\n");
- } }



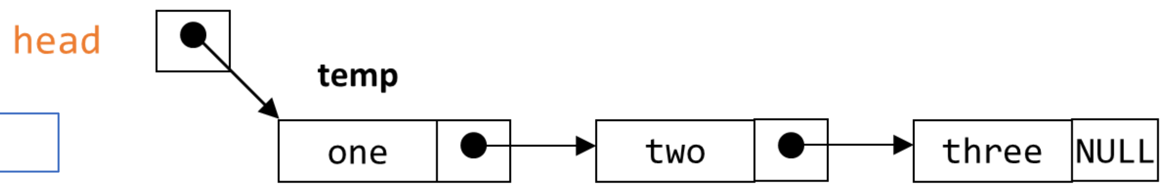
DELETING A NODE IN SLL



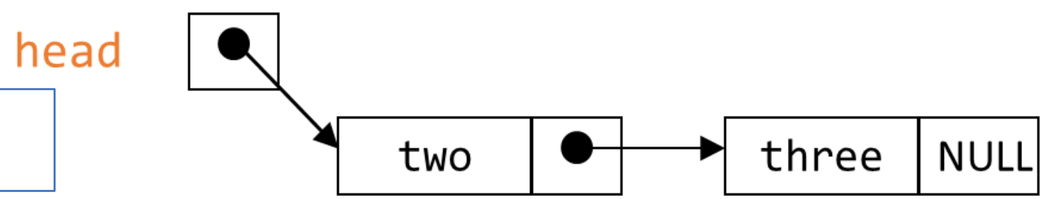
- In a linked list, deleting a node has the following possible cases.
- Delete at the front
- Delete at the last
- Delete in the middle

DELETE AT THE FRONT

- temp=head;



head=temp->next;
free(temp);



DELETE AT THE FRONT

- void delete_front() {
-    struct node *temp;
-    if (head == NULL) {
-       printf("\nList is empty\n");
-    } else {
-       temp = head;
-       head = head->next;
-       free(temp);
-       printf("\nNode deleted from the beginning...\n");
-    }
- }

DELETE AT THE LAST

- void delete_last() {
- struct node *temp, *temp1;
- if (head == NULL) {
- printf("\nList is empty");
- } else if (head->next == NULL) {
- // Only one node in the list
- temp = head;
- head = NULL;
- free(temp);
- printf("\nOnly node of the list deleted...\n");
- } else {

DELETE AT THE LAST

- // More than one node
-       temp = head;
-       // Traverse to the second last node
-       while (temp->next != NULL) {
-          temp1 = temp;
-          temp = temp->next;
-       }
-       temp1->next = NULL;
-       free(temp);
-       printf("\nDeleted node from the end...\n");
-    }
- }

DELETE IN THE MIDDLE

- void delete_middle() {
-     struct node *temp, *temp1;
-     int loc, i;
-     if (head == NULL) {
-         printf("\nList is empty\n");
-         return;   }
-     printf("\nEnter the location (position) of the node to delete (starting from 1): ");
-     scanf("%d", &loc);
-     if (loc == 1) {
-         // Special case: deleting the first node
-         temp = head;
-         head = head->next;
-         free(temp);
-         printf("\nNode at position 1 deleted.\n");
-         return;   }

DELETE IN THE MIDDLE

- temp = head;
- // Traverse to the node just before the one to delete
- for (i = 1; i < loc; i++) {
-     temp1 = temp;
-     temp = temp->next;
-     if (temp == NULL) {
-         printf("\nCan't delete. Position %d doesn't exist.\n", loc);
-         return;   }   }
- temp1->next = temp->next;
- free(temp);
- printf("\nNode at position %d deleted.\n", loc);
- }

ADDITIONAL LIST OPERATIONS IN LINKED LISTS

SEARCH AN ELEMENT IN SLL

- void search() {
- struct node *temp;
- int item, i = 1, found = 0;
- temp = head;
- if (temp == NULL) {
- printf("\nEmpty List\n");
- return; }
- printf("\nEnter item which you want to search: ");
- scanf("%d", &item);

SEARCH AN ELEMENT IN SLL

- while (temp != NULL) {
- if (temp->data == item) {
- printf("Item found at position %d\n", i);
- found = 1;
- break; // remove this if you want to search all occurrences }
- temp = temp->next;
- i++;   }
- if (found == 0) {
- printf("Item not found\n");
- } }

REVERSE THE SLL

- void reverse_list() {
- struct node *prev = NULL, *current = head, *next = NULL;
- if (head == NULL) {
- printf("\nList is empty\n");
- return; }
- while (current != NULL) {
- next = current->next; // store next node
- current->next = prev; // reverse the link
- prev = current; // move prev to current
- current = next; // move current to next }
- head = prev; // update head to new first node
- printf("\nList has been reversed\n"); }

DISPLAY A SLL

- void display_list() {
- struct node *temp;
- temp = head;
- if (head == NULL) {
- printf("\nList is empty\n");
- } else {
- printf("\nLinked List elements:\n");
- while (temp != NULL) {
- printf("%d -> ", temp->data);
- temp = temp->next; }
- printf("NULL\n");
- } }

CONCATENATING TWO SLL

- void concatenate() {
-     struct node *temp;
-     // If first list is empty, concatenated list is just the second list
-     if (head1 == NULL) {
-         head1 = head2;
-         return;   }
-     // If second list is empty, nothing to concatenate
-     if (head2 == NULL) { return;   }
-     temp = head1;
-     // Traverse to the last node of the first list
-     while (temp->next != NULL) {
-         temp = temp->next;   }
-     // Link last node of first list to head of second list
-     temp->next = head2; }

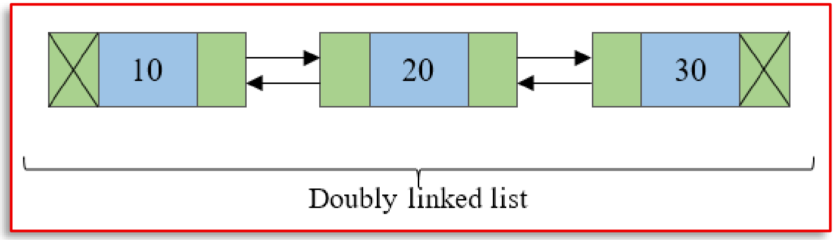
TRY

- Counting how many nodes are present in the list.
- Dividing a linked list into two sublists (e.g., even and odd positioned nodes).
- Deleting/freeing all nodes to release memory.

DOUBLY LINKED LISTS (DLL)

A Doubly Linked List (DLL) is a linear data structure where each node contains:

- Data – the value stored in the node.
- Pointer to the Next Node (next).
- Pointer to the Previous Node (prev).
- This allows bi-directional traversal of the list (both forward and backward).



SLL vs DLL

Feature	Singly Linked List (SLL)	Doubly Linked List (DLL)
Structure	Each node has one pointer (next)	Each node has two pointers (prev and next)
Memory Usage	Requires less memory per node	Requires extra memory for the prev pointer
Traversal Direction	Only forward traversal possible	Both forward and backward traversal possible
Insertion/Deletion	Simpler if at the beginning	Easier at both ends and specific positions
Deleting a node	Need access to previous node explicitly	Can delete a node if a pointer to it is known
Implementation Simplicity	Easier to implement (fewer pointers to handle)	Slightly complex due to handling two pointers
Efficiency in Reversing	Requires traversal and pointer change	Efficient reversing by just swapping head/tail
Use Case Suitability	Suitable when memory is limited and backward traversal isn't needed	Useful in applications requiring bi-directional navigation (e.g., browser history)

STRUCTURE OF A DLL NODE

- struct node
- {
-    int data;
-    struct node *prev;
-    struct node *next;
- };

INSERTION AT BEGINNING IN DLL

- void insert_beginning() {
-     struct node *newNode;
-     int item;
-     newNode = (struct node*)malloc(sizeof(struct node));
-     if (newNode == NULL) {
-         printf("\nOVERFLOW\n"); return;   }
-     printf("\nEnter value: ");
-     scanf("%d", &item);
-     newNode->data = item;
-     newNode->prev = NULL;
-     newNode->next = head;
-     if (head != NULL)
-         head->prev = newNode;
-     head = newNode;
-     printf("\nNode inserted at beginning\n"); }

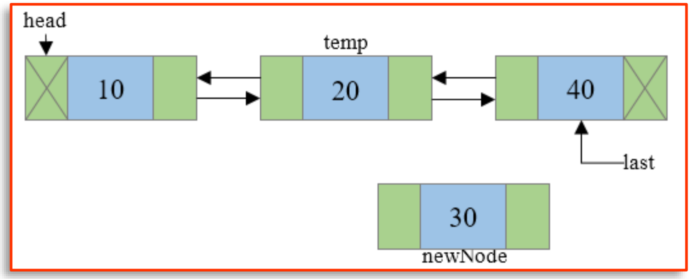
INSERTION AT END IN DLL

- void insert_end() {
-    struct node *newNode, *temp;
-    int item;
-    newNode = (struct node*)malloc(sizeof(struct node));
-    if (newNode == NULL) {
-       printf("\nOVERFLOW\n"); return;   }
-    printf("\nEnter value: ");
-    scanf("%d", &item);
-    newNode->data = item;
-    newNode->next = NULL;
-    if (head == NULL) {
-       newNode->prev = NULL;
-       head = newNode;
-    }

- else {
-       temp = head;
-       while (temp->next != NULL)
-          temp = temp->next;
-       temp->next = newNode;
-       newNode->prev = temp;   }
-    printf("\nNode inserted at end\n"); }

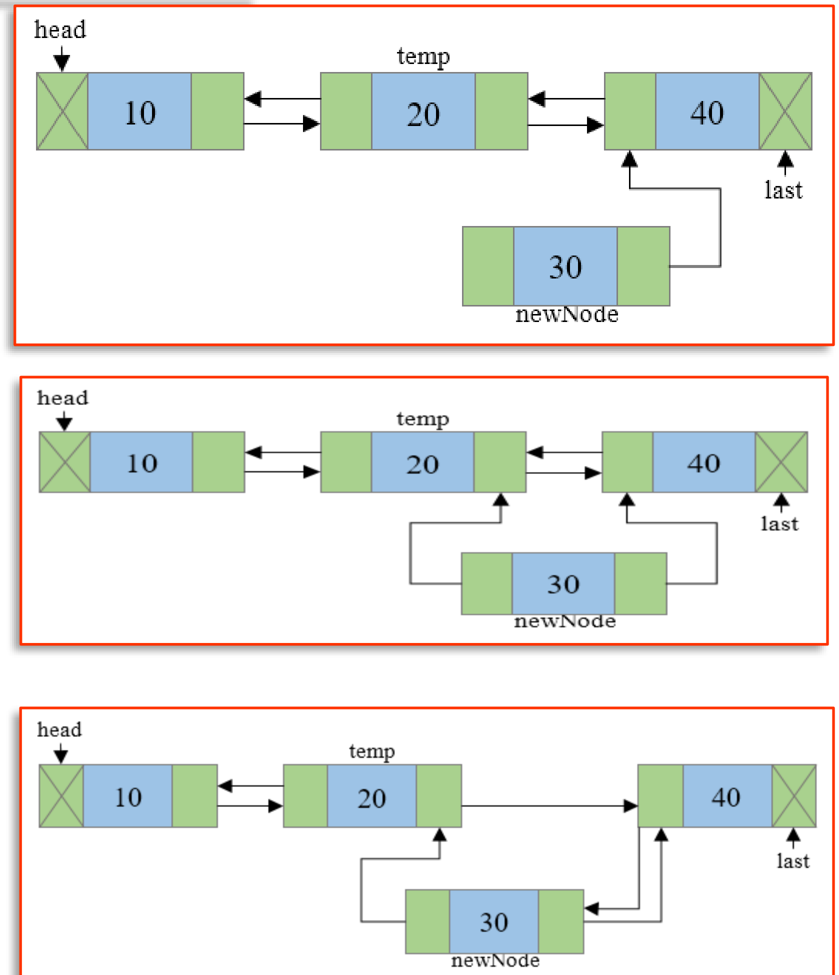
INSERTION AT SPECIFIC POSITION IN DLL

- void insert_middle() {
-     struct node *newNode, *temp;
-     int loc, i, item;
-     printf("\nEnter location after which to insert: ");
-     scanf("%d", &loc);
-     temp = head;
-     for (i = 1; i < loc; i++) {
-         if (temp == NULL) {
-             printf("\nPosition exceeds list size\n");
-             return;   }
-         temp = temp->next;   }
-     newNode = (struct node*)malloc(sizeof(struct node));
-     if (newNode == NULL) {
-         printf("\nOVERFLOW\n");
-         return;   }

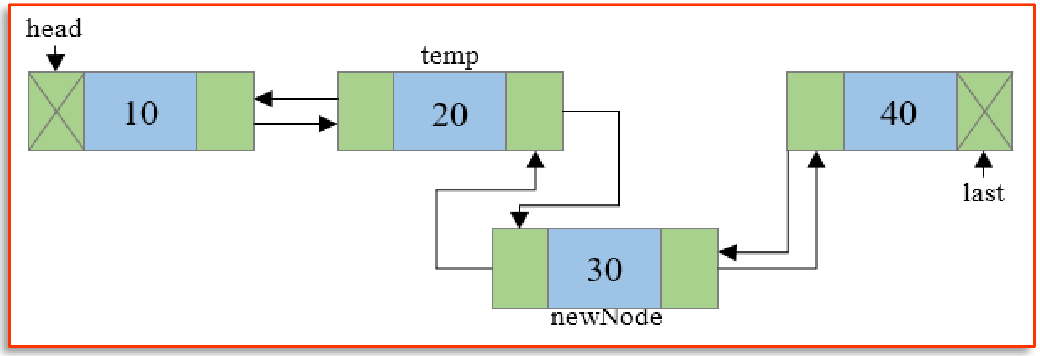


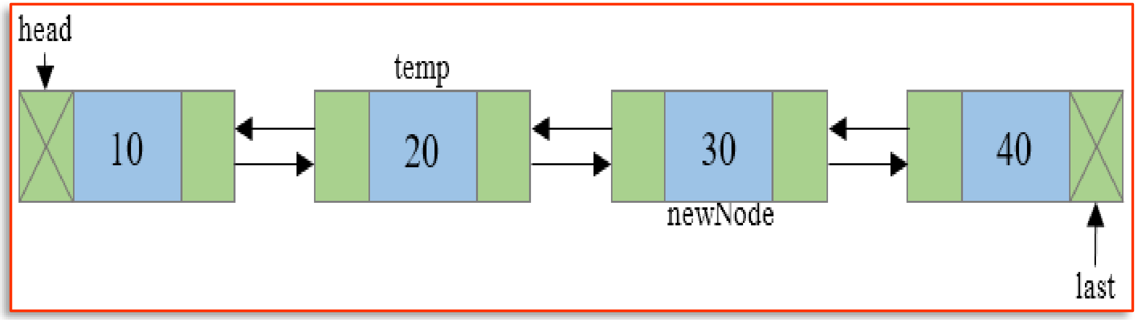
INSERTION AT SPECIFIC POSITION IN DLL

- printf("\nEnter value: ");
- scanf("%d", &item);
- newNode->data = item;
- newNode->next = temp->next;
- newNode->prev = temp;
- if (temp->next != NULL)
- temp->next->prev = newNode;
- temp->next = newNode;
- printf("\nNode inserted at position %d\n", loc+1);
- }



INSERTION AT SPECIFIC POSITION IN DLL





DELETE FROM BEGINNING IN DLL

- void delete_beginning() {
- struct node *temp;
- if (head == NULL) {
- printf("\nList is empty\n");
- return;
- }
- temp = head;
- head = head->next;
- if (head != NULL)
- head->prev = NULL;
- free(temp);
- printf("\nNode deleted from beginning\n");
- }

DELETE FROM END IN DLL

- void delete_end() {
-     struct node *temp;
-     if (head == NULL) {
-         printf("\nList is empty\n");
-         return;   }
-     temp = head;
-     while (temp->next != NULL)
-         temp = temp->next;
-     if (temp->prev != NULL)
-         temp->prev->next = NULL;
-     else
-         head = NULL;
-     free(temp);
-     printf("\nNode deleted from end\n");
- }

DELETE FROM SPECIFIC POSITION IN DLL

- void delete_middle() {
- struct node *temp;
- int loc, i;
- printf("\nEnter location to delete: ");
- scanf("%d", &loc);
- if (head == NULL) {
- printf("\nList is empty\n");
- return; }
- temp = head;
- for (i = 1; i < loc; i++) {
- if (temp == NULL) {
- printf("\nPosition exceeds list size\n");
- return; }
- temp = temp->next; }

DELETE FROM SPECIFIC POSITION IN DLL

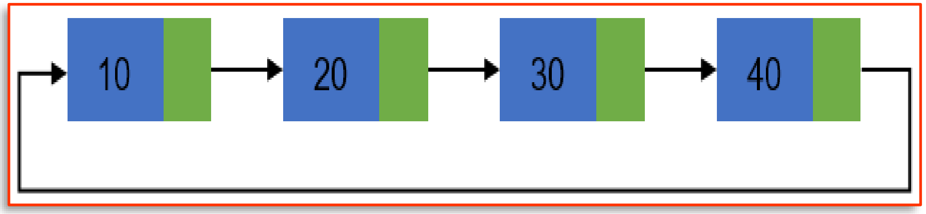
- if (temp == NULL) {
- printf("\nCan't delete. Position doesn't exist\n");
- return;
- }
- if (temp->prev != NULL)
- temp->prev->next = temp->next;
- if (temp->next != NULL)
- temp->next->prev = temp->prev;
- if (temp == head)
- head = temp->next;
- free(temp);
- printf("\nNode at position %d deleted\n", loc); }

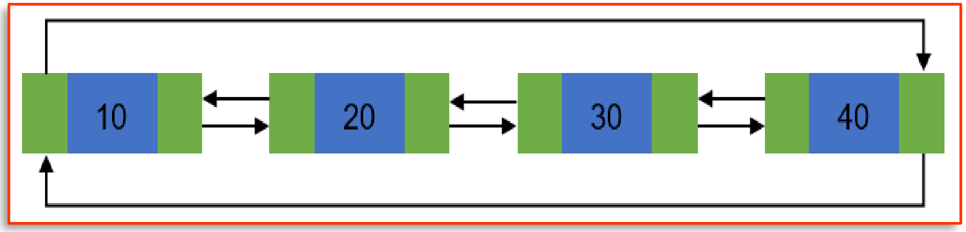
TRY

- Search in DLL
- Forward and Reverse Display of DLL
- Concatenate Two DLLs
- Counting how many nodes are present in the DLL.

CIRCULAR LINKED LIST (CLL)

- A Circular Linked List (CLL) is a variation of linked list where:
- The last node points back to the first node.
- Can be **Singly Circular (only next pointer) or Doubly Circular (both next and prev pointers).**





ADVANTAGES OF CLL OVER SLL AND DLL

Advantage	SLL/DLL Limitation	CLL Advantage
Efficient Circular Traversal	SLL/DLL cannot automatically go back to head	In CLL, after reaching last node, next is head
No need to handle NULL in traversal	In SLL/DLL, need NULL checks to detect end of list	CLL has no NULL, looped back to start
Efficient Queue Implementation	In SLL/DLL, rear to front linking is tricky	In CLL, front and rear are easily connected
Can traverse from any node	SLL/DLL traversal starts only from head	CLL can start traversal from any node
Better for CPU Scheduling (Round Robin)	SLL/DLL need extra management for cyclic processes	CLL naturally cycles through nodes

CONS & USE OF CLL

Disadvantages

- More complex insertion/deletion logic
- Risk of infinite loop

When to Prefer Circular Linked List?

- Implementing **circular queues**.
- **Round Robin scheduling** in operating systems.
- Applications where **repeated cyclic access** is needed.

STRUCTURE OF A CLL NODE

- struct cnode {
-     int data;
-     struct cnode *link; };
- typedef struct cnode* CNODE;

INSERTING AT BEGINNING USING THE LAST PTR

- CNODE insfrl(CNODE last) {
- CNODE temp = (struct cnode *)malloc(sizeof(struct cnode));
- if (temp == NULL) {
- printf("Memory allocation failed\n");
- return last; }
- printf("\nEnter the element:\n");
- scanf("%d", &temp->data);
- if (last == NULL) {
- // First node in the CLL
- last = temp;
- last->link = last; }
- else {
- temp->link = last->link;
- last->link = temp; }
- return last; }

INSERTING AT END USING THE LAST PTR

- CNODE inslast(CNODE last) {
- CNODE temp = (struct cnode *)malloc(sizeof(struct cnode));
- if (temp == NULL) {
- printf("Memory allocation failed\n");
- return last; }
- printf("\nEnter the element:\n");
- scanf("%d", &temp->data);
- if (last == NULL) {
- // First node in the CLL
- last = temp;
- last->link = last; }

- else {
- temp->link = last->link;
- last->link = temp;
- last = temp; }
- return last; }

INSERTING AT END USING THE FIRST PTR

- CNODE insrt(CNODE head) {
-     CNODE temp = (CNODE)malloc(sizeof(struct cnode));
-     CNODE cur;
-     if (temp == NULL) {
-         printf("Memory allocation failed\n");
-         return head;   }
-     printf("Enter the value to be inserted: ");
-     scanf("%d", &temp->data);
-     temp->link = NULL;
-     if (head == NULL) {
-         // First node in CLL
-         head = temp;
-         temp->link = head; }
- else {
-         cur = head;
-         // Traverse to the last node
-         while (cur->link != head)
-             cur = cur->link;
-         cur->link = temp;
-         temp->link = head;   }
-     return head;   }

INSERTING AT BEGINNING USING THE FIRST PTR

```
CNODE insfrnt(CNODE head) {
   CNODE temp = (CNODE)malloc(sizeof(struct cnode));
   CNODE cur;
   if (temp == NULL) {
      printf("Memory allocation failed\n");
      return head;   }
   printf("Enter the value to be inserted: ");
   scanf("%d", &temp->data);
   temp->link = NULL;
   if (head == NULL) {
      // First node in CLL
      head = temp;
      temp->link = head; }
else {
      temp->link = head;
      cur = head;
      // Traverse to the last node
      while (cur->link != head)
         cur = cur->link;
      cur->link = temp;
      head = temp; }
   return head; }
```

DISPLAYING THE CLL

```
• void print(CNODE head) {
•    CNODE h = head;
•    if (head == NULL) {
•       printf("List is empty.\n");
•       return;    }
•    printf("%d ", h->data);
•    h = h->link;
•    while (h != head) {
•       printf("%d ", h->data);
•       h = h->link;
•    }
•    printf("\n");
• }
```

DELETING AN ELEMENT FROM THE BEGINNING USING LAST POINTER

- CNODE dellb(CNODE last) {
- CNODE cur;
- if (last == NULL) {
- printf("No nodes to delete\n");
- return NULL; }
- // Only one node in the list
- if (last->link == last) {
- printf("Element deleted is: %d\n", last->data);
- free(last);
- return NULL; }
- cur = last->link; // First node to be deleted
- last->link = cur->link; // Update last's link to second node
- printf("Item deleted: %d\n", cur->data);
- free(cur);
- return last; }

DELETING AN ELEMENT FROM THE BEGINNING USING FIRST POINTER

- CNODE delfb(CNODE head) {
- CNODE cur;
- if (head == NULL) {
- printf("No nodes to delete\n");
- return NULL; }
- // Only one node in the list
- if (head->link == head) {
- printf("Element deleted is: %d", head->data);
- free(head);
- return NULL; }
- cur = head;
- // Traverse to the last node
- while (cur->link != head) {
- cur = cur->link; }
- CNODE temp = head;
- head = head->link;
- cur->link = head;
- printf("Item deleted: %d", temp->data);
- free(temp);
- return head; }

DELETING AN ELEMENT FROM THE END USING A LAST POINTER

- CNODE delle(CNODE last) {
- if (last == NULL) {
- printf("No elements to delete.\n");
- return NULL; }
- // Only one node in the list
- if (last->link == last) {
- printf("Element deleted is: %d", last->data);
- free(last);
- return NULL; }
- CNODE cur = last->link; // Start from first node
- // Traverse to the node before last
- while (cur->link != last) {
- cur = cur->link; }
- cur->link = last->link;
- printf("Item deleted: %d", last->data);
- free(last);
- last = cur;
- return last; }

DELETING AN ELEMENT FROM THE END USING FIRST POINTER

- CNODE delfe(CNODE head) {
-     CNODE cur, t;
-     if (head == NULL) {
-         printf("No records to delete\n");
-         return NULL;   }
-     // Only one node in the list
-     if (head->link == head) {
-         printf("Deleted item: %d\n", head->data);
-         free(head);
-         return NULL;   }
-     cur = head;
- // Traverse to the second last node
-     while (cur->link->link != head) {
-         cur = cur->link;   }
-     t = cur->link; cur->link = head;
-     printf("Item deleted: %d\n", t->data);
-     free(t);
-     return head;   }

APPLICATIONS USING LINKED LISTS- POLYNOMIALS

$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \ldots + a_0 x^{e_0}$$

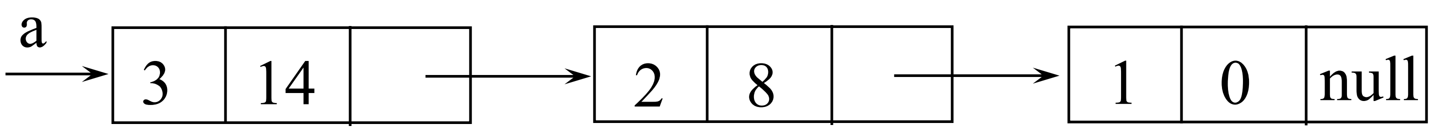
Representation

- typedef struct poly_node *poly_pointer;
- typedef struct poly_node {
- int coef;
- int expon;
- poly_pointer link;
- };
- poly_pointer a, b, c;

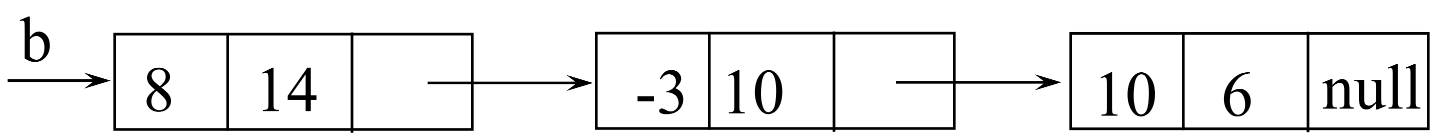
coef	expon	link

EXAMPLES

$$a = 3x^{14} + 2x^{8} + 1$$

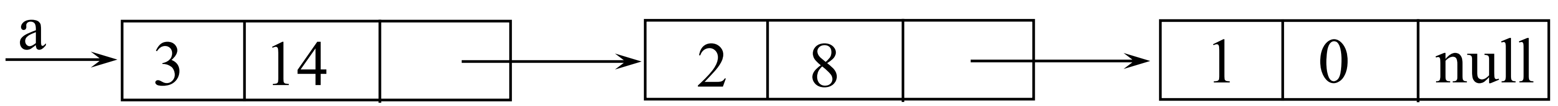


$$b = 8x^{14} - 3x^{10} + 10x^{6}$$

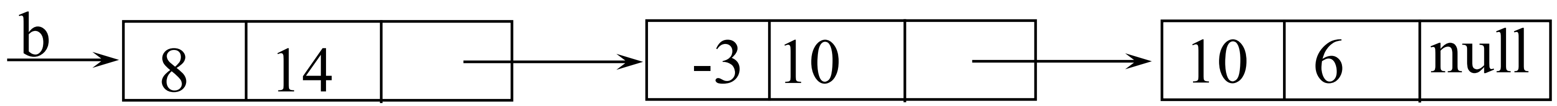


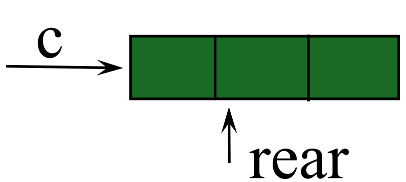
ADDING POLYNOMIALS

$$a = 3x^{14} + 2x^{8} + 1$$

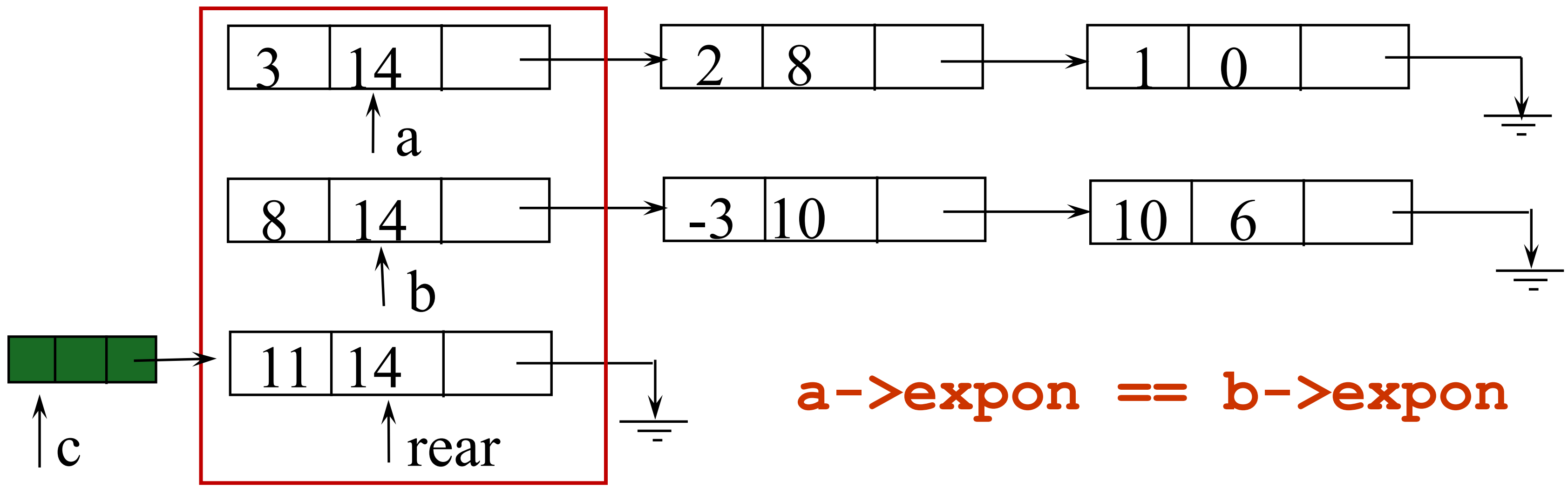


$$b = 8x^{14} - 3x^{10} + 10x^{6}$$

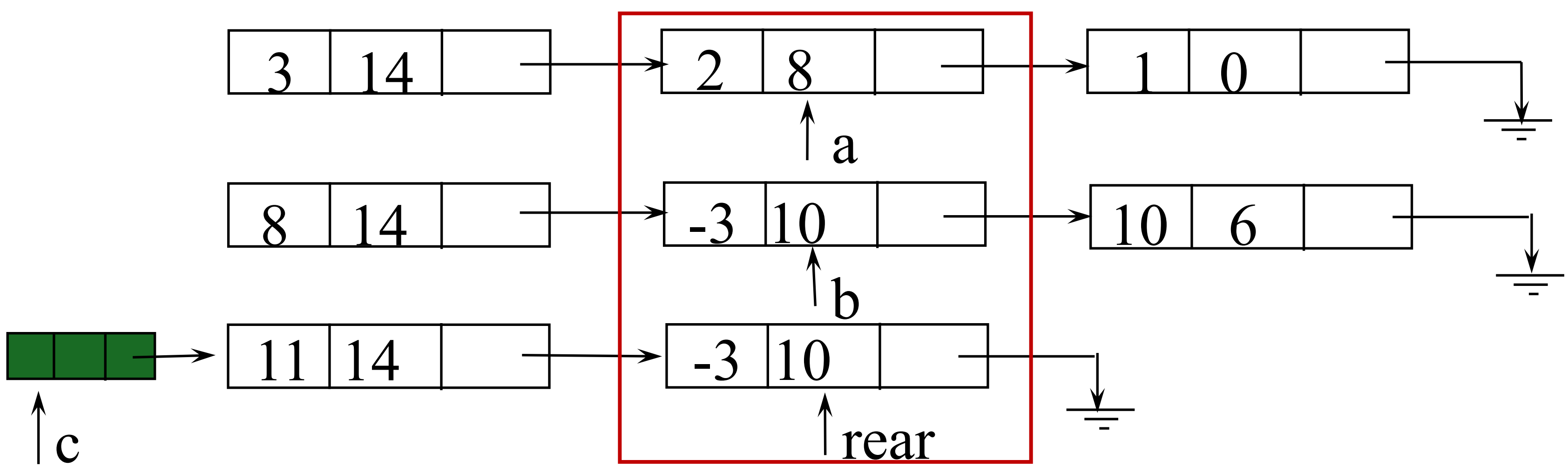




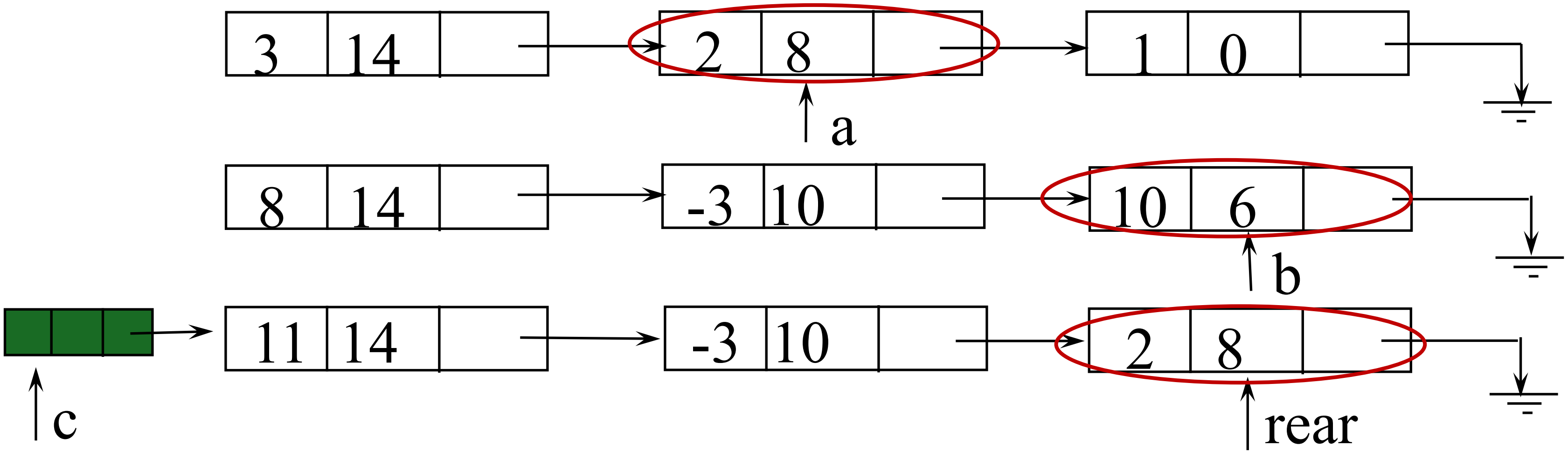
Dummy node



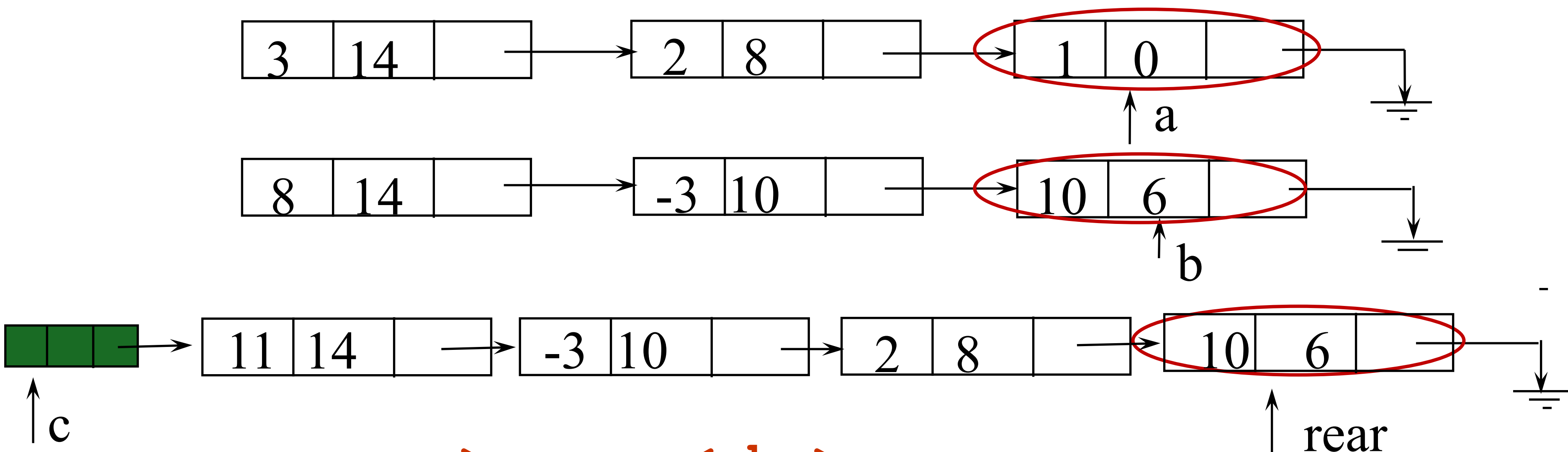
a->expon == b->expon



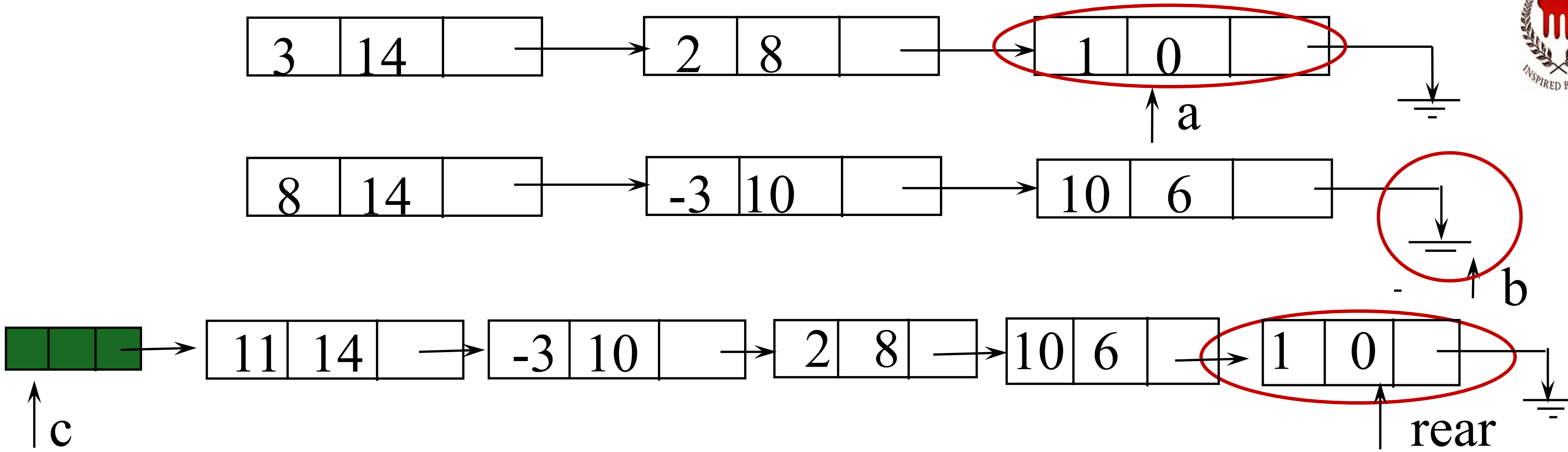
a->expon < b->expon



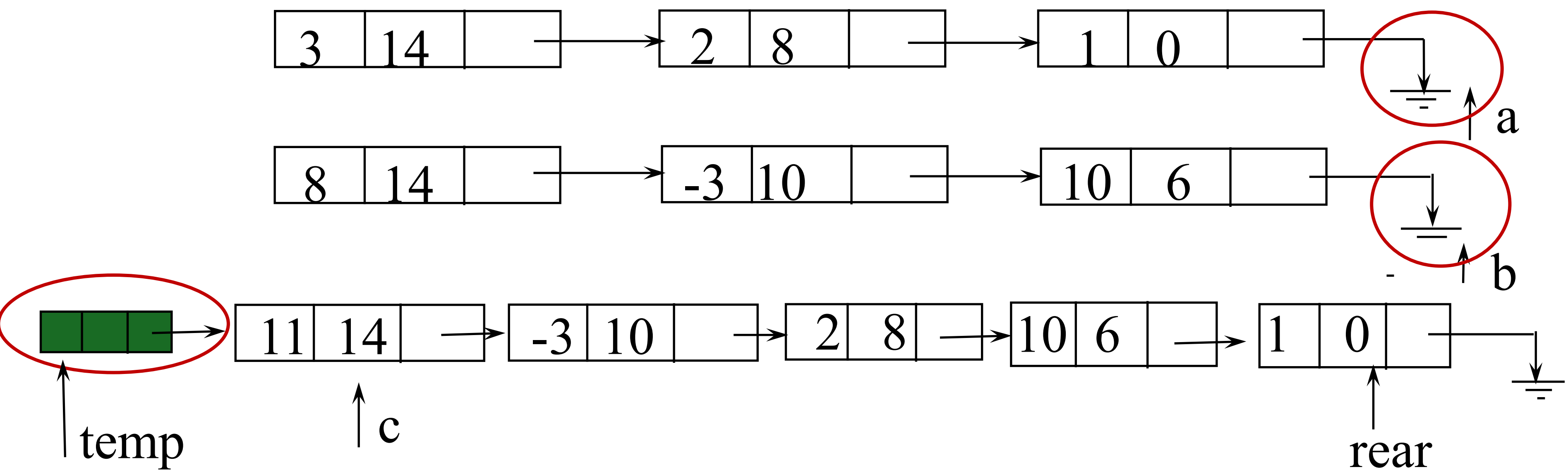
a->expon > b->expon



a->expon < b->expon



b-> NULL Attach remaining nodes from a



Free the dummy node

SINGLY LINKED LIST FOR POLYNOMIAL REPRESENTATION

Structure Definition

- typedef struct poly_node {
- int coef;
- int expon;
- struct poly_node *link;
- } *poly_pointer;

- // Macro to check if memory allocation failed
- #define IS_FULL(ptr) (!(ptr))

GET_NODE() FUNCTION (SLL)

```
• poly_pointer get_node()
• {
•    poly_pointer temp = (poly_pointer)malloc(sizeof(struct poly_node));
•    if (temp == NULL) {
•        printf("Memory allocation failed\n");
•        exit(1);  // Exit the program if memory allocation fails
•    }
•    temp->link = NULL;  // Initialize link to NULL
•    return temp;
• }
```

ADDING POLYNOMIALS - ATTACH FUNCTION (SLL)

- **void attach(int coefficient, int exponent, poly_pointer *ptr)** {
-    poly_pointer temp;
-    temp = (poly_pointer)malloc(sizeof(struct poly_node));
-    if (IS_FULL(temp)) {
-       printf("The memory is full\n");
-       exit(1);
-    }
-    temp->coef = coefficient;
-    temp->expon = exponent;
-    temp->link = NULL;
-    (*ptr)->link = temp; // Attach new node to the list
-    *ptr = temp;   // Move rear pointer to the new node
- }

POLYNOMIAL ADDITION FUNCTION (SLL)

- // Assumed COMPARE macro/function
- #define COMPARE(x, y) (((x) < (y)) ? -1 : ((x) == (y)) ? 0 : 1)
- **poly_pointer padd(poly_pointer a, poly_pointer b)** {
-     poly_pointer c, rear, temp;
-     int sum;
-     // Create dummy node
-     rear = (poly_pointer)malloc(sizeof(struct poly_node));
-     if (rear == NULL) {
-         printf("Memory allocation failed\n");
-         exit(1);
-     }
-     rear->link = NULL;
-     c = rear;

POLYNOMIAL ADDITION FUNCTION (SLL)

```
while (a && b) {
   switch (COMPARE(a->expon, b->expon)) {
      case -1:  // a->expon < b->expon
         attach(b->coef, b->expon, &rear);
         b = b->link;
         break;
      case 0:   // a->expon == b->expon
         sum = a->coef + b->coef;
         if (sum != 0)
            attach(sum, a->expon, &rear);
         a = a->link;
         b = b->link;
         break;
```

POLYNOMIAL ADDITION FUNCTION (SLL)

```
case 1:   // a->expon > b->expon
            attach(a->coef, a->expon, &rear);
            a = a->link;
            break;      }   }
// Attach remaining terms from a
  for (; a; a = a->link)
     attach(a->coef, a->expon, &rear);
  // Attach remaining terms from b
  for (; b; b = b->link)
     attach(b->coef, b->expon, &rear);
  rear->link = NULL;
  // Delete dummy node
  temp = c;    c = c->link;
  free(temp);    return c; }
```

ERASE FUNCTION (FREE THE ENTIRE POLYNOMIAL LIST) (SLL)

- **void erase(poly_pointer *ptr)** {
-     poly_pointer temp;
-     while (*ptr) {
-         temp = *ptr;
-         *ptr = (*ptr)->link;
-         free(temp);
-     }
- }

USAGE EXAMPLE IN MAIN (EXPRESSION E(X) = A(X) * B(X) + D(X)): (SLL)

- **int main()** {
- poly_pointer a, b, d, e, temp;
- // Assume readPoly() reads a polynomial and returns its head pointer
- a = readPoly(); // Read polynomial a
- b = readPoly(); // Read polynomial b
- d = readPoly(); // Read polynomial d
- temp = pmult(a, b); // Multiply a(x) * b(x) --> returns polynomial temp
- e = padd(temp, d); // Add d(x) to temp --> result stored in e(x)
- printPoly(e); // Function to print polynomial e(x)
- erase(&temp); // Free memory allocated to temp polynomial
- erase(&e); // Free result polynomial e
- erase(&a); // Free input polynomials
- erase(&b);
- erase(&d);
- return 0; }

TRY

Polynomial Multiplication

Approach

- For each term in polynomial a, multiply it with every term in polynomial b.
- Insert the result (product term) into a result list.
- If exponents are the same during insertion, combine (add) the coefficients.
- Use attach() to add new terms to the rear.
- After multiplication, remove the dummy node and return the result polynomial.

DOUBLY LINKED LIST FOR POLYNOMIAL REPRESENTATION

- **Structure Definition**
- typedef struct poly_node {
-     int coef;
-     int expon;
-     struct poly_node *prev;
-     struct poly_node *next;
- } *poly_pointer;

POLYNOMIAL ADDITION FUNCTION (DLL)

- **poly_pointer padd(poly_pointer a, poly_pointer b)** {
- poly_pointer c, rear;
- int sum;
- // Create dummy head for result polynomial c
- c = get_node();
- rear = c;
- while (a != NULL && b != NULL) {
- if (a->expon < b->expon) {
- attach(b->coef, b->expon, &rear);
- b = b->next;
- }

POLYNOMIAL ADDITION FUNCTION (DLL)

- else if (a->expon == b->expon) {
- sum = a->coef + b->coef;
- if (sum != 0)
- attach(sum, a->expon, &rear);
- a = a->next;
- b = b->next; }
- else {
- attach(a->coef, a->expon, &rear);
- a = a->next; } }
- // Attach remaining terms from a
- while (a != NULL) {
- attach(a->coef, a->expon, &rear);
- a = a->next; }

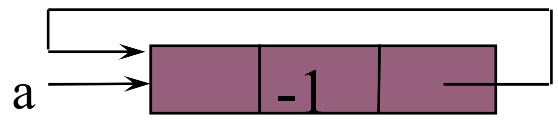
POLYNOMIAL ADDITION FUNCTION (DLL)

- // Attach remaining terms from b
-     while (b != NULL) {
-         attach(b->coef, b->expon, &rear);
-         b = b->next;   }
-     return c->next; // Skip dummy head and return actual result
- }

POLYNOMIALS AS CIRCULARLY LINKED LISTS

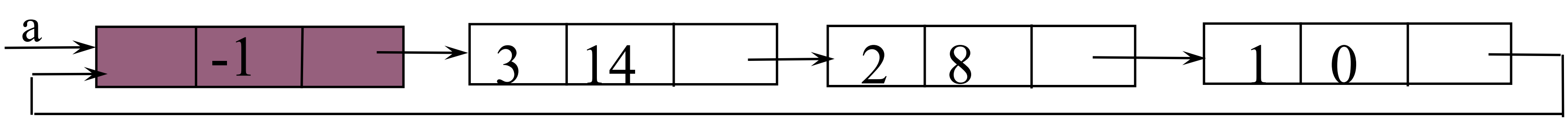
Represent polynomial as circular list.

(1) zero



Zero polynomial

(2) others



$$a = 3x^{14} + 2x^{8} + 1$$

CPADD FUNCTION

- **poly_pointer cpadd(poly_pointer a, poly_pointer b)** {
- poly_pointer c, rear;
- int sum;
- // Move one node ahead to skip the header nodes
- a = a->link;
- b = b->link;
- // Create a dummy header node for the resultant polynomial c
- c = get_node();
- c->expon = -1;
- rear = c;
- // Loop until either a or b reaches their respective header nodes
- while (a->expon != -1 && b->expon != -1) {
-     switch (COMPARE(a->expon, b->expon)) {

CPADD FUNCTION

- case -1: // a->expon < b->expon
-             attach(b->coef, b->expon, &rear);
-             b = b->link;
-             break;
- case 0:   // a->expon == b->expon
-             sum = a->coef + b->coef;
-             if (sum != 0)
-                attach(sum, a->expon, &rear);
-             a = a->link;
-             b = b->link;
-             break;
- case 1:   // a->expon > b->expon
-             attach(a->coef, a->expon, &rear);
-             a = a->link;
-             break;   }   }

CPADD FUNCTION

- // Attach remaining terms of polynomial a
- while (a->expon != -1) {
-     attach(a->coef, a->expon, &rear);
-     a = a->link;
- }
- // Attach remaining terms of polynomial b
- while (b->expon != -1) {
-     attach(b->coef, b->expon, &rear);
-     b = b->link;
- }
- // Complete the circular link (rear to header node c)
- rear->link = c;
- return c; // Return header node of resultant polynomial
- }