Data Structure & Algorithm
CS 102

Dr. Sambit Bakshi

Problem With Array

- Fixed Length (Occupy a Block of memory)

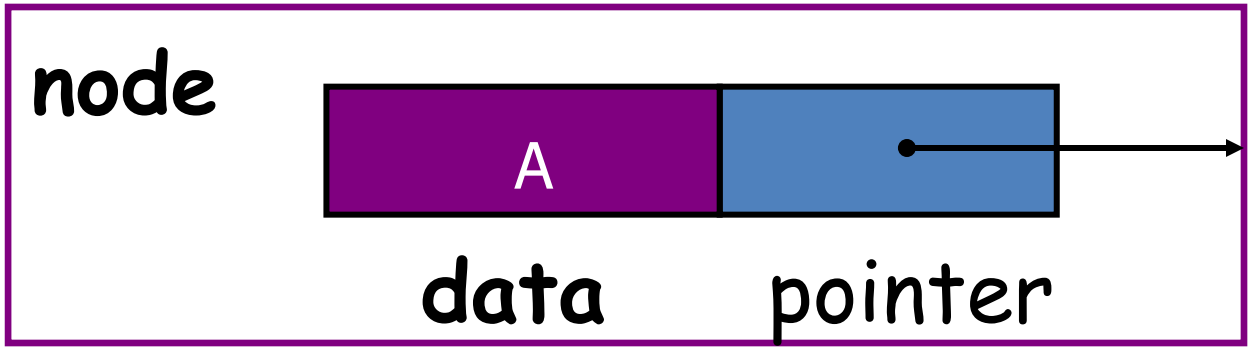
- To expand an Array
  - create a new array, longer in size and
  - copy the contents of the old array into the new array

- Insertion or Deletion

Solution

- Attach a pointer to each item in the array, which points to the next item
  - This is a *linked list*
  - An data item plus its pointer is called a ***node***

Linked List node



- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list

malloc & free

- **void * malloc(n);** -- allocates a memory block of **n** bytes dynamically and returns the base address of the block
- int * ptr;

  ptr = malloc(sizeof(int))

malloc & free

- **void free(void * ptr);** -- deallocates memory block (dynamically created) pointed by ptr
- free(ptr)

Node structure in C

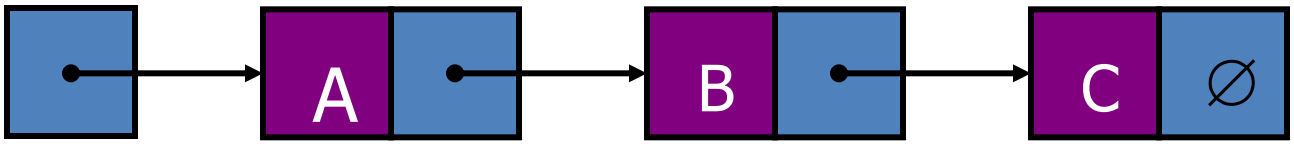
```
struct NODE {
        int DATA;
        struct NODE * Next;
};
```

```
struct NODE *ptr;
ptr = malloc(sizeof(struct NODE))
```

Linked List

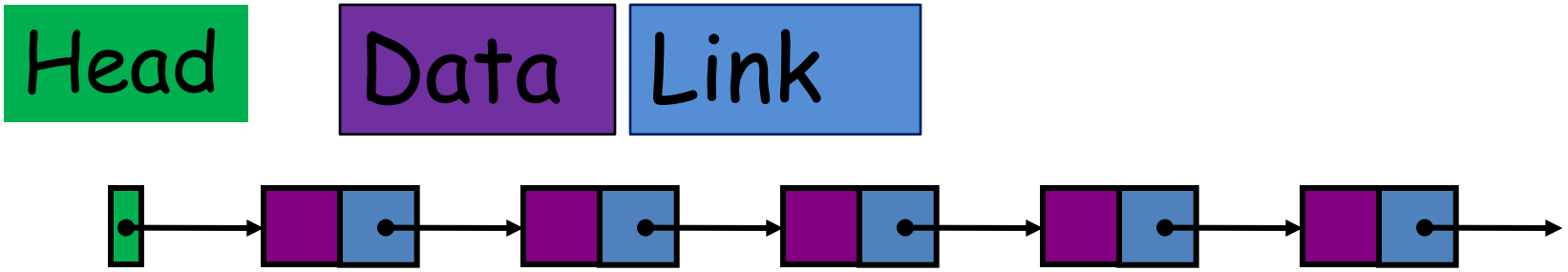
- A linked list, or one-way list, is a linear collection of data elements , called **nodes**, where the linear order is given by means of **pointer**

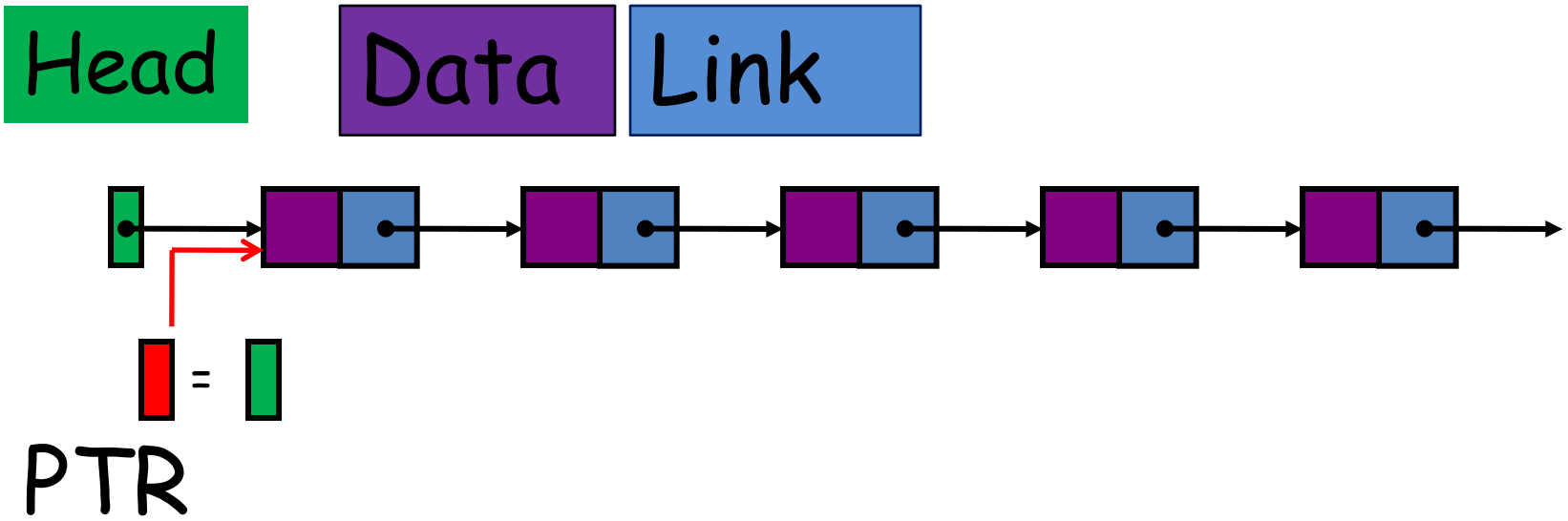
Linked List

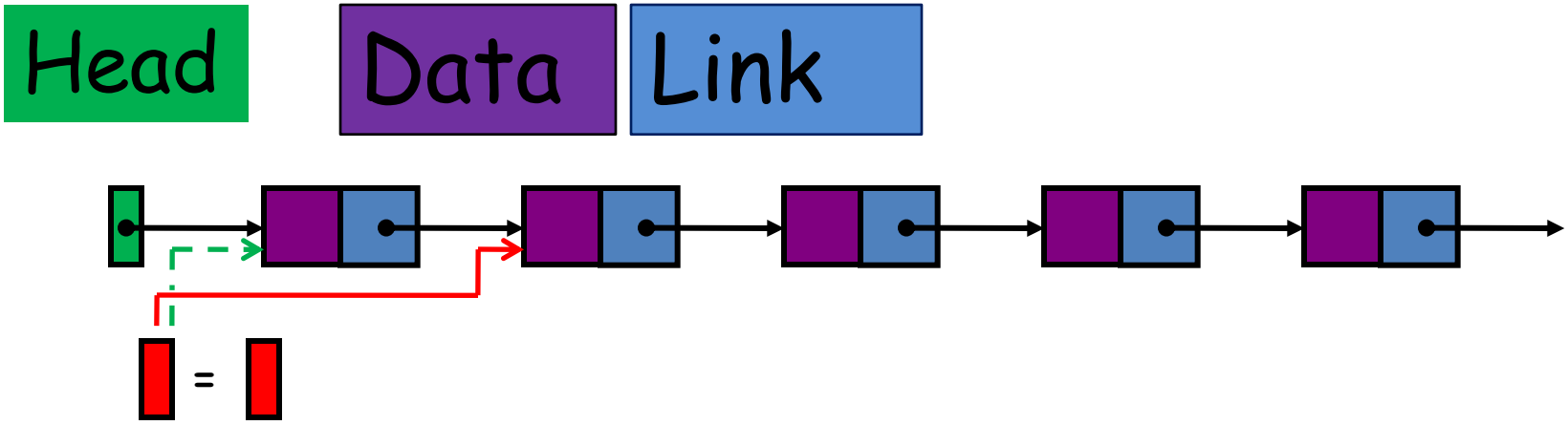


- A **linked list** is a series of connected **nodes**
- **Head** : pointer to the first node
- The last node points to **NULL**

List Traversal




Head
Data
Link
=
PTR



PTR = PTR - > Link

List Traversal

Let Head be a pointer to a linked list in memory. Write an algorithm to print the contents of each node of the list

List Traversal

Algorithm

1. set PTR = Head
2. Repeat step 3 and 4 while PTR ≠ NULL
3. Print PTR->DATA
4. Set PTR = PTR -> LINK
5. Stop

Search for an ITEM

- Let Head be a pointer to a linked list in memory. Write an algorithm that finds the location LOC of the node where ITEM first appears in the list, or sets LOC = NULL if search is unsuccessful.

Search for an ITEM

Algorithm

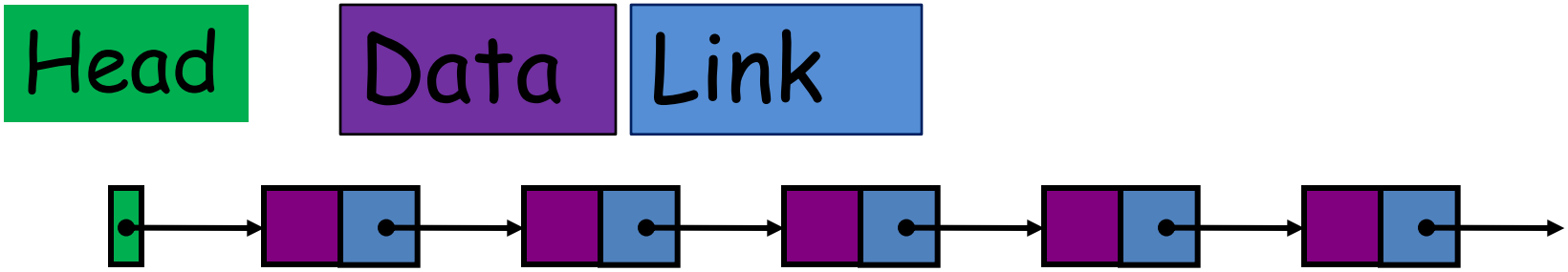
1. Set PTR = Head
2. Repeat step 3 while PTR ≠ NULL
3. if ITEM == PTR -> DATA, then
   Set LOC = PTR, and Exit
   else
   Set PTR = PTR -> LINK
4. Set LOC = NULL /*search unsuccessful */
5. Stop

Search for an ITEM

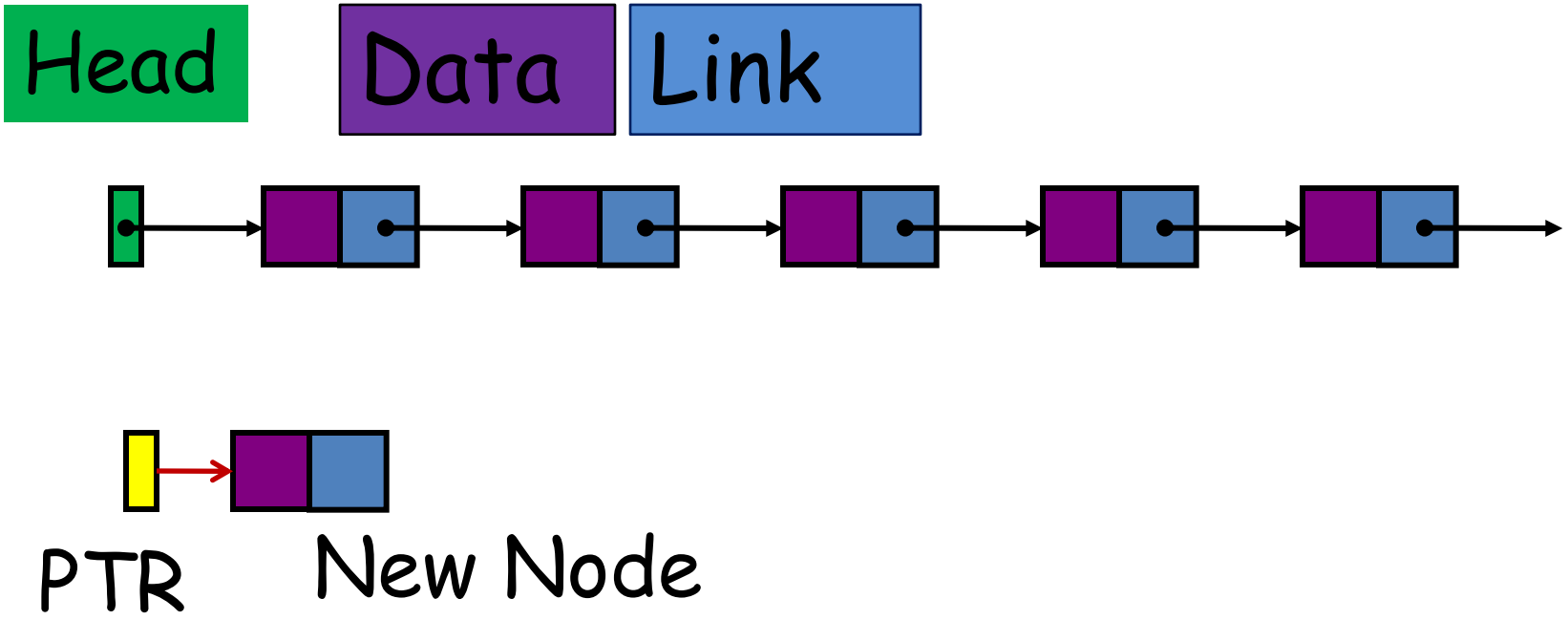
Algorithm [Sorted]

1. Set PTR = Head
2. Repeat step 3 while PTR ≠ NULL
3. if ITEM < PTR -> DATA, then
   Set PTR = PTR->LINK,
   else if ITEM == PTR->DATA, then
   Set LOC = PTR, and Exit
   else
   Set LOC = NULL, and Exit
4. Set LOC = NULL /*search unsuccessful */
5. Stop

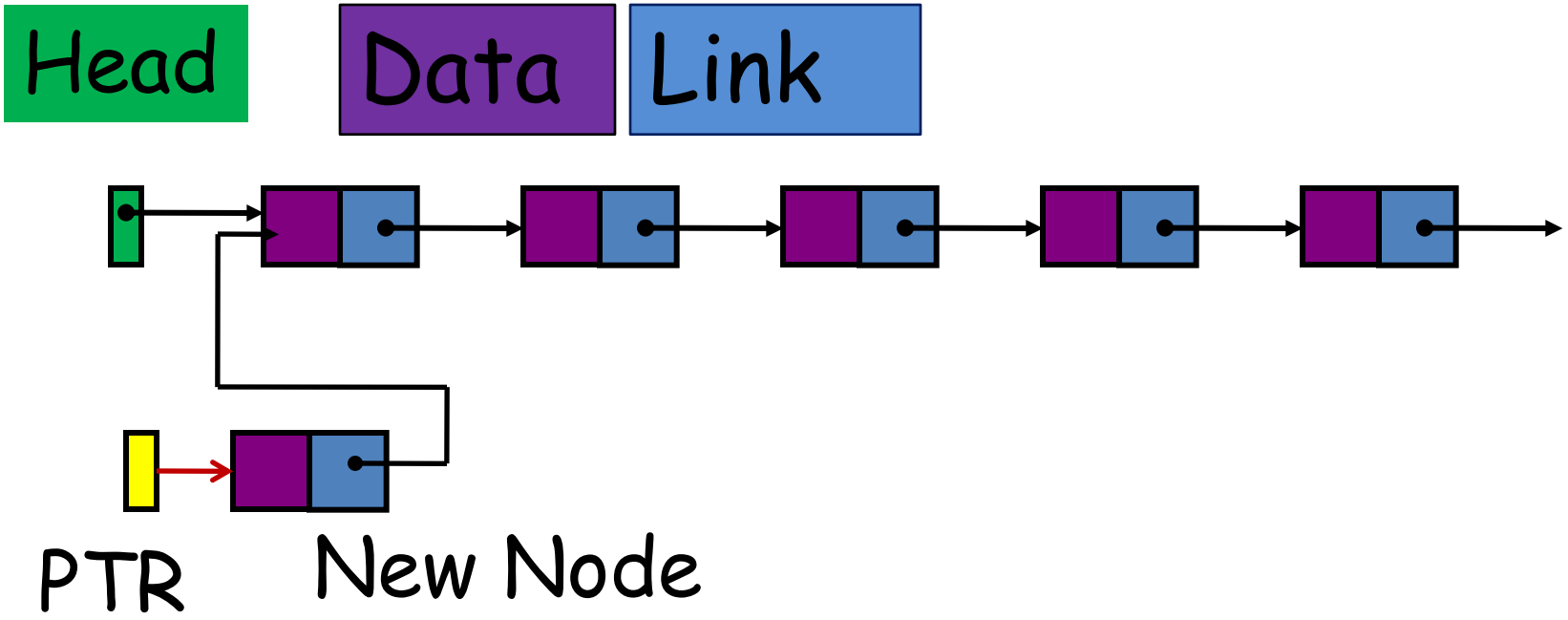
Insertion to a Linked List



Insertion to Beginning

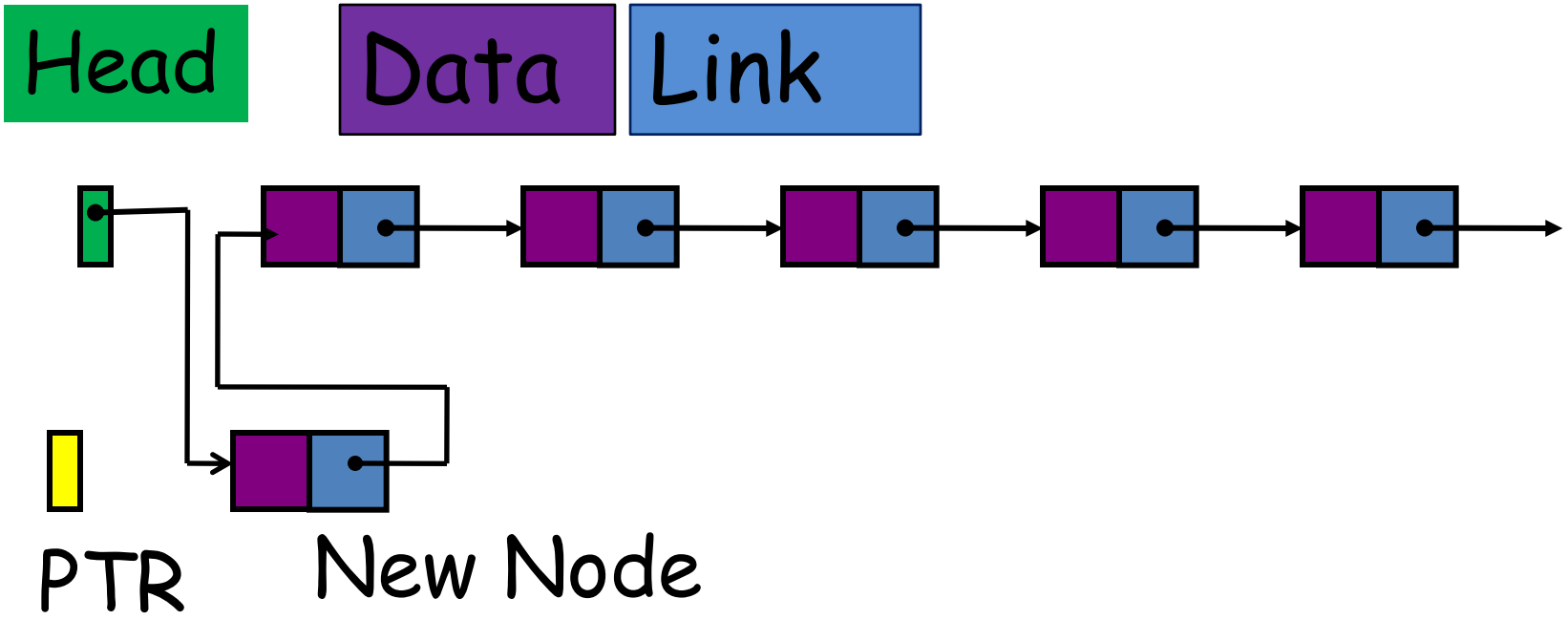


Insertion to Beginning



PTR->Link = Head

Insertion to Beginning



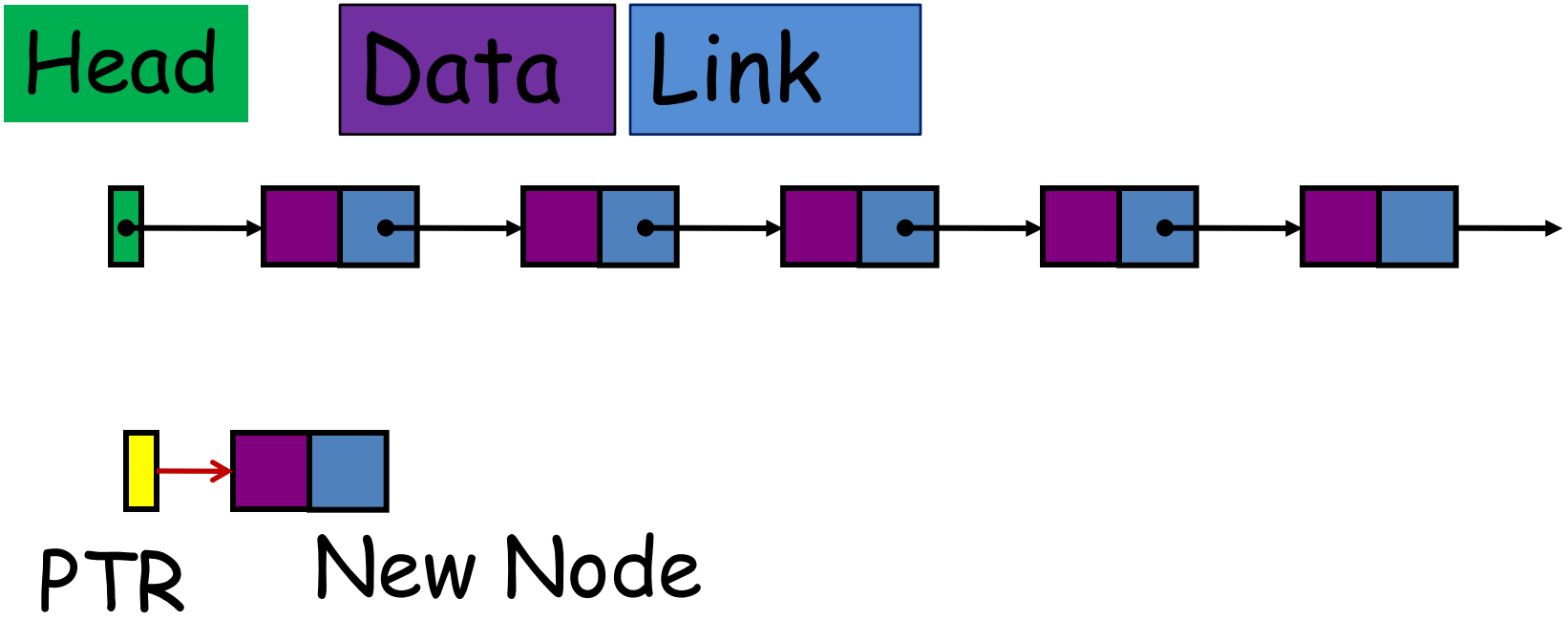
PTR->Link = Head , Head = PTR

Overflow and Underflow

- **Overflow** : A new Data to be inserted into a data structure but there is no available space.

- **Underflow**: A situation where one wants to delete data from a data structure that is empty.

Insertion to Beginning



Overflow , PTR == NULL

Insertion at the Beginning

Let **Head** be a pointer to a linked list in memory. Write an algorithm to insert **node PTR** at the **beginning** of the List.

Insertion at the Beginning

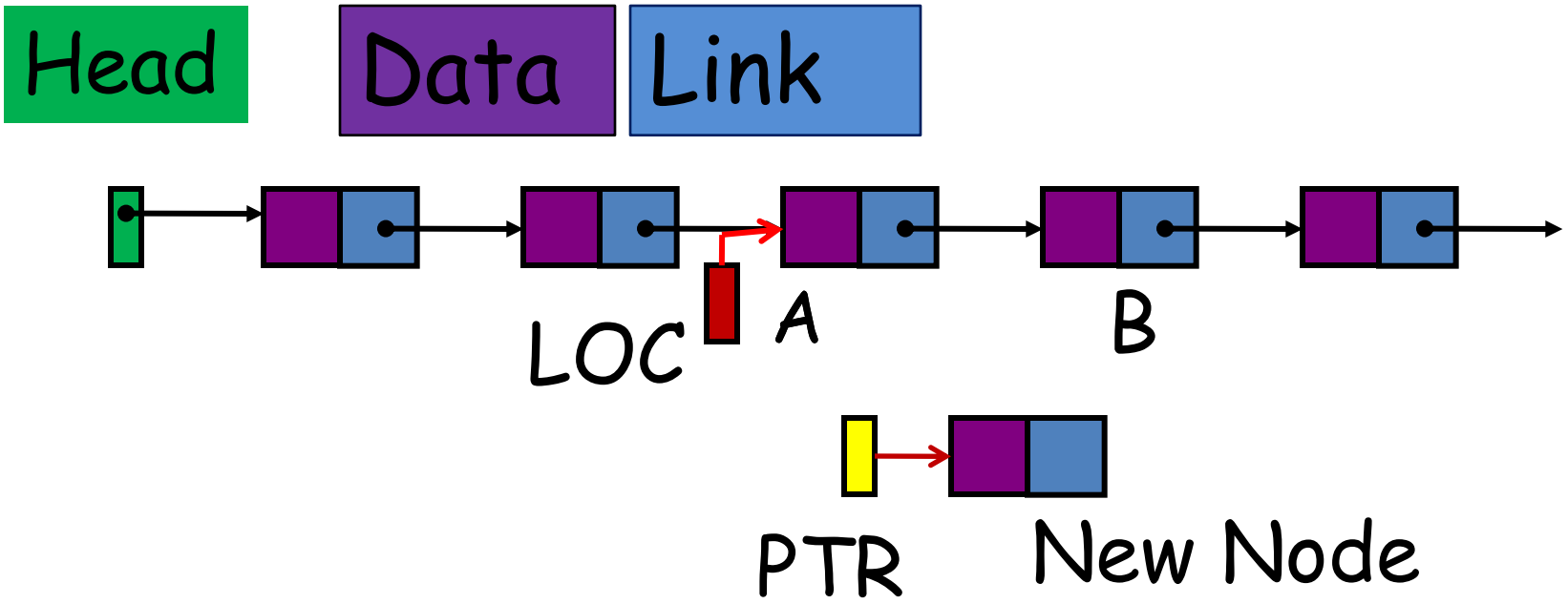
Algorithm

1. PTR = create new node
2. If PTR == NULL , then Write Overflow and Exit
3. Set PTR -> DATA = ITEM
4. Set PTR -> LINK = Head
5. Set Head = PTR
6. Exit

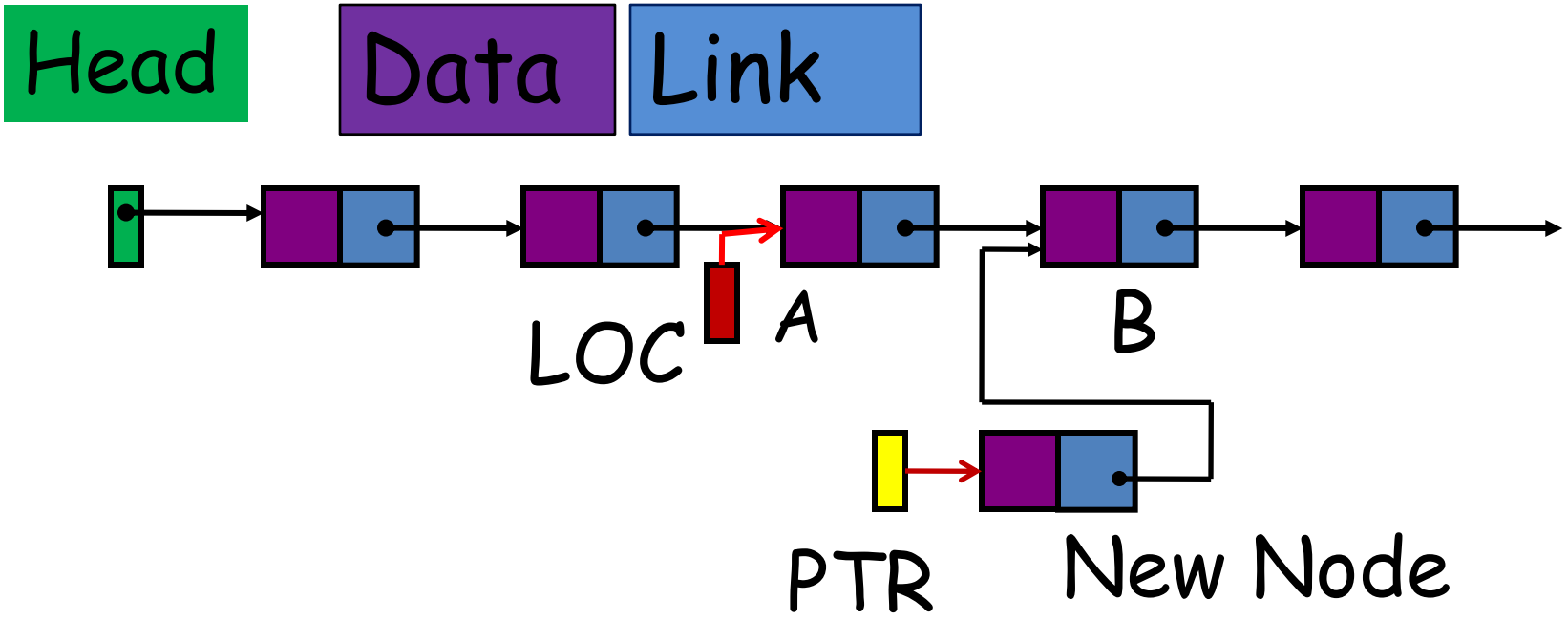
Insertion After a Given Node

Let **Head** be a pointer to a linked list in memory. Write an algorithm to insert **ITEM** so that ITEM follows the node with location **LOC** or insert ITEM as the first node when **LOC == NULL**

Insertion at a Given Node

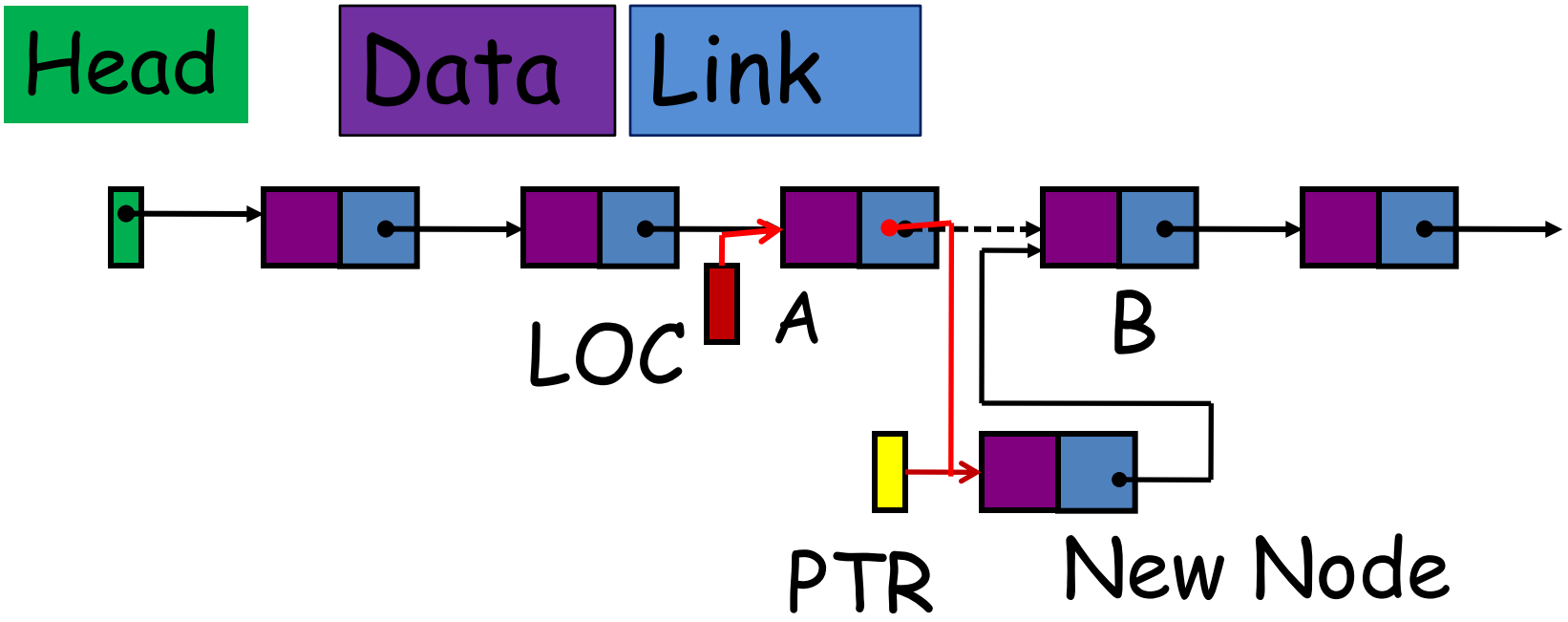


Insertion at a Given Node



PTR->Link = LOC->Link

Insertion at a Given Node



LOC->Link = PTR

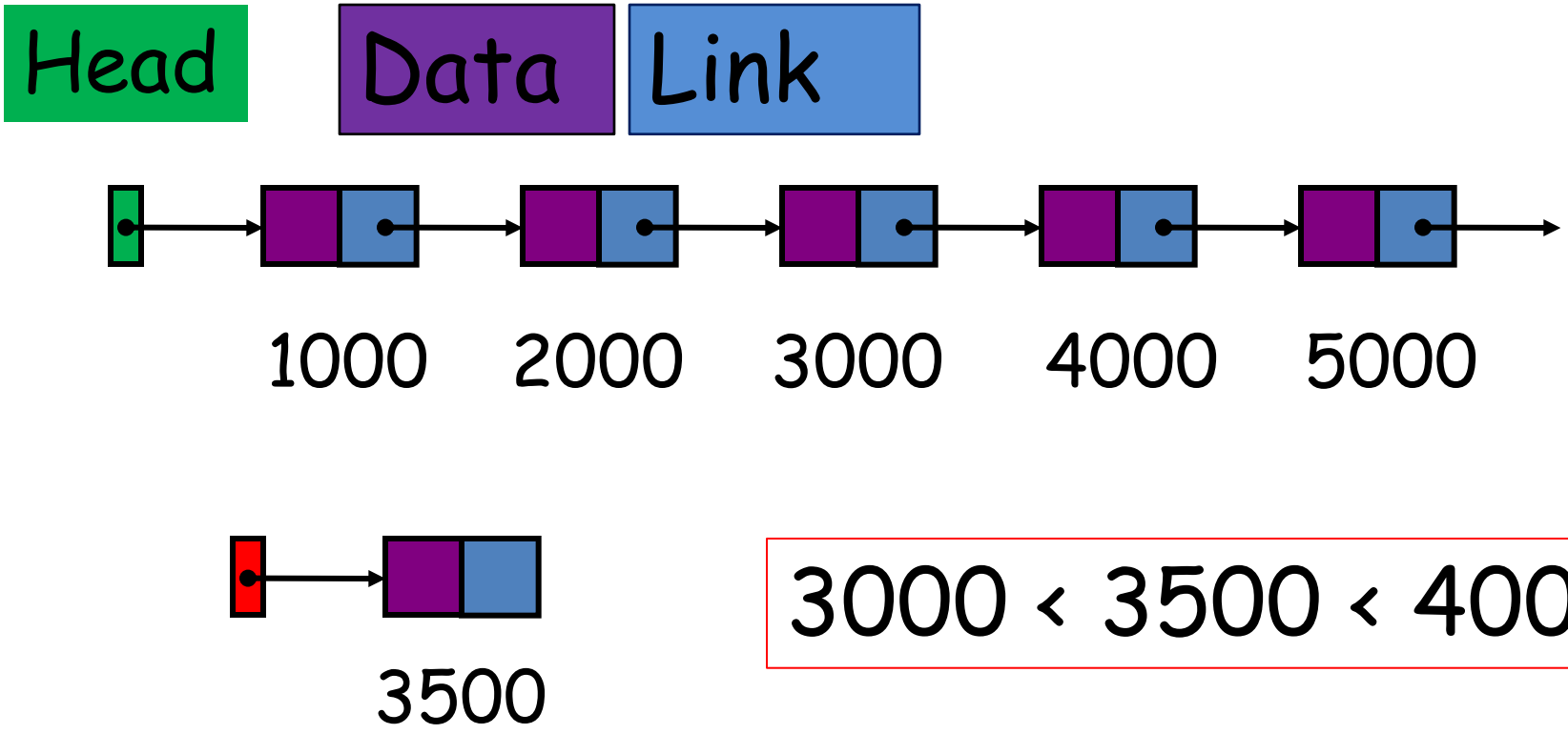
Insertion After a Given Node

Algorithm

1. PTR = create new node
2. If PTR == NULL , then Write Overflow and Exit
3. Set PTR -> DATA = ITEM
4. If LOC == NULL
   - Set PTR -> LINK = Head
   - Set Head = PTR

   Else
   - Set PTR->Link = LOC->Link
   - Set LOC->Link = PTR
5. Exit

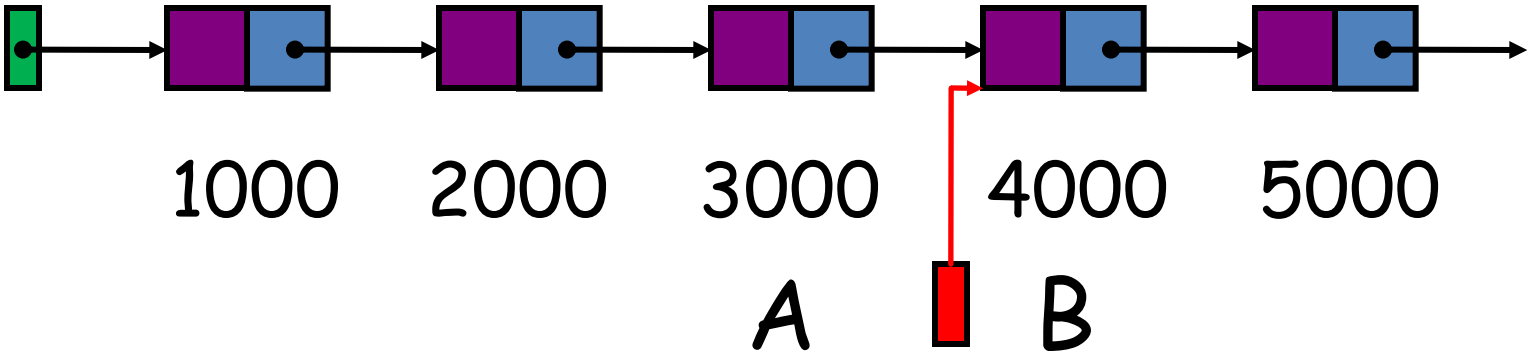
Insertion into a Sorted Linked List



3000 < 3500 < 4000

Insertion into a Sorted Linked List

To Insert Between Node A and B We have to Remember the Pointer to Node A which is the predecessor Node of B



3500 < 4000

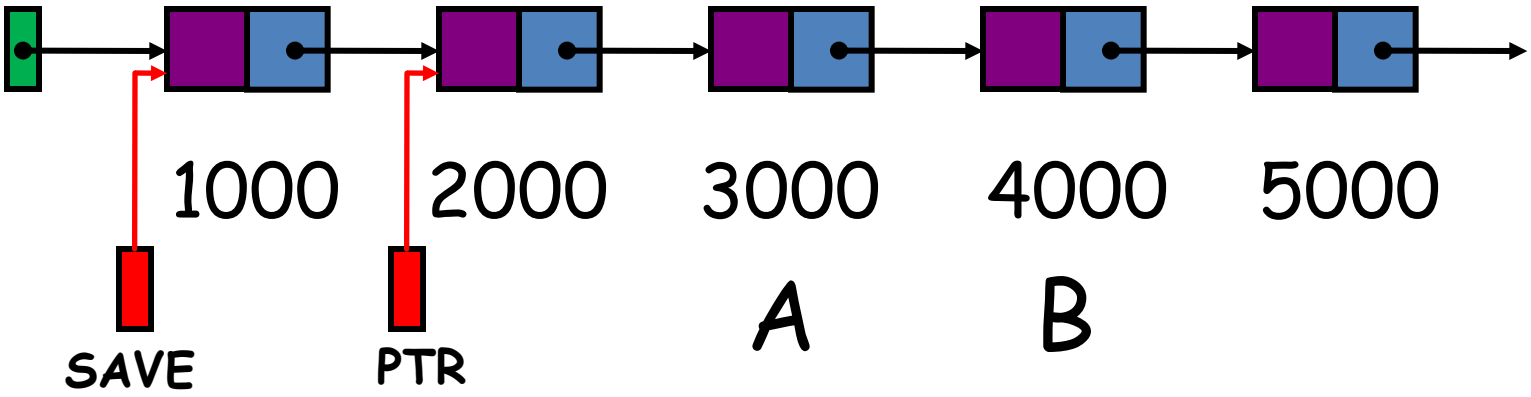
Insertion into a Sorted Linked List

Steps to Find the LOC of Insertion

1. If Head == NULL, then Set LOC = NULL and Return
2. If ITEM < Head ->Data , then Set LOC = NULL and Return
3. Set SAVE = Head and PTR = Head -> Link
4. Repeat Steps 5 and 6 while PTR ≠ NULL
5. If ITEM < PTR -> Data then
   LOC = SAVE and Return
6. Set SAVE = PTR and PTR = PTR->Link
7. Set LOC = SAVE
8. Return

Insertion into a Sorted Linked List

ITEM = 3500

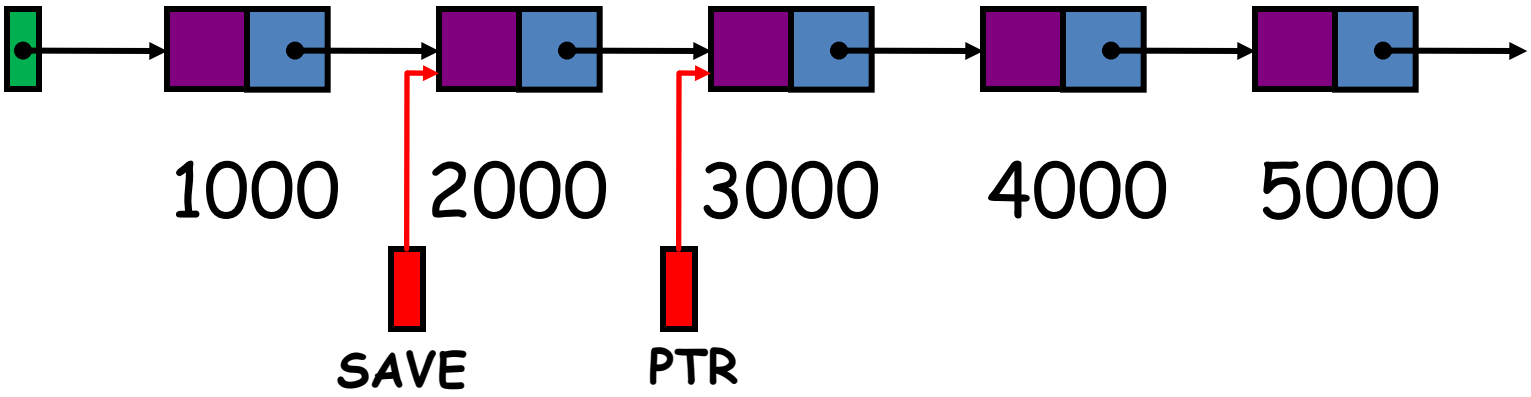


SAVE = Head;

PTR = HEAD -> Link;

Insertion into a Sorted Linked List

ITEM = 3500

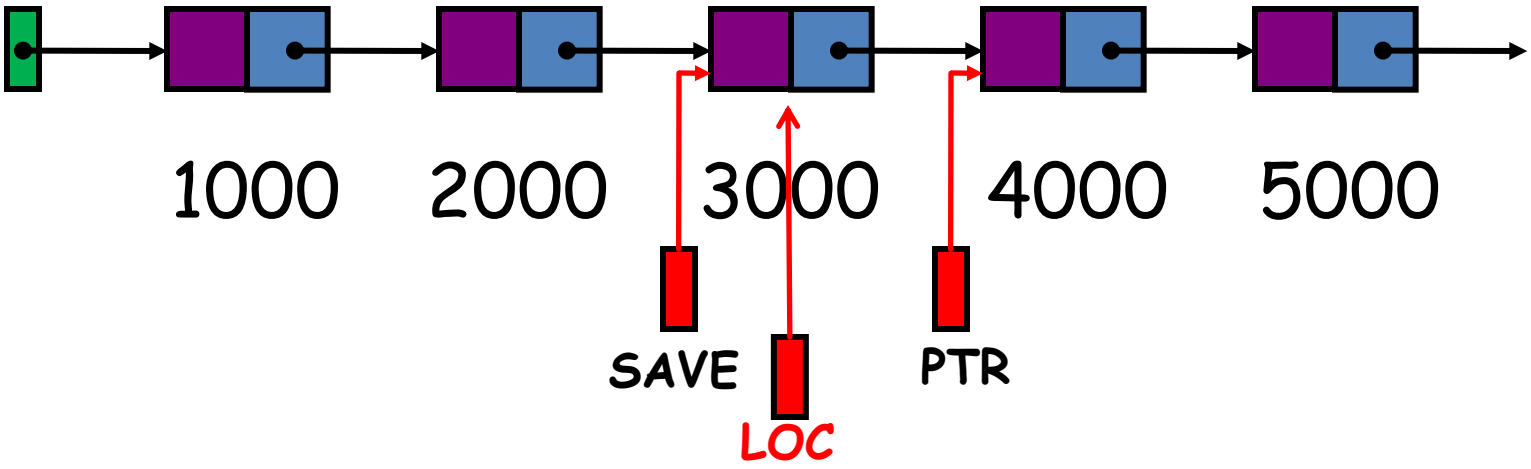


SAVE = PTR;

PTR = PTR -> Link;

Insertion into a Sorted Linked List

ITEM = 3500

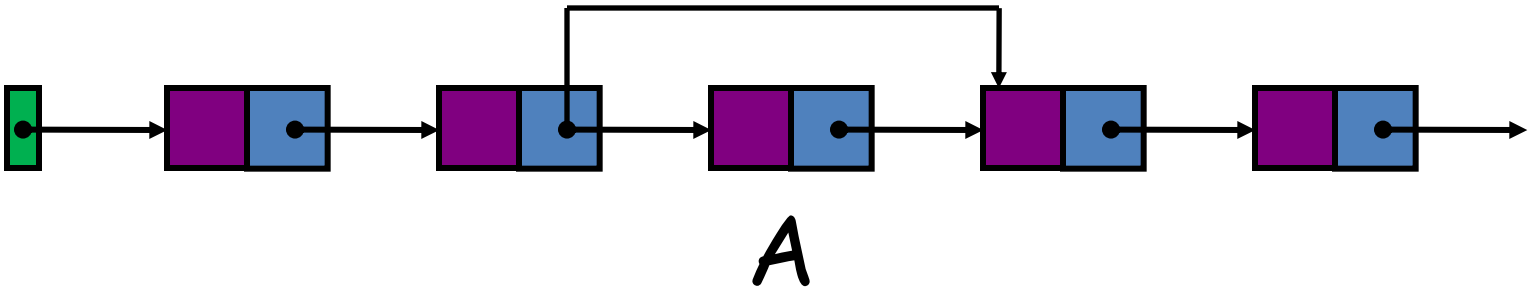


SAVE = PTR;

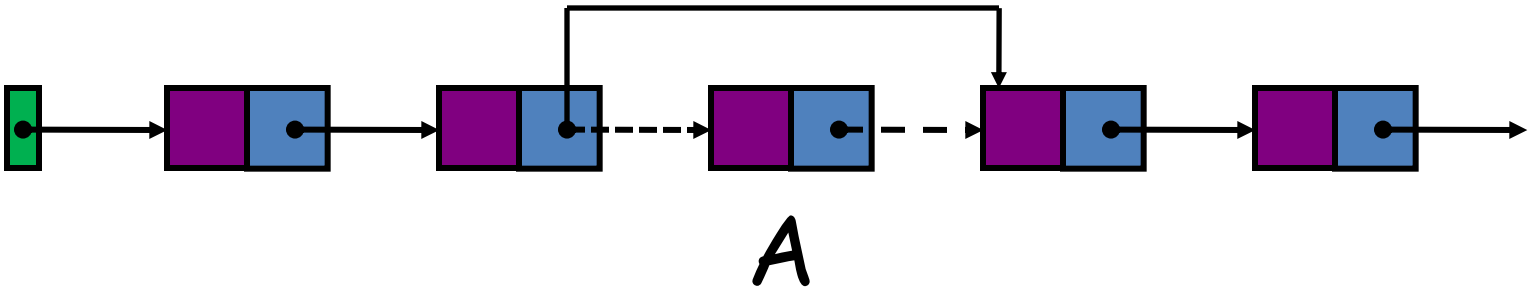
PTR = PTR -> Link;

LOC = SAVE

Deletion Algorithm



Deletion Algorithm



Delete the Node Following a Given Node

- Write an Algorithm that deletes the **Node N** with location **LOC**. **LOCP** is the location of the node which precedes **N** or when **N** is the first node **LOCP = NULL**

Delete the Node Following a Given Node

- Algorithm: Delete(Head, LOC, LOCP)

1 If LOCP = NULL then
    Set Head = Head ->Link. [**Deletes the 1st Node]**
  Else
    Set LOCP->Link = LOC->Link [**Deletes Node N**]

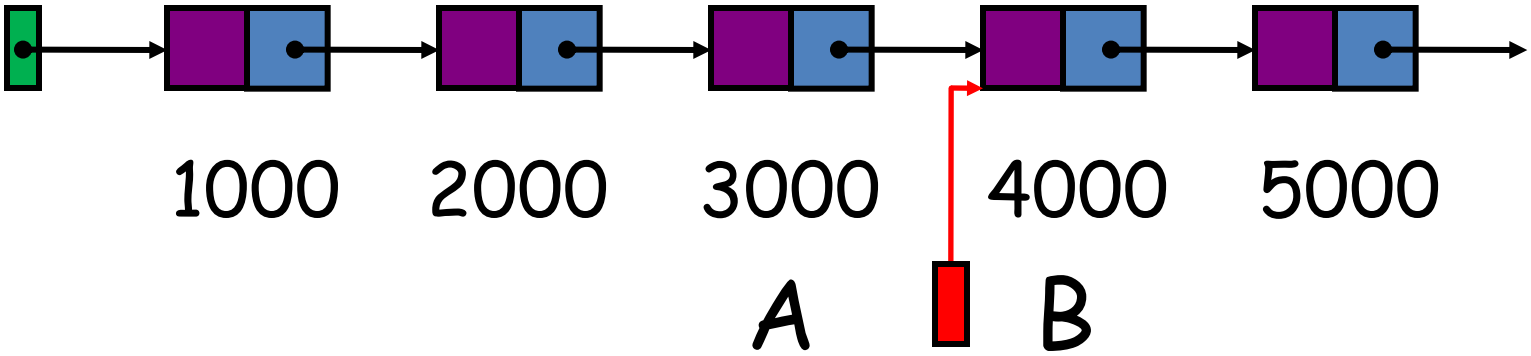
  2. Exit

Delete an Item

Let **Head** be a pointer to a linked list in memory that contains integer data. Write an algorithm to delete node which contains **ITEM**.

Delete an Item

To delete a Node [**Node B**] We have to Remember the Pointer to its predecessor [Node A]



4000

Deletion of an ITEM

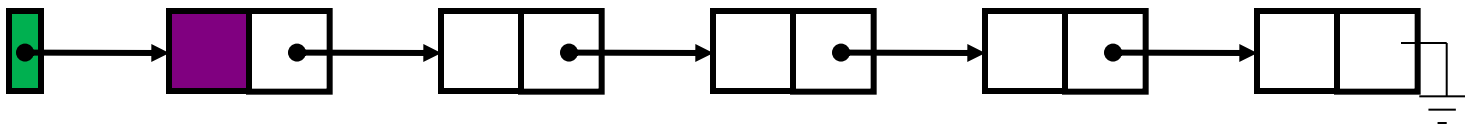
Algorithm

1. Set PTR=Head and SAVE = Head
2. If Head->DATA == ITEM
   Head = Head -> Link
   PTR -> Link = NULL;
3. Else
   PTR = PTR -> Link
4. Repeat step 5 while PTR ≠ NULL
5. If PTR->DATA == ITEM, then
   Set SAVE->LINK = PTR -> LINK, exit
   else
   SAVE = PTR
   PTR = PTR -> LINK
6. Stop

Header Linked Lists

- A **header linked** list is a linked list which always contains a **special node** called **header node**

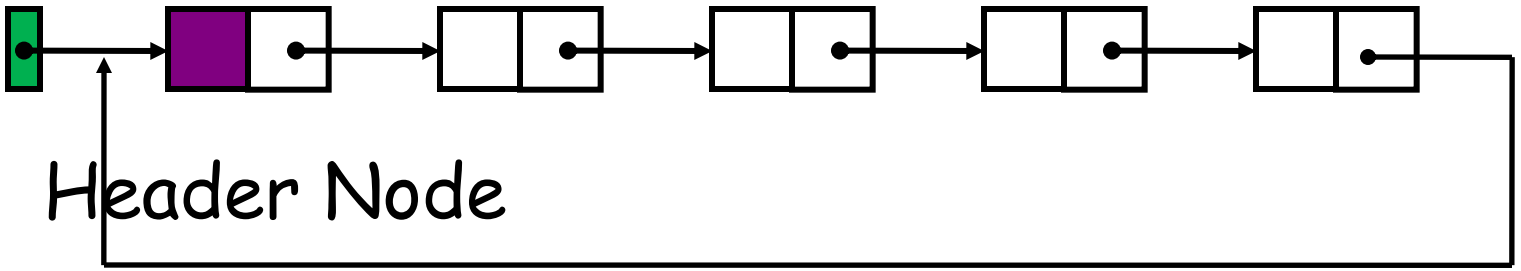
- **Grounded Header List:** A header list where the last node contains the NULL pointer.



Header Node

Header Linked Lists

- **Circular Header List:** A header list where the **last node** points back to the **header node**.



Header Linked Lists

- Pointer **Head** always points to the **header node**.
- **Head->Link == NULL** indicates that a grounded header list is empty
- **Head->Link == Head** indicates that a circular header list is empty

Header Linked Lists

- The **first node** in a header list is the node following the **header node**
- Circular Header list are frequently used instead of ordinary linked list
  - Null pointer are not used, hence all pointer contain valid addresses
  - Every node has a predecessor, so the first node may not require a special case.

Traversing a Circular Header List

- Let **Head** be a circular header list in memory. Write an algorithm to **print Data** in each node in the list.

Traversing a Circular Header List

Algorithm

1. Set PTR = Head->Link;
2. Repeat Steps 3 and 4 while PTR ≠ Head
3. Print PTR->Data
4. Set PTR = PTR ->Link
5. Exit

Locating an ITEM

- Let **Head** be a circular header list in memory. Write an algorithm to find the location LOC of the first node in the list which contains ITEM or return LOC = NULL when the item is not present.

Locating an ITEM

Algorithm

1. Set PTR = Head->Link
2. Repeat while **PTR ≠ Head**

   If PTR->Data == ITEM then

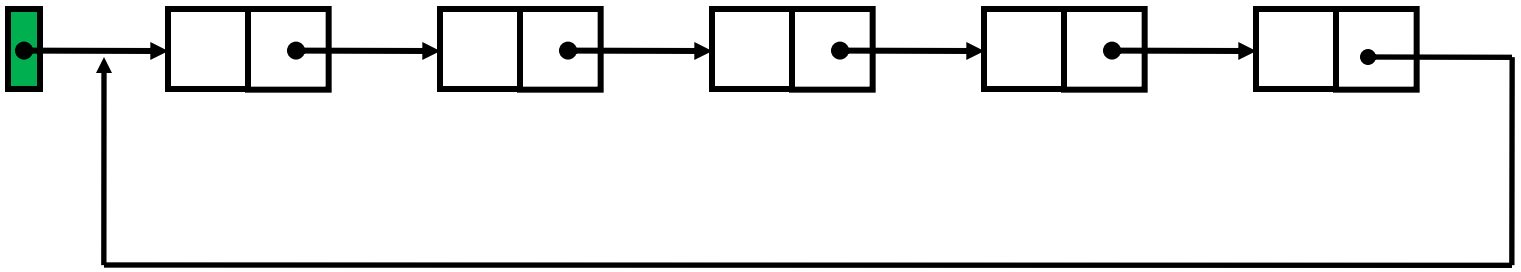
   Set LOC = PTR and exit

   Else

   Set PTR = PTR ->Link
3. Set LOC = NULL
4. Exit

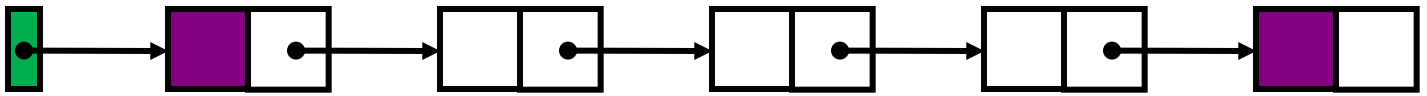
Other variation of Linked List

- A linked list whose last node points back to the first node instead of containing a NULL pointer called a **circular list**



Other variation of Linked List

- A linked list which contains both a special header node at the beginning of the list and a special trailer node at the end of list



Header Node

Trailer Node

Applications of Linked Lists

1. Polynomial Representation and operation on Polynomials

   Ex: $10X^6 + 20X^3 + 55$

2. Sparse Matrix Representation

0	0	11
22	0	0
0	0	66

Polynomials

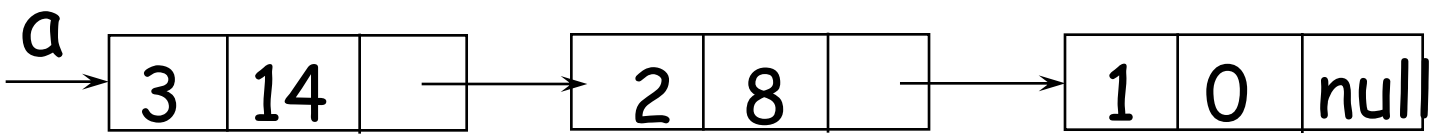
$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \ldots + a_0 x^{e_0}$$

coef	expon	link

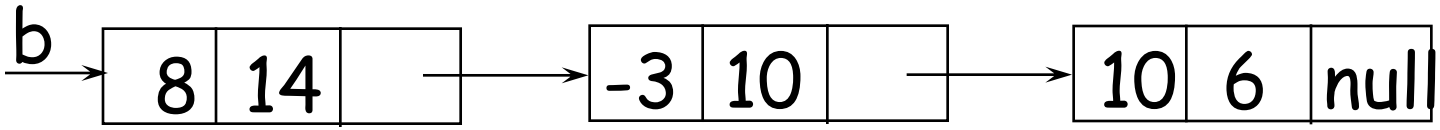
Representation of Node

Example

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



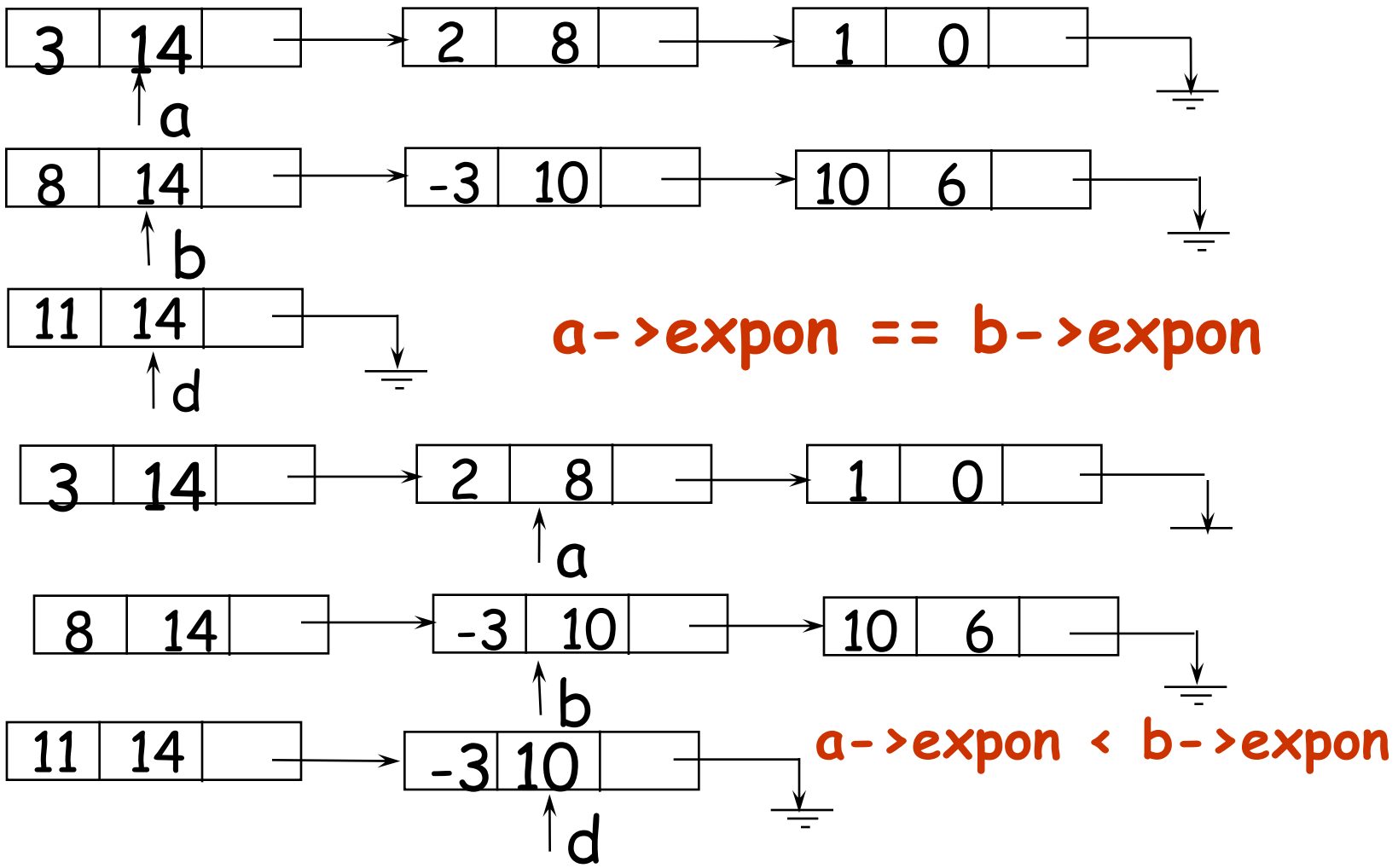
$$b = 8x^{14} - 3x^{10} + 10x^{6}$$



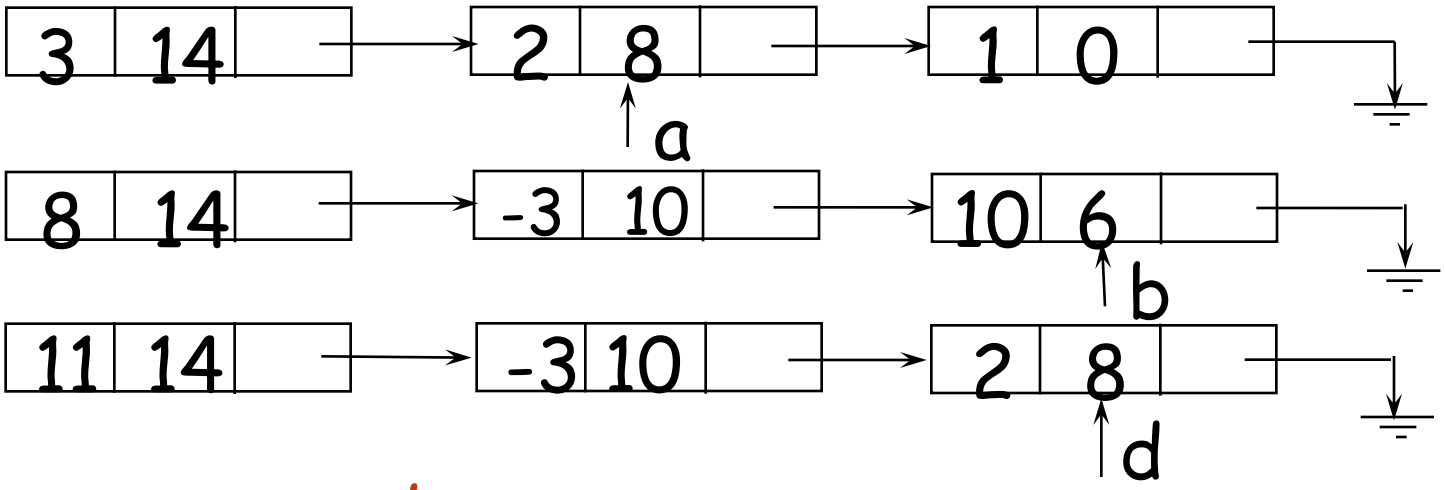
Polynomial Operation

1. **Addition**
2. **Subtraction**
3. **Multiplication**
4. **Scalar Division**

Polynomial Addition

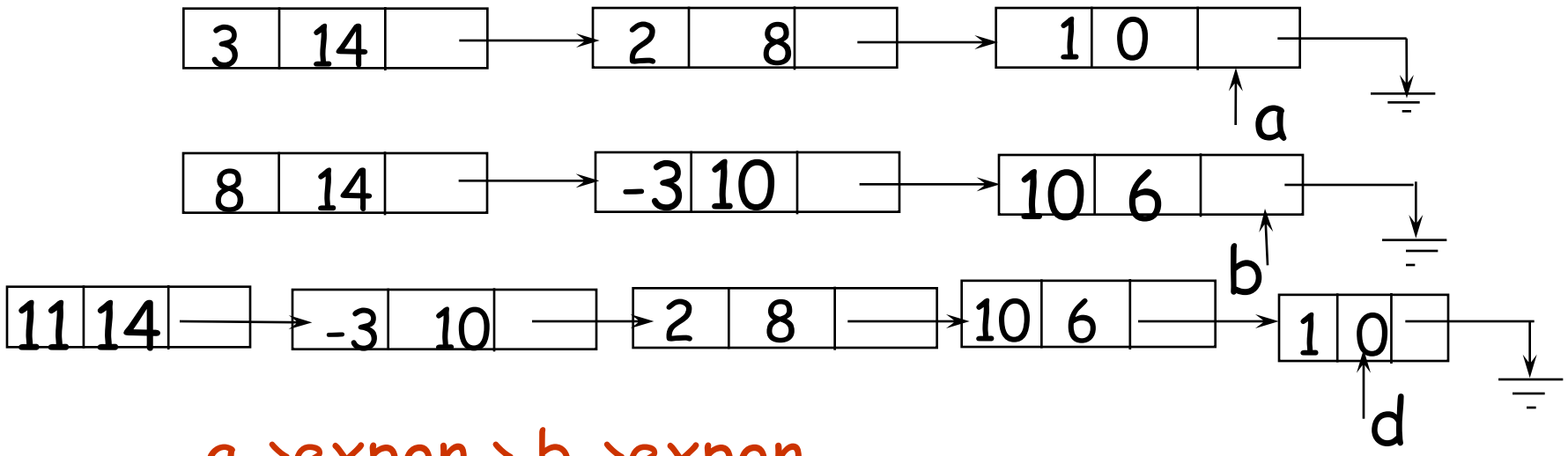


Polynomial Addition



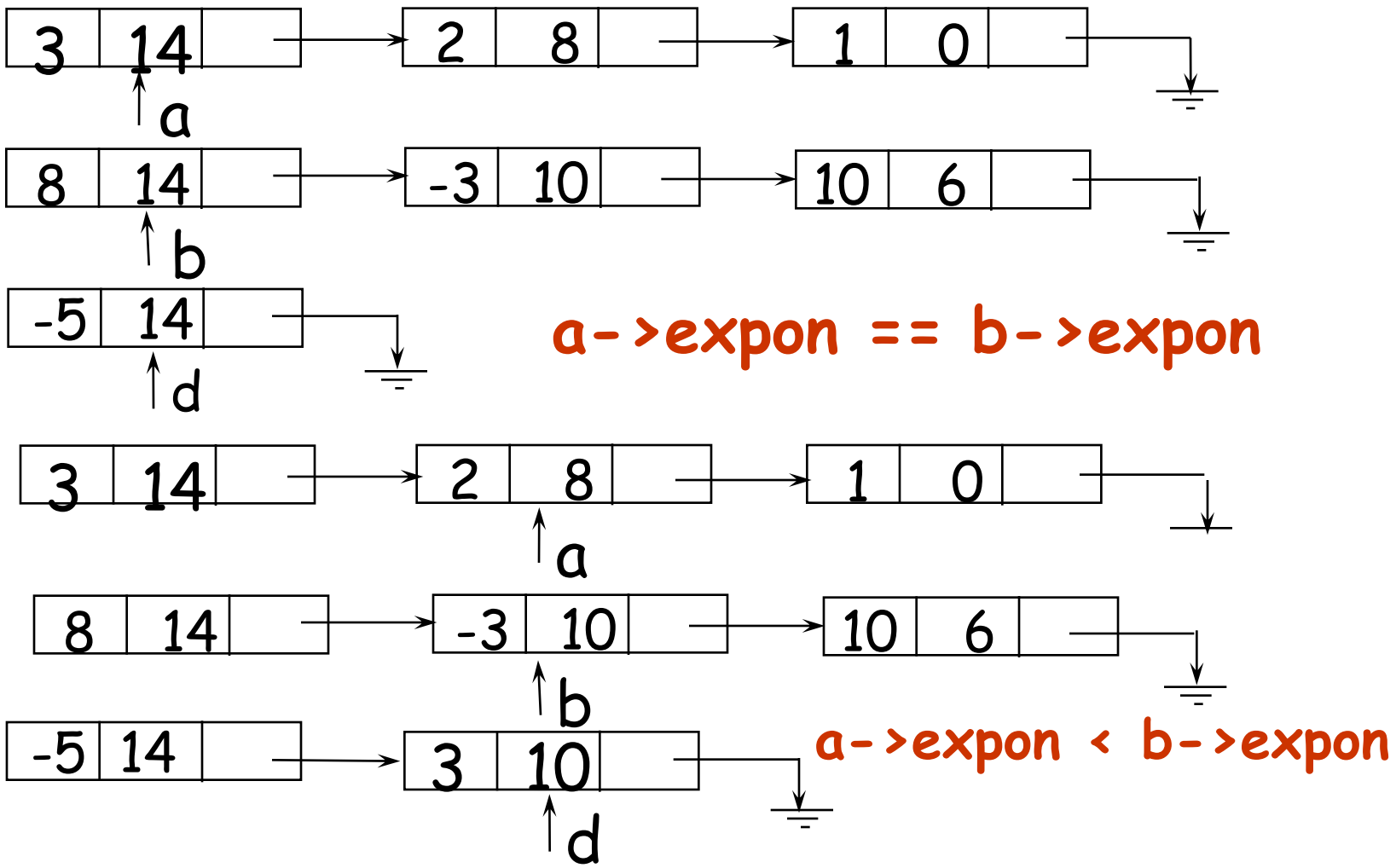
a->expon > b->expon

Polynomial Addition (cont'd)

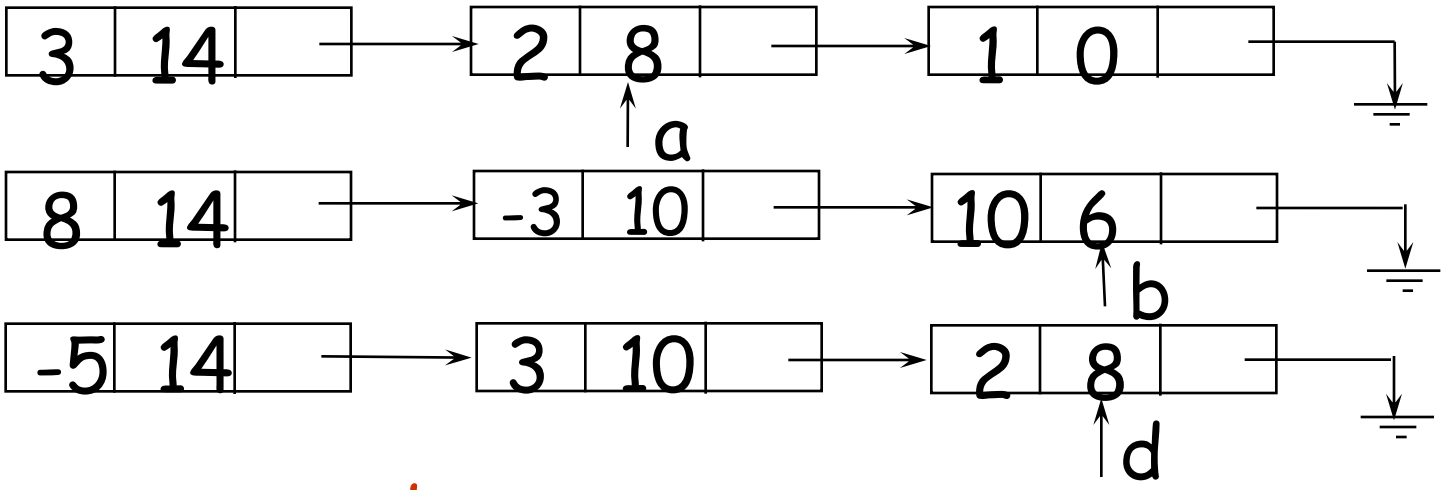


a->expon > b->expon

Polynomial Subtraction

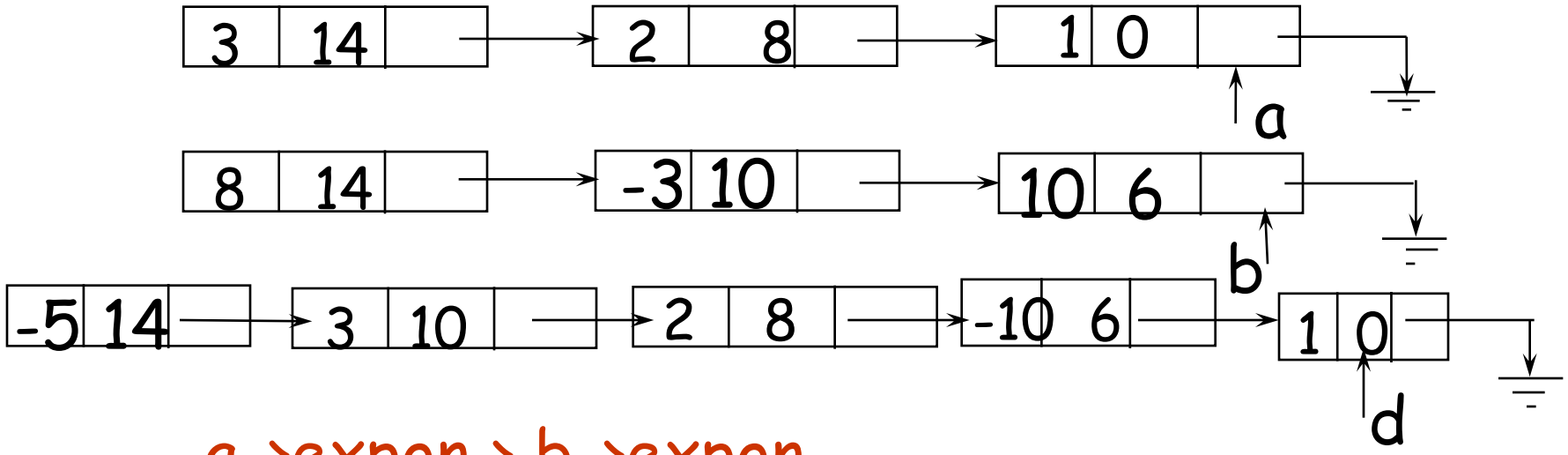


Polynomial Subtraction (cont'd)



a->expon > b->expon

Polynomial Subtraction (cont'd)



a->expon > b->expon

Two-Way List

- What we have discussed till now is a **one-way** list [Only one way we can traversed the list]
- Two-way List : Can be traversed in two direction
  - Forward : From beginning of the list to end
  - Backward: From end to beginning of the list

Two-Way List

- A **two-way list** is a linear collection of data element called nodes where each node **N** is divided into **three parts**:
  - A **information field** INFO which contains the data of N
  - A **pointer field** FORW which contains the location of the next node in the list
  - A **pointer field** BACK which contains the location of the preceding node in the list

Two-Way List

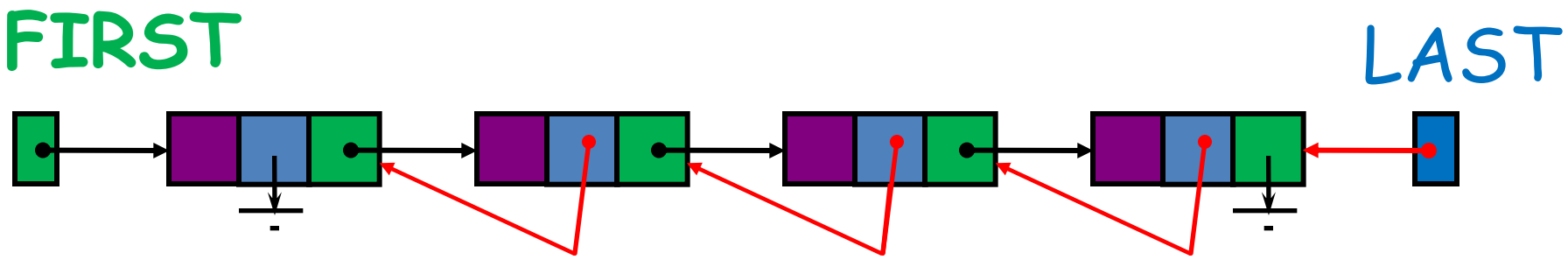
- List requires two pointer variables:
  - **FIRST**: which points to the first node in the list
  - **LAST**: which points to the last node in the list

INFO field

BACK pointer

FORW pointer

Two-Way List



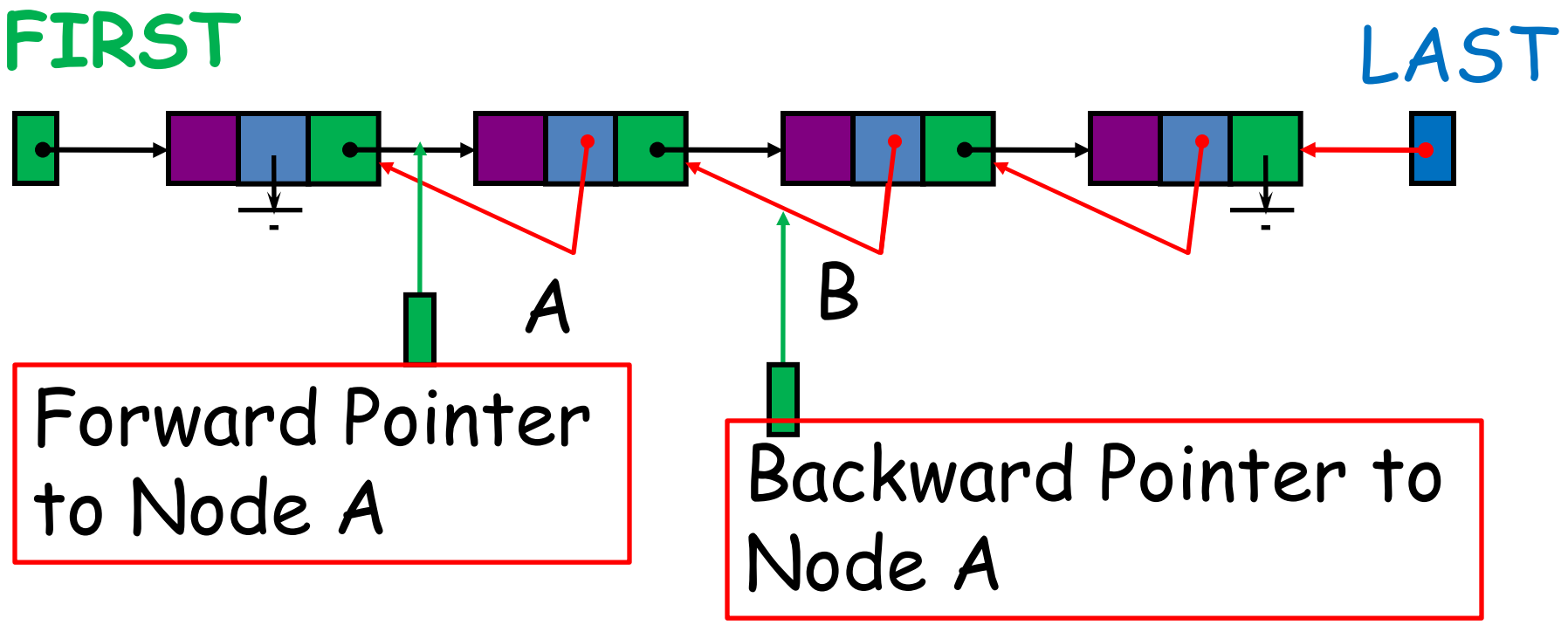
Two-Way List

Suppose **LOCA** and **LOCB** are the locations of **nodes A and B** respectively in a two-way list.

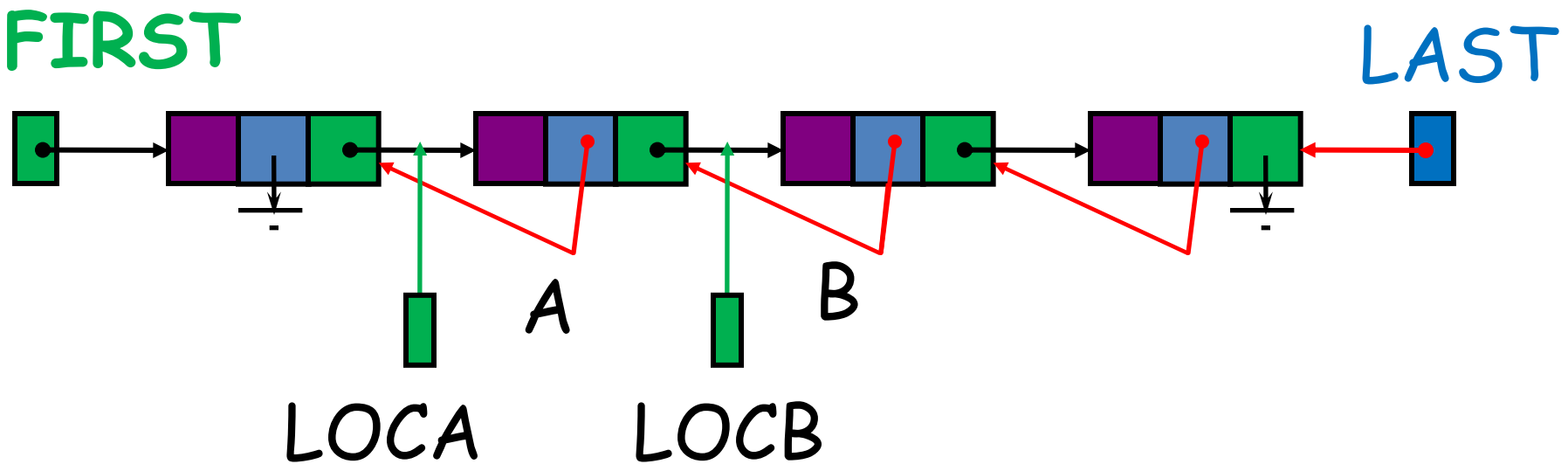
The statement that node B follows node A is equivalent to the statement that node A precedes node B

Pointer Property: **LOCA->FORW** = **LOCB** if and only if **LOCB->BACK** = **LOCA**

Two-Way List


FIRST
LAST
A
B
Forward Pointer to Node A
Backward Pointer to Node A

Two-Way List



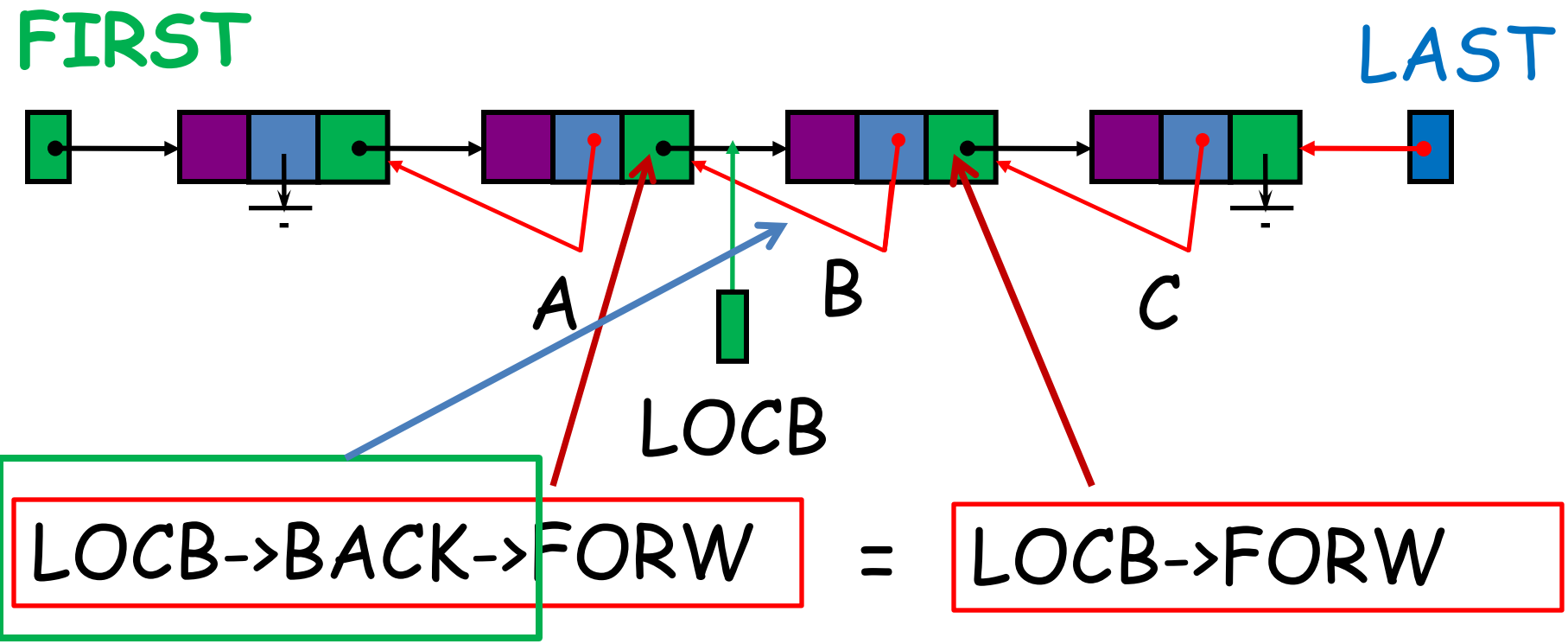
LOCA->FORW = LOCB if and only if
LOCB->BACK = LOCA

Operation in two-way list

- Traversing
- Searching
- Deleting
- Inserting

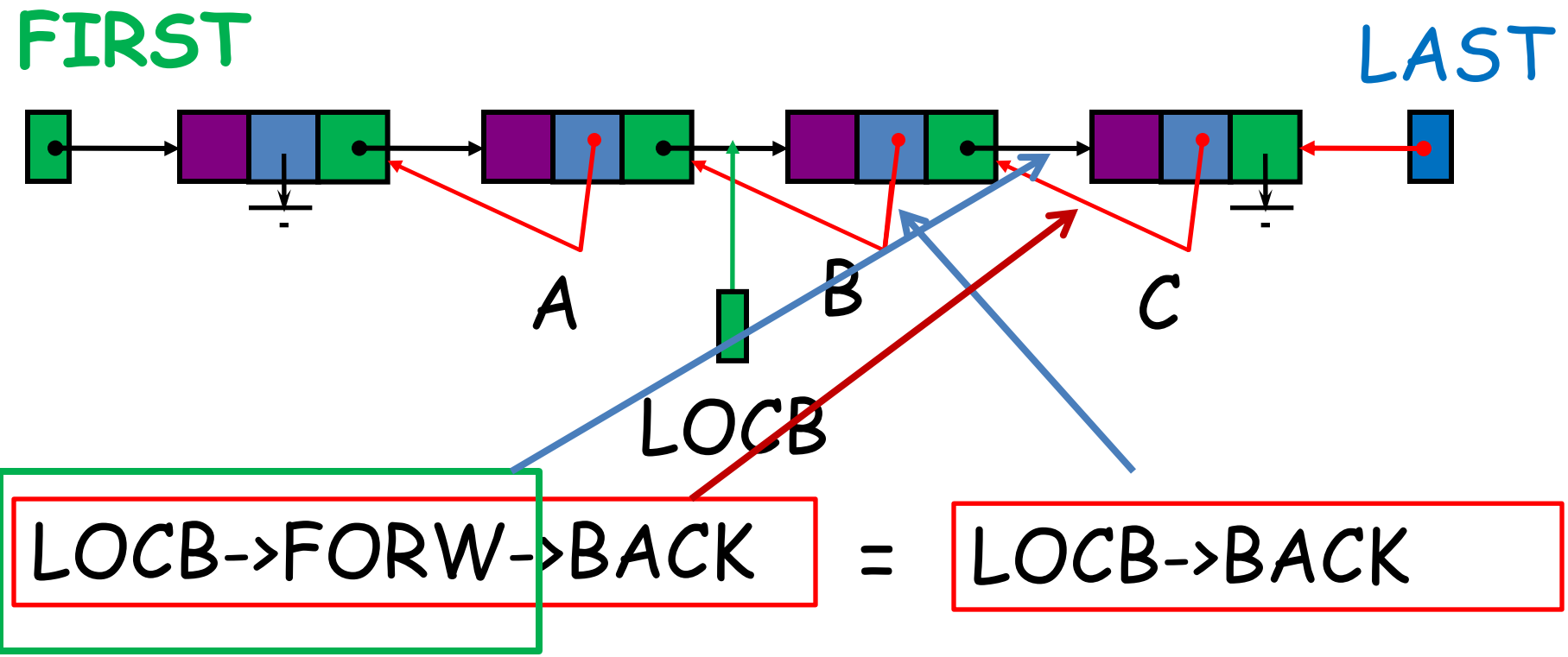
Deletion in Two-Way List

DELETE NODE B



Deletion in Two-Way List

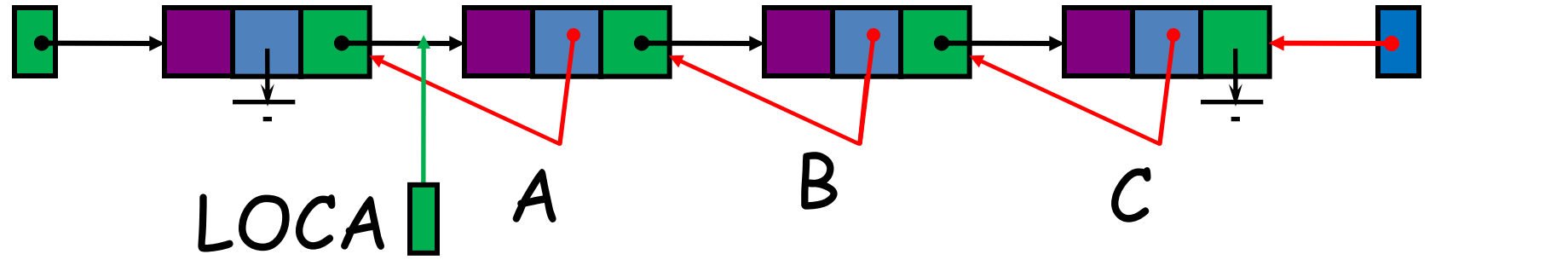
DELETE NODE B

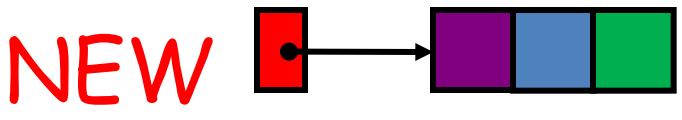


Insertion in Two-Way List

INSERT NODE NEW

LOCA->FORW->BACK = NEW
NEW->FORW = LOCA->FORW





LOCA->FORW = NEW
NEW->BACK = LOCA