KTU NOTES

LINKED LIST & MEMORY MANAGEMENT

MODULE III

Jacob P Cherian
Asst.Professor
Dept.of CSE, Saintgits College of Engineering

CONTENTS

Self Referential Structures

Dynamic Memory Allocation

Singly Linked List

Operations on Linked List

Doubly Linked List

Circular Linked List

Stacks and Queues using Linked List

Polynomial representation using Linked List

Memory allocation and deallocation

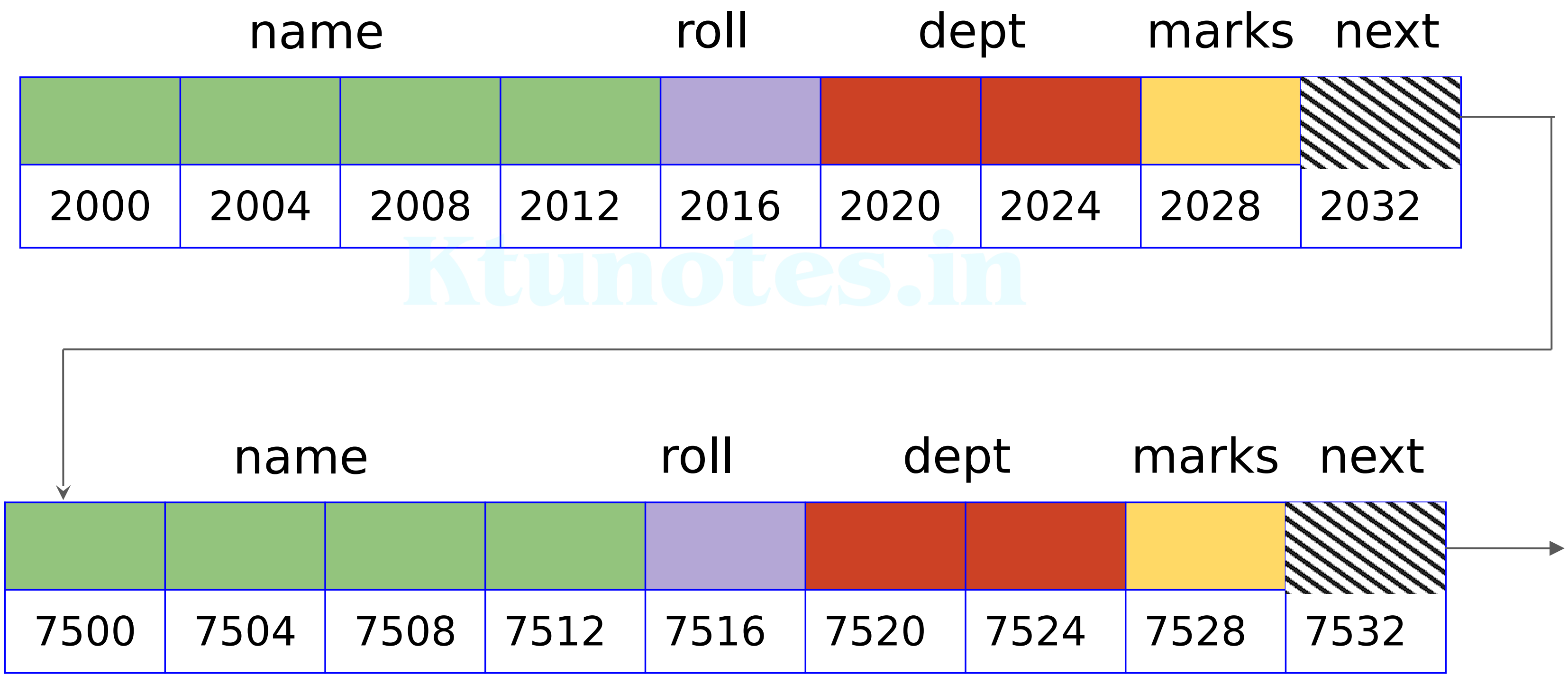
First-fit, Best-fit and Worst-fit allocation schemes

Self Referential Structures

Self-referential structures are those which have structure pointer(s) of the same type as their member(s).

```
struct student {
    char name[16];
    int roll;
       char dept[8];
    int marks;
    struct student *next;
};
```

Self Referential Structures-Memory Visualization



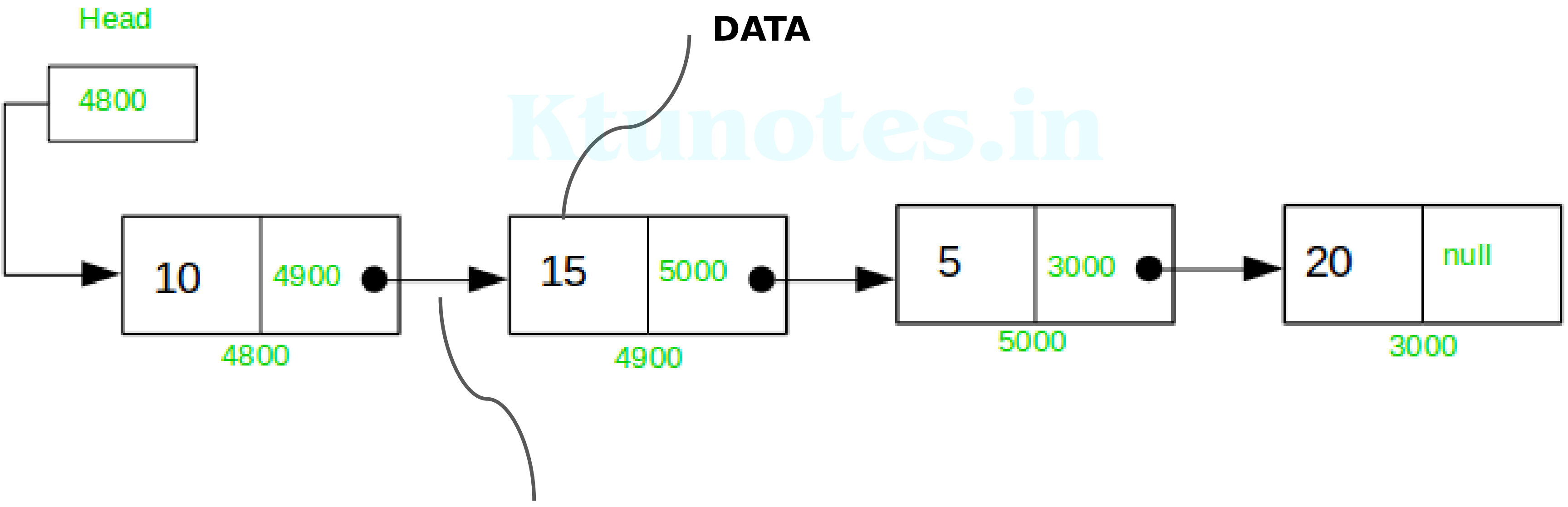
name roll dept marks next

2000 2004 2008 2012 2016 2020 2024 2028 2032

name roll dept marks next

7500 7504 7508 7512 7516 7520 7524 7528 7532

Linked List- Quick Overview



DATA

POINTER TO NEXT NODE

Types of Linked Lists

Singly Linked Lists

Doubly Linked Lists

Circular Linked Lists

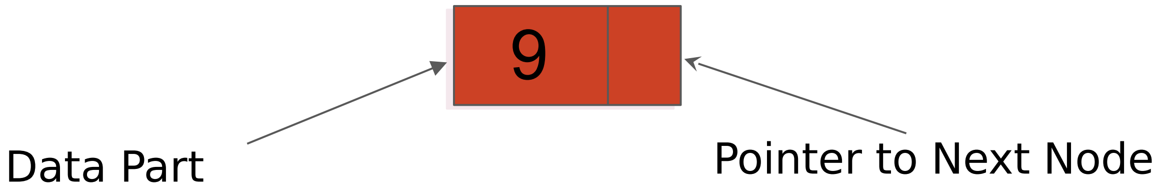
SINGLY LINKED LIST

Singly Linked List

Each element in a linked list is called a **node**.

A single node contains *data* and a pointer to the *next* node which helps in maintaining the structure of the list.

A singly linked list allows traversal only in a single direction



9

Data Part

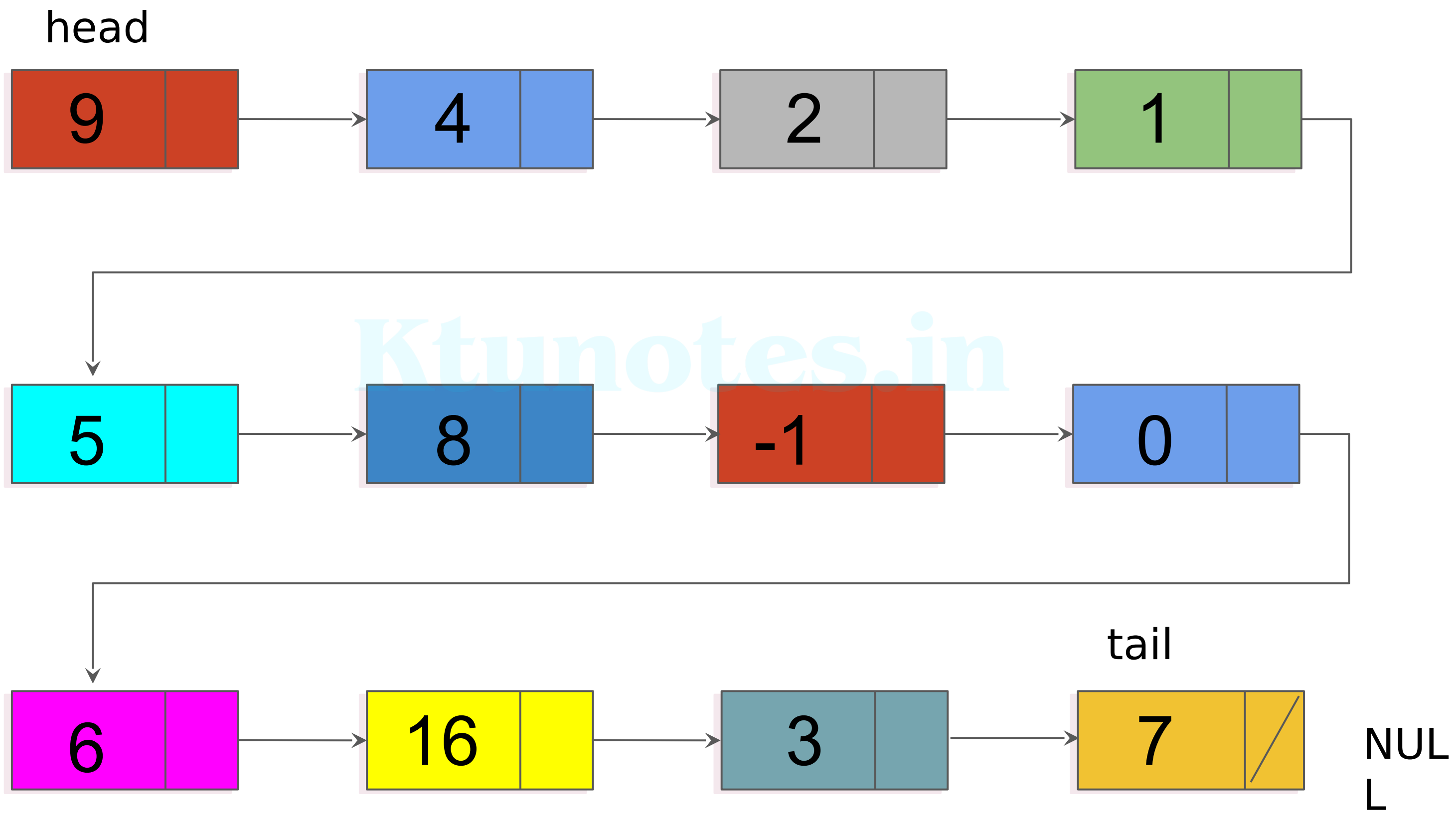
Pointer to Next Node

Singly Linked List

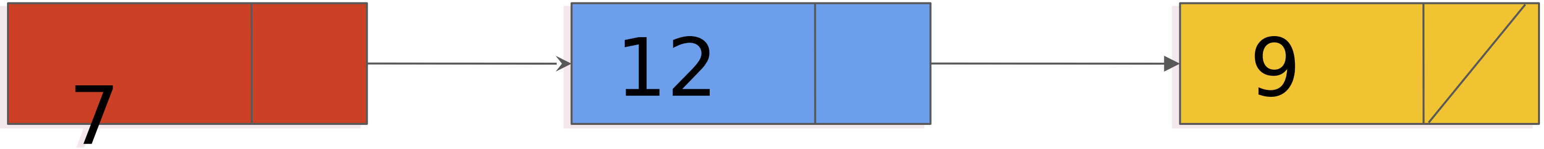
The first node is called the **head**; it points to the first node of the list and helps us access every other element in the list.

The last node, also sometimes called the **tail**, points to *NULL* which helps us in determining when the list ends.

Visual Representation



Representation of Linked List in Memory



7	2016			12	4020			
2000	2004	2008	2012	2016	2020	2024	2028	...

MAIN MEMORY

					9	NULL		
...	4004	4008	4012	4016	4020	4024	4028	4032

Common Linked List Operations

Search for a node in the List

Add a node to the List

Remove a node from the List

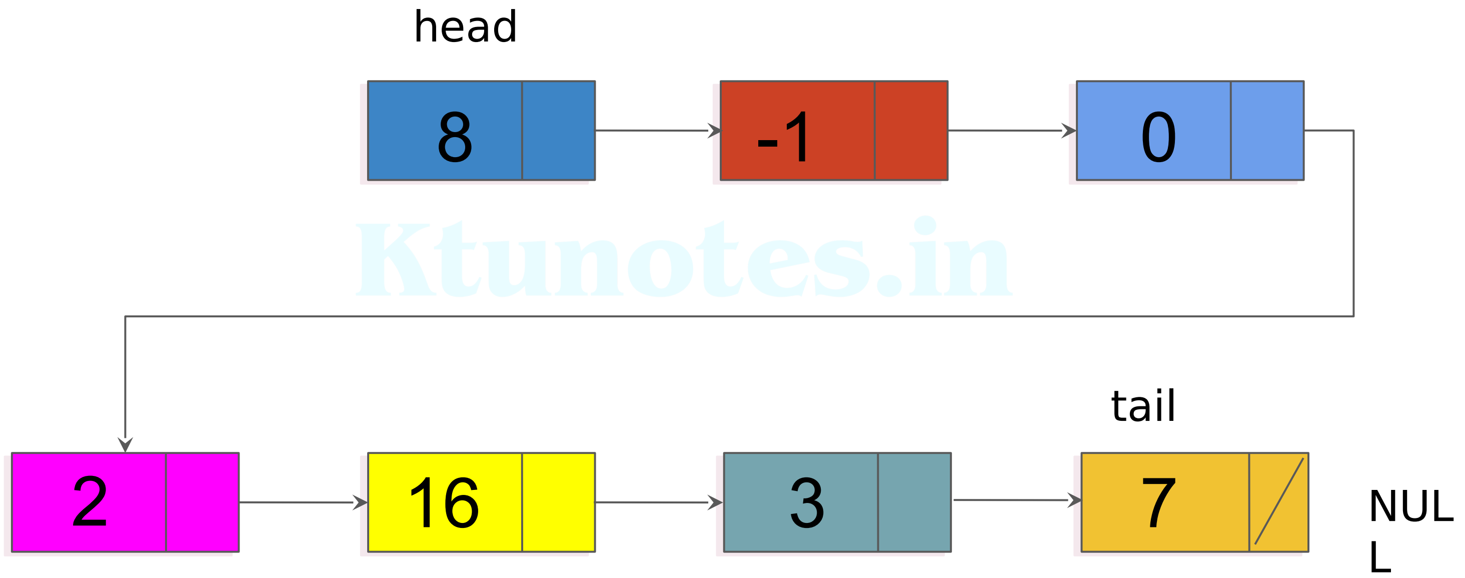
Basic Operations: Insertion at the Beginning

Create a new node with given data.

Point new node’s next to old head.

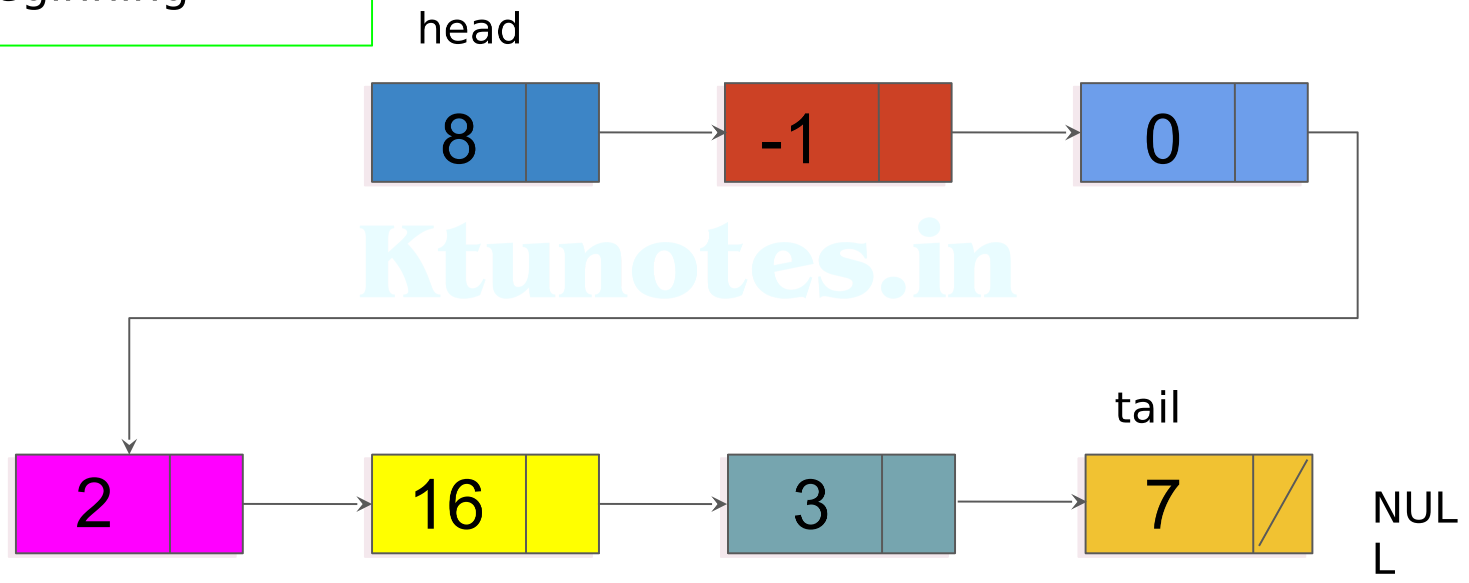
Point head to this new node.

Basic Operations: Insertion at the Beginning

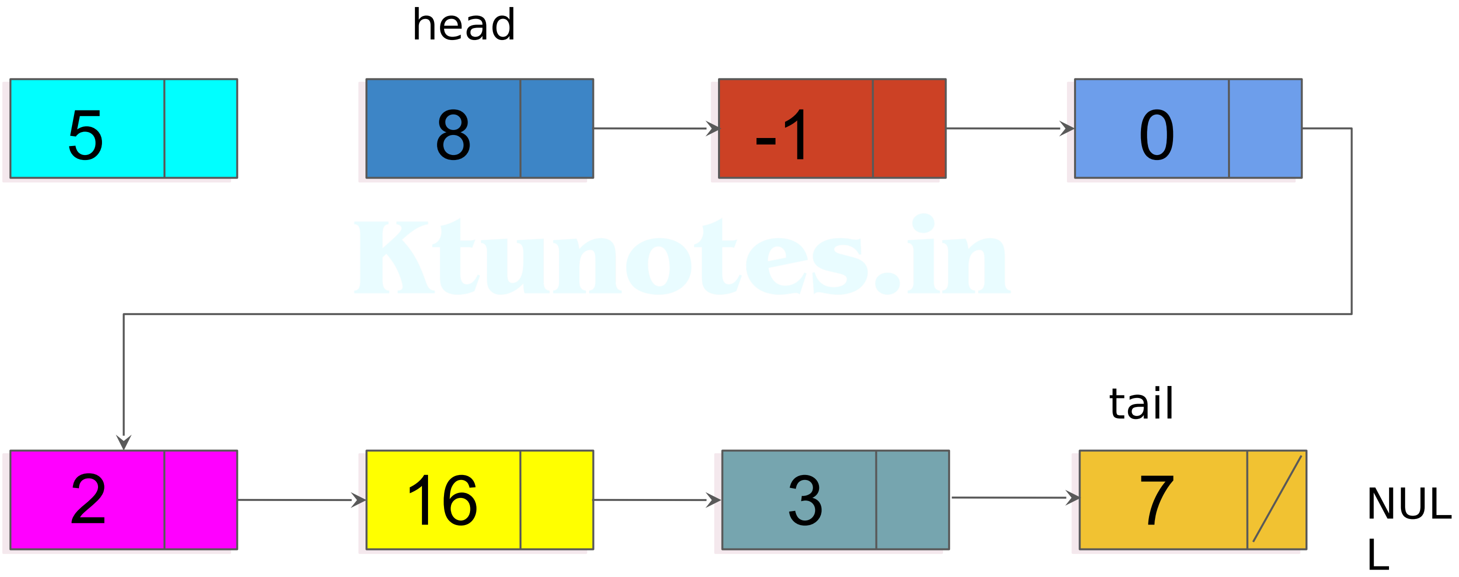


Basic Operations: Insertion at the Beginning

Insert a New Node 5
At the beginning

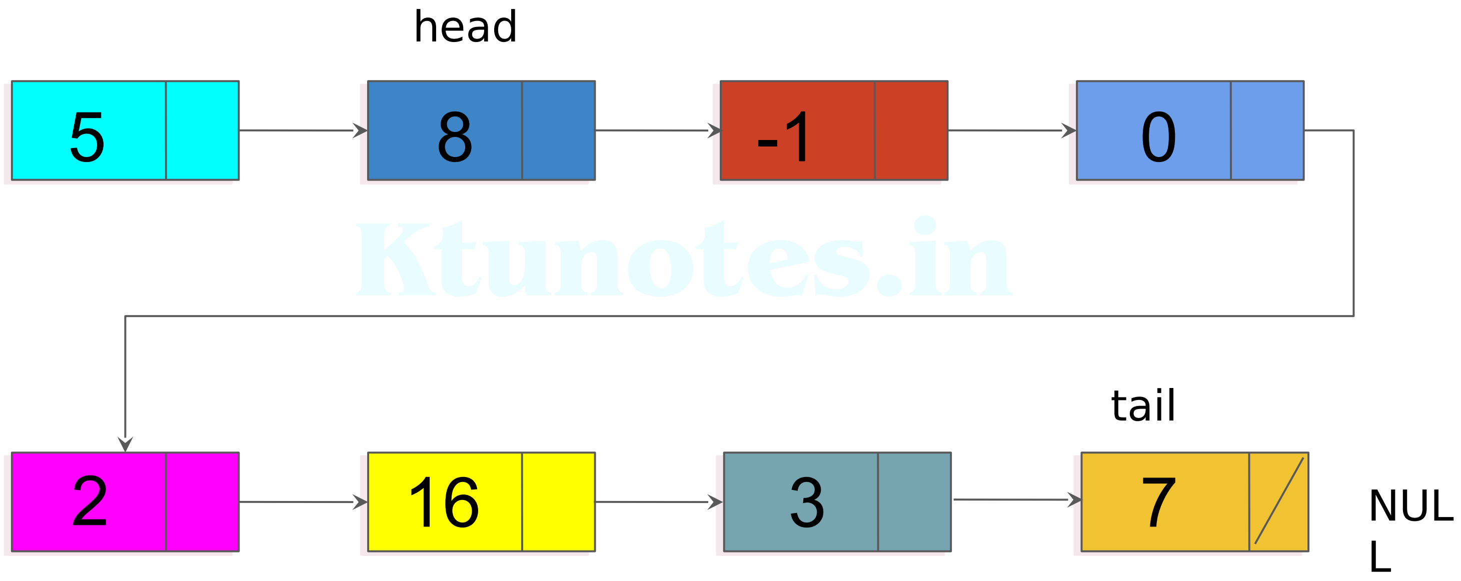


Basic Operations: Insertion at the Beginning



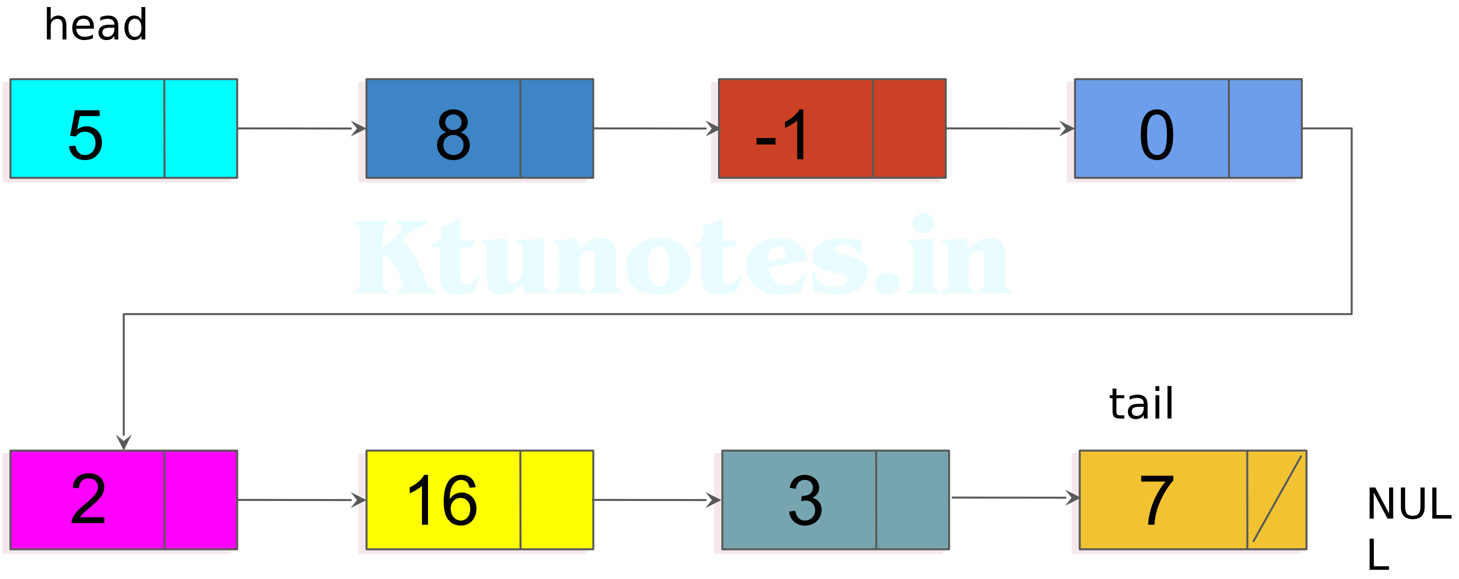
Create a new node with given data.

Basic Operations: Insertion at the Beginning



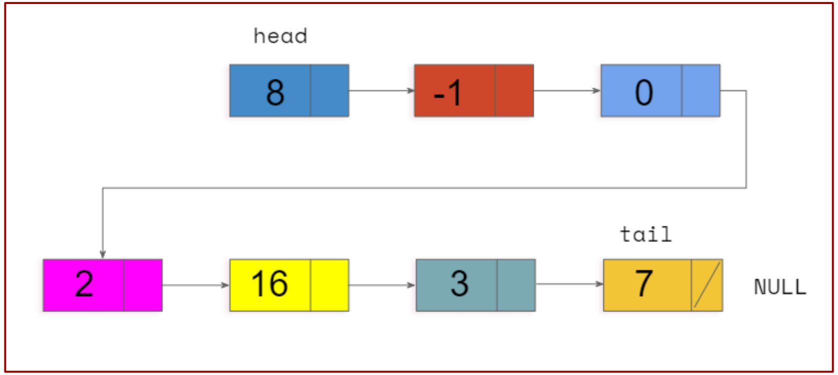
Point new node’s next to old head.

Basic Operations: Insertion at the Beginning



Point head to the new node.

Memory Visualization

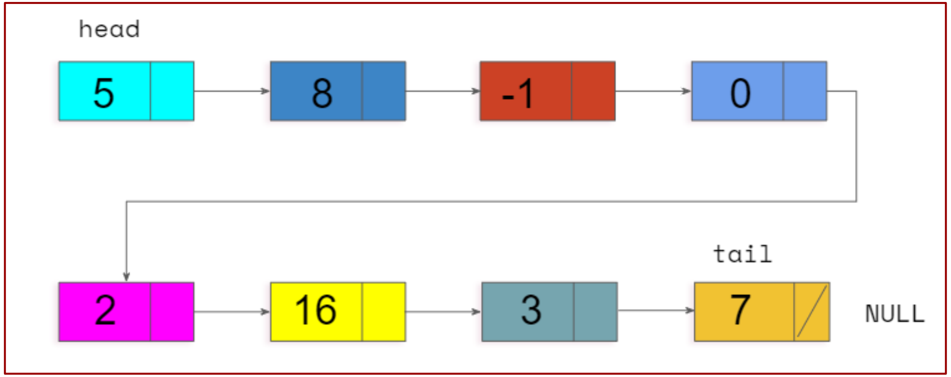


8	3016			-1	4004			
2000	2004	2008	...	3016	3020	3024	3028	...

0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

3	5012		7	NULL				
5000	5004	5008	5012	5016	...	6024	6028	...

Memory Visualization

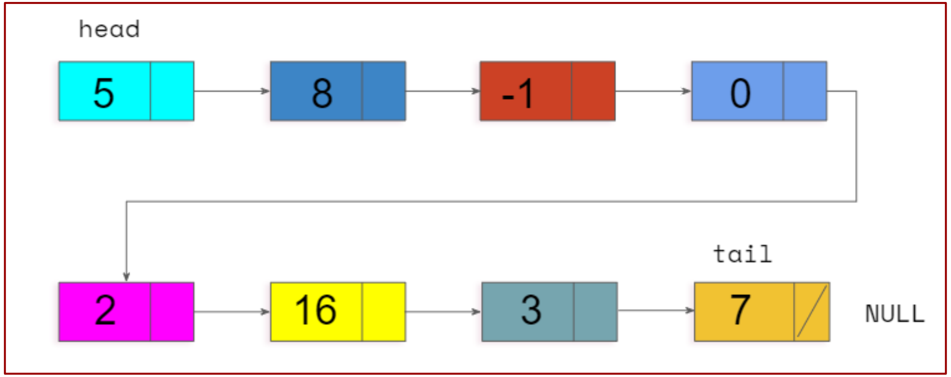


8	3016			-1	4004			
2000	2004	2008	...	3016	3020	3024	3028	...

0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

3	5012		7	NULL		5		
5000	5004	5008	5012	5016	...	6024	6028	...

Memory Visualization



8	3016			-1	4004			
2000	2004	2008	...	3016	3020	3024	3028	...

0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

3	5012		7	NULL		5	2000	
5000	5004	5008	5012	5016	...	6024	6028	...

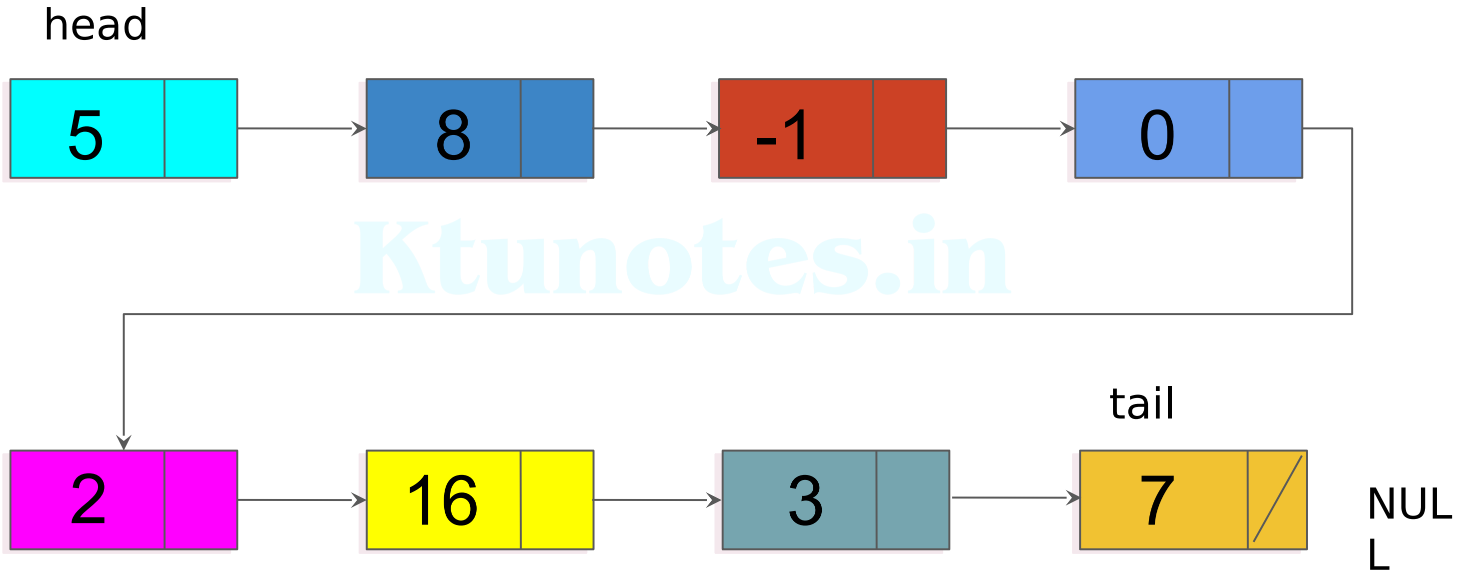
Basic Operations: Insertion at Middle/End or Insertion after Node X

Create a new node with given data.

Point new node’s next to old X’s next.

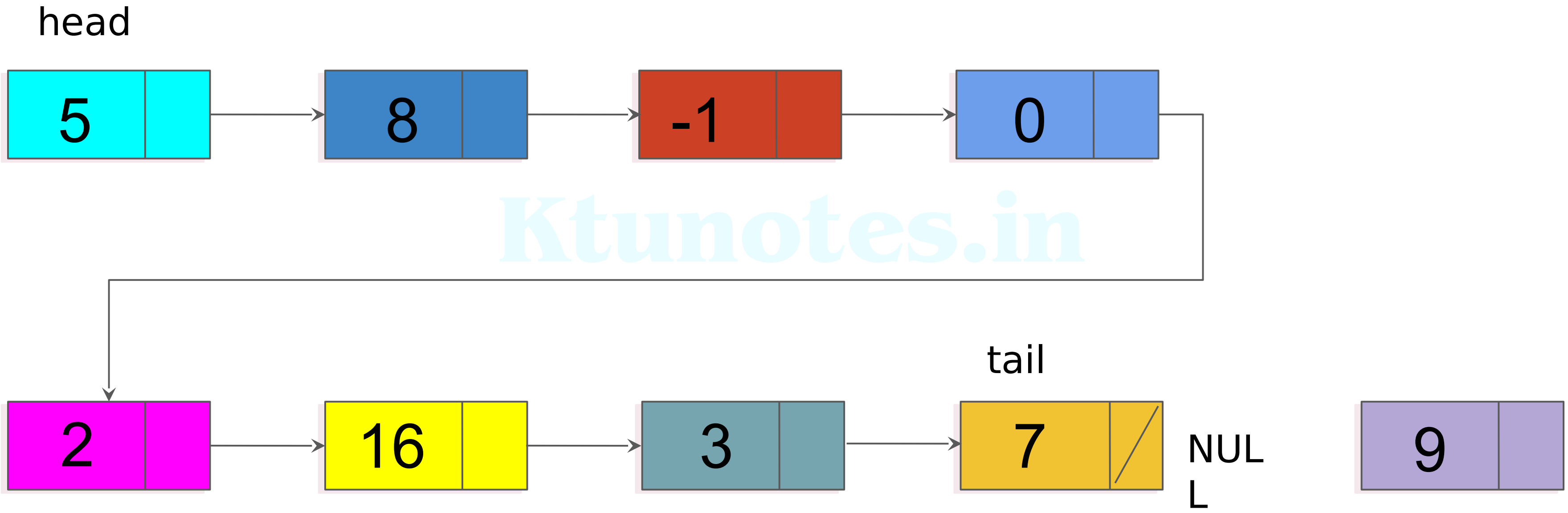
Point X’s next to the new node

Basic Operations: Insertion at the End



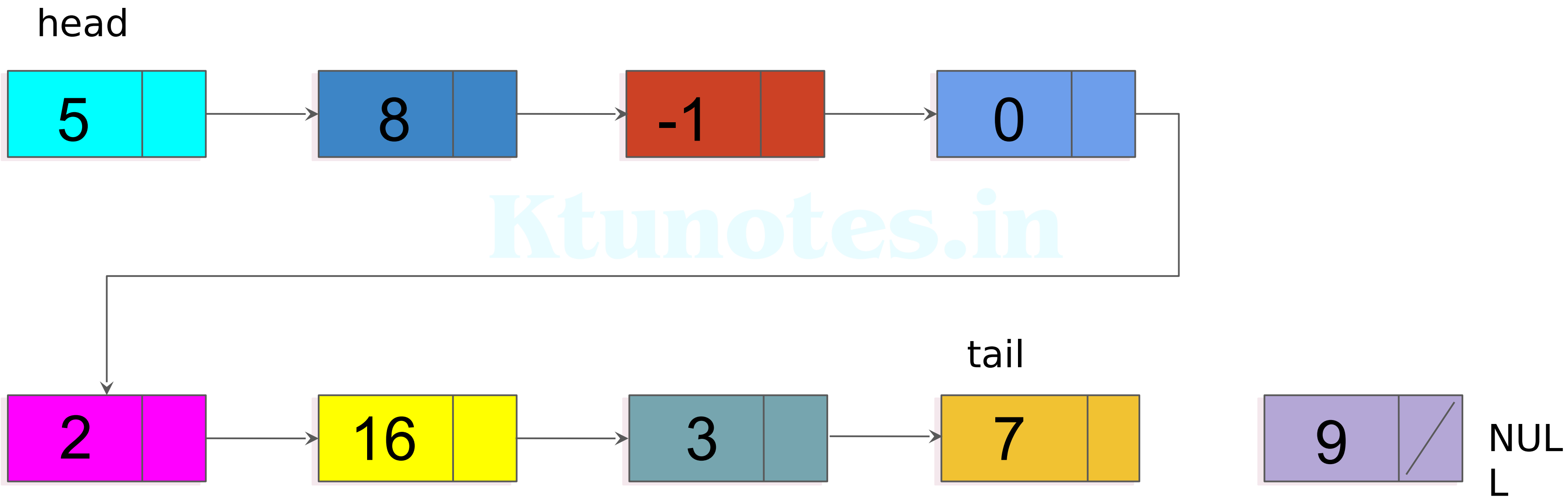
Insert a New Node 9
At the End

Basic Operations: Insertion at the End



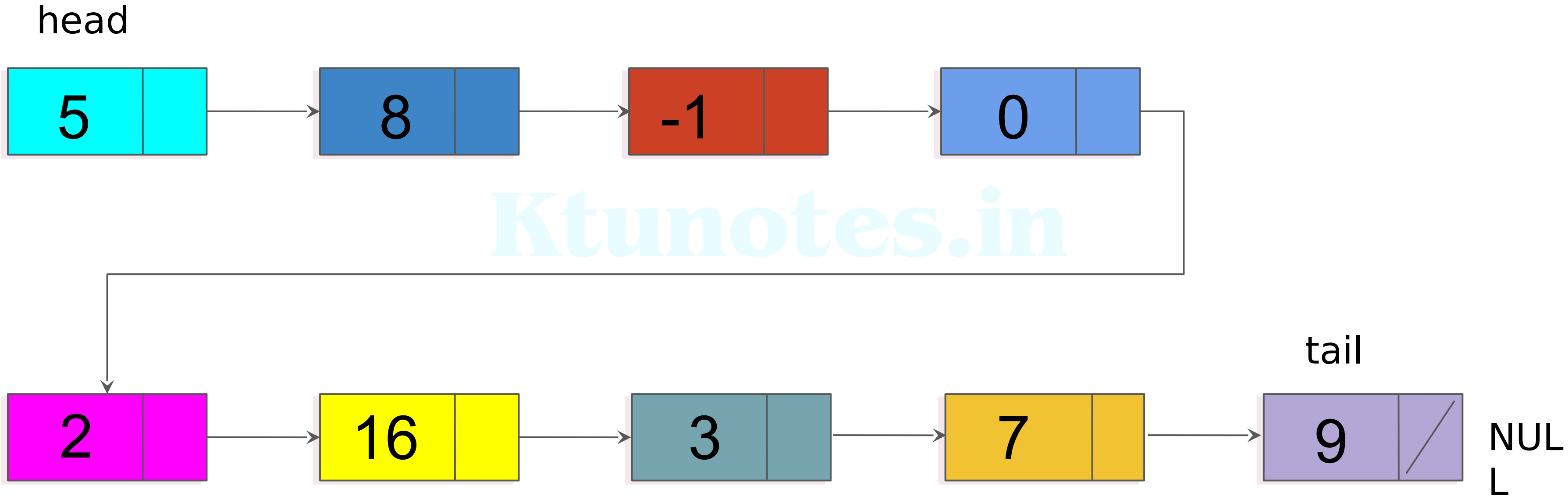
Create a new node with given data.

Basic Operations: Insertion at the End



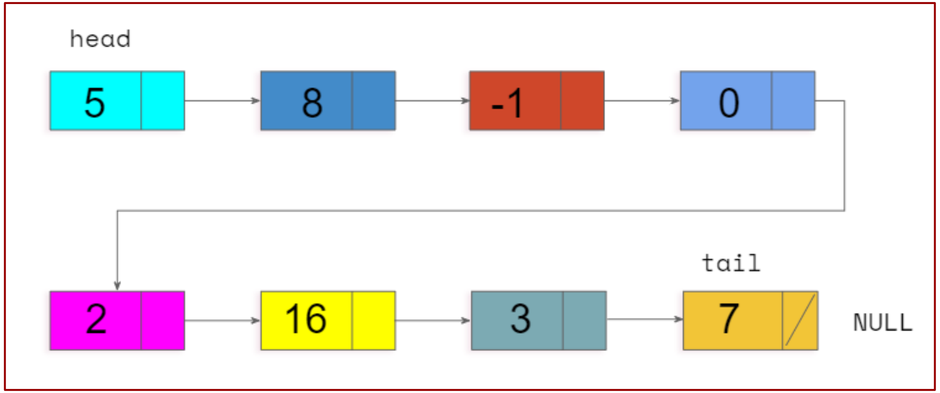
Point new node’s next to old X’s next.

Basic Operations: Insertion at the End



Point X’s next to the new node

Memory Visualization

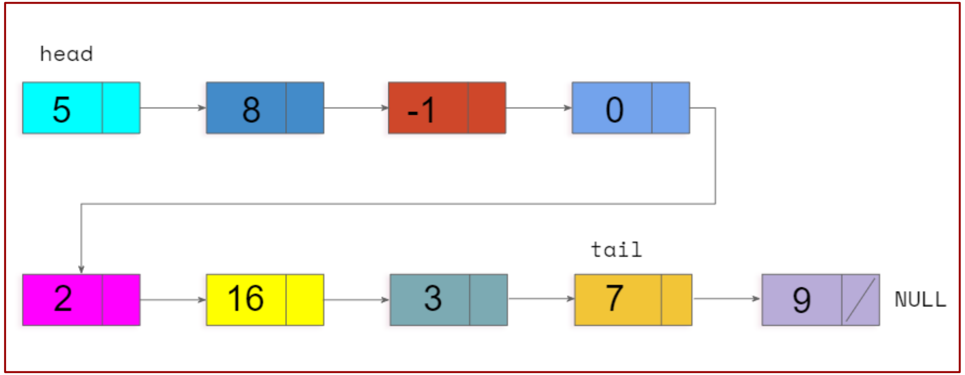


8	3016			-1	4004			
2000	2004	2008	...	3016	3020	3024	3028	...

0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

3	5012		7	NULL		5	2000	
5000	5004	5008	5012	5016	...	6024	6028	...

Memory Visualization

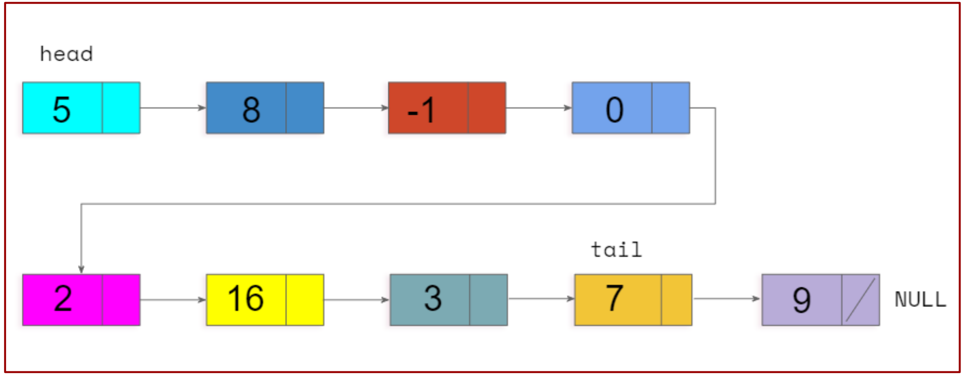


8	3016			-1	4004			
2000	2004	2008	...	3016	3020	3024	3028	...

0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

3	5012		7	8000		5	2000		9	
5000	5004	5008	5012	5016		6024	6028	...	8000	8004

Memory Visualization

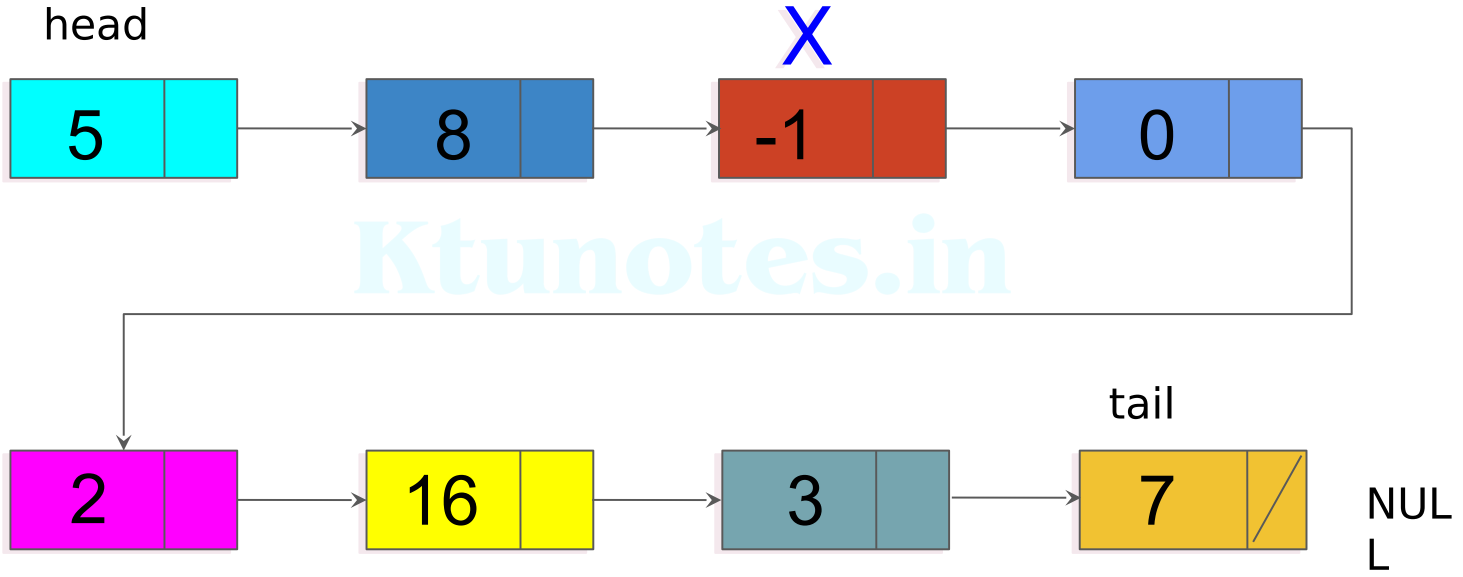


8	3016			-1	4004			
2000	2004	2008	...	3016	3020	3024	3028	...

0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

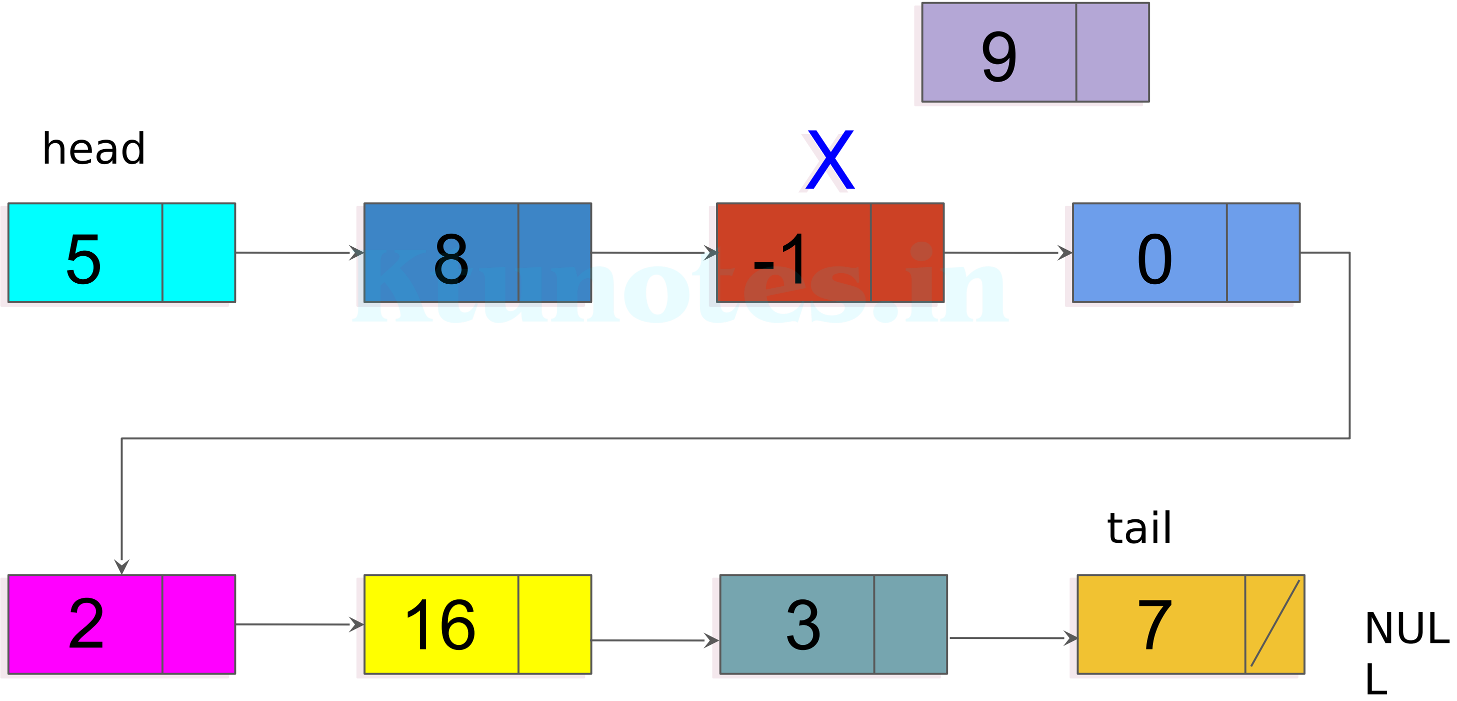
3	5012		7	8000		5	2000		9	NULL
5000	5004	5008	5012	5016		6024	6028	...	8000	8004

Basic Operations: Insertion after Position X



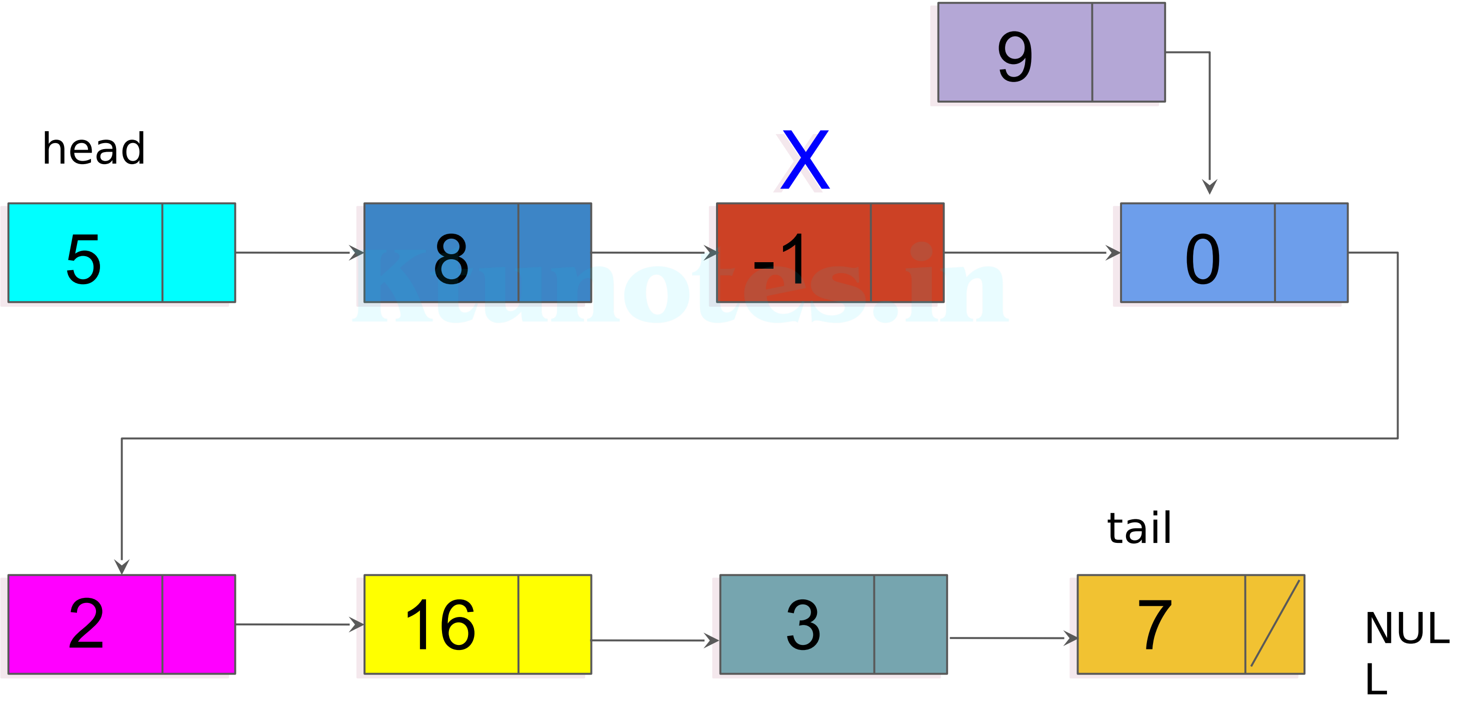
Insert a New Node 9
After position 3

Basic Operations: Insertion after Position X



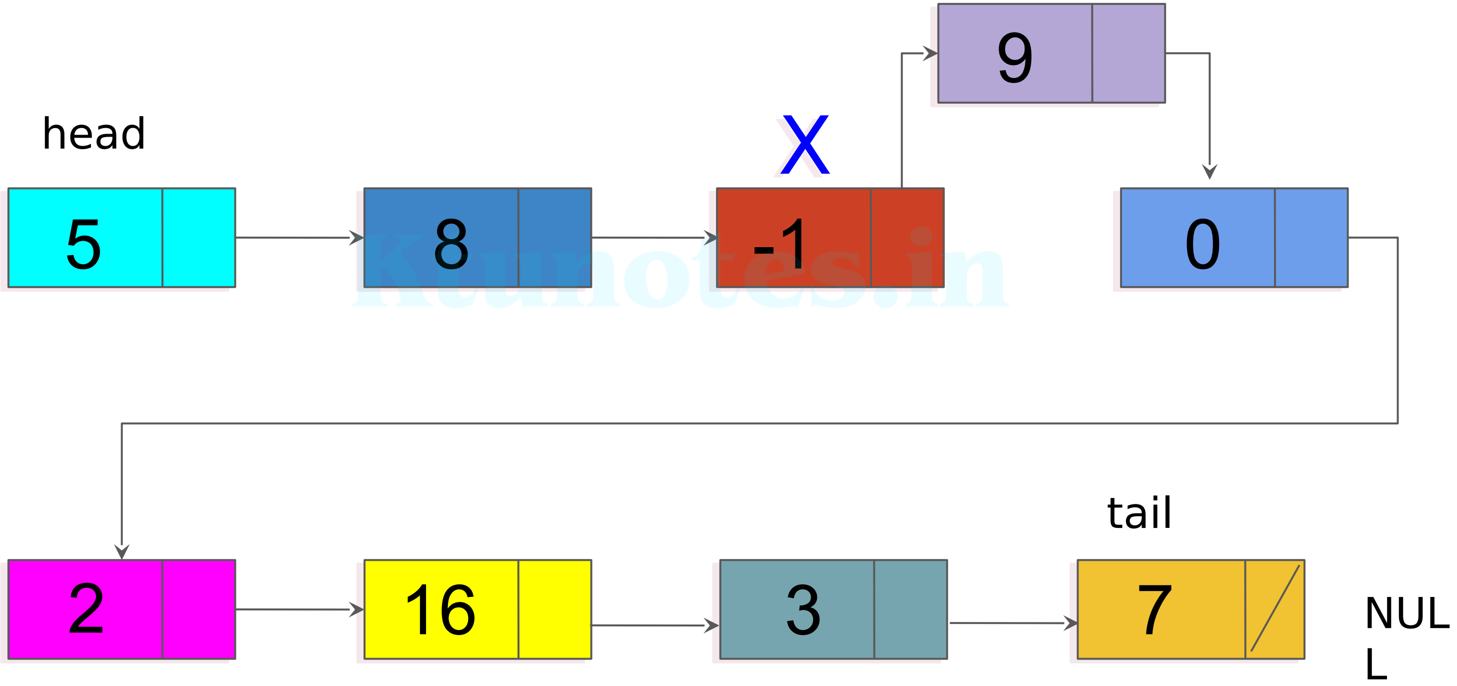
Create a new node with given data.

Basic Operations: Insertion after Position X



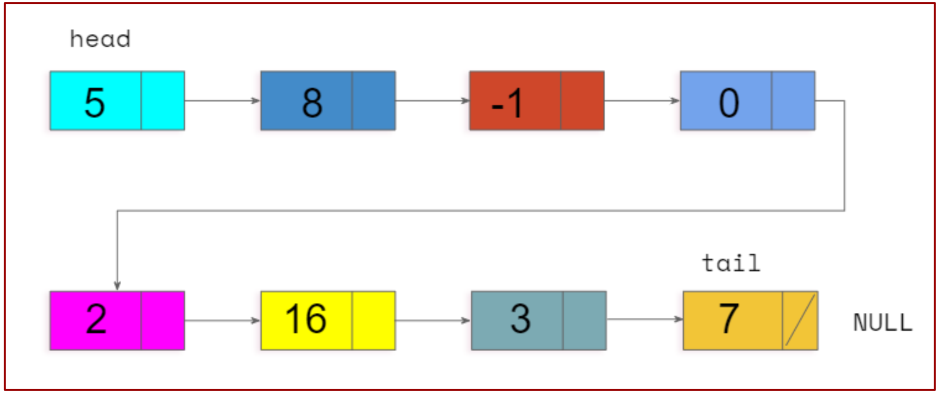
Point new node’s next to old X’s next.

Basic Operations: Insertion at a particular position



Point X’s next to the new node

Memory Visualization

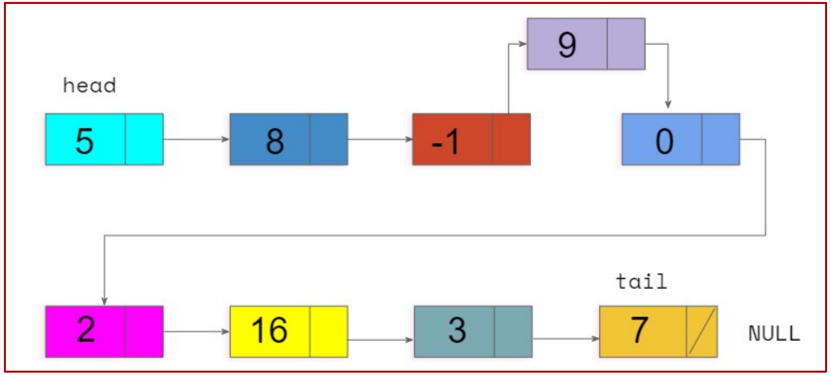


8	3016			-1	4004			
2000	2004	2008	...	3016	3020	3024	3028	...

0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

3	5012		7	NULL		5	2000	
5000	5004	5008	5012	5016	...	6024	6028	...

Memory Visualization



8	3016			-1	8000			
2000	2004	2008	...	3016	3020	3024	3028	...

0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

3	5012		7	800 0		5	2000		9	400 4
5000	5004	5008	5012	5016		6024	6028	...	8000	8004

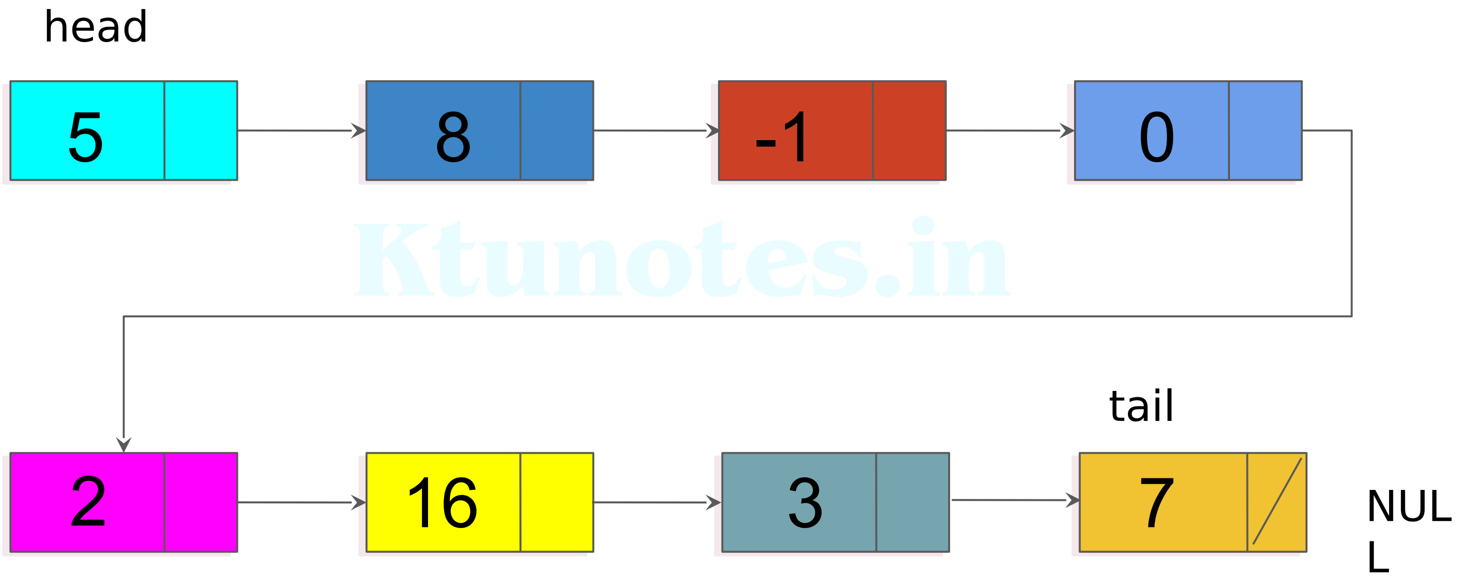
Basic Operations: Deletion at the Beginning

Get the node pointed by head as *temp*

Point head to *temp’s* next

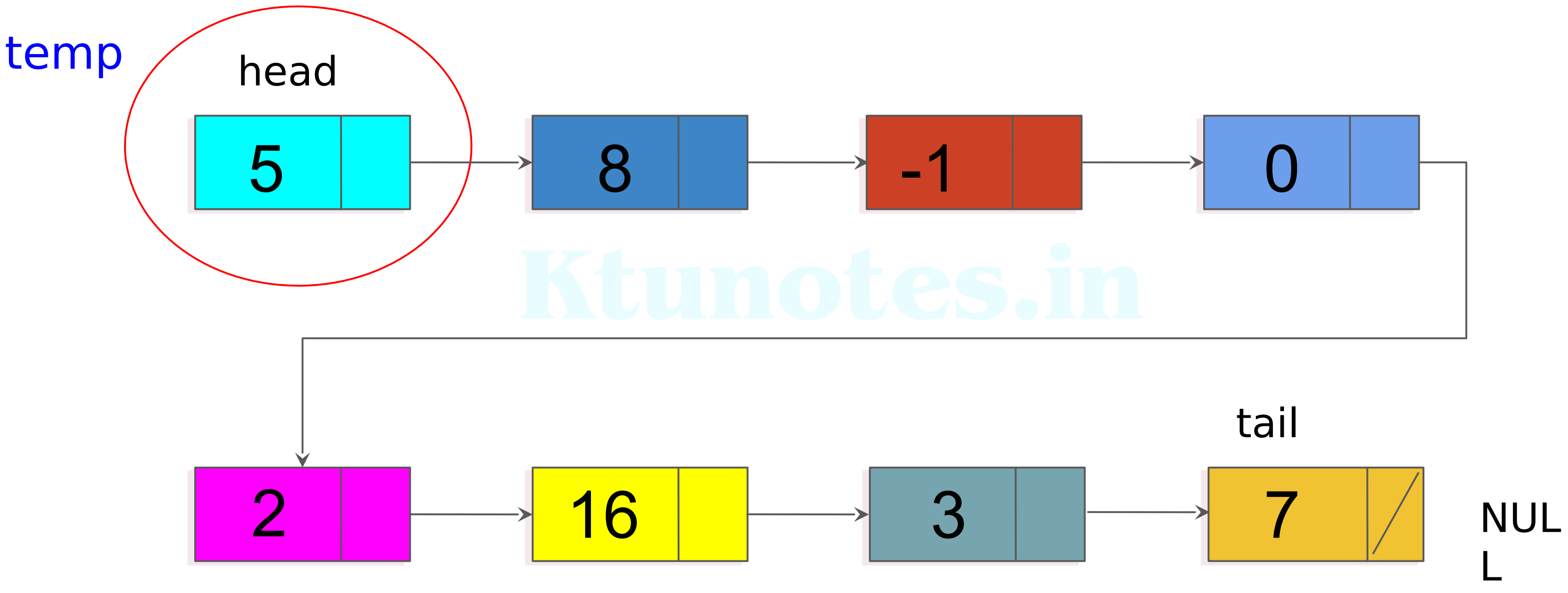
Free the memory used by node *temp*

Basic Operations: Deletion at the Beginning



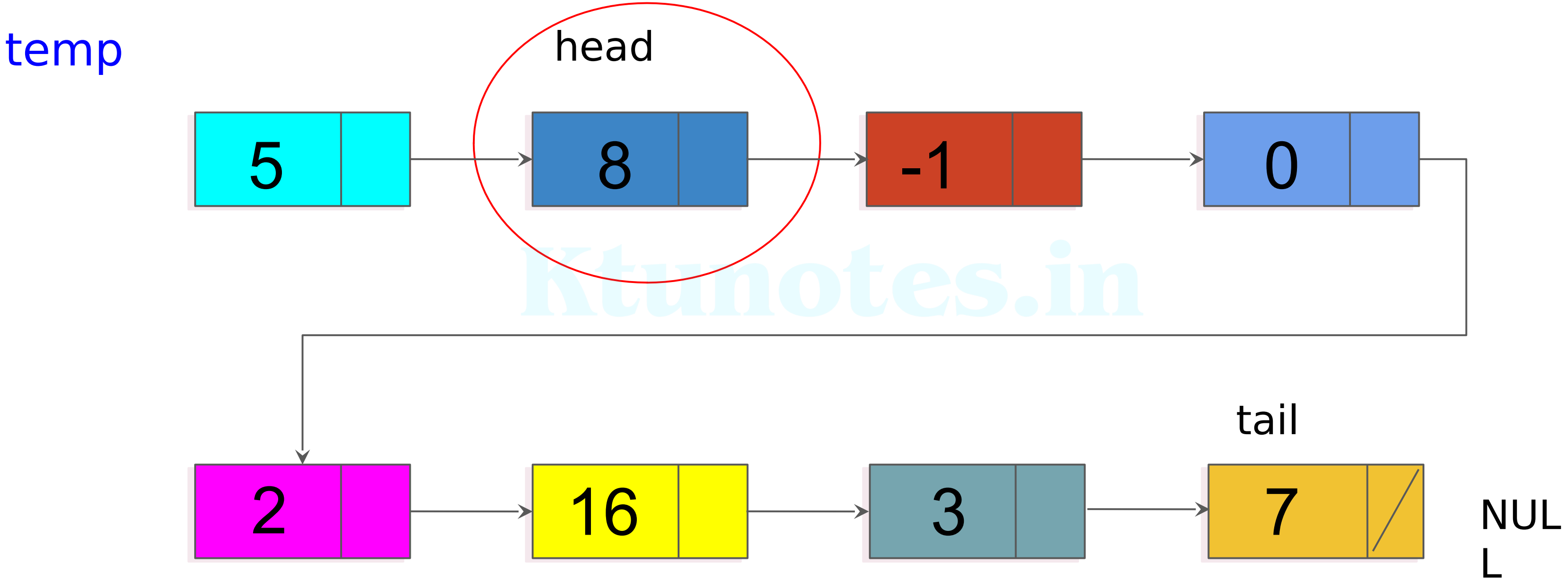
Delete the node from the beginning

Basic Operations: Deletion at the Beginning



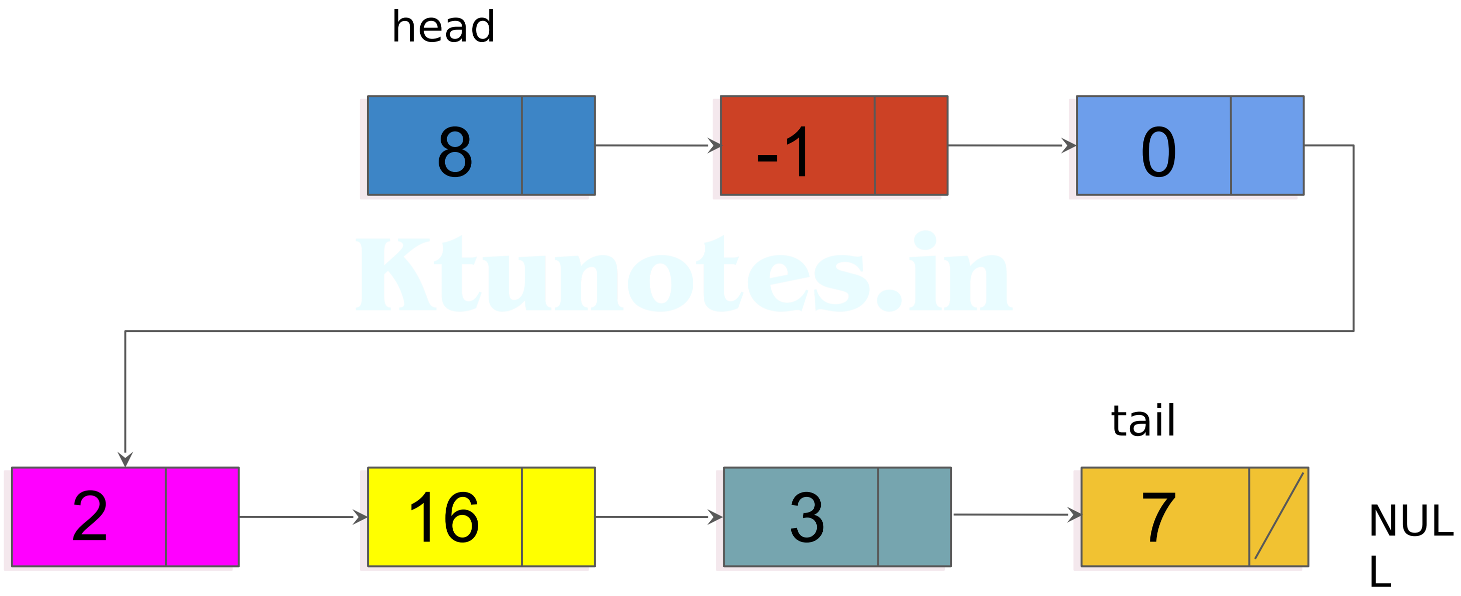
Get the node pointed by head as *temp*

Basic Operations: Deletion at the Beginning



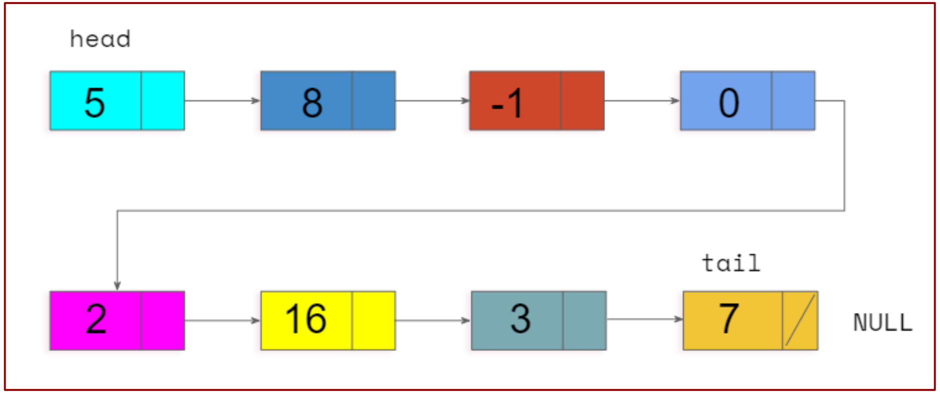
Point head to *temp's* next

Basic Operations: Deletion at the Beginning



Free the memory used by node *temp*

Memory Visualization

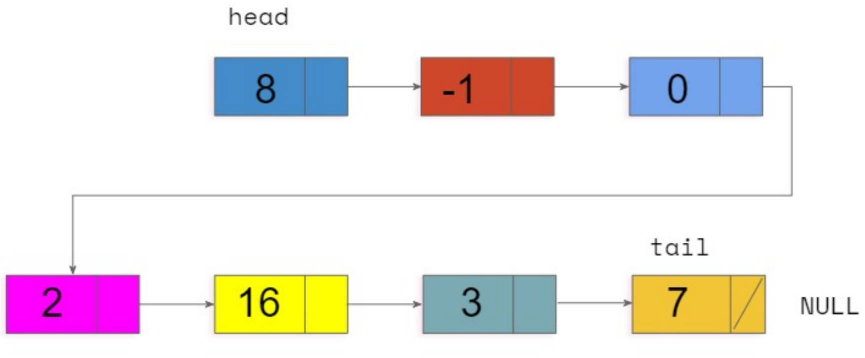


8	3016			-1	4004			
2000	2004	2008	...	3016	3020	3024	3028	...

0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

3	5012		7	NULL		5	2000	
5000	5004	5008	5012	5016	...	6024	6028	...

Memory Visualization



▨ Freed Memory
(Can be re-allocated)

8	3016			-1	4004			
2000	2004	2008	...	3016	3020	3024	3028	...

0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

3	5012		7	NULL		5	2000	
5000	5004	5008	5012	5016	...	6024	6028	...

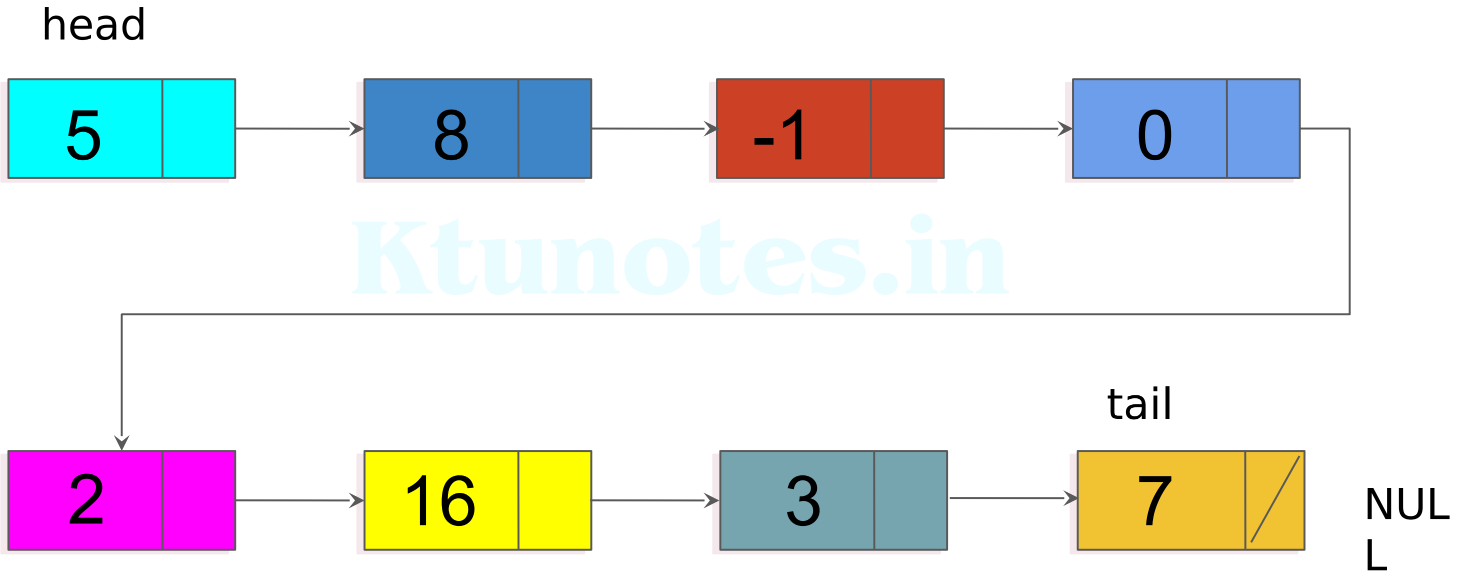
Basic Operations: Deletion at Middle/End or Deletion after Node X

Get the node pointed by X as temp

Point X’s next to temp’s next.

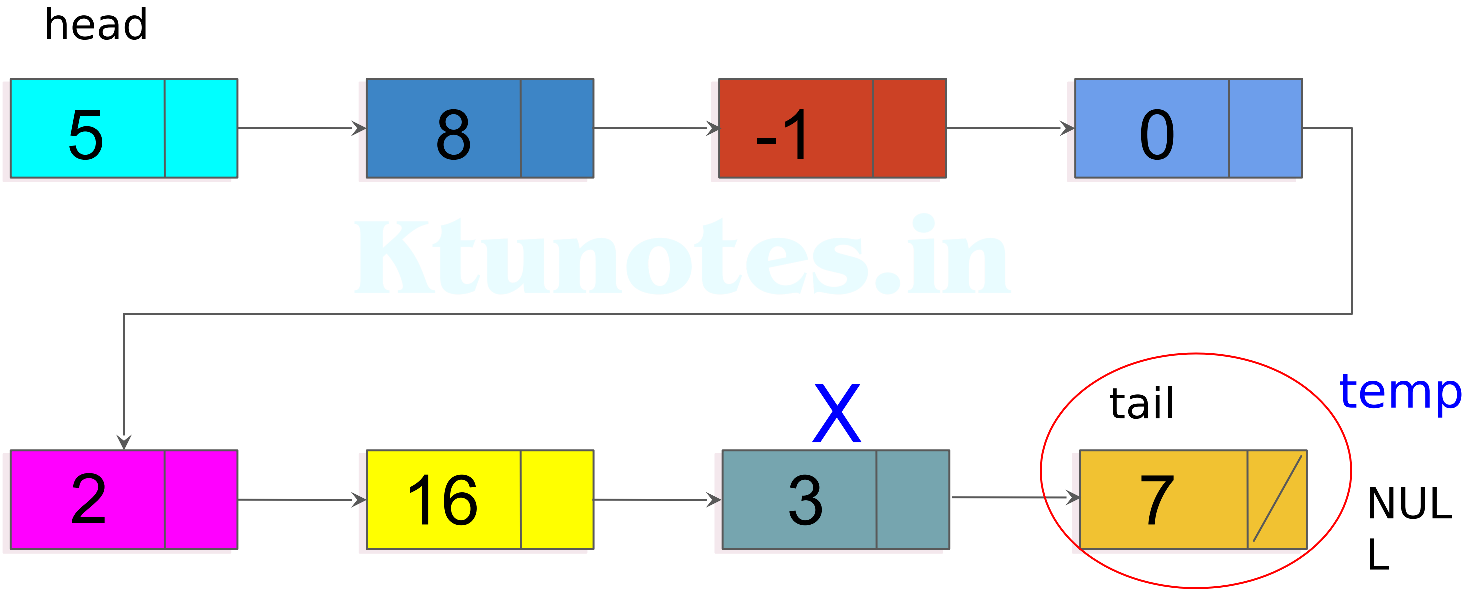
Free memory used by temp node

Basic Operations: Deletion at the End



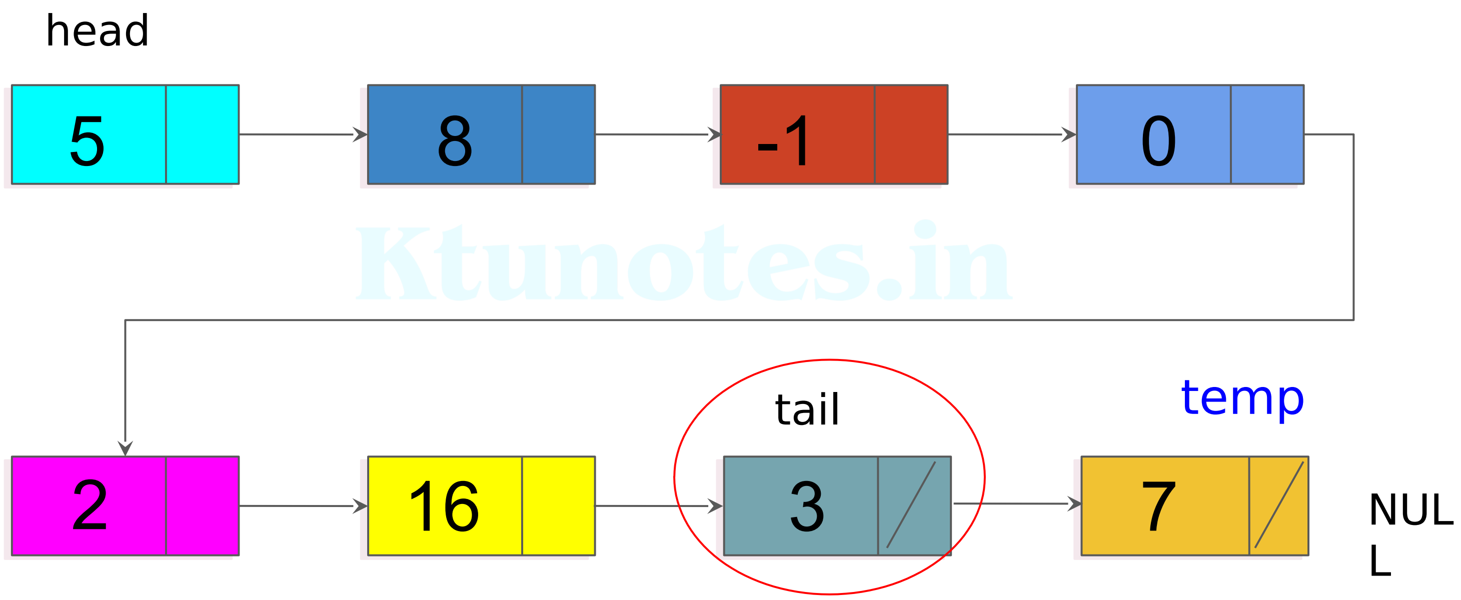
Delete the node from the end

Basic Operations: Deletion at the End



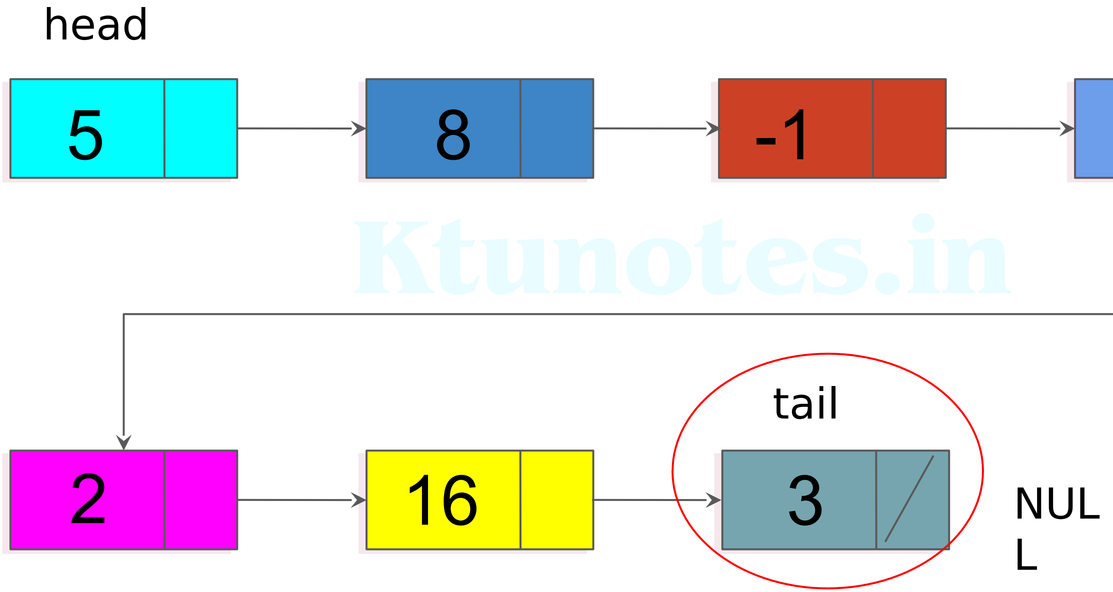
Get the node pointed by X as *temp*

Basic Operations: Deletion at the End



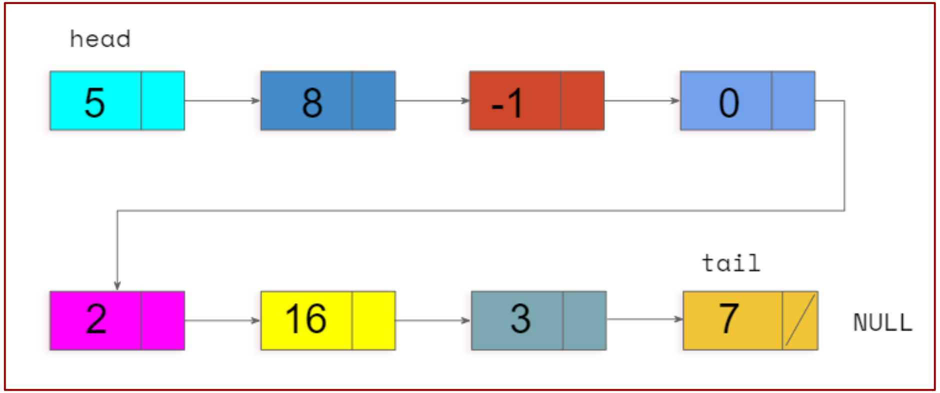
Point X’s next to temp’s next.

Basic Operations: Deletion at the End



Free memory used by temp node

Memory Visualization

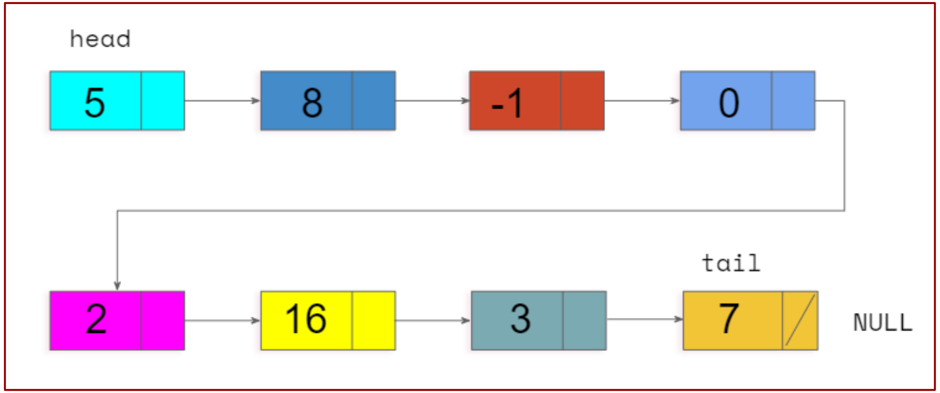


8	3016			-1	4004			
2000	2004	2008	...	3016	3020	3024	3028	...

0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

3	5012		7	NULL		5	2000	
5000	5004	5008	5012	5016	...	6024	6028	...

Memory Visualization



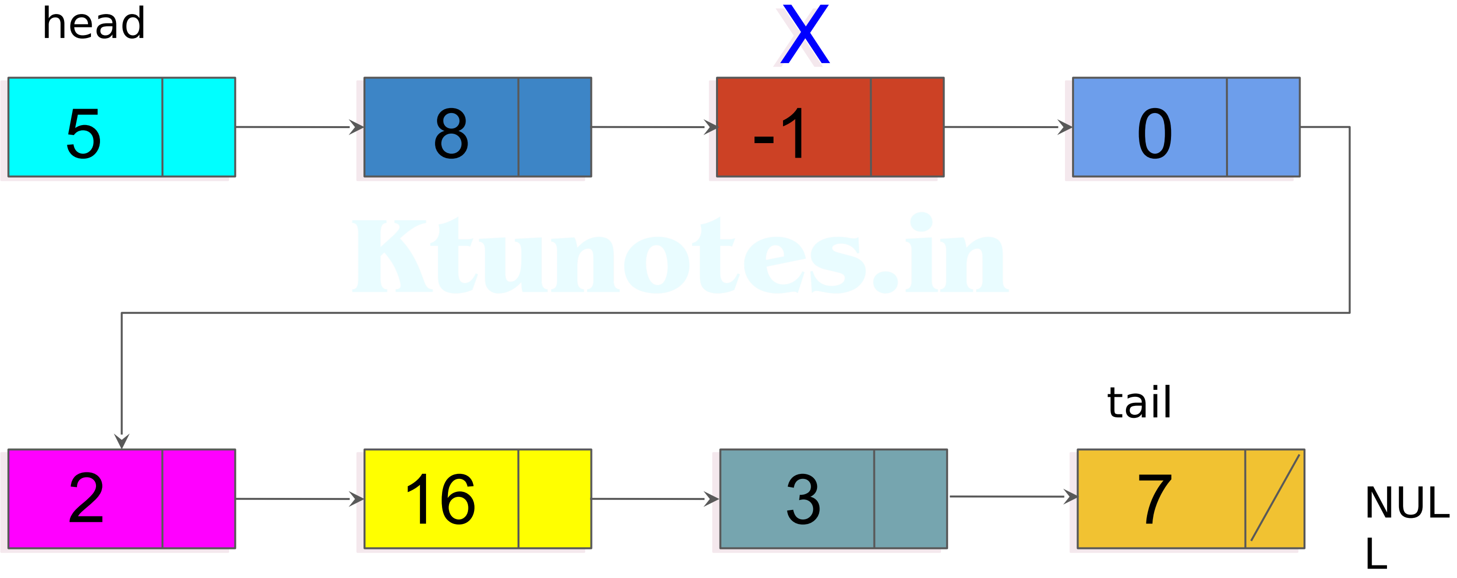
⬚ Freed Memory
(Can be re-allocated)

8	3016			-1	4004			
2000	2004	2008	...	3016	3020	3024	3028	...

0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

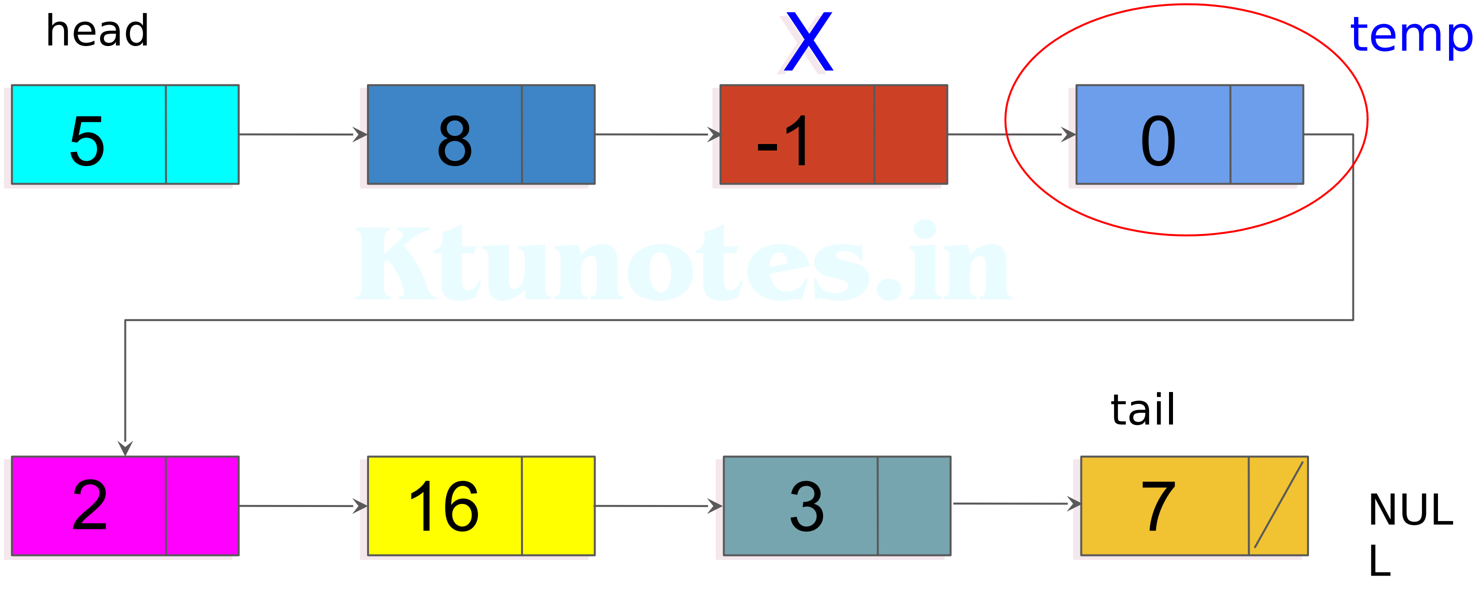
3	NULL		7	NULL		5	2000	
5000	5004	5008	5012	5016	...	6024	6028	...

Basic Operations: Deletion after position X



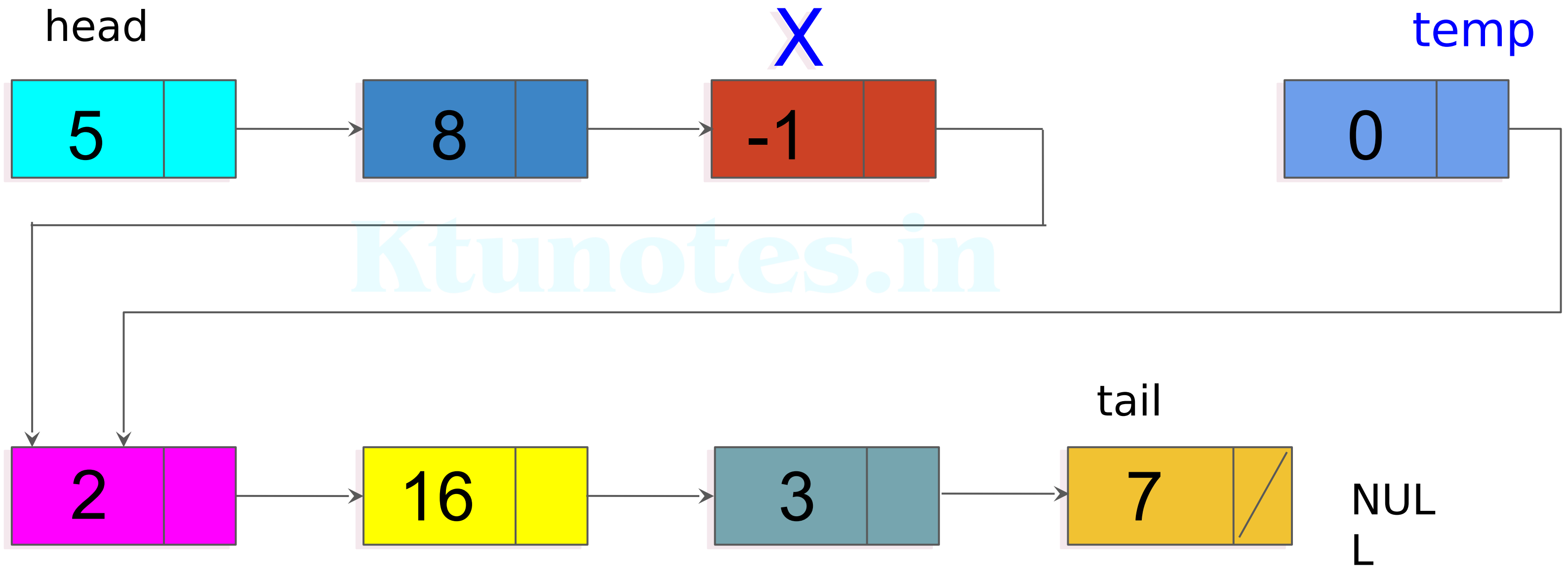
Delete after position 3

Basic Operations: Deletion after position X



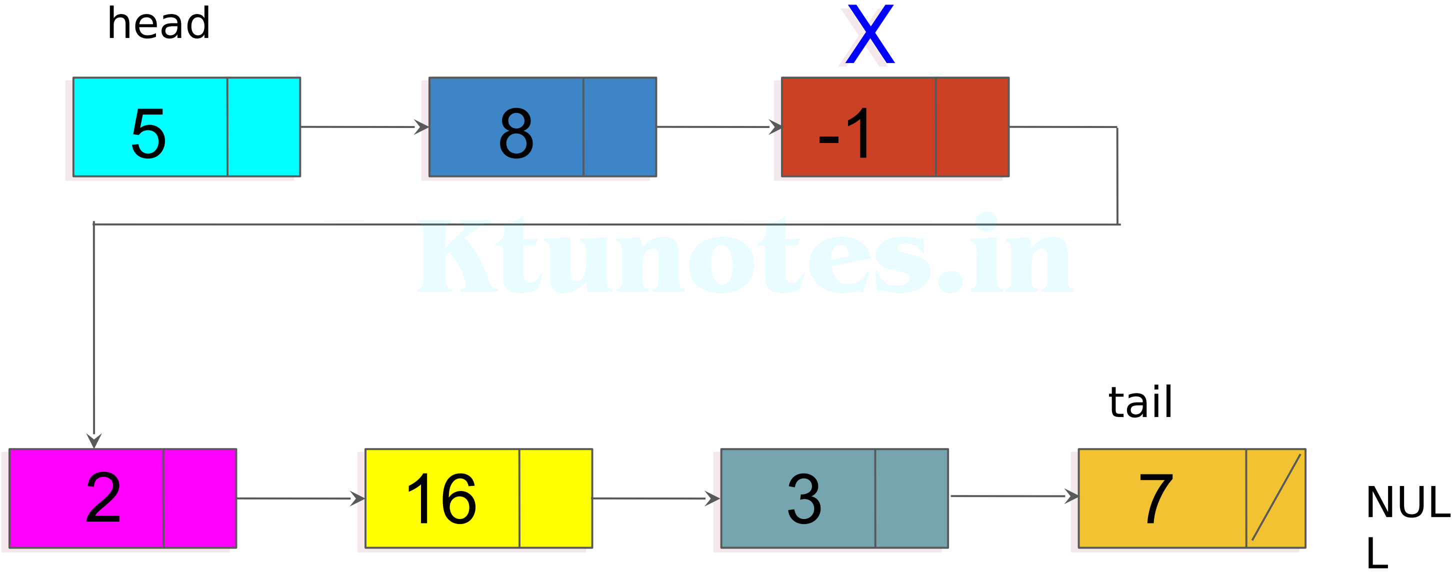
Get the node pointed by X as *temp*

Basic Operations: Deletion after position X



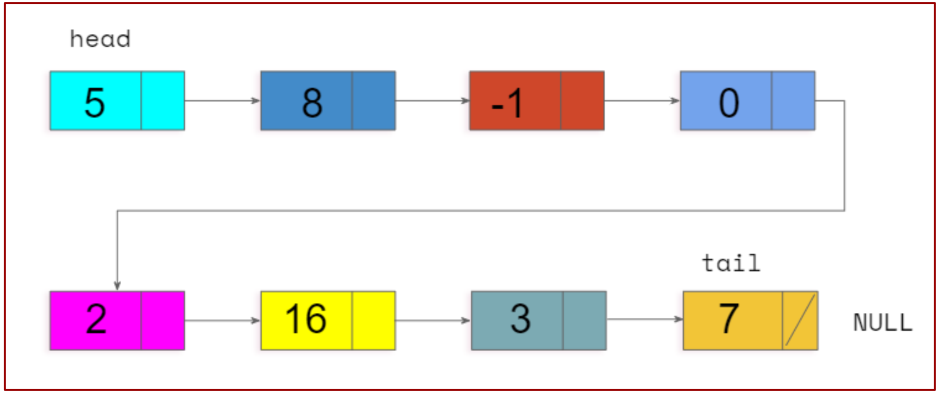
Point X’s next to temp’s next.

Basic Operations: Deletion after position X



Free memory used by temp node

Memory Visualization



░ Freed Memory
(Can be re-allocated)

8	3016			-1	4616			
2000	2004	2008	...	3016	3020	3024	3028	...

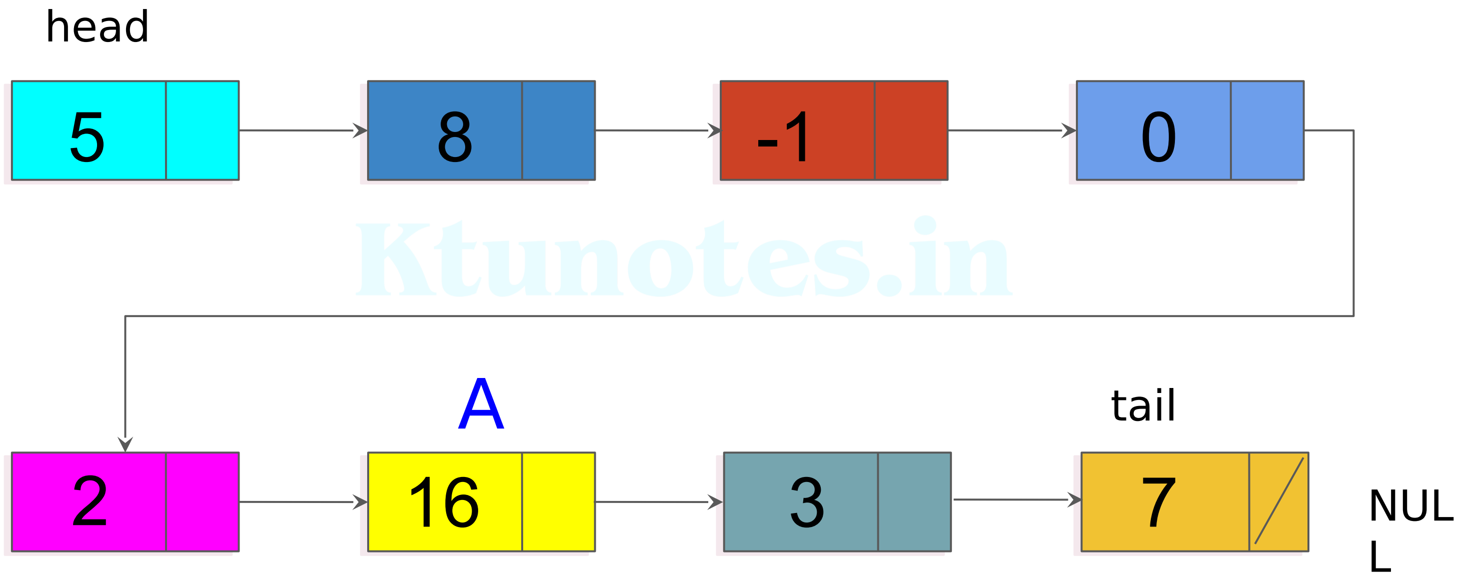
0	4616		2	4800			16	5000
4004	4008	...	4616	4620	4624	...	4800	4804

3	5012		7	NULL		5	2000	
5000	5004	5008	5012	5016	...	6024	6028	...

Traversing a Node

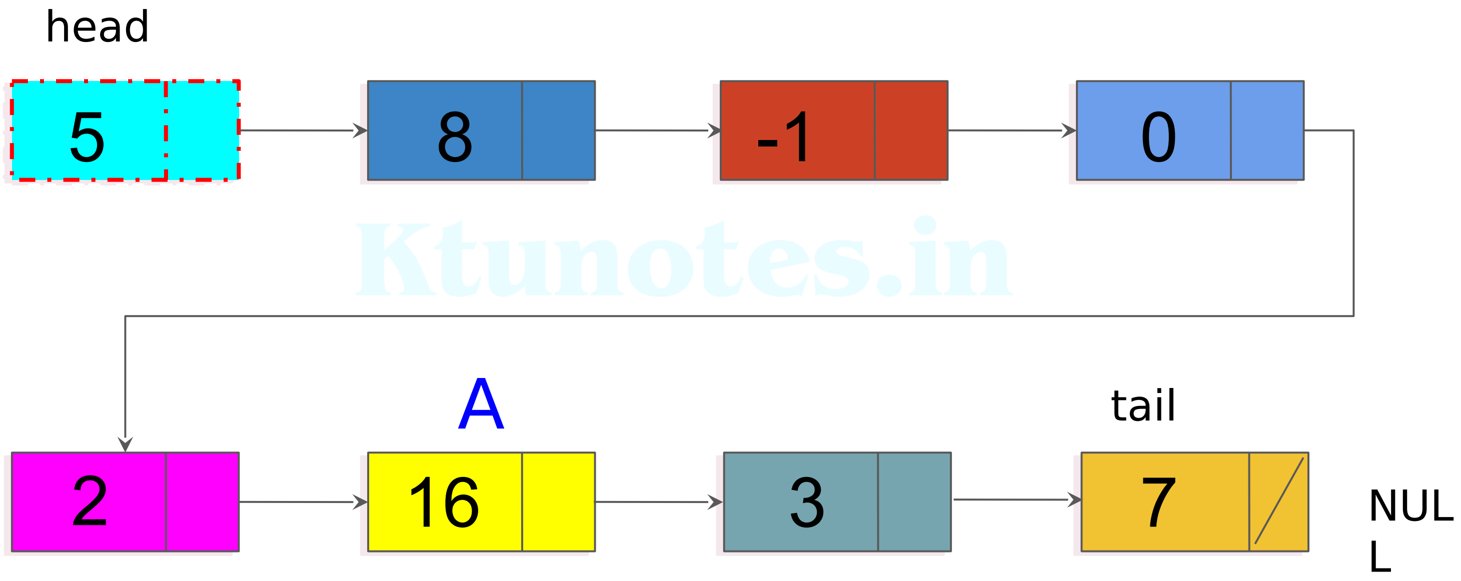
Displaying Elements, Searching Elements

Traversing Nodes in LL



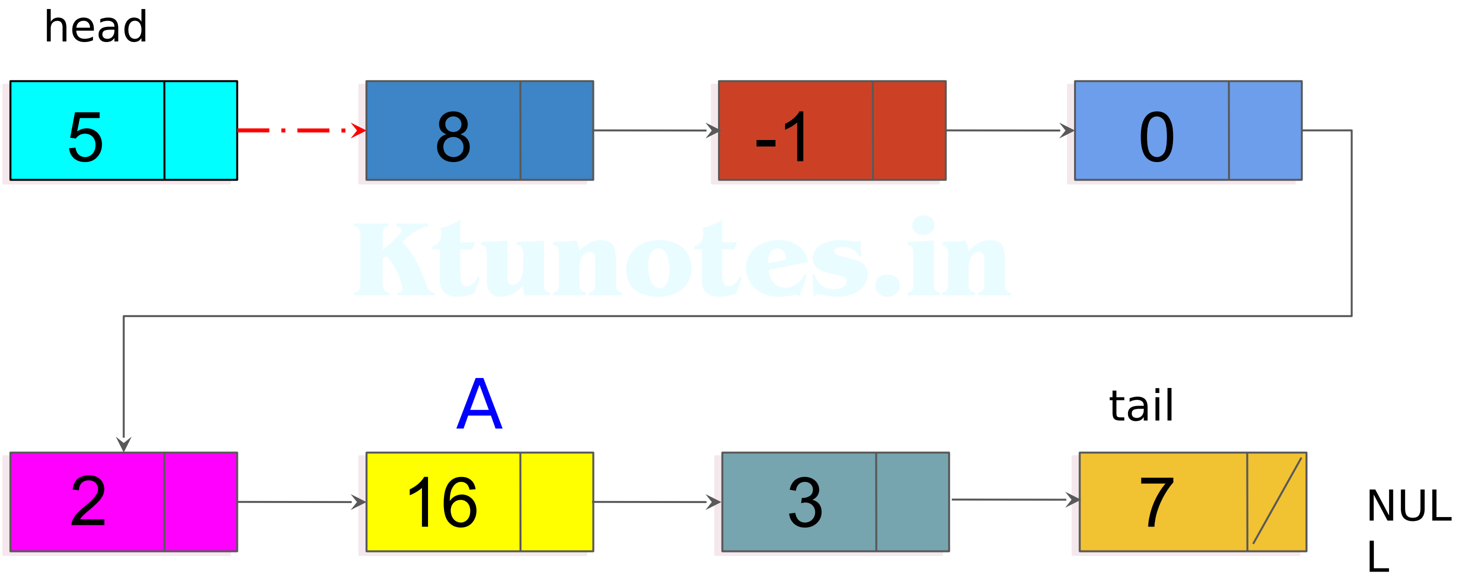
Traverse to Node A

Traversing Nodes in LL



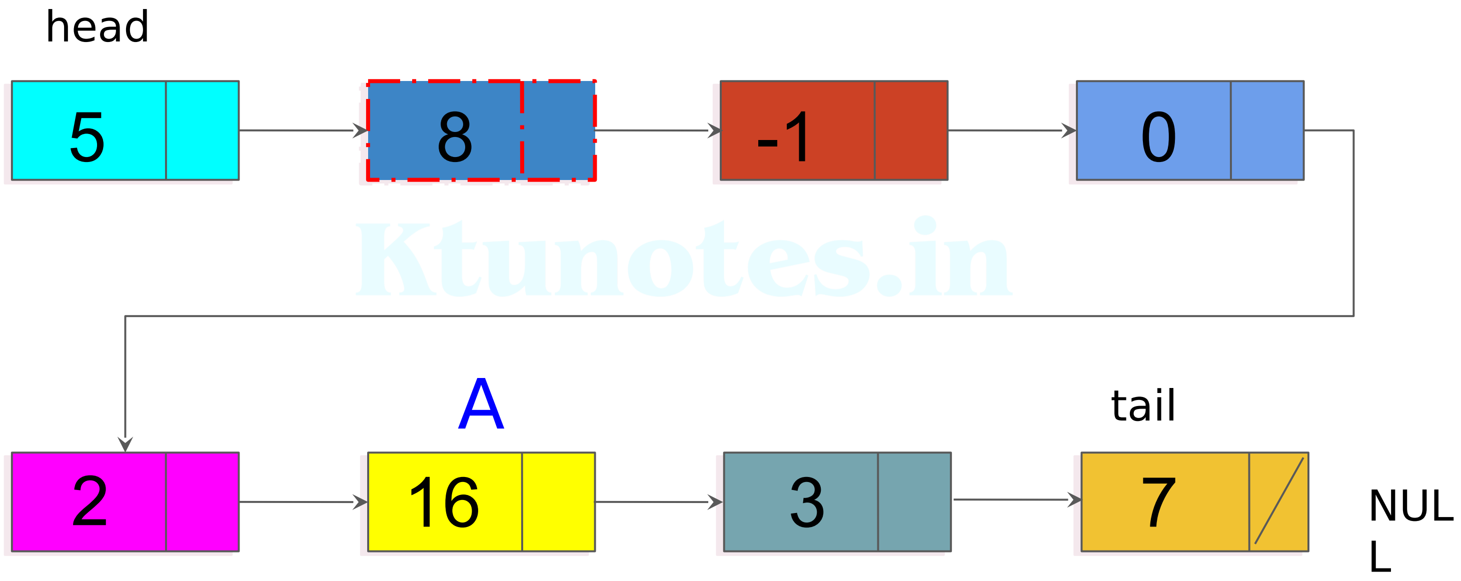
Start from head

Traversing Nodes in LL



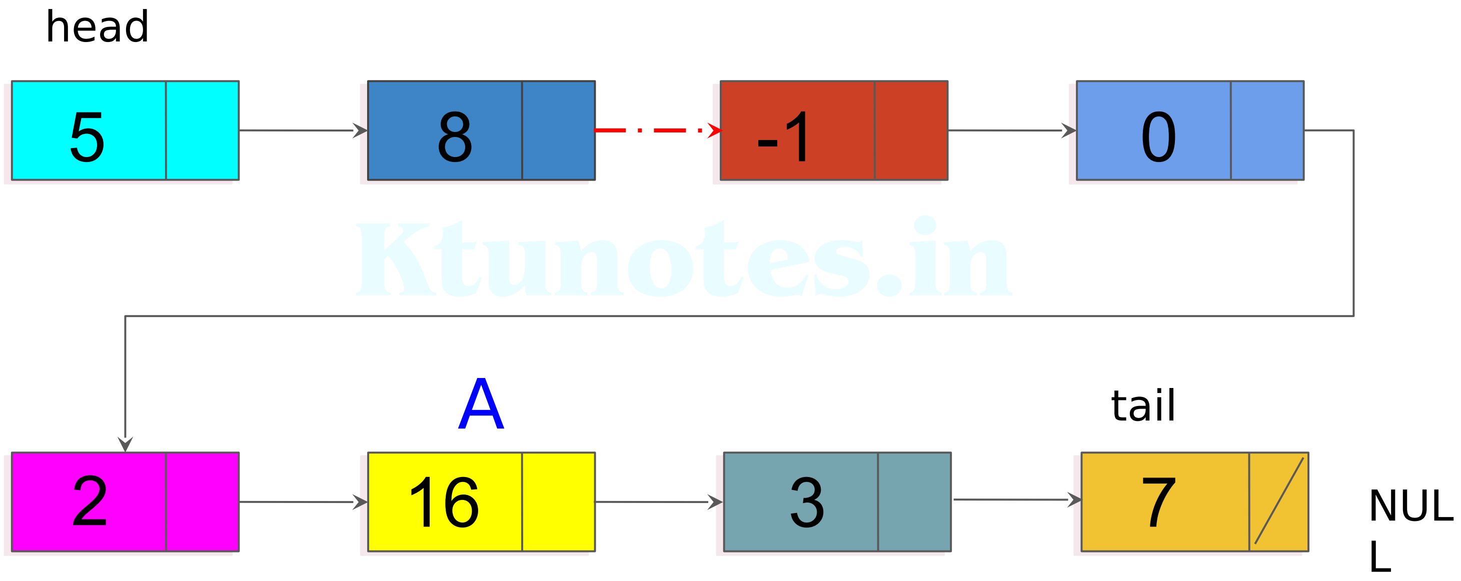
Traverse to Next Node through next pointer

Traversing Nodes in LL



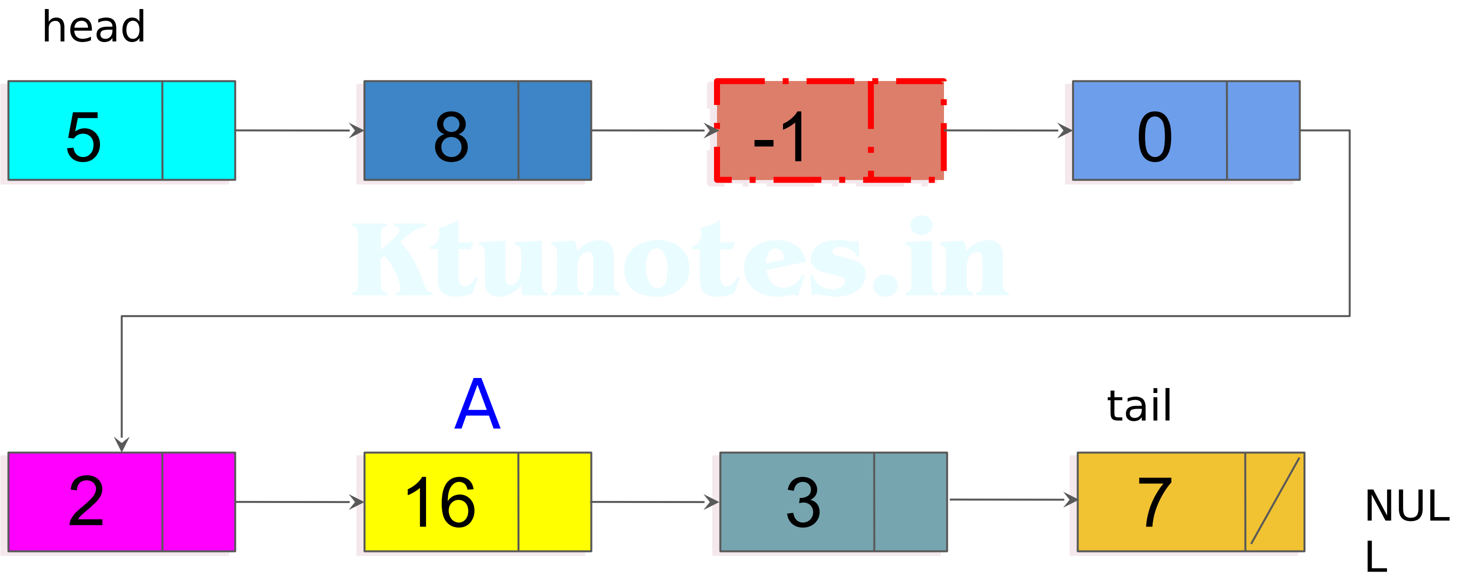
Traverse to Next Node through next pointer

Traversing Nodes in LL

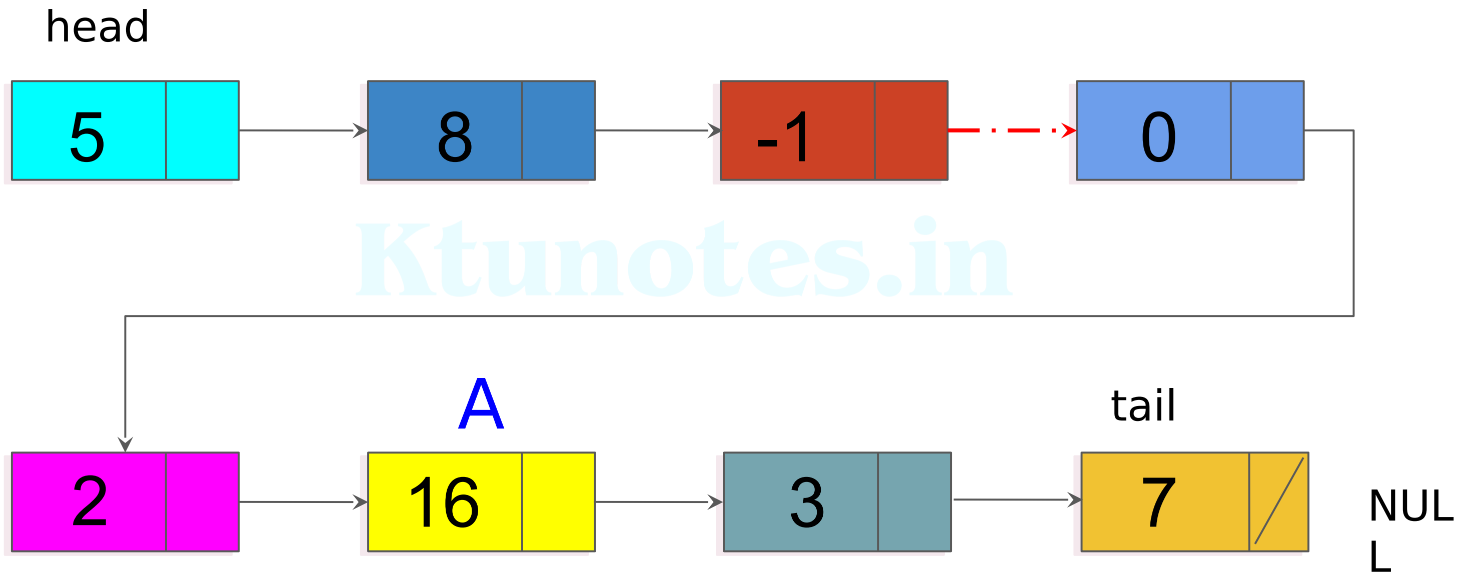


Traverse to Next Node through next pointer

Traversing Nodes in LL

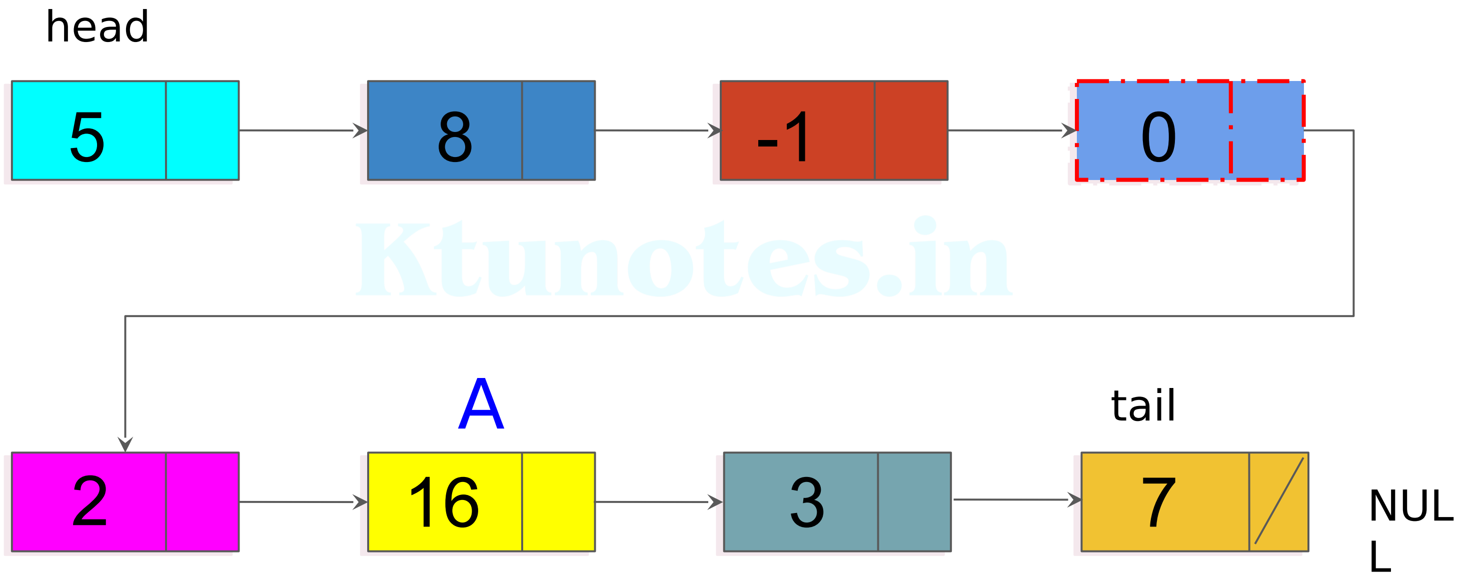


Traversing Nodes in LL

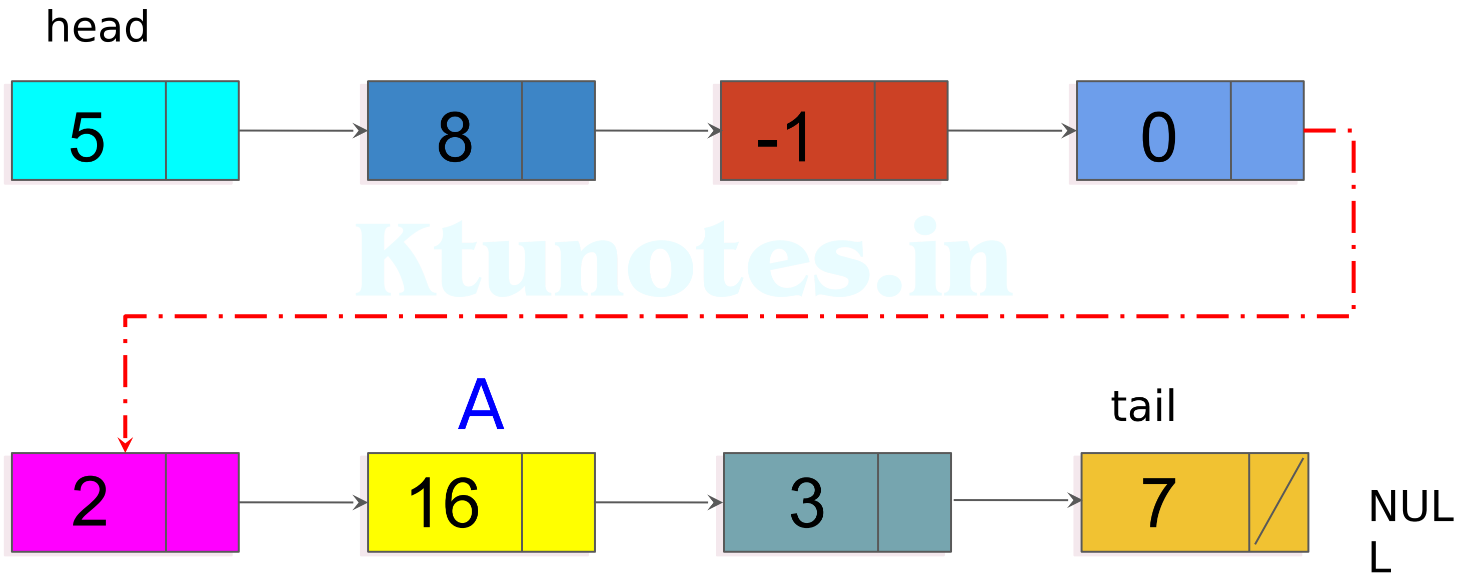


Traverse to Next Node through next pointer

Traversing Nodes in LL

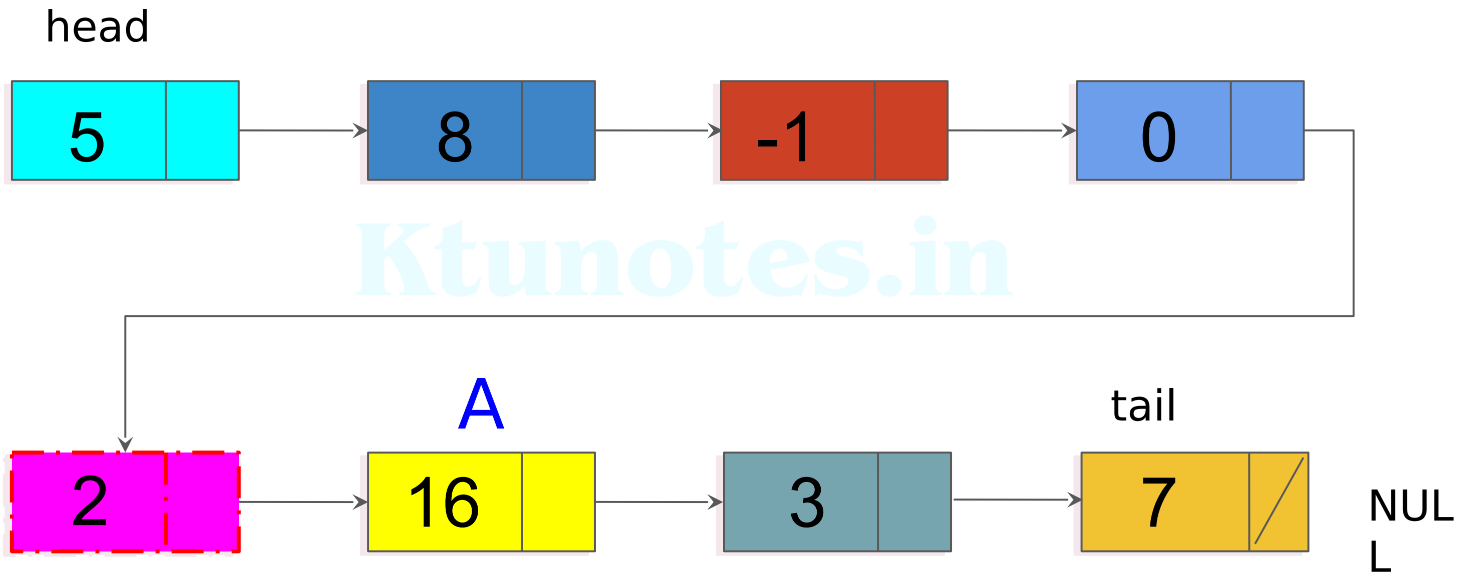


Traversing Nodes in LL

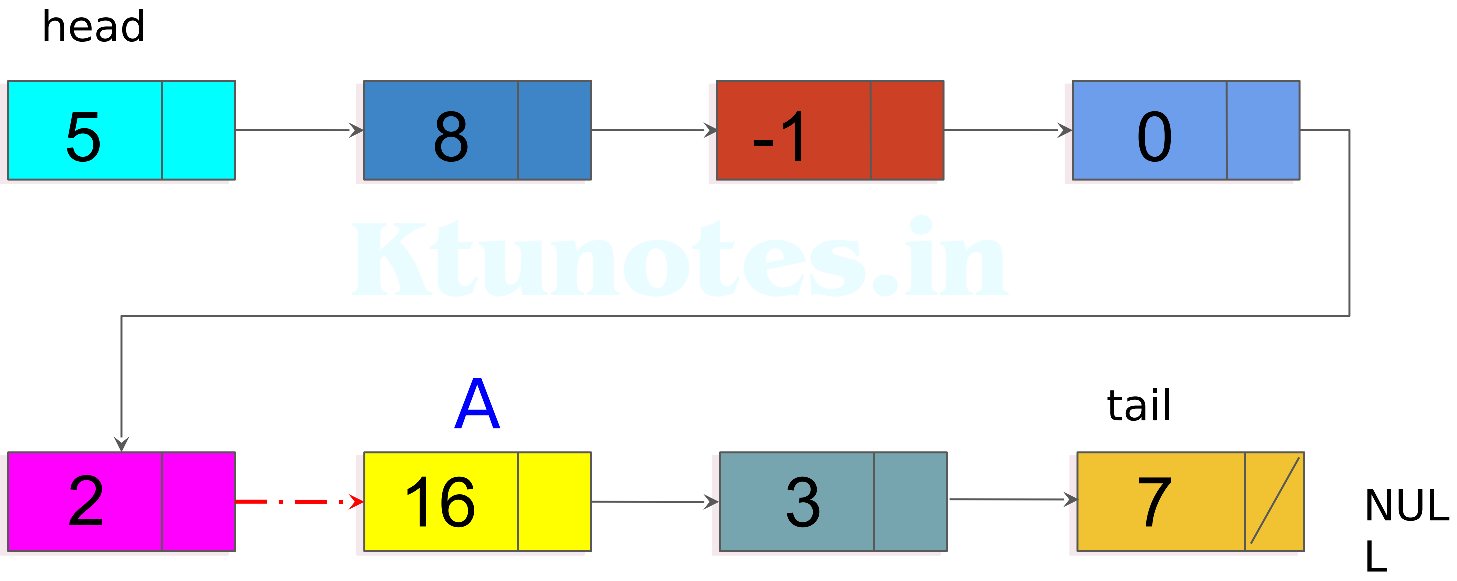


Traverse to Next Node through next pointer

Traversing Nodes in LL

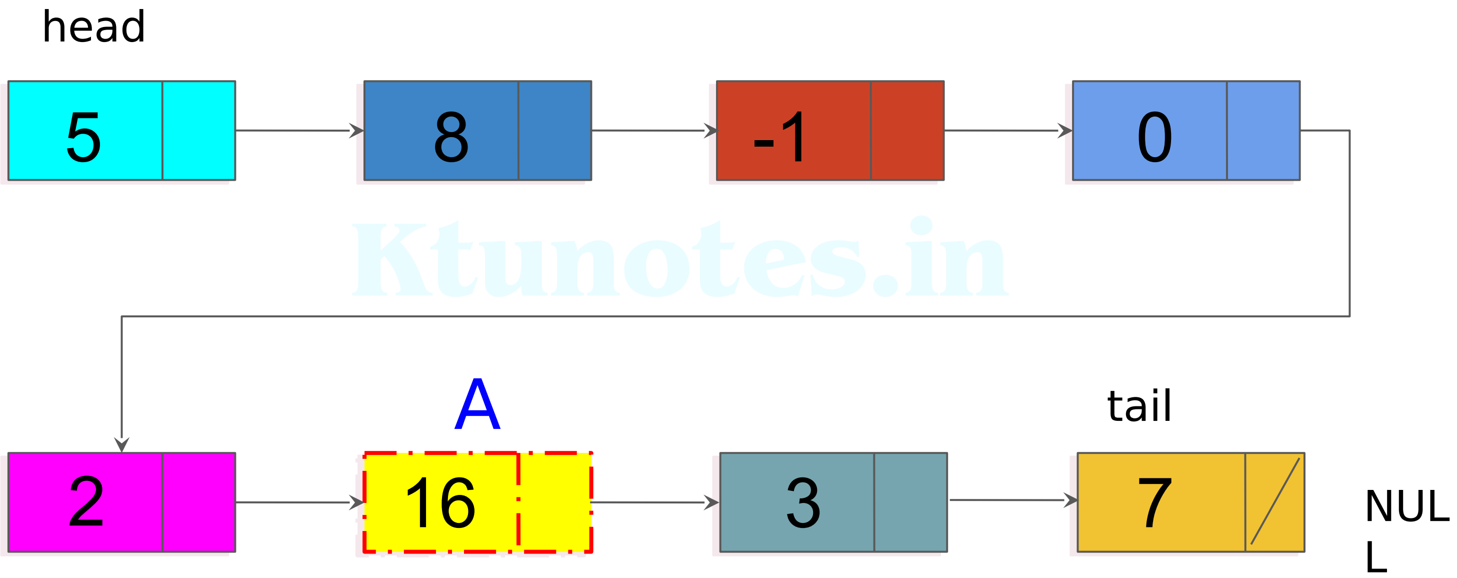


Traversing Nodes in LL



Traverse to Next Node through next pointer

Traversing Nodes in LL



Reached Node A

Dynamic Memory Allocation

Static Memory Allocation

Memory for named variables is allocated by the compiler.

Exact size and type of storage must be known at compile time.

For standard array declarations, this is why the size has to be constant.

Dynamic Memory Allocation

Memory allocated "on the fly" during run time

Exact amount of space or number of items does not have to be known by the compiler in advance.

For dynamic memory allocation, pointers are crucial

Dynamic Memory Allocation in C

Function	Description
malloc	allocates the specified number of bytes
realloc	increases or decreases the size of the specified block of memory, moving it if necessary
calloc	allocates the specified number of bytes and initializes them to zero
free	releases the specified block of memory back to the system

Self Referential Structure for LL

```
struct node

{

      int data;

      struct node *next;

};
```

```
struct node *head=NULL,*newnode=NULL,*current;
```

Doubly Linked Lists

Doubly Linked Lists

Doubly linked list is a type of linked list in which each node apart from storing its data has two links.

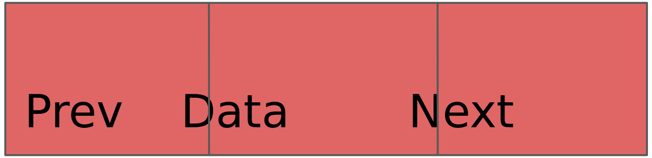
The first link points to the previous node in the list and the second link points to the next node in the list.

The first node of the list has its previous link pointing to NULL , similarly the last node of the list has its next node pointing to NULL.

Doubly Linked Lists

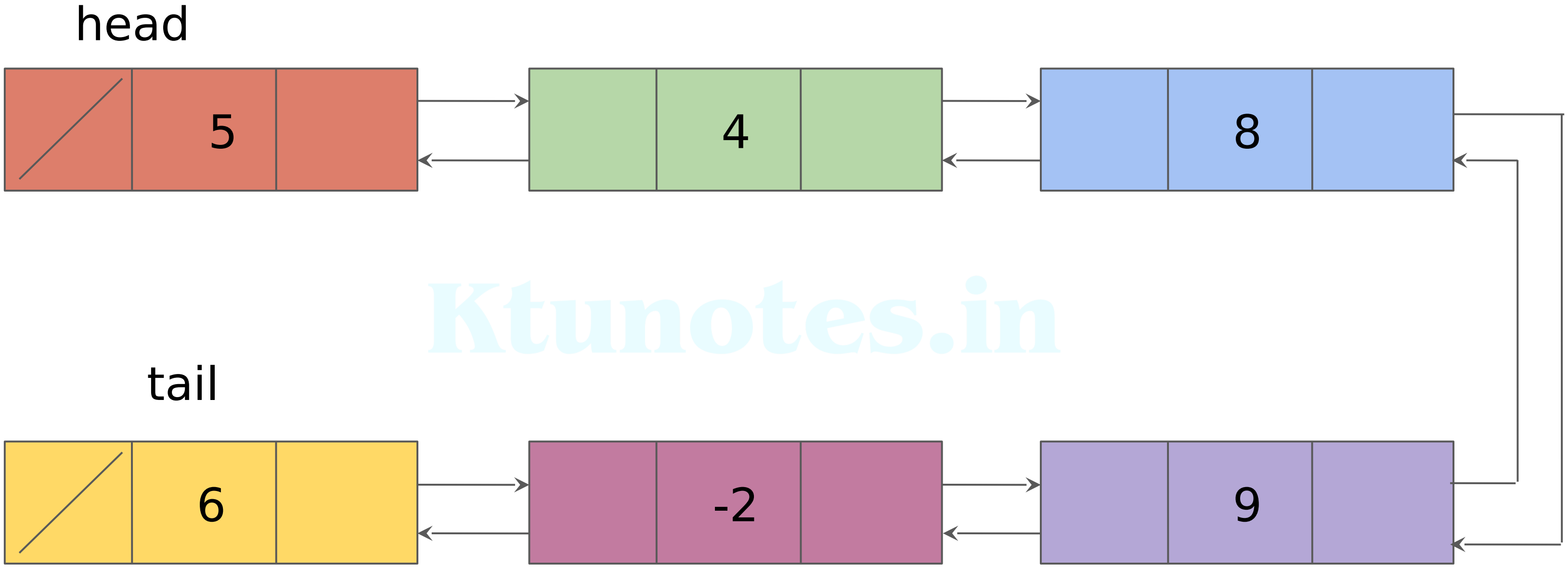
A doubly linked list allows traversal in both directions.

```
struct node

{

     int data;      // Data

     node *prev;       // A reference to the previous node

     node *next;       // A reference to the next node

  };
```

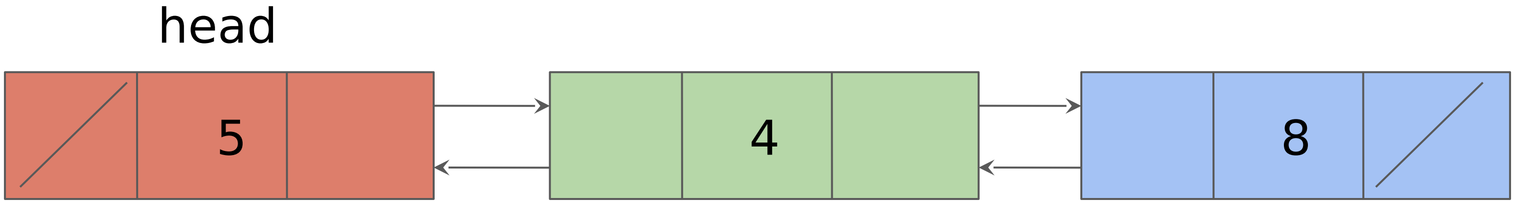


Prev Data Next

Visualization



Representation of Doubly Linked List in Memory



NULL	5	2020			2000	4	4016	
2000	2004	2008	2012	2016	2020	2024	2028	...

MAIN MEMORY

				2020	8	NULL		
...	4004	4008	4012	4016	4020	4024	4028	4032

Basic Operations: Insertion at the Beginning

Create a new node with given data.

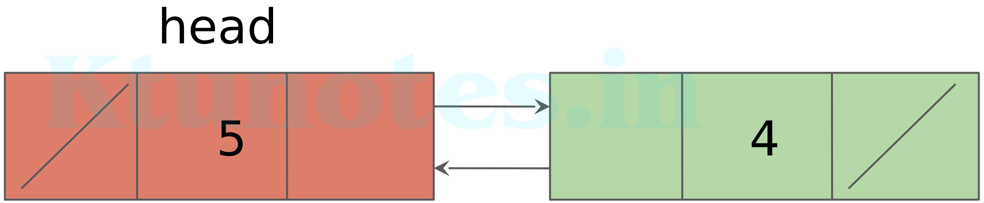
Point new node’s next to head

Point prev of head to new node

Make prev of new node to NULL

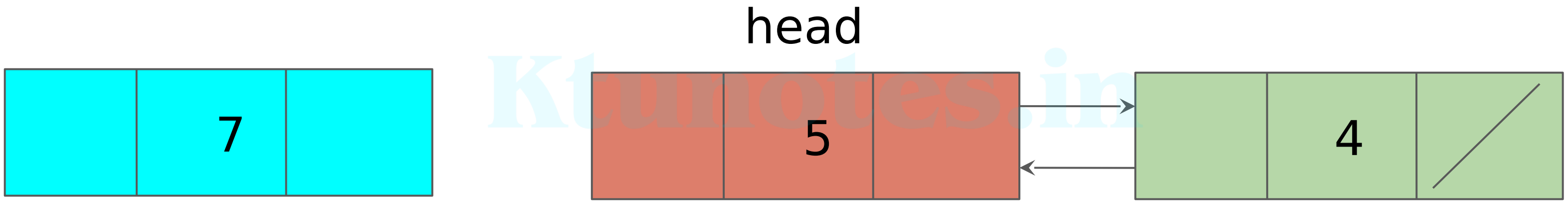
Point head to the new node.

Insertion at the Beginning



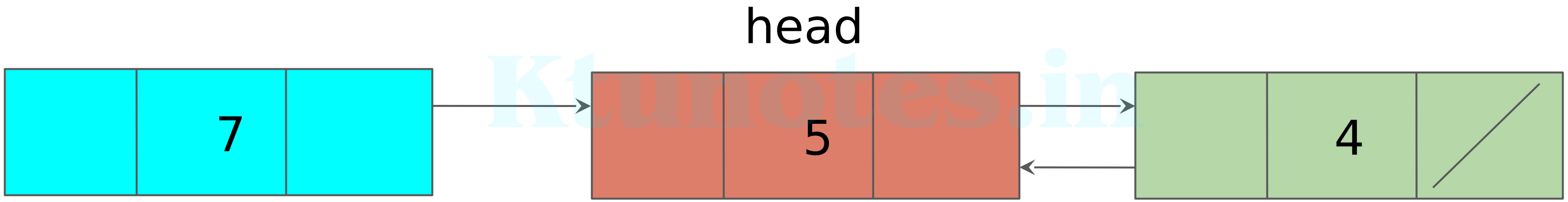
Insert a new node with value 7 at the beginning

Insertion at the Beginning



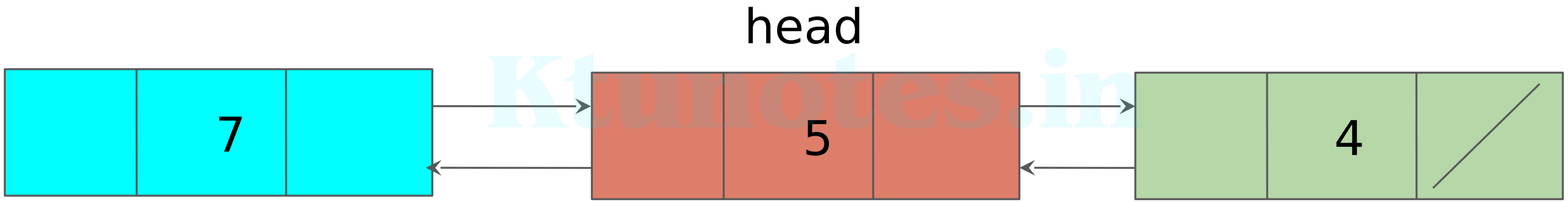
Create a new node with given data.

Insertion at the Beginning



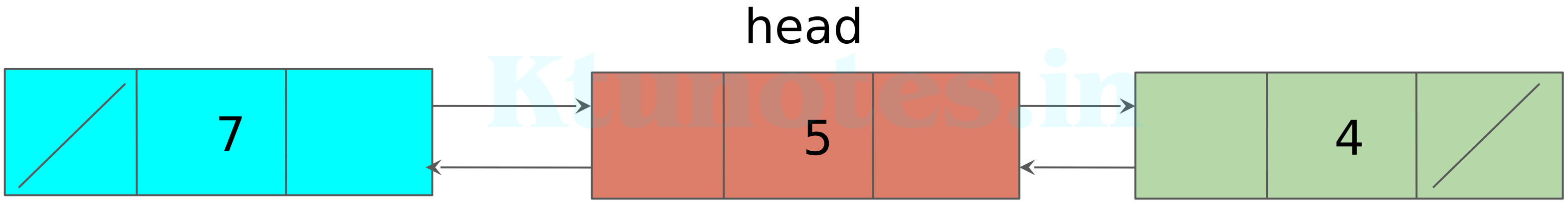
Point new node’s next to old head

Insertion at the Beginning



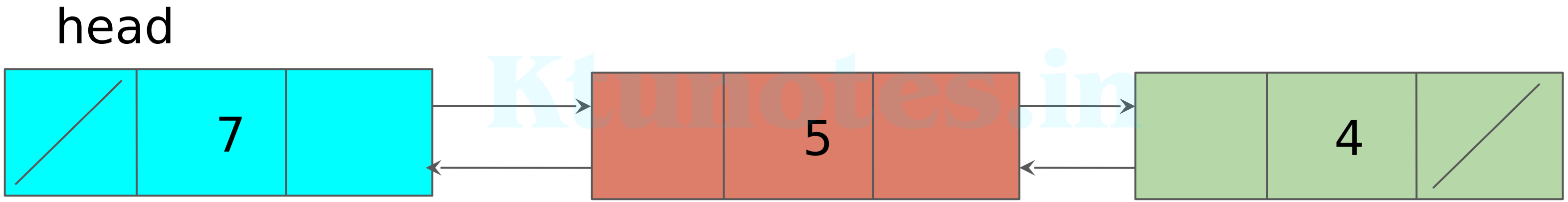
Point prev of head to new node

Insertion at the Beginning



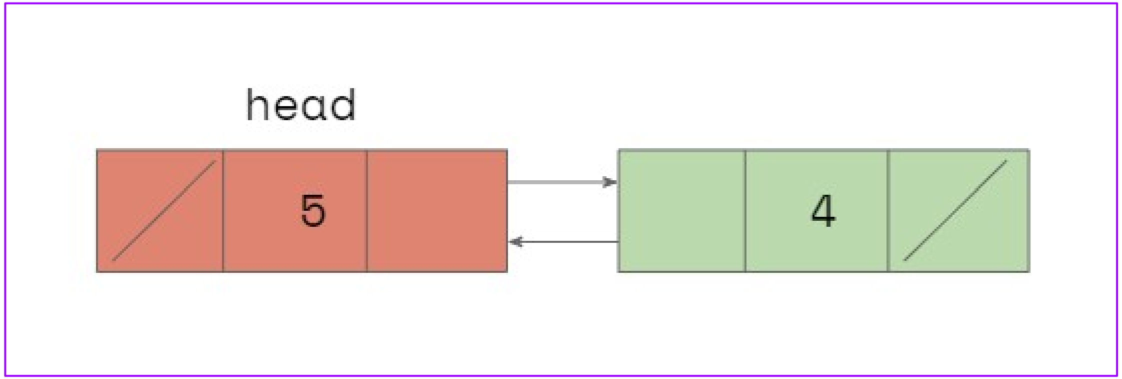
Make prev of new node to NULL

Insertion at the Beginning



Point head to this new node.

Representation of Doubly Linked List in Memory

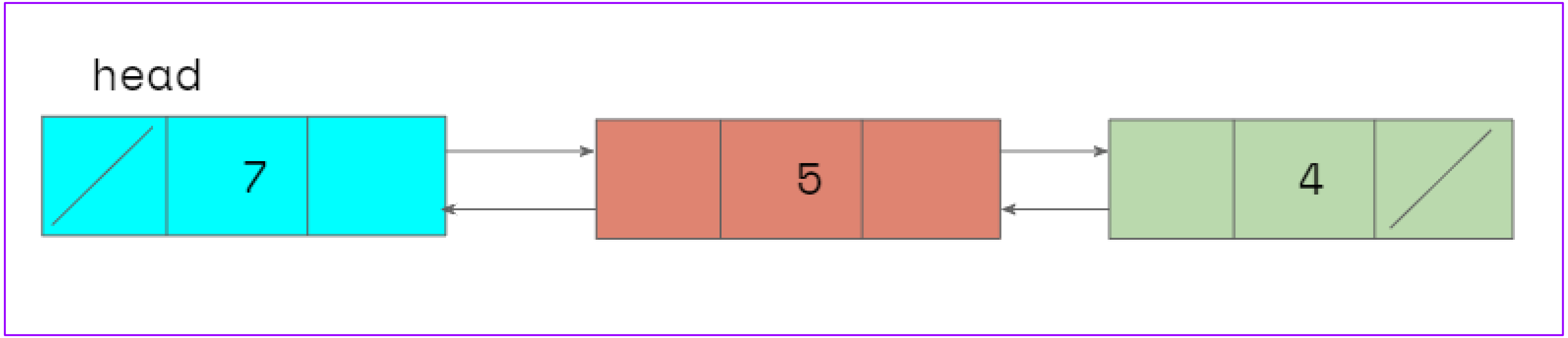


NULL	5	2020			2000	4	NULL	
2000	2004	2008	2012	2016	2020	2024	2028	...

MAIN MEMORY

...	4004	4008	4012	4016	4020	4024	4028	4032

Representation of Doubly Linked List in Memory



4012	5	2020			2000	4	NULL	
2000	2004	2008	2012	2016	2020	2024	2028	...

MAIN MEMORY

			NULL	7	2000			
...	4004	4008	4012	4016	4020	4024	4028	4032

Basic Operations: Insertion at the End

Create a new node with given data.

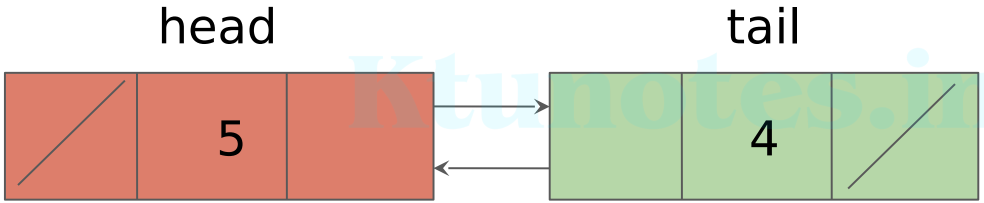
Point next of tail to new node

Point prev of new node to tail

Make next of new node to NULL

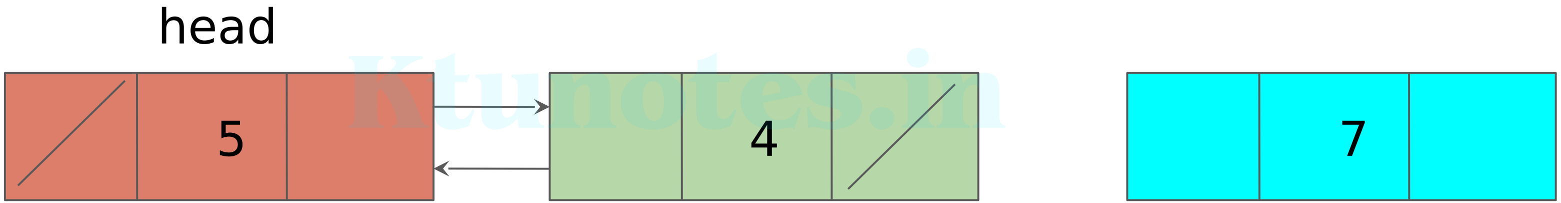
Point tail to the new node.

Insertion at the End



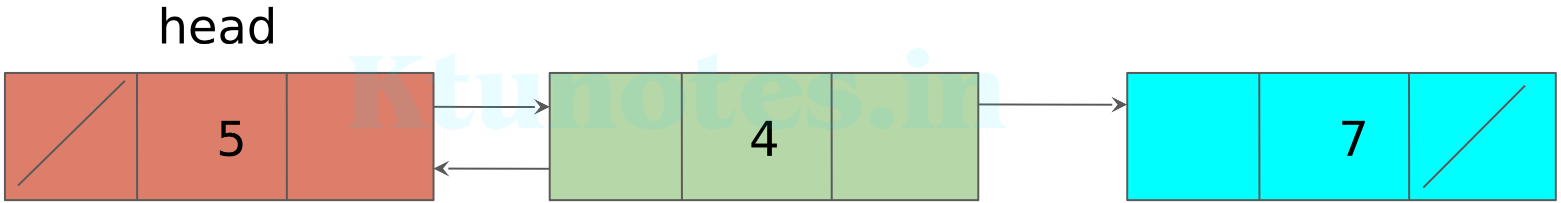
Insert a new node with value 7 at the end

Insertion at the End



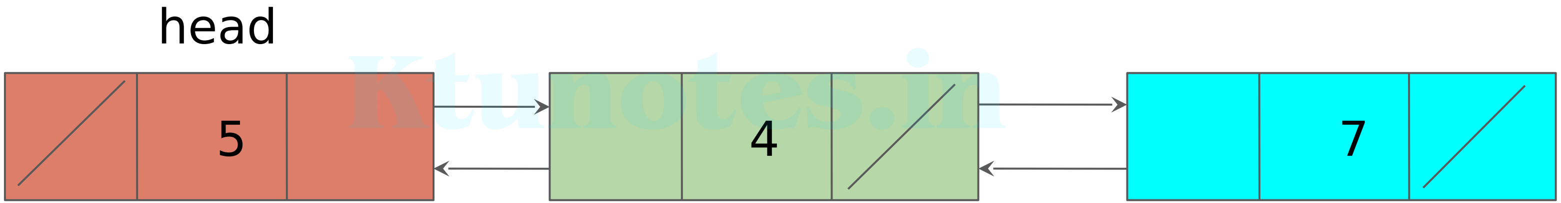
Create a new node with given data.

Insertion at the End



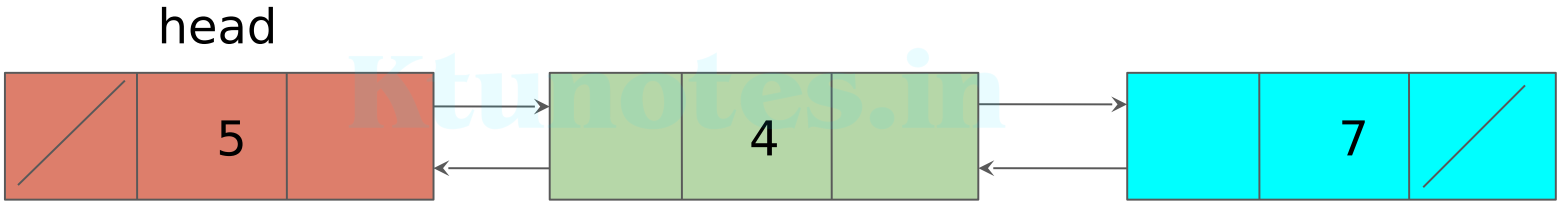
Point next of tail to new node

Insertion at the End



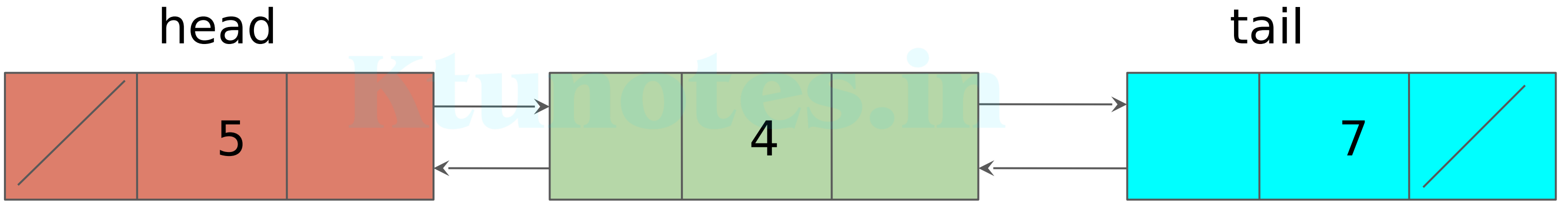
Point prev of new node to tail

Insertion at the End



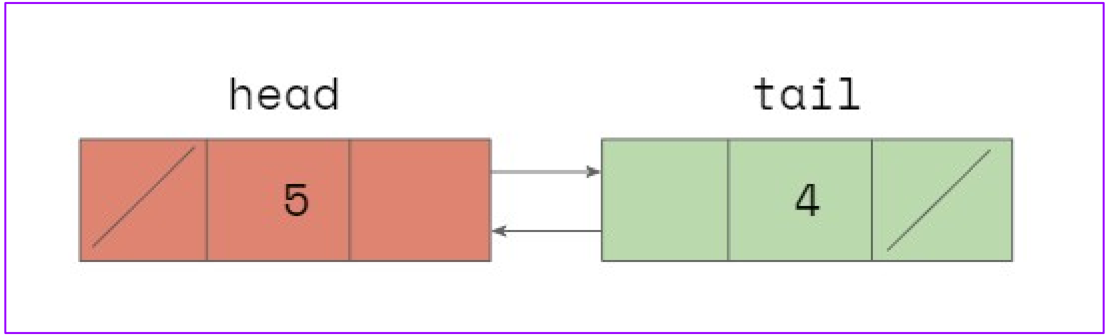
Make next of new node to NULL

Insertion at the End



Point tail to last node

Representation of Doubly Linked List in Memory

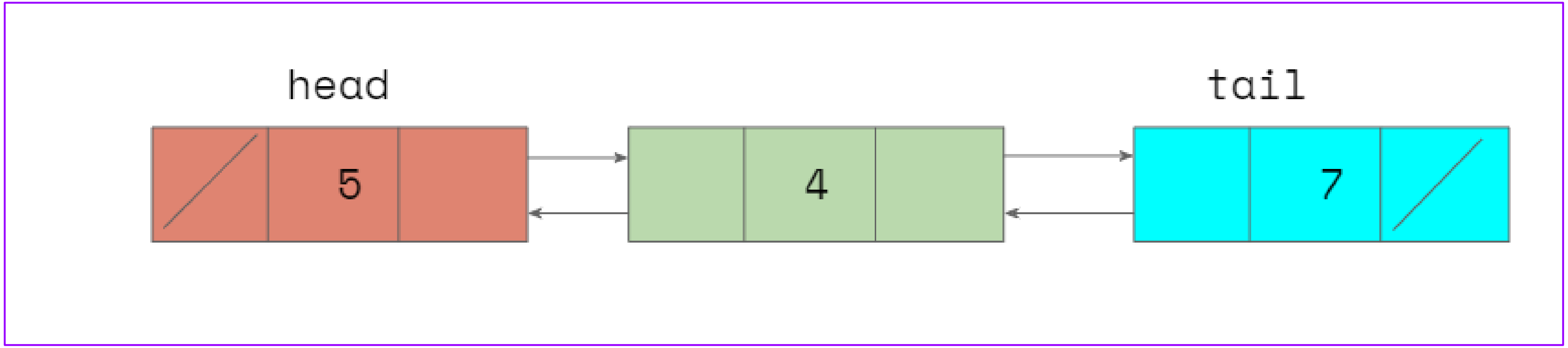


NULL	5	2020			2000	4	NULL	
2000	2004	2008	2012	2016	2020	2024	2028	...

MAIN MEMORY

...	4004	4008	4012	4016	4020	4024	4028	4032

Representation of Doubly Linked List in Memory



NULL	5	2020			2000	4	4016	
2000	2004	2008	2012	2016	2020	2024	2028	...

MAIN MEMORY

				2020	7	NULL		
...	4004	4008	4012	4016	4020	4024	4028	4032

Basic Operations: Insertion between node X and node Y

Create a new node with given data.

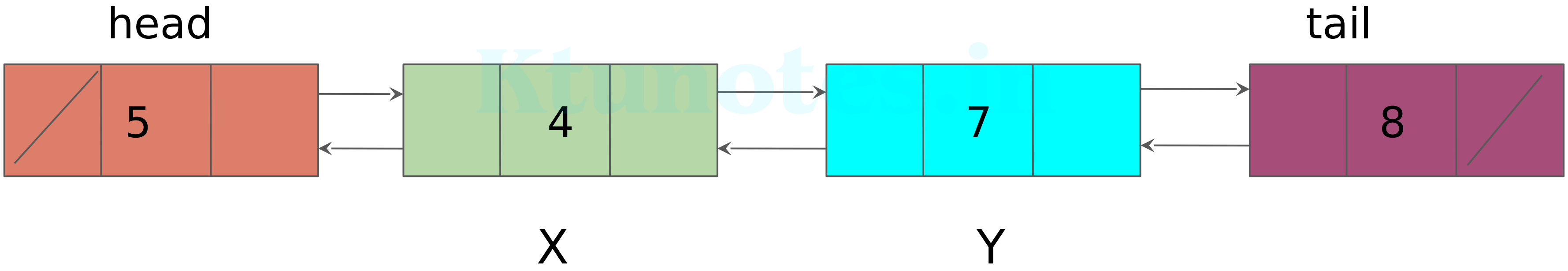
Point next of X to new node

Point prev of new node to X

Point next of new node to Y

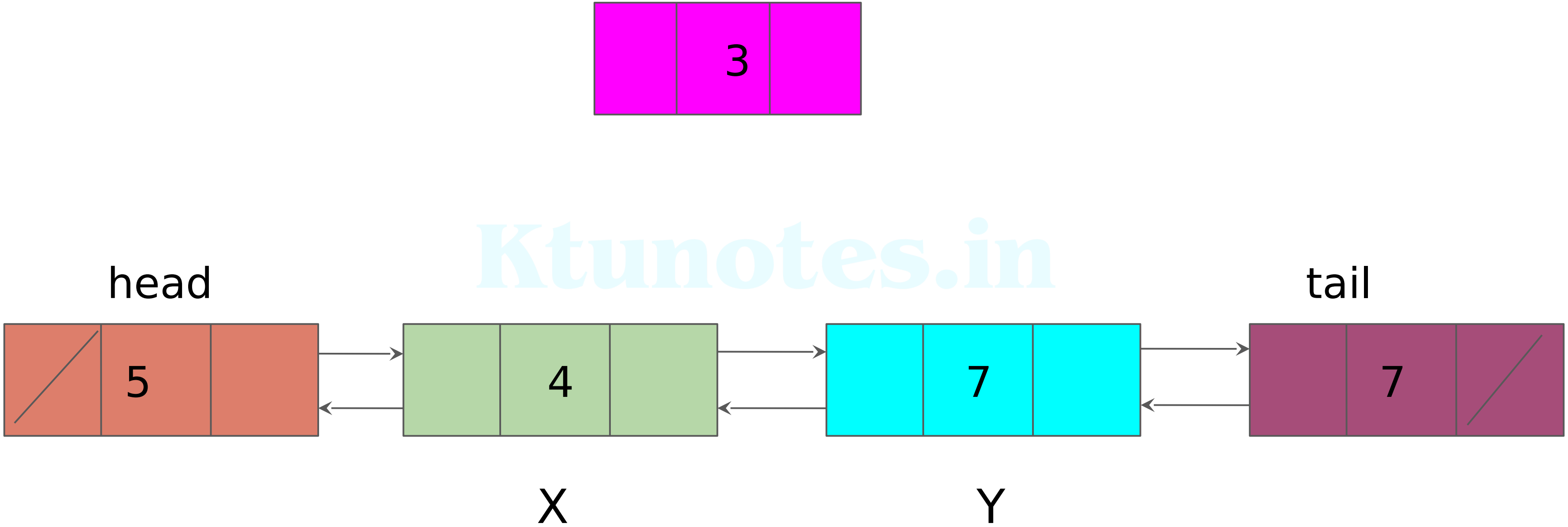
Point prev of Y to new node.

Insertion between node X and Y



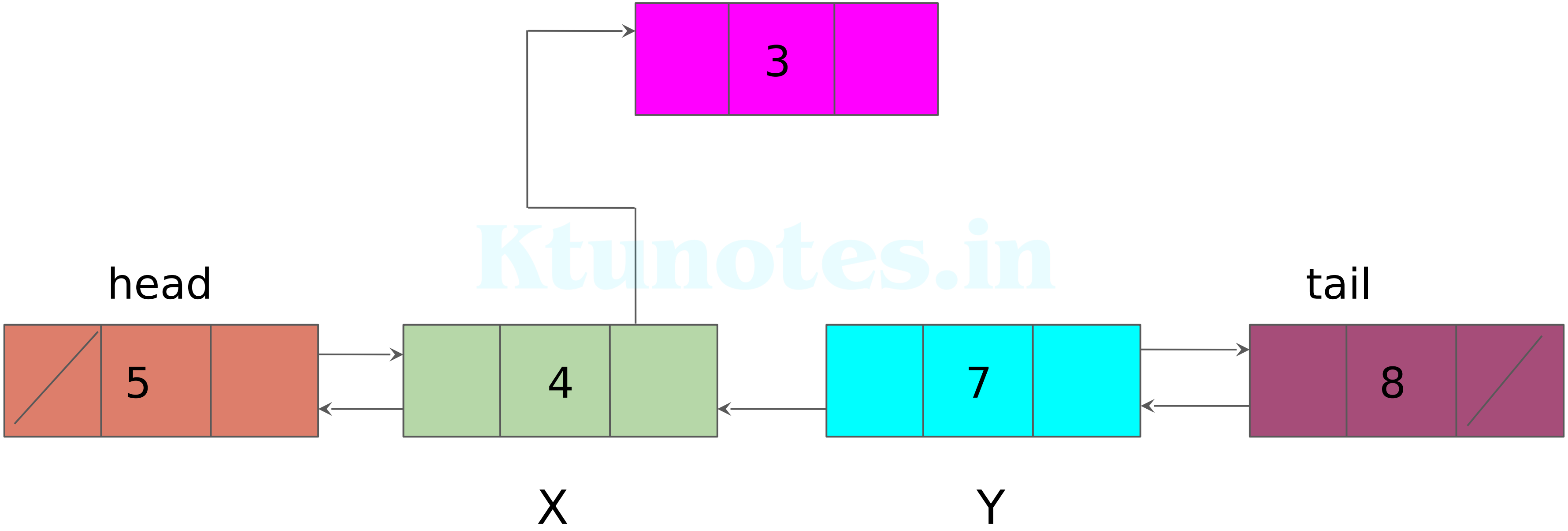
Insert a new node with value 3 between node X and Y

Insertion between node X and Y



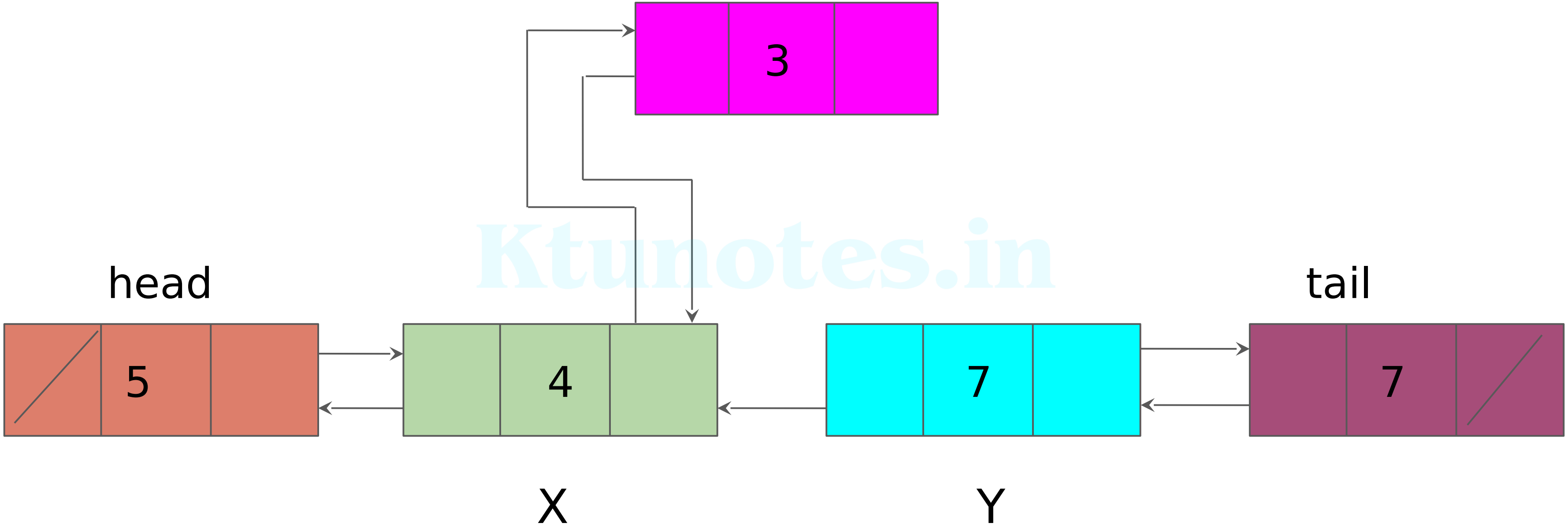
Create a new node with given data.

Insertion between node X and Y



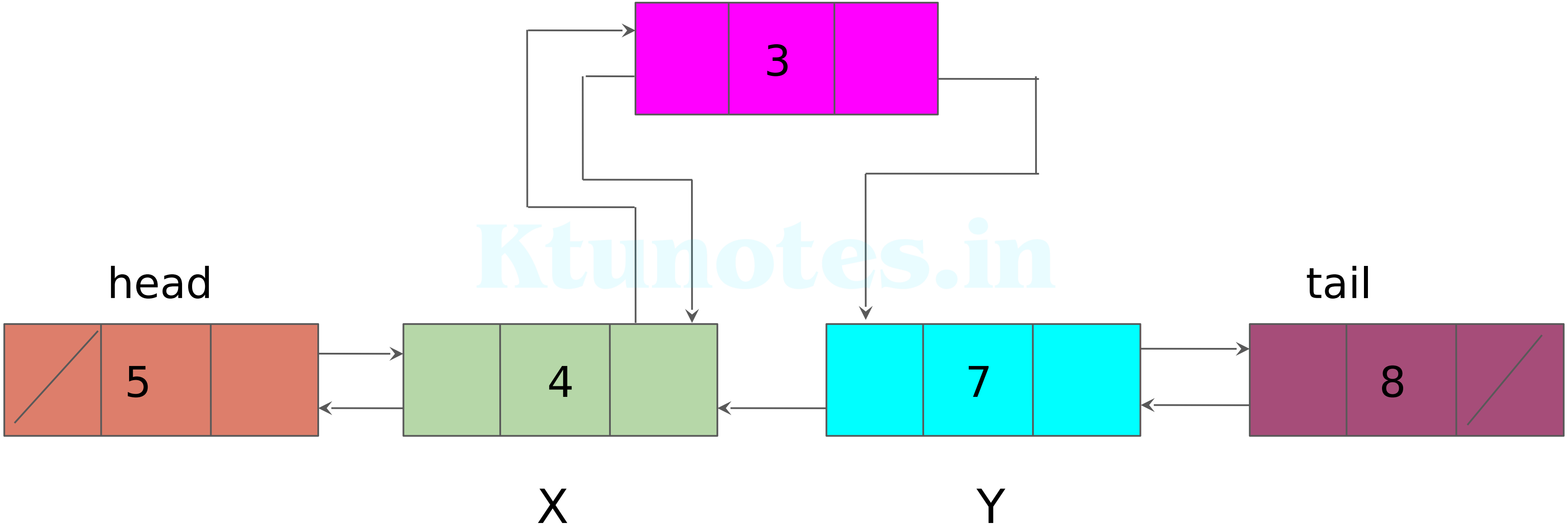
Point next of X to new node

Insertion between node X and Y



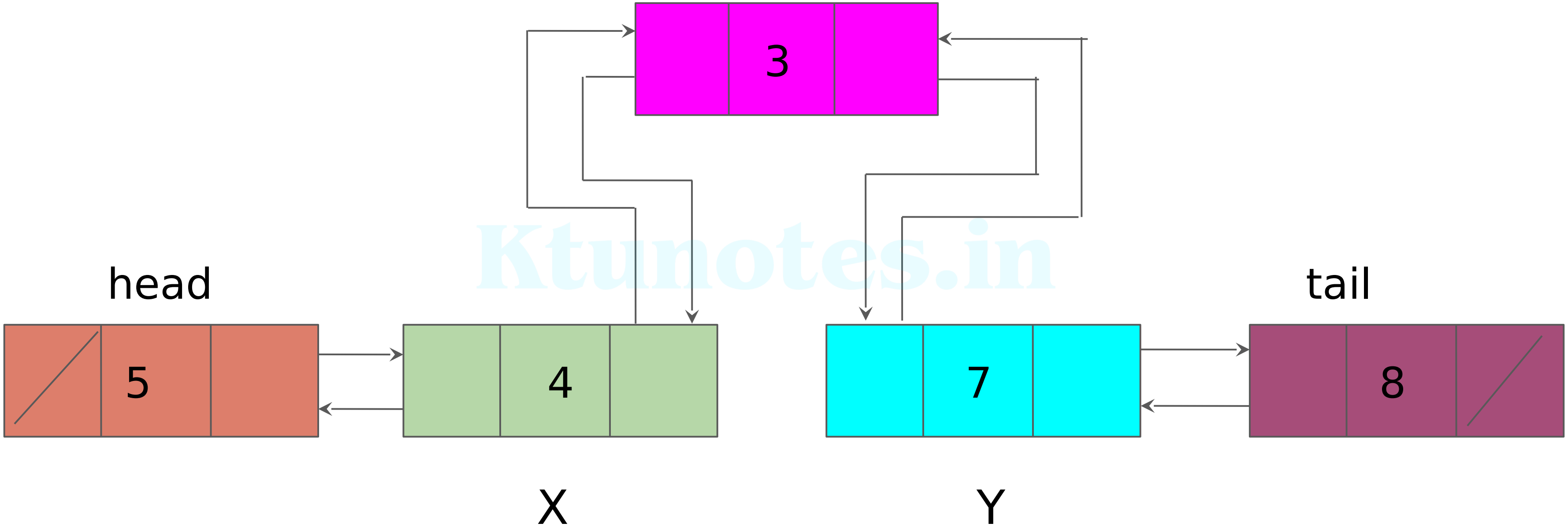
Point prev of new node to X

Insertion between node X and Y



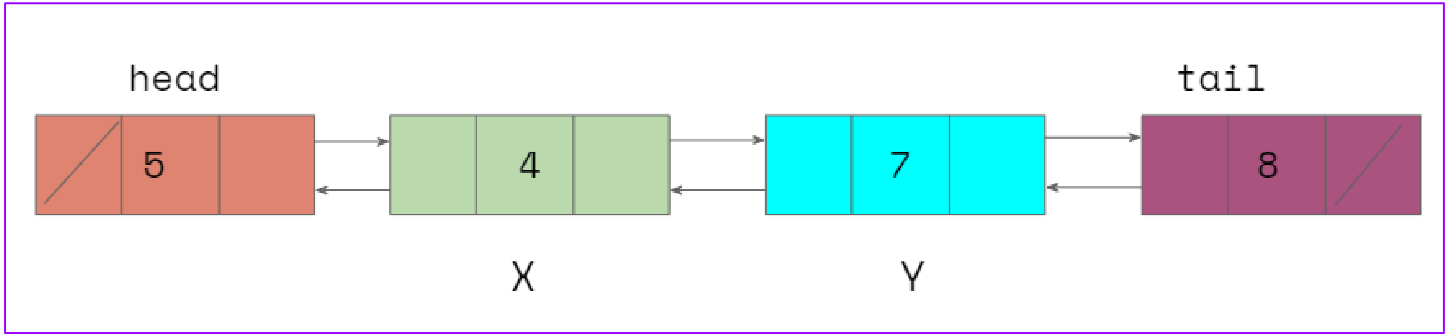
Point next of new node to Y

Insertion between node X and Y



Point prev of Y to new node.

Representation of Doubly Linked List in Memory

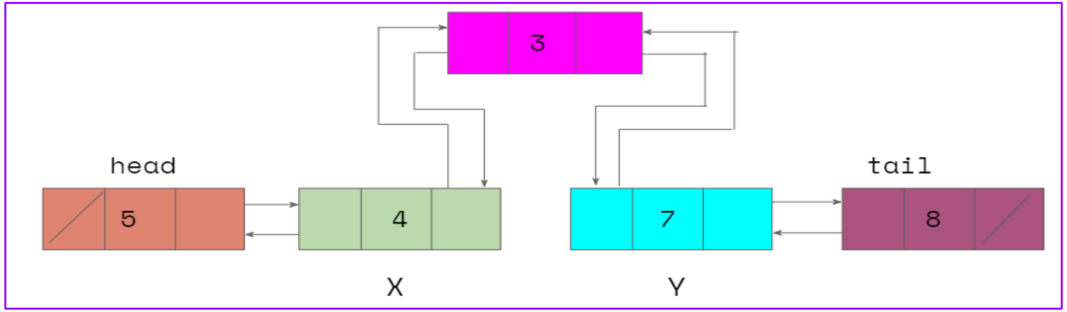


NULL	5	2020			2000	4	4004	
2000	2004	2008	2012	2016	2020	2024	2028	...

	2020	7	4024			4004	8	NULL
...	4004	4008	4012	4016	4020	4024	4028	4032

...	7200	7204	7208	7212	7216	7220	7224	7228

Representation of Doubly Linked List in Memory



NULL	5	2020			2000	4	7208	
2000	2004	2008	2012	2016	2020	2024	2028	...

	7208	7	4024			4004	8	NULL
...	4004	4008	4012	4016	4020	4024	4028	4032

			2020	3	4004			
...	7200	7204	7208	7212	7216	7220	7224	7228

Basic Operations: Deletion at the Beginning

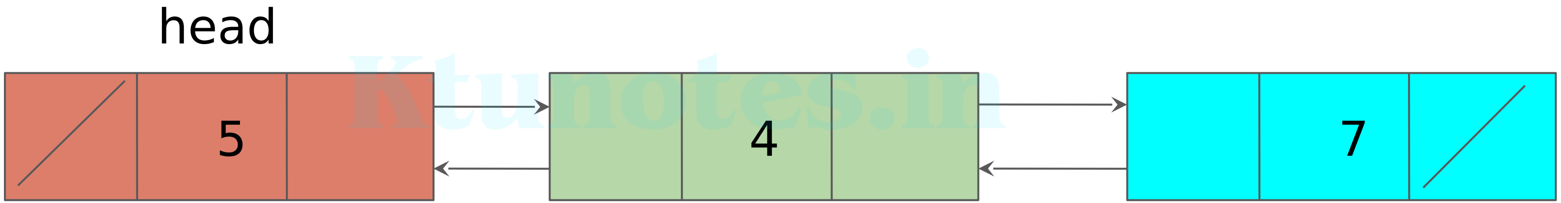
Get the node pointed by head as *temp*

Point head to *temp’s* next

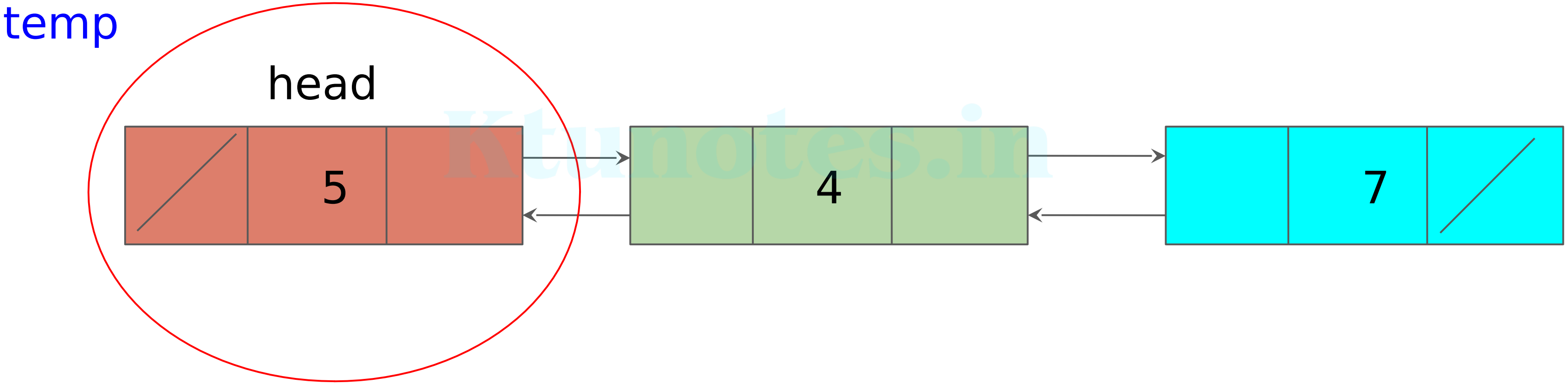
Free the memory used by node *temp*

Set prev of head to NULL

Deletion at the Beginning

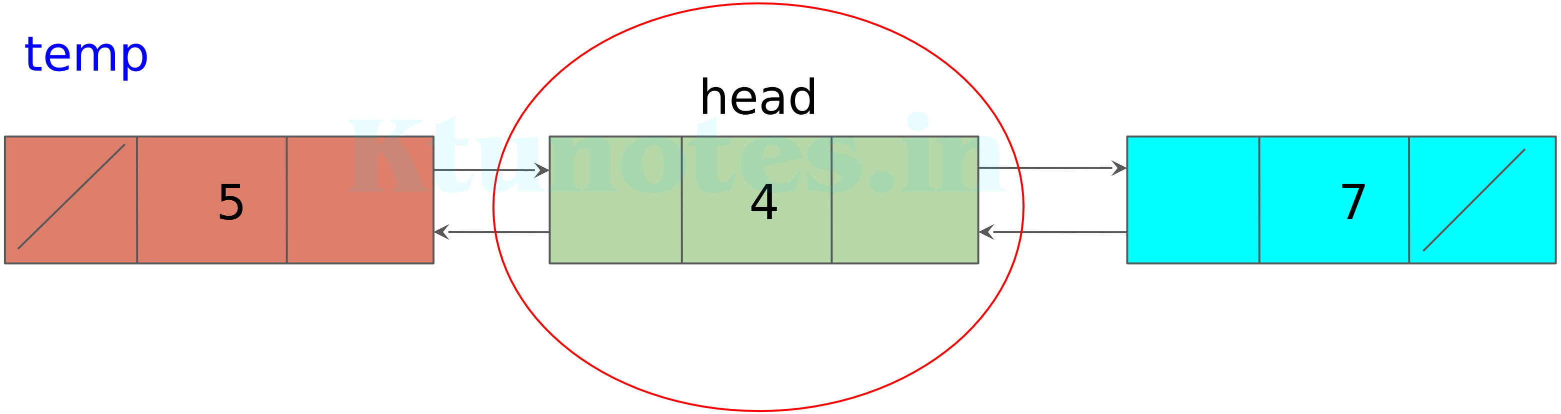


Deletion at the Beginning



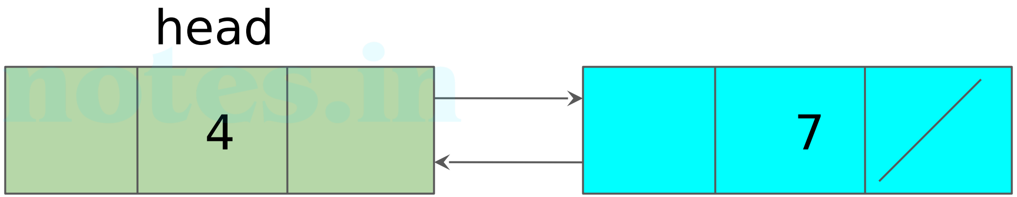
Get the node pointed by head as *temp*

Deletion at the Beginning



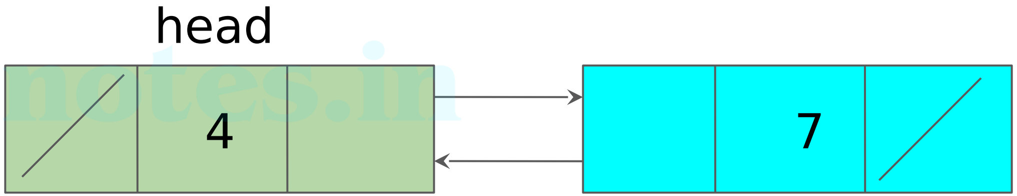
Point head to *temp's* next

Deletion at the Beginning



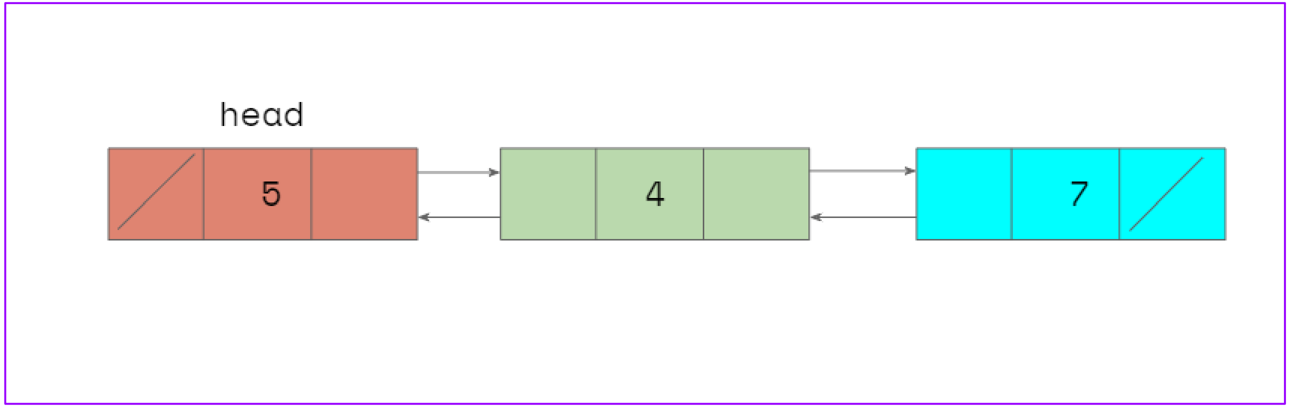
Free the memory used by node *temp*

Deletion at the Beginning



Set prev of head to NULL

Representation of Doubly Linked List in Memory

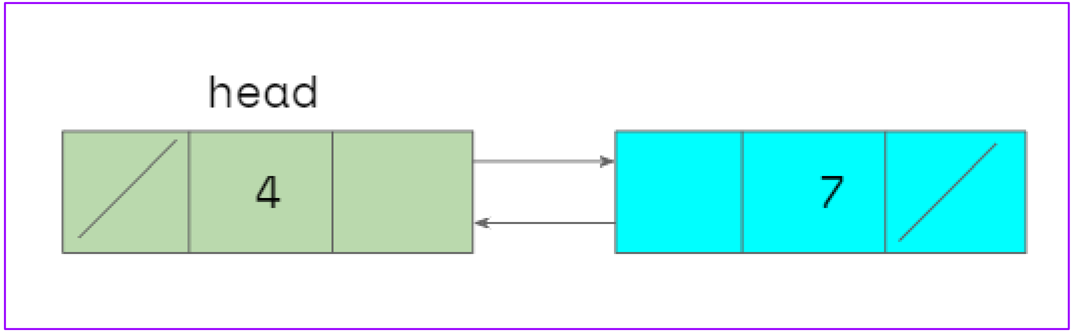


NULL	5	2020			2000	4	4012	
2000	2004	2008	2012	2016	2020	2024	2028	...

MAIN MEMORY

			2020	7	NULL			
...	4004	4008	4012	4016	4020	4024	4028	4032

Representation of Doubly Linked List in Memory



▨ *Freed Memory*

NULL	*5*	*2020*			NULL	4	4012	
2000	2004	2008	2012	2016	2020	2024	2028	...

MAIN MEMORY

			2020	7	NULL			
...	4004	4008	4012	4016	4020	4024	4028	4032

Basic Operations: Deletion at the End

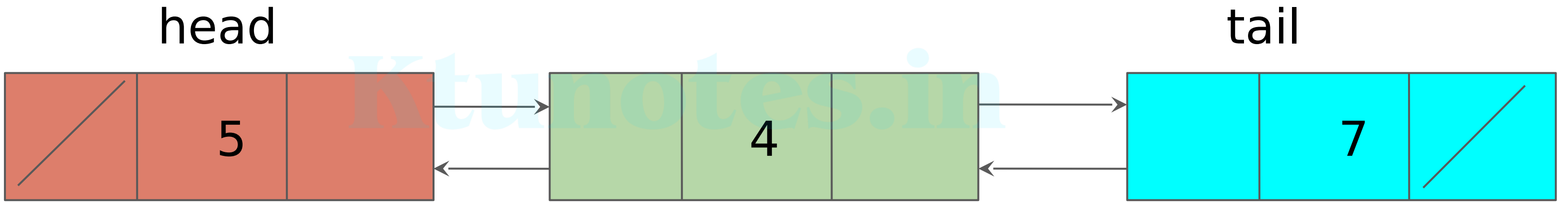
Get the node pointed by tail as *temp*

Point tail to *temp's* previous

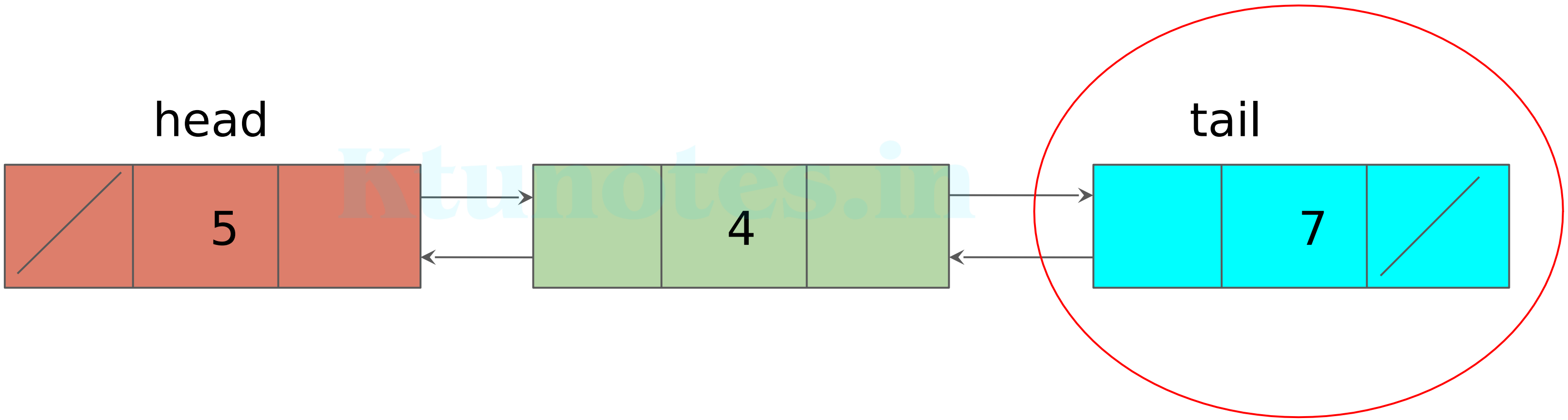
Set next of tail to NULL

Free the memory used by node *temp*

Deletion at the End

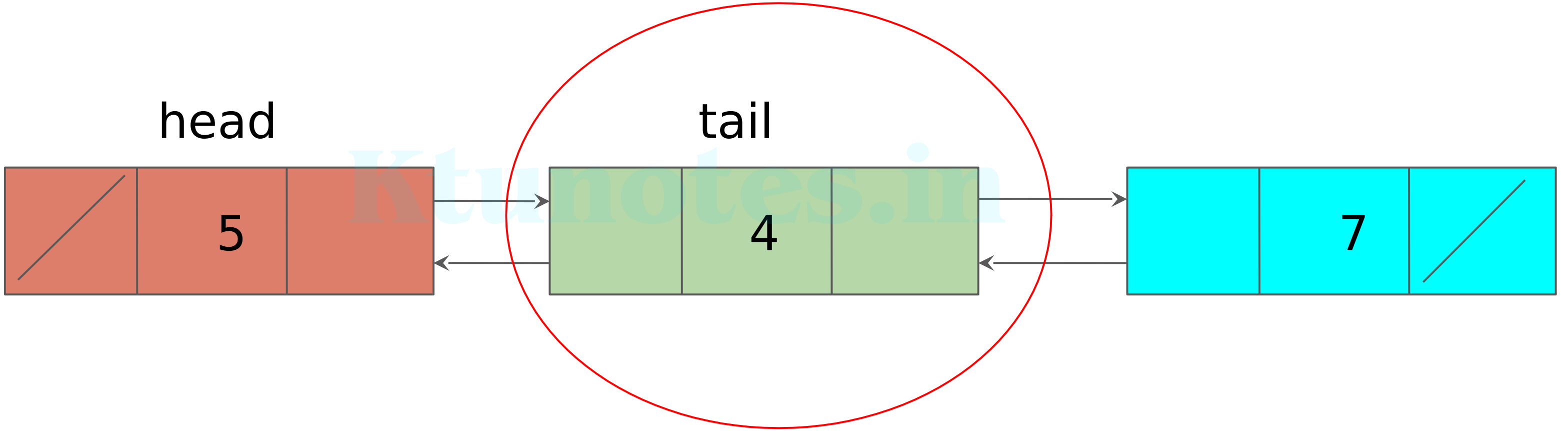


Deletion at the End



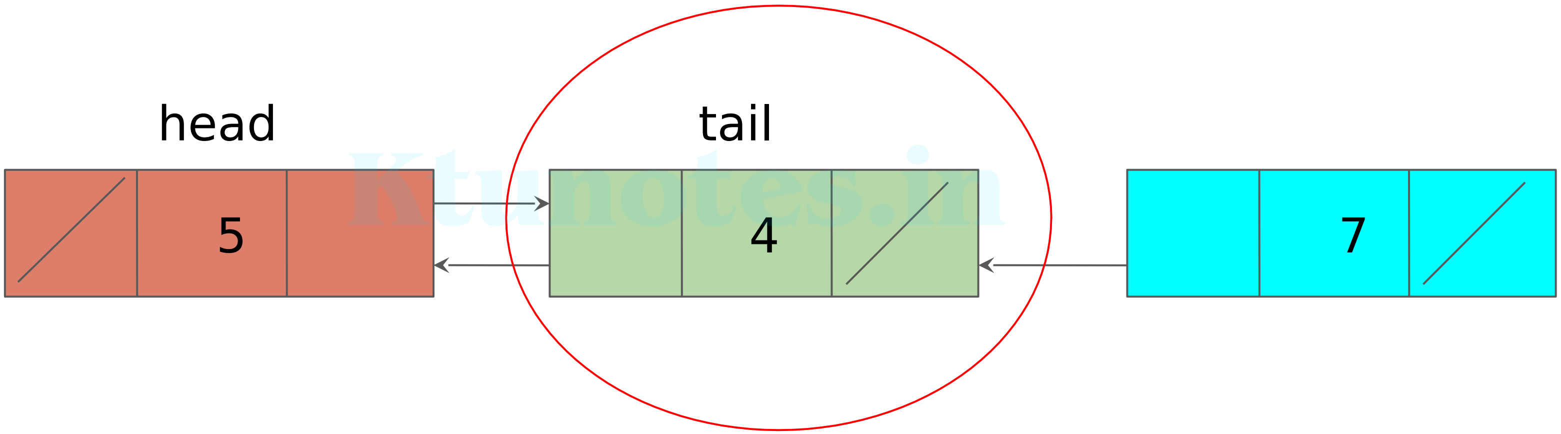
Get the node pointed by tail as *temp*

Deletion at the End



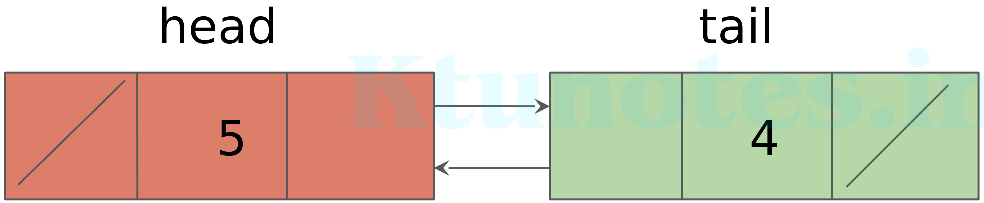
Point tail to *temp’s* previous

Deletion at the End



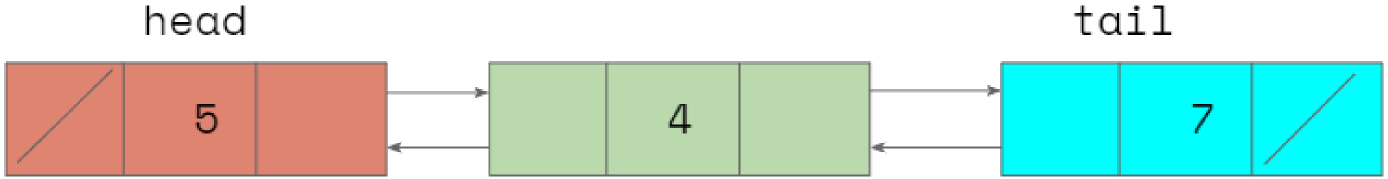
Set next of tail to NULL

Deletion at the End



Free the memory used by node *temp*

Representation of Doubly Linked List in Memory

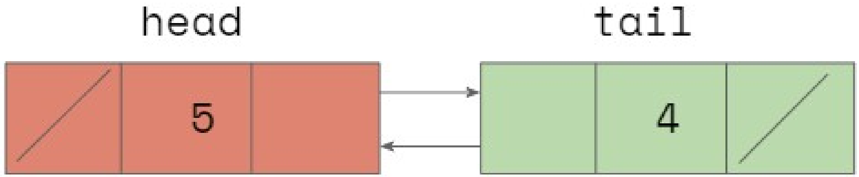


NULL	5	2020			2000	4	4012	
2000	2004	2008	2012	2016	2020	2024	2028	...

MAIN MEMORY

			2020	7	NULL			
...	4004	4008	4012	4016	4020	4024	4028	4032

Representation of Doubly Linked List in Memory



⬜ *Freed Memory*

NULL	5	2020			2000	4	NULL	
2000	2004	2008	2012	2016	2020	2024	2028	...

MAIN MEMORY

			2020	*7*	*NULL*			
...	4004	4008	4012	4016	4020	4024	4028	4032

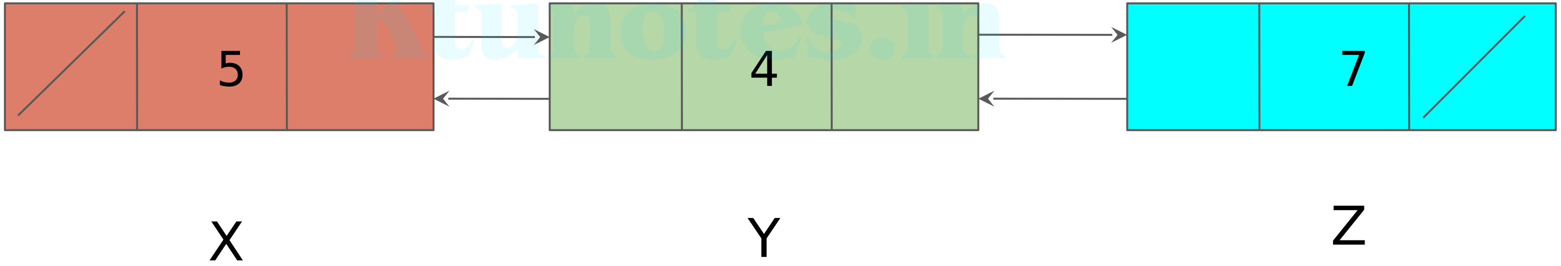
Basic Operations: Deletion of a node Y between node X and node Z

Point next of X to Z

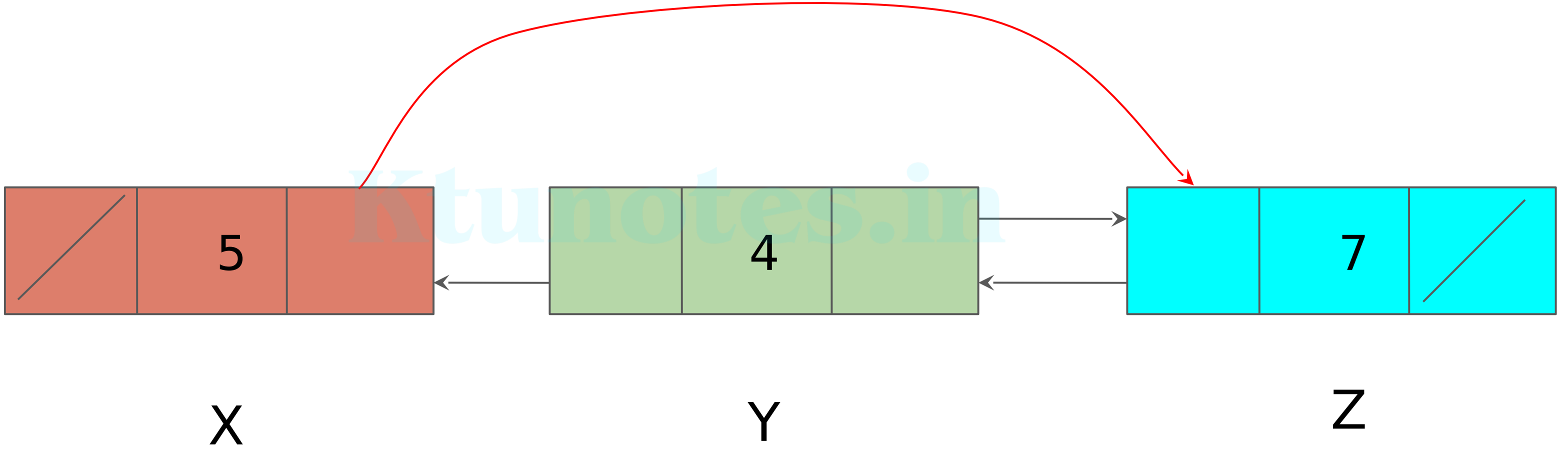
Point prev of Z to X

Free up the memory space used by node Y

Basic Operations: Deletion of a node Y between node X and node Z

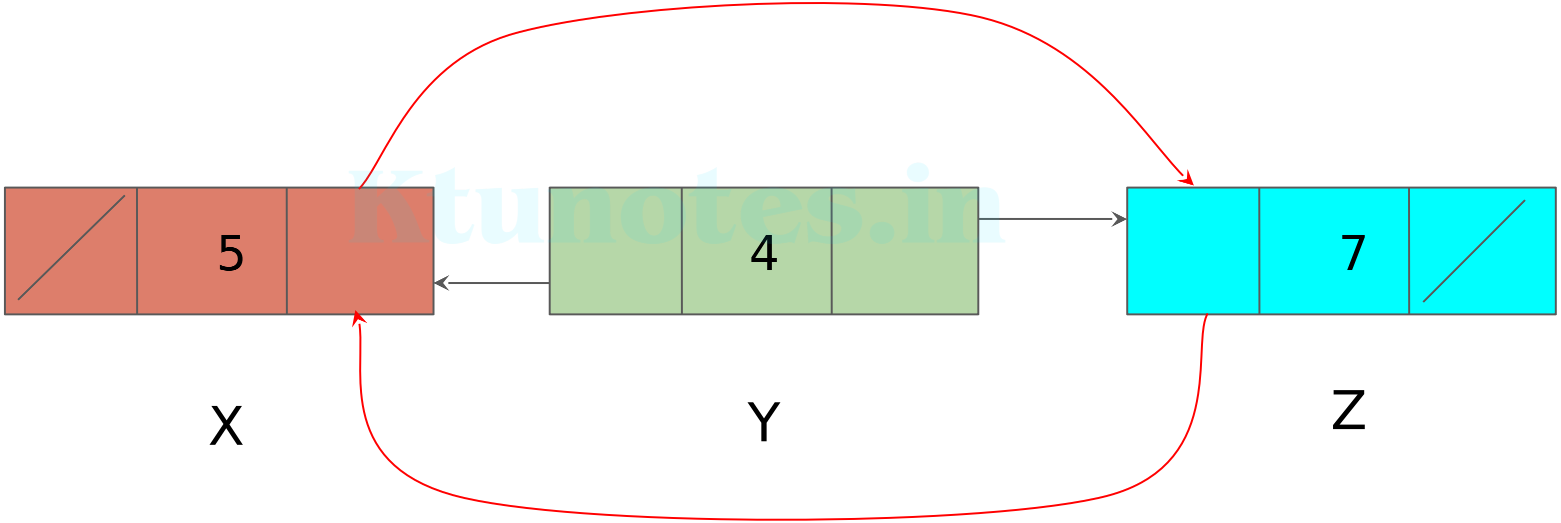


Basic Operations: Deletion of a node Y between node X and node Z



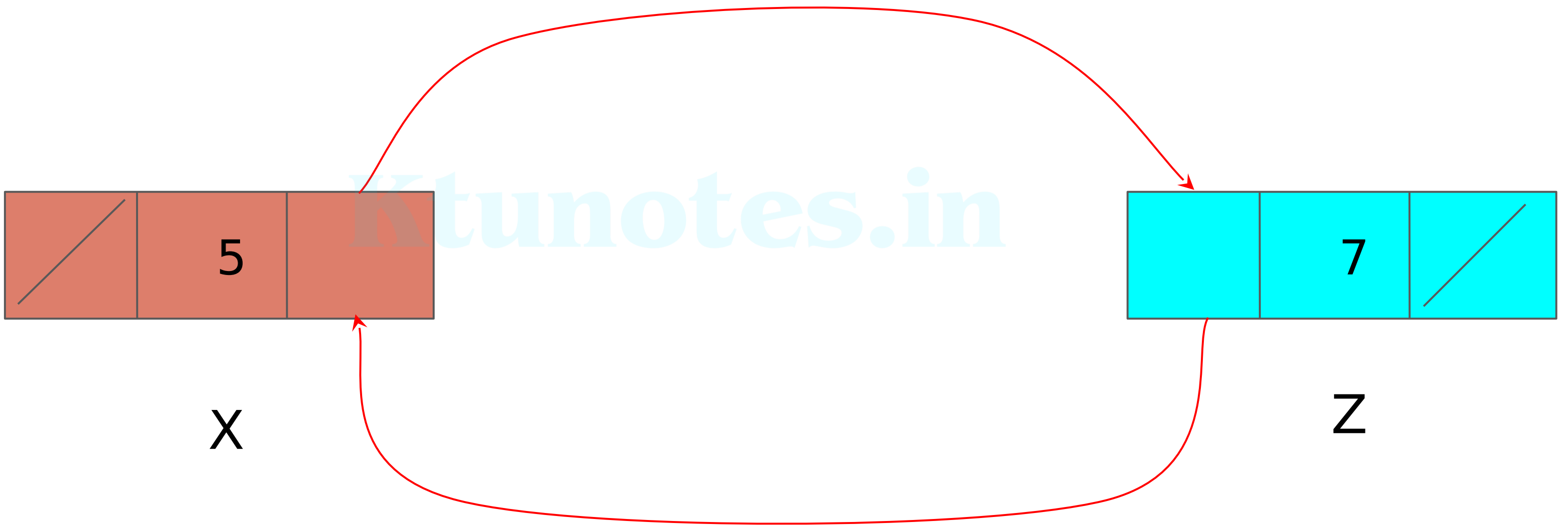
Point next of X to Z

Basic Operations: Deletion of a node Y between node X and node Z



Point prev of Z to X

Basic Operations: Deletion of a node Y between node X and node Z



Free up the memory space used by node Y

Traversal through a DLL

Forward Traversal

Follows next pointer

Backward Traversal

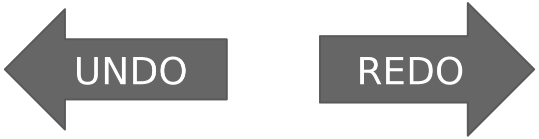
Follows prev pointer

Applications of DLL

Doubly linked list can be used in navigation systems where both front and back navigation is required.

It is used by browsers to implement backward and forward navigation of visited web pages

It is also used by various **application** to implement Undo and Redo functionality.



UNDO

REDO

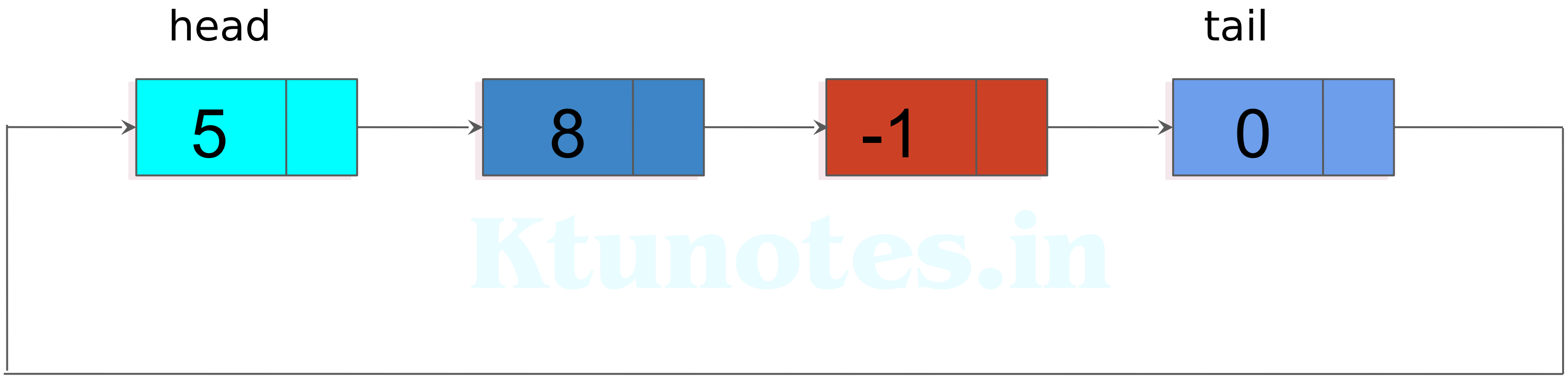
Circular Linked Lists

Circular LL

Variation of Linked list in which the first element points to the last element and the last element points to the first element.

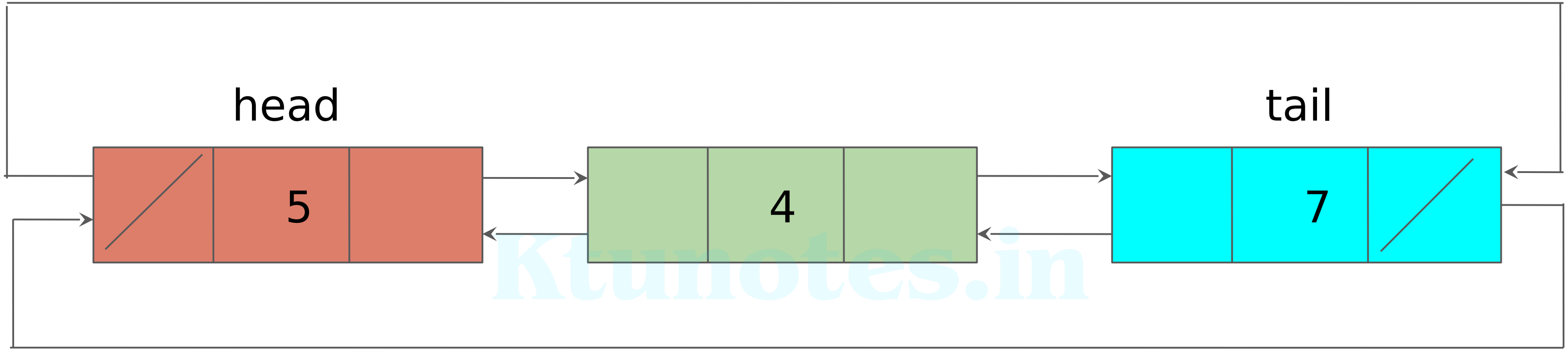
Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

SLL as a Circular LL



The next pointer of tail is connected to the head

DLL as a Circular LL



The next pointer of tail is connected to the head

The prev pointer of head is connected to the tail

DLL as a Circular LL

Operations

Basic Operations: Insertion at the Beginning

Create a new node with given data.

Point new node’s next to head

Point prev of head to new node

Make prev of new node to tail

Point head to the new node.

Basic Operations: Insertion at the End

Create a new node with given data.

Point next of tail to new node

Point prev of new node to tail

Make next of new node to head

Point tail to the new node.

Basic Operations: Deletion at the Beginning

Get the node pointed by head as *temp*

Point head to *temp's* next

Free the memory used by node *temp*

Set prev of head to tail

Basic Operations: Deletion at the End

Get the node pointed by tail as *temp*

Point tail to *temp's* previous

Set next of tail to head

Free the memory used by node *temp*

Deletion/Insertion After Node X

Same as Doubly Linked List

Implementing Stack

Using Linked List

Stack Implementation

A stack can be implemented using a singly linked list or a doubly linked list.

Insertion (Push Operation) from the BEGINNING

&

Deletion (Pop Operation) from the BEGINNING

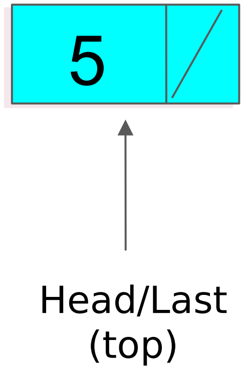
OR

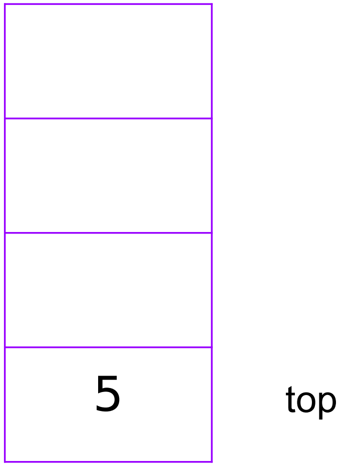
Insertion (Push Operation) from the END

&

Deletion (Pop Operation) from the END

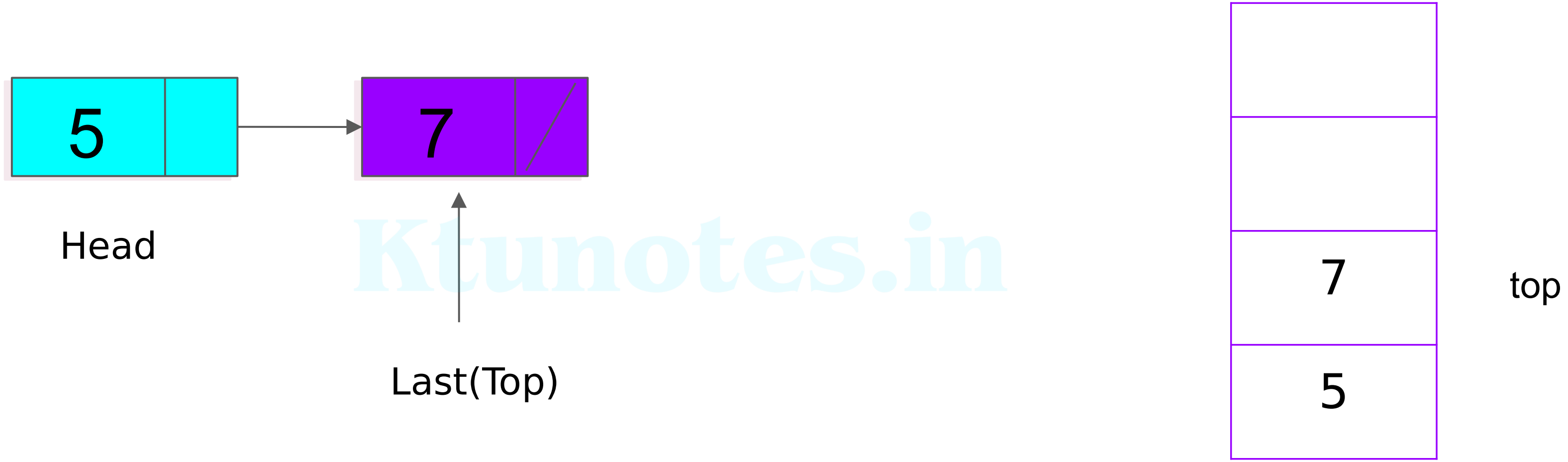
Push Operation(Insertion)





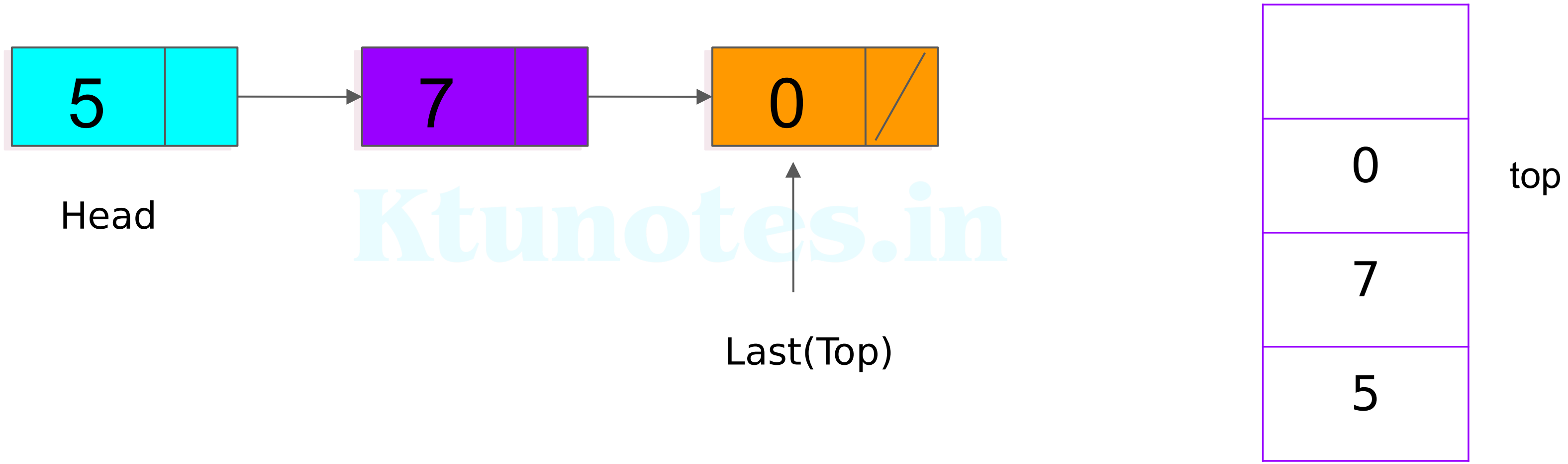
Pushing the First Element to the Stack

Push Operation(Insertion)



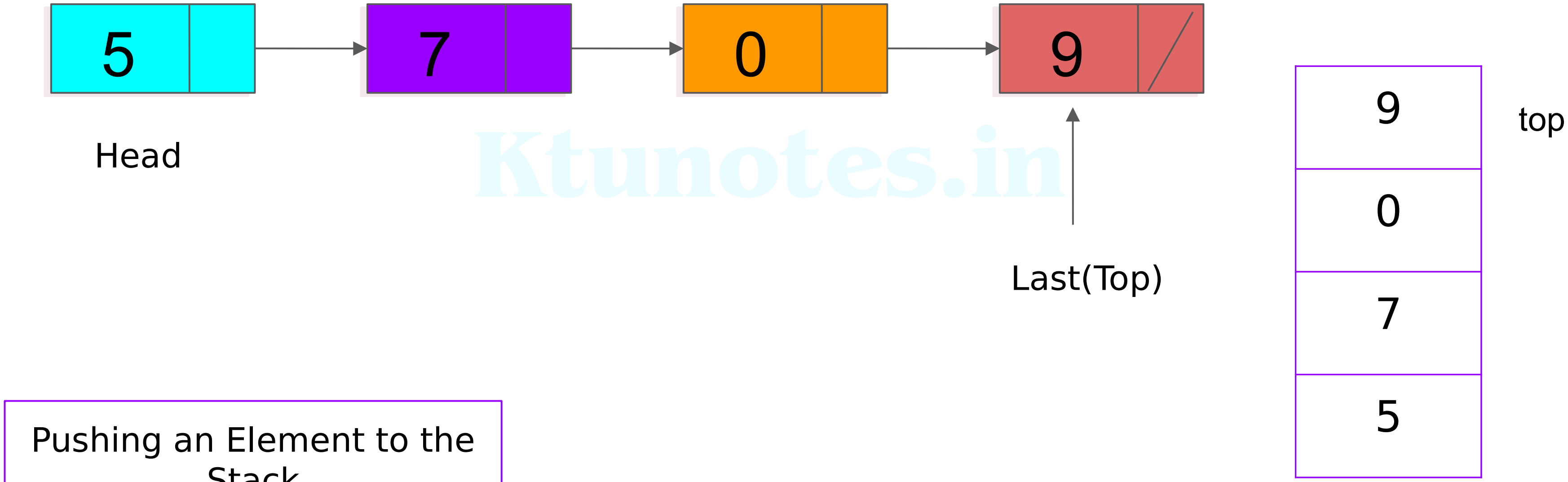
Pushing an Element to the Stack

Push Operation(Insertion)



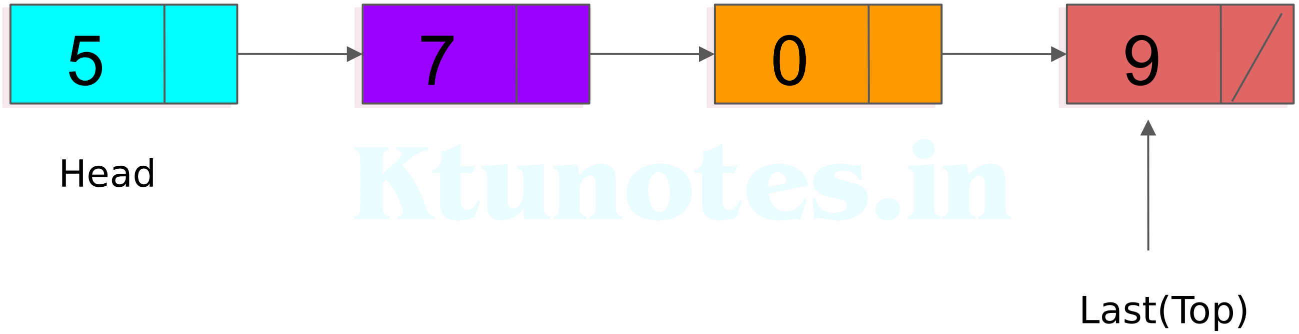
Pushing an Element to the Stack

Push Operation(Insertion)



Pushing an Element to the Stack

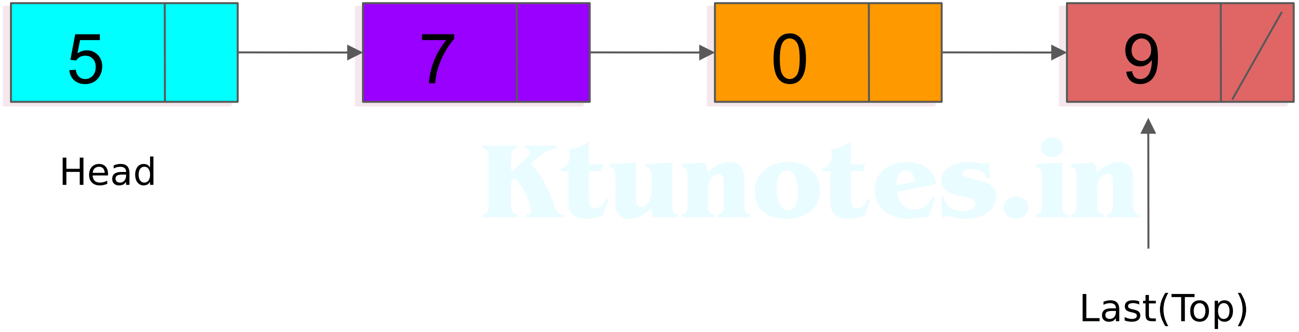
Push Operation(Insertion)

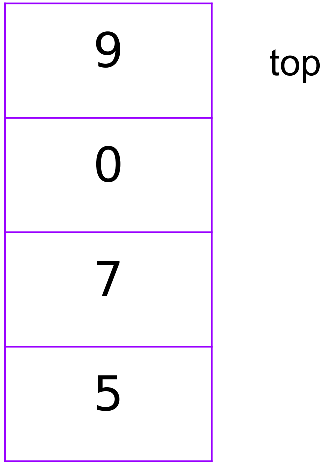


Stack Full Condition

When no:of elements is Equal to SIZE, stack becomes Full

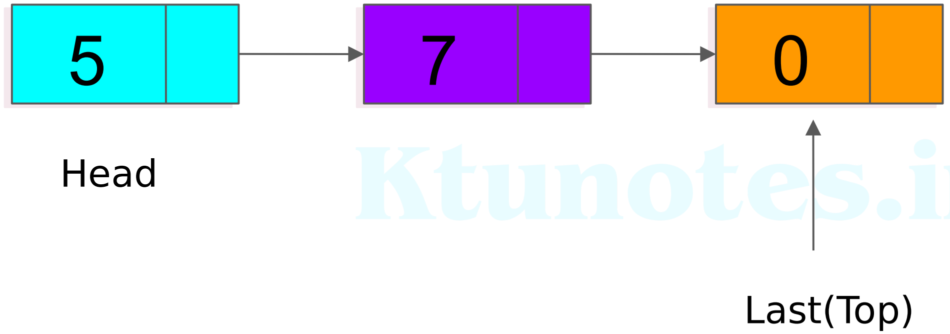
Pop Operation(Deletion)

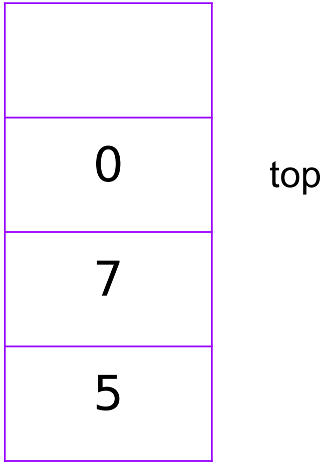




Pop an Element

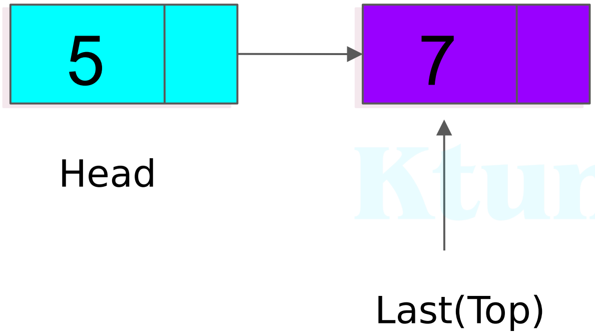
Pop Operation(Deletion)

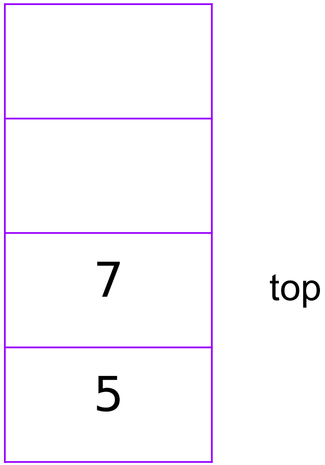




Pop an Element

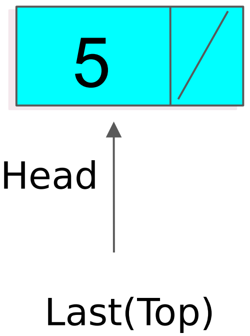
Pop Operation(Deletion)





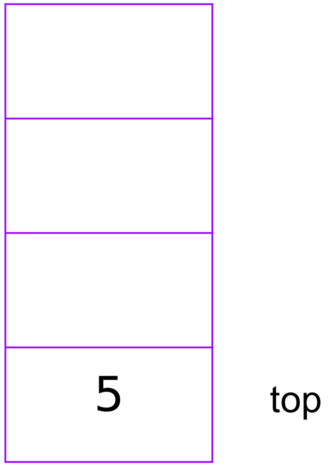
Pop an Element

Pop Operation(Deletion)



While deleting last element,

set top=NULL



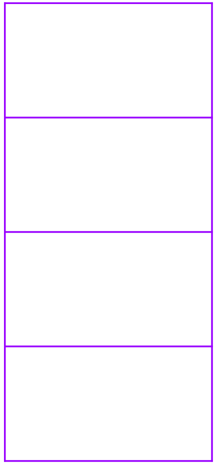
Pop an Element

Pop Operation(Deletion)

Stack Empty

Condition

Check if top==NULL



Implementing Queue

Using Linked List

Queue Implementation

A queue can be implemented using a singly linked list or a doubly linked list.

Insertion (Enqueue Operation) from the BEGINNING

&

Deletion (Dequeue Operation) from the END

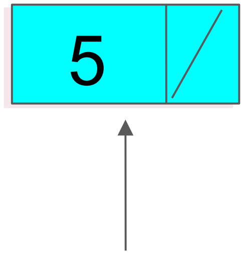
OR

Insertion (Enqueue Operation) from the END

&

Deletion (Dequeue Operation) from the BEGINNING

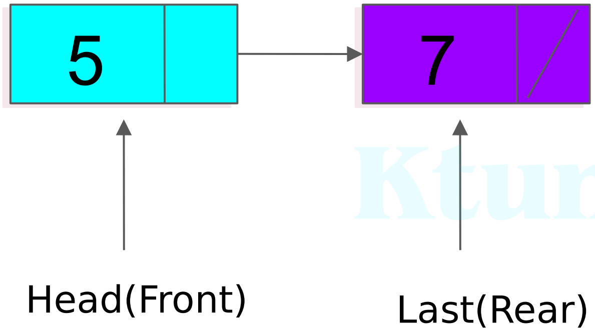
Enqueue Operation(Insertion)



Head/Last
(Front/Rear)

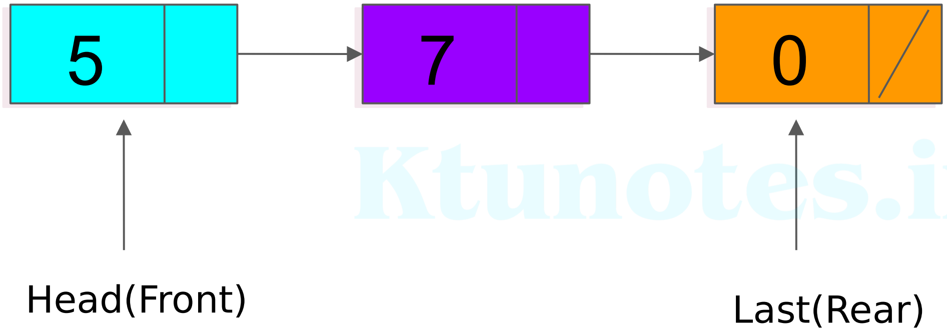
Inserting an Element to the Queue

Enqueue Operation(Insertion)



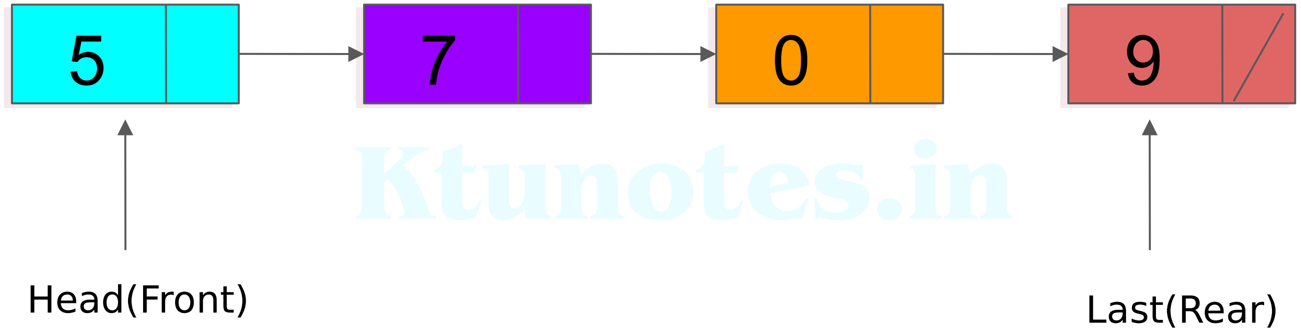
Inserting an Element to the Queue

Enqueue Operation(Insertion)



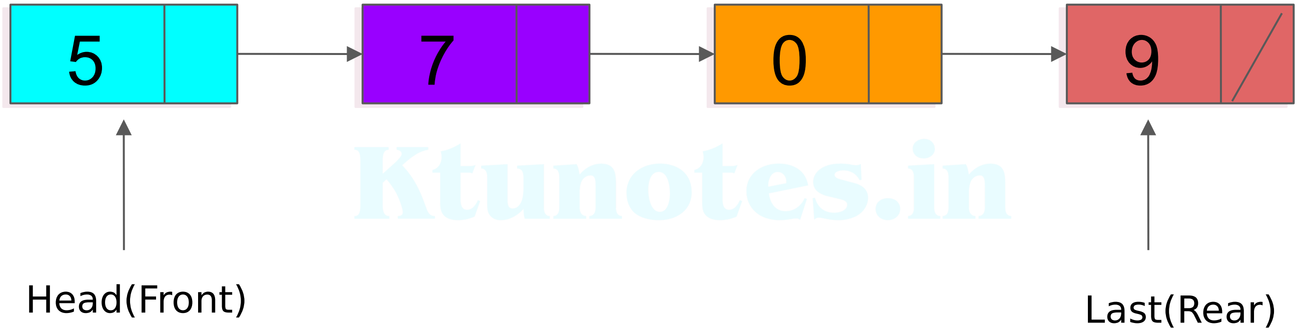
Inserting an Element to the Queue

Enqueue Operation(Insertion)



Inserting an Element to the Queue

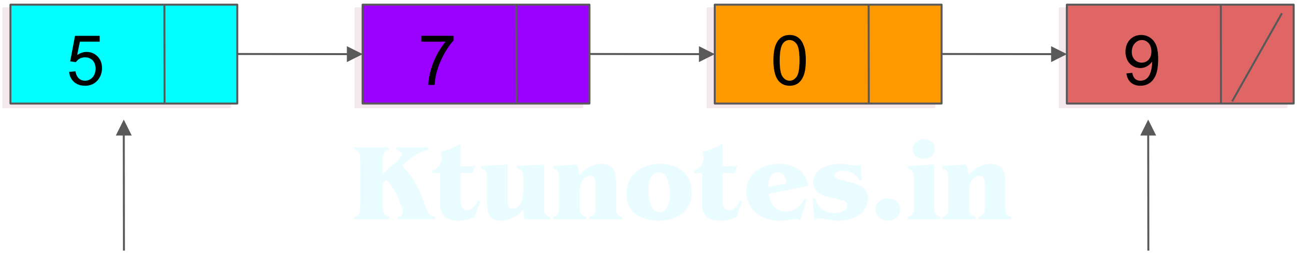
Enqueue Operation(Insertion)



Queue Full Condition

When no:of elements is SIZE, queue becomes Full

Dequeue Operation(Deletion)

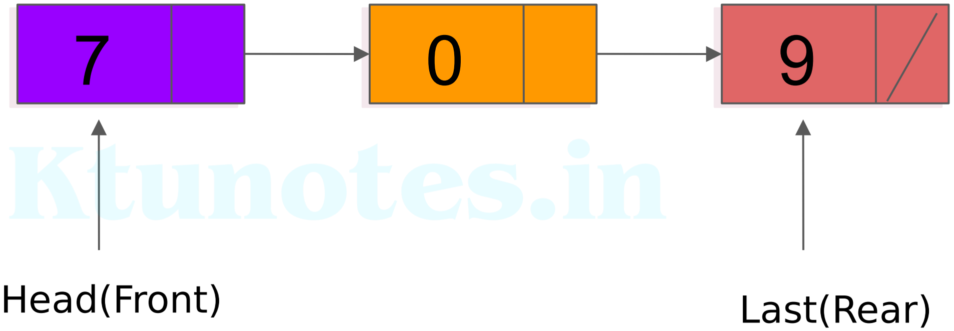


Head(Front)

Last(Rear)

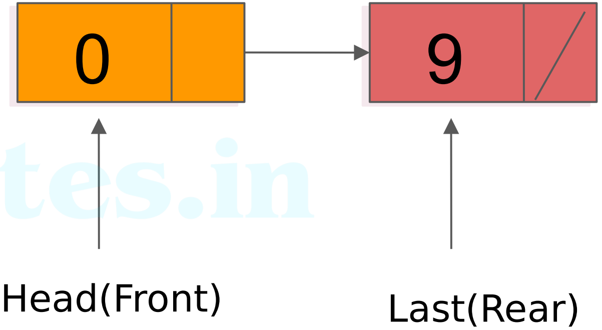
Deleting an Element from the Queue

Dequeue Operation(Deletion)



Deleting an Element from the Queue

Dequeue Operation(Deletion)

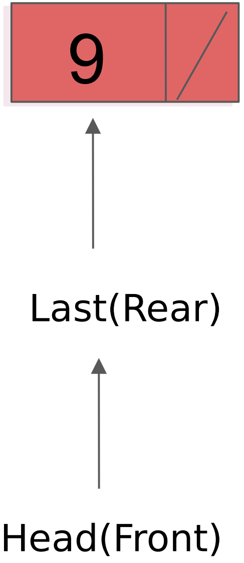


Deleting an Element from the Queue

Dequeue Operation(Deletion)

When front=rear, Queue contain only one element

Set front=rear=NULL, while deleting last element



Deleting an Element from the Queue

Pop Operation(Deletion)

Queue Empty

Condition

Check if front=rear=NULL

Memory Allocation Strategies

Memory Allocation

For visualization purpose, memory can be viewed as a single array of variable sized blocks.

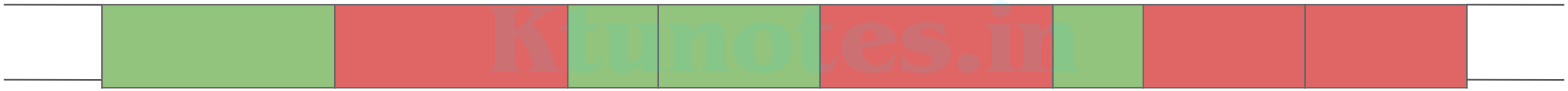
Some of the blocks are ***free blocks*** and some are ***reserved blocks*** or already allocated.

The free blocks are linked together to form a ***freelist*** used for servicing future memory requests.

Click for Web Reference

Dynamic Memory Allocation

Memory is made up of a series of variable-size blocks, some allocated and some free.



🟥 Memory currently allocated

🟩 Unused memory, for future allocation

Memory Managers

Memory Managers receive memory requests.

They should find some block on the freelist that is large enough to service the request.

If no such block is found, then the memory manager must resort to a failure policy such as garbage collection.

Fragmentation

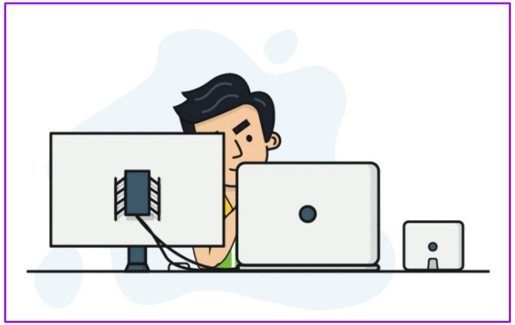
External fragmentation happens when a series of memory requests which results in lots of small sized memory blocks, none of which is useful for servicing memory requests.

Internal fragmentation occurs when more than m words are allocated to a request for m words, wasting free storage.

Demonstrating External Fragmentation

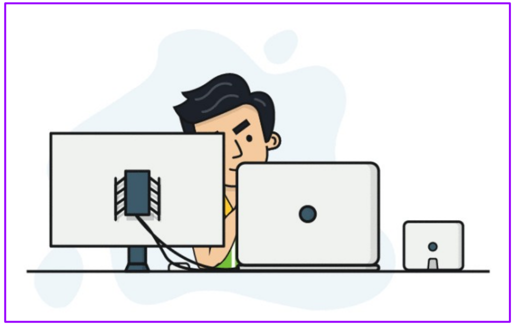
Allocate me 300 bytes of memory

Demonstrating External Fragmentation



300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Demonstrating External Fragmentation

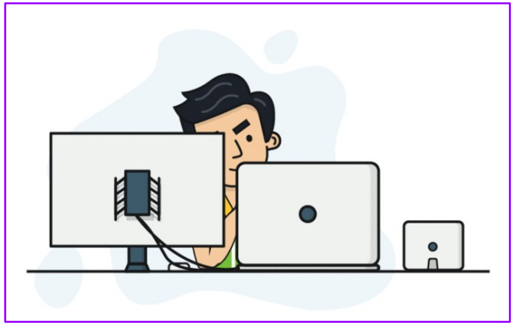


🟥 300 Bytes	🟩 300 Bytes	🟩 500 Bytes	🟩 600 Bytes	🟩 200 Bytes	🟩 200 Bytes

Demonstrating External Fragmentation

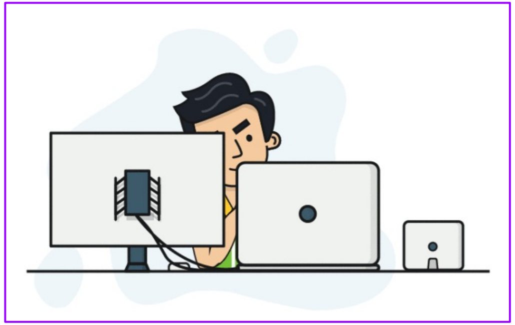
Allocate me 570 bytes of memory

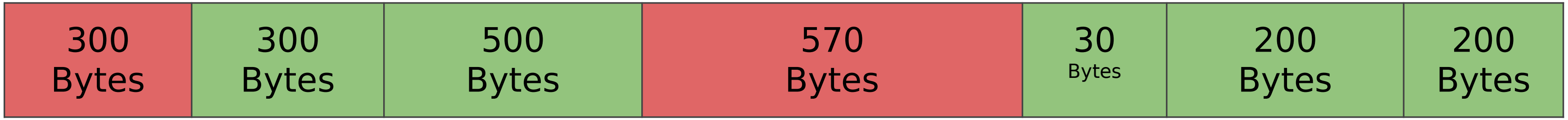
Demonstrating External Fragmentation



300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Demonstrating External Fragmentation



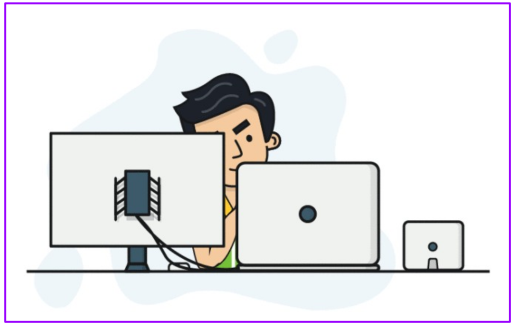


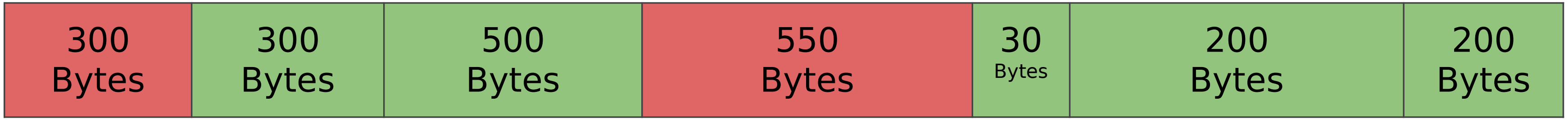
300 Bytes	300 Bytes	500 Bytes	570 Bytes	30 Bytes	200 Bytes	200 Bytes

Demonstrating External Fragmentation

Allocate me 180 bytes of memory

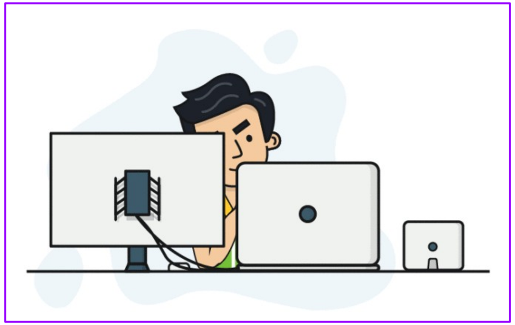
Demonstrating External Fragmentation





🟥 300 Bytes	🟩 300 Bytes	🟩 500 Bytes	🟥 550 Bytes	🟩 30 Bytes	🟩 200 Bytes	🟩 200 Bytes

Demonstrating External Fragmentation



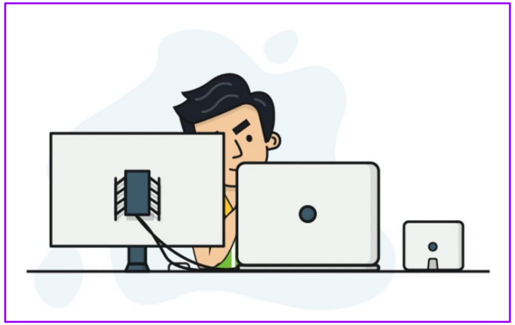
Small block: External Fragmentation
(Too small to satisfy memory requests)

🟥 300 Bytes	🟩 300 Bytes	🟩 500 Bytes	🟥 570 Bytes	🟩 30 Bytes	🟥 180 Bytes	🟩 20 Bytes	🟩 200 Bytes

Demonstrating Internal Fragmentation

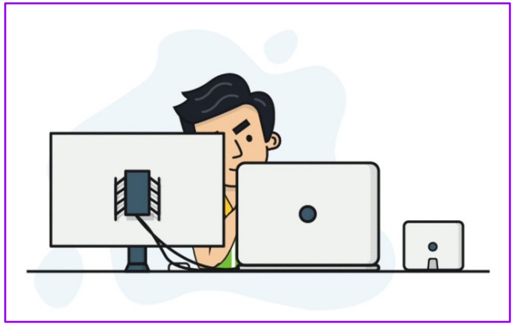
Allocate me 600 bytes of memory

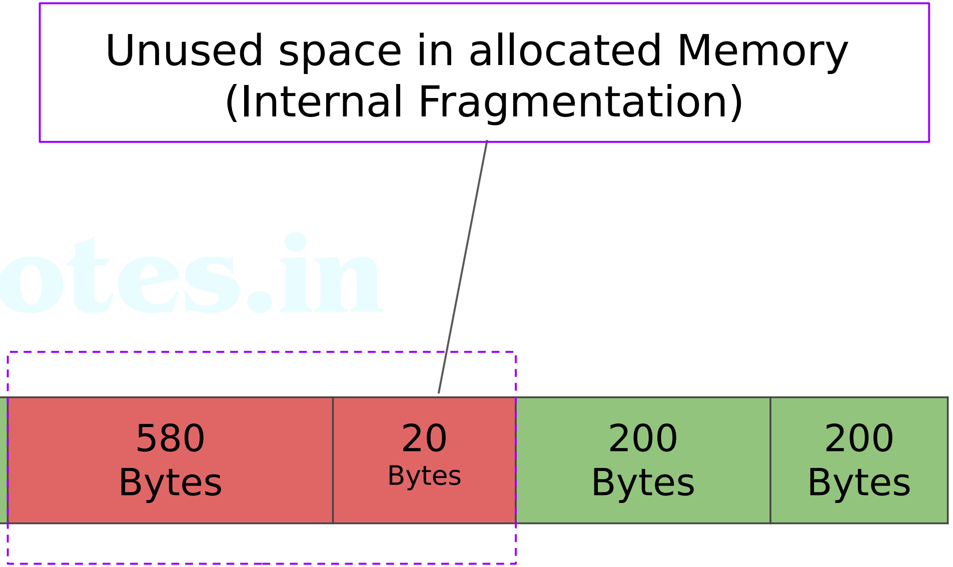
Demonstrating Internal Fragmentation



300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Demonstrating Internal Fragmentation





Unused space in allocated Memory (Internal Fragmentation)

300 Bytes	300 Bytes	500 Bytes	580 Bytes	20 Bytes	200 Bytes	200 Bytes

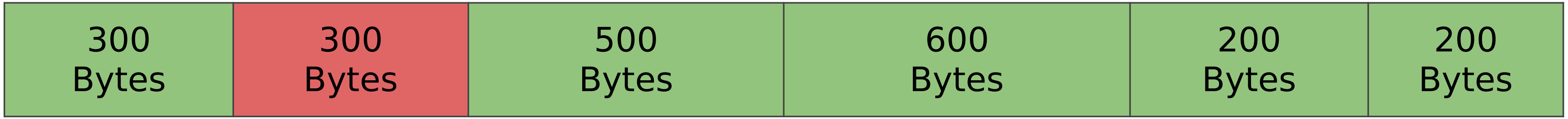
First Fit Strategy

Move down the free block list until a block of size at least greater than the requested size is found.

Any remaining space in this block is left on the freelist.

Click for Web Reference

First Fit-Demonstration



300 Bytes

300 Bytes

500 Bytes

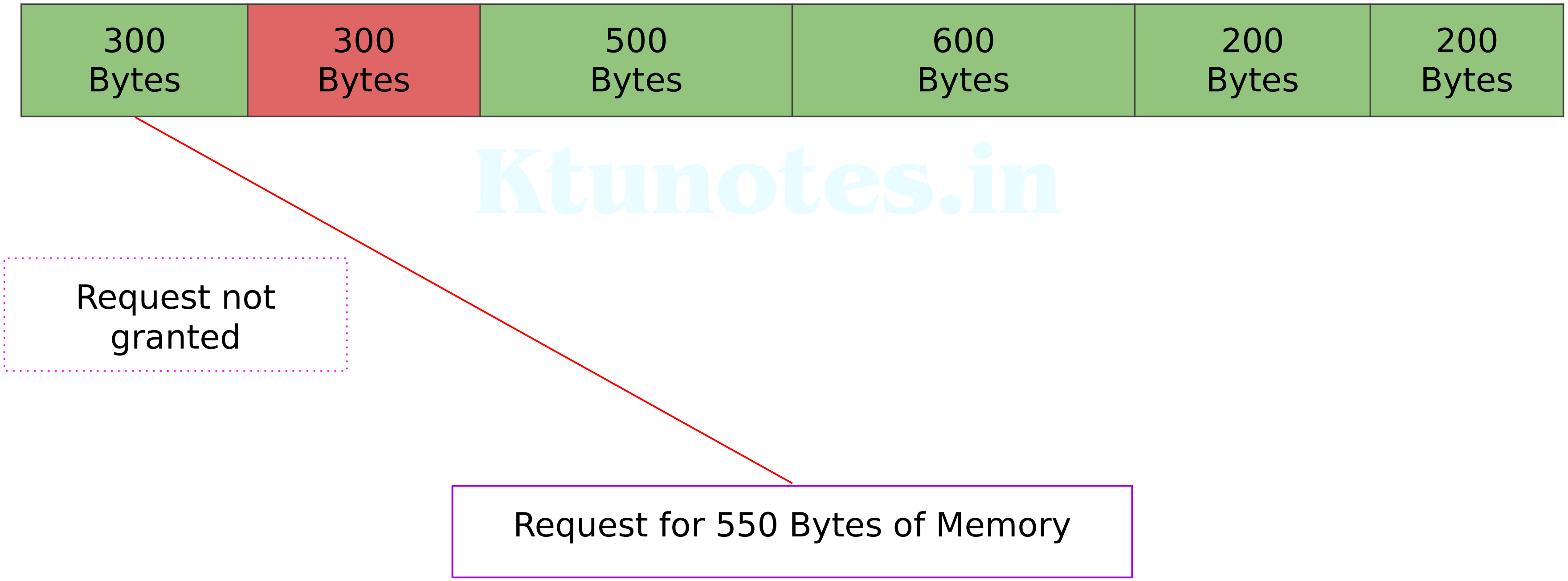
600 Bytes

200 Bytes

200 Bytes

Request for 550 Bytes of Memory

First Fit-Demonstration

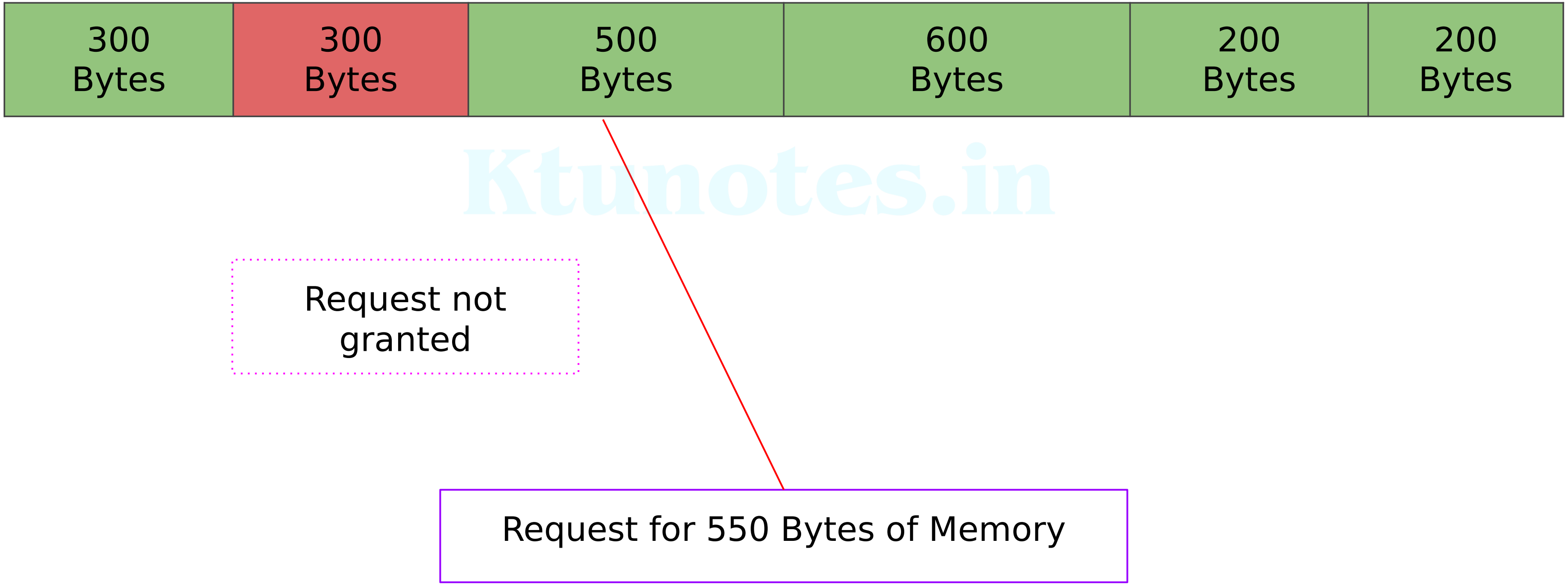


300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Request not granted

Request for 550 Bytes of Memory

First Fit-Demonstration



300 Bytes

🟥 300 Bytes

500 Bytes

600 Bytes

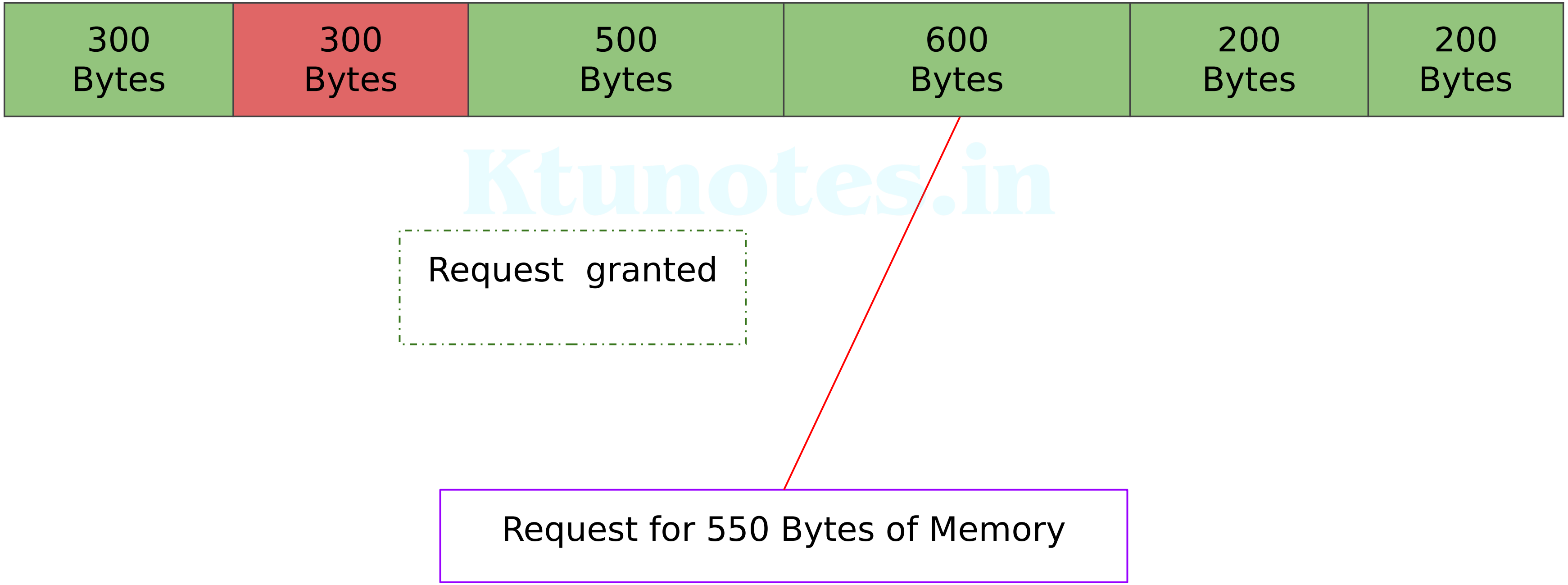
200 Bytes

200 Bytes

Request not granted

Request for 550 Bytes of Memory

First Fit-Demonstration

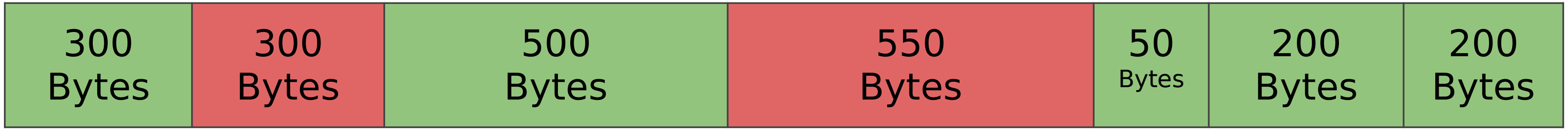


300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Request granted

Request for 550 Bytes of Memory

First Fit-Demonstration



300 Bytes	300 Bytes	500 Bytes	550 Bytes	50 Bytes	200 Bytes	200 Bytes

Request for 550 Bytes of Memory

First Fit-Pros & Cons

👍 As the processor allocates the nearest available memory partition to the job, it is very fast in execution.

👎 Wastage of Memory as large blocks of memory may be allocated to serve memory requests with low storage requirements

Best Fit Strategy

Best fit looks at the entire list and picks the smallest block that is at least as large as the request.

Provides the "best" or closest fit to the request.

Click for Web Reference

Best Fit-Implementation

A	B	C	D	E	F
300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Block	Free Space(After Allotment of Requested Block)
A	120 Bytes
B	ALREADY FULL
C	320 Bytes
D	420 Bytes
E	20 Bytes
F	ALREADY FULL

Request for 180 Bytes of Memory

Best Fit-Implementation

A	B	C	D	E	F
300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Current Winner

BLOCK A

Block	Free Space(After Allotment of Requested Block)
A	120 Bytes
B	ALREADY FULL
C	320 Bytes
D	420 Bytes
E	20 Bytes
F	ALREADY FULL

Request for 180 Bytes of Memory

Best Fit-Implementation

A	B	C	D	E	F
🟩 300 Bytes	🟥 300 Bytes	🟩 500 Bytes	🟩 600 Bytes	🟩 200 Bytes	🟥 200 Bytes

Current Winner

BLOCK A

Block	Free Space(After Allotment of Requested Block)
A	120 Bytes
B	ALREADY FULL
→ C	320 Bytes
D	420 Bytes
E	20 Bytes
F	ALREADY FULL

Request for 180 Bytes of Memory

Best Fit-Implementation

A	B	C	D	E	F
300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Current Winner

BLOCK A

Block	Free Space(After Allotment of Requested Block)
A	120 Bytes
B	ALREADY FULL
C	320 Bytes
D	420 Bytes
E	20 Bytes
F	ALREADY FULL

Request for 180 Bytes of Memory

Best Fit-Implementation

A	B	C	D	E	F
300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

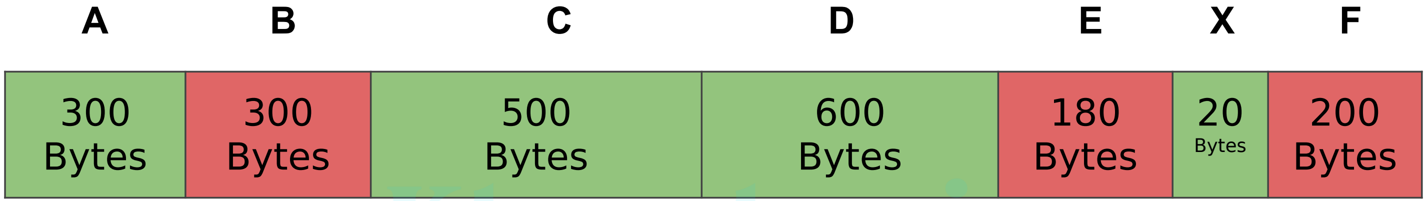
Current Winner

BLOCK E

Block	Free Space(After Allotment of Requested Block)
A	120 Bytes
B	ALREADY FULL
C	320 Bytes
D	420 Bytes
E	20 Bytes
F	ALREADY FULL

Request for 180 Bytes of Memory

Best Fit-Implementation



A	B	C	D	E	X	F
300 Bytes	300 Bytes	500 Bytes	600 Bytes	180 Bytes	20 Bytes	200 Bytes

BLOCK E ALLOCATED

NEW BLOCK (X) FORMED

Best Fit-Pros & Cons

👍 Memory Efficient- allocates the best possible block, thereby reducing memory wastage by external fragmentation

👎 Slow Process - checking the entire memory to find the best possible block is time consuming.

Worst Fit Strategy

Worst fit looks at the entire list and picks the largest block that is available to serve the request.

Provides the "worst" or largest fit to the request.

Click for Web Reference

Worst Fit-Implementation

A	B	C	D	E	F
300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Block	Free Space(After Allotment of Requested Block)
A	120 Bytes
B	ALREADY FULL
C	320 Bytes
D	420 Bytes
E	20 Bytes
F	ALREADY FULL

Request for 180 Bytes of Memory

Worst Fit-Implementation

A	B	C	D	E	F
300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Current Winner

BLOCK A

Block	Free Space(After Allotment of Requested Block)
A	120 Bytes
B	ALREADY FULL
C	320 Bytes
D	420 Bytes
E	20 Bytes
F	ALREADY FULL

Request for 180 Bytes of Memory

Worst Fit-Implementation

A	B	C	D	E	F
300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Current Winner

BLOCK C

Block	Free Space(After Allotment of Requested Block)
A	120 Bytes
B	ALREADY FULL
C	320 Bytes
D	420 Bytes
E	20 Bytes
F	ALREADY FULL

Request for 180 Bytes of Memory

Worst Fit-Implementation

A	B	C	D	E	F
300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Current Winner

BLOCK D

Request for 180 Bytes of Memory

Block	Free Space(After Allotment of Requested Block)
A	120 Bytes
B	ALREADY FULL
C	320 Bytes
D	420 Bytes
E	20 Bytes
F	ALREADY FULL

Worst Fit-Implementation

A	B	C	D	E	F
300 Bytes	300 Bytes	500 Bytes	600 Bytes	200 Bytes	200 Bytes

Current Winner

BLOCK D

Block	Free Space(After Allotment of Requested Block)
A	120 Bytes
B	ALREADY FULL
C	320 Bytes
D	420 Bytes
E	20 Bytes
F	ALREADY FULL

Request for 180 Bytes of Memory

Worst Fit-Implementation

A	B	C	D	X	E	F
🟩 300 Bytes	🟥 300 Bytes	🟩 500 Bytes	🟥 180 Bytes	🟩 420 Bytes	🟩 200 Bytes	🟥 200 Bytes

BLOCK D ALLOCATED

NEW BLOCK (X) FORMED

Worst Fit- Pros & Cons

👍

Since this process chooses the largest block, therefore there will be large internal fragmentation.

This internal fragmentation will be quite big so that other small processes can also be placed in that left over block.

👎

Slow Process- Traverses all the blocks in the memory and then selects the largest block among all the blocks, which is a time consuming process.

Next Fit Allocation

Next fit is a modified version of ‘first fit’.

It begins as the first fit to find a free block but when called next time it starts searching from where it left off, not from the beginning.

This helps in, to avoid the usage of memory always from the head (beginning) of the free block chain.