Week 3: List ADT—Linked

CS-250 Data Structure and Algorithms

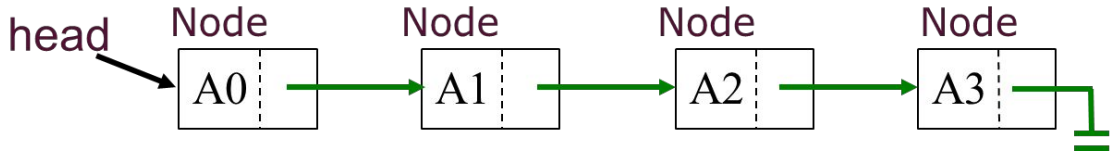
DR. Mehwish Fatima | Assist. Professor
Department of AI & DS | SEECS, NUST

Limitations of Arrays

- Arrays are stored in contiguous memory blocks. And have advantages and disadvantages due to it.
  - It is very easy to access any data element from array using **index**.
  - We need to know **size** of array before hand.
  - We cannot resize array. That's why arrays are called **static data structure**.
    - We can relocate existing array to new array, but still expensive.
  - **Contiguous** block cannot be guaranteed—insufficient blocks size.
  - Insertion and deletion is very **expensive** because it needs **shifting** of elements.

Limitations of Arrays

- Solution: **Linked list**
  - A **dynamic** data structure in which each data element is linked with next element through some link.
  - Because each element is **connected/linked**, it will be easy to insert and delete an element without shifting.
  - Linked list is a linear collection of homogeneous data elements where each element is connected through a link.

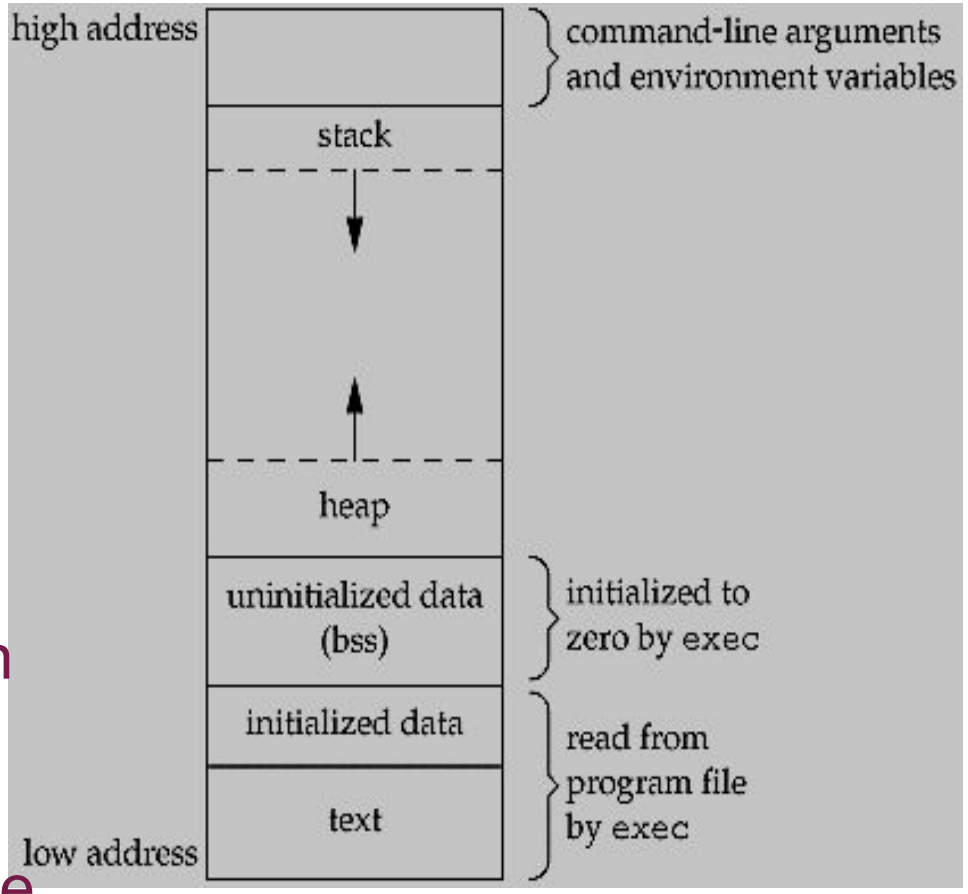


Memory Allocation

Operating System has to allocate memory to each application.

There are two ways that memory gets allocated for data storage:

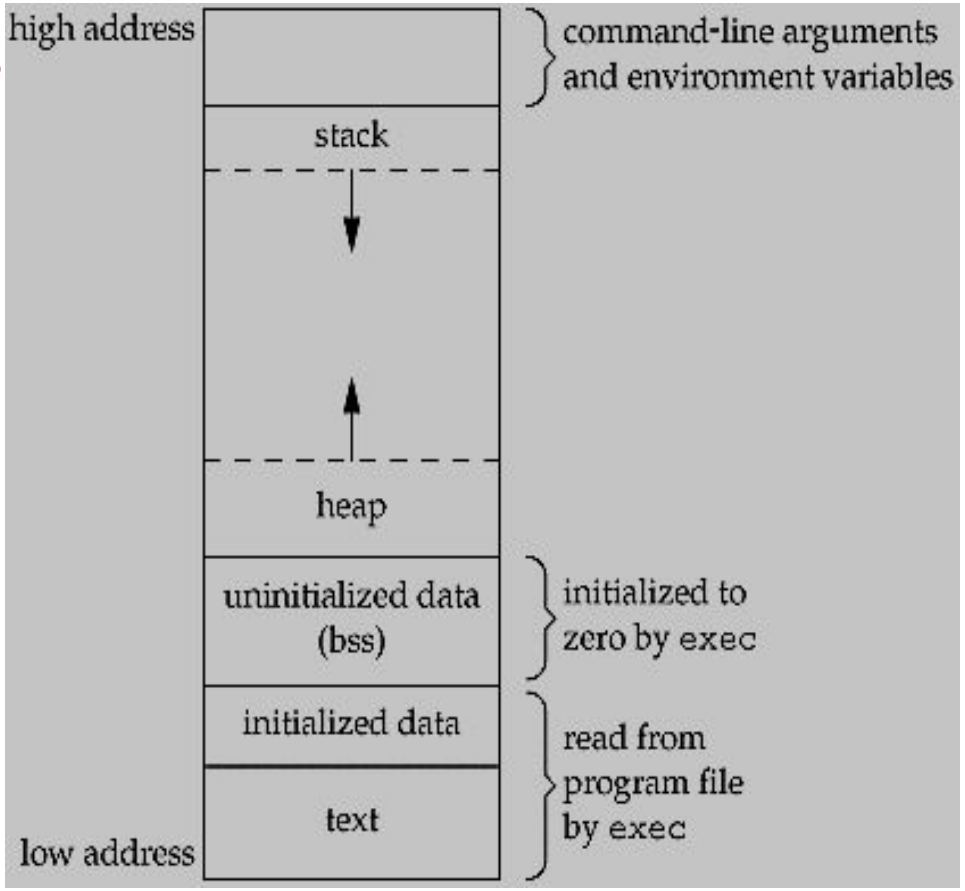
- **Compile Time** (Static) Allocation
  - Memory for named variables is allocated by the compiler
  - Exact size and type of storage must be known at compile time
  - For standard array declarations, this is why the size has to be constant



VIRTUAL MEMORY- PROGRAM ADDRESS SPACE

Memory Allocation

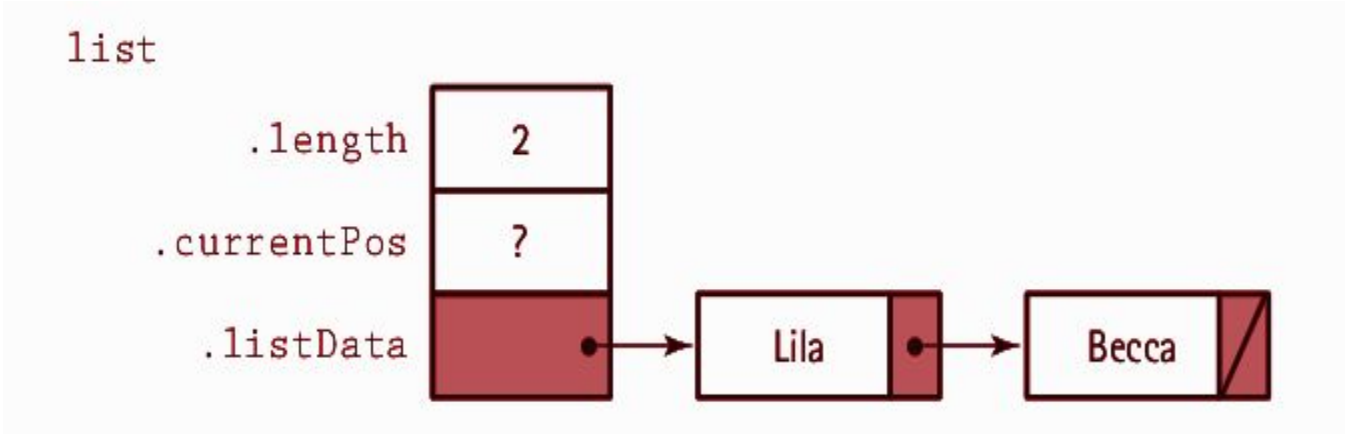
- **Dynamic Memory** Allocation
  - Memory allocated **"on the fly"** during run time
  - dynamically allocated space usually placed in a program segment known as the heap or the free store
  - Exact amount of space or number of items does not have to be known by the compiler in advance
  - For dynamic memory allocation, pointers are crucial



VIRTUAL MEMORY- PROGRAM ADDRESS SPACE

List ADT-Linked List

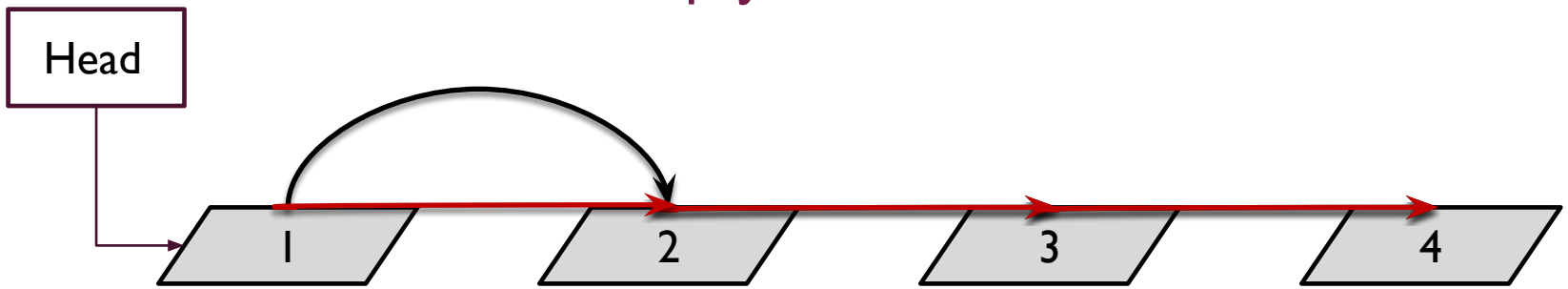
```
// Header file for Unsorted List ADT.
template <class ItemType>
struct NodeType;
// Assumption: ItemType is a type for which the operators "<"
// and "==" are defined-either an appropriate built-in type or
// a class that overloads these operators.
```



List ADT-Linked List

- A single element in linked list is normally called **Node**.
- Every node has two parts:
  - **Data**: actual information
  - **Next Link**: a reference to next node in memory

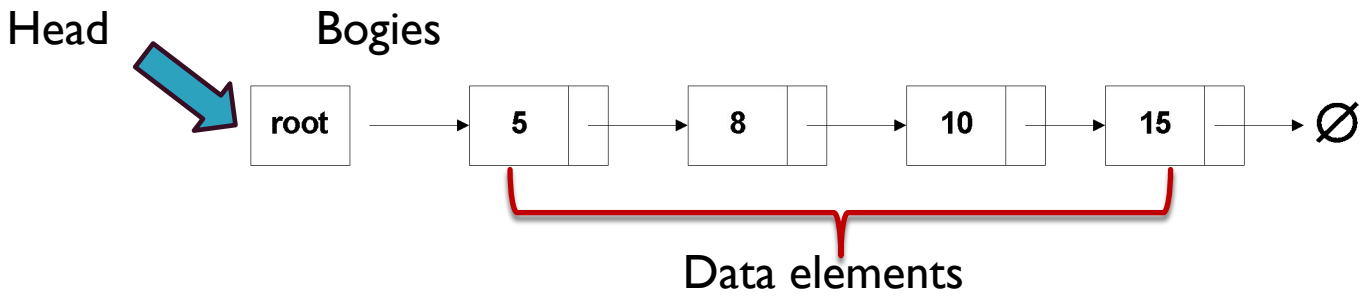
- To maintain the list, we need a start/head pointer.
  - A null Head is an empty list.

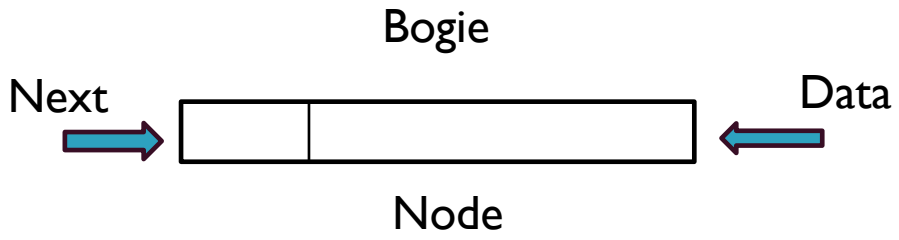


In order to access the elements of the list in dynamic presentation: **all we need is a starting point and a link from one element to the next.**

List ADT-Linked List

- This organization rids us from the requirement to maintain the physical adjacency.
  - Now all we need to maintain is the logical sequences and two logically adjacent elements need to not be physically next to each other.





List ADT-Linked List

- The nodes are connected.

- The nodes are accessed through the **links** between them.
  - For each node the node that is in front of it is called **predecessor**.
  - The node that is after it is called **successor**.

- **Head (front)**
  - Head is a special pointer because it contains address of the **first node**.
  - The first node without predecessor (the node that starts the lists) can be considered as header node.

List ADT-Linked List

- **Tail (end)**
  - Tail is a special pointer because it contains address of **last node**.
  - The last node without **successor** (the node that end the lists) can be considered as trailer node.

- **Current node**
  - The node being processed.
  - From the current node we can access the next node.

Node

- Node can be represented using either structure or class

- Node Operations
  - Constructing a new node
  - Accessing the data value
  - Accessing the next pointer

```
Node* node=new Node
```

C++

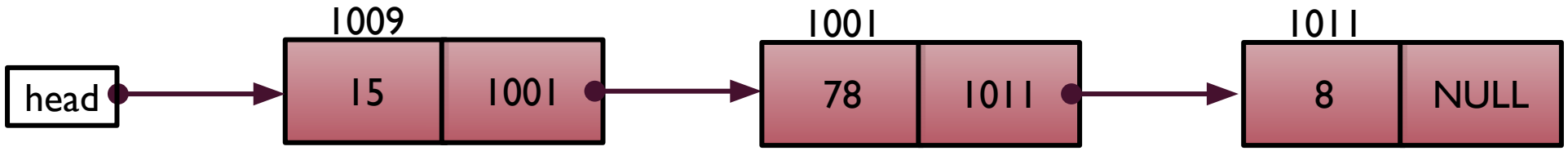
```
struct Node{
int data;
Node* next;
}
```

```
node->data
```

```
node->next
```

Linked List Memory Representation

- Linked list has nodes and memory has cells, nodes of list are distributed in **memory cells**, each node is an object that is **dynamically** created at run time and a free memory cell is allocated to that node.
  - Let say head node of a linked list is located at memory cell 1009. Its data is only an integer value.
  - It points to list's 2nd node which is located at memory location 1001
  - 2nd node points to 3rd node which is located at 1011.

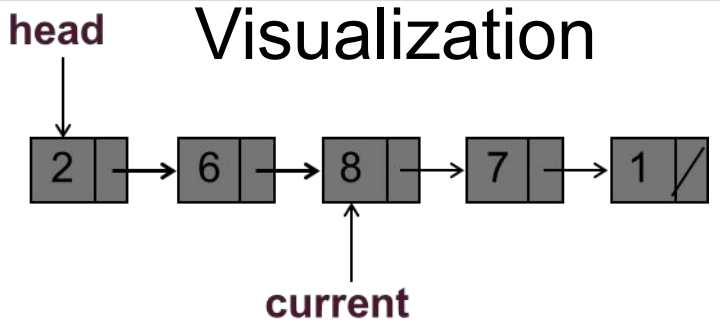


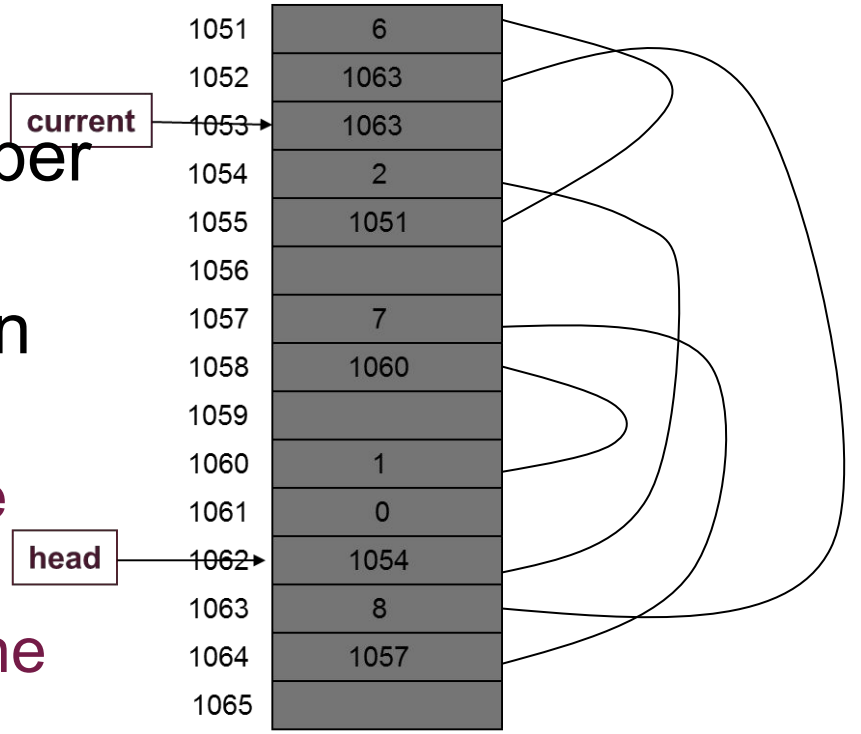
  - 3rd node points to NULL address, means it is end of list

Linked List

- A linked list is a very efficient data structure for sorted list that will go through many insertions and deletions.

- A linked list is a suitable structure if a large number of insertions and deletions are needed, but searching a linked list is slower that searching an array.
  - For example, a linked list could be used to hold the records of students in a school.
  - Each quarter or semester, new students enroll in the school and some students leave or graduate.





Actual picture in memory

Unsorted List ADT

Structure

```
template <class ItemType>
class UnsortedType
{
   public:
      UnsortedType(); // Class constructor.
      ~UnsortedType(); // Class destructor.
      bool IsFull() const; // Determines whether list is full.
      int LengthIs() const; // Determines the number of elements in
      list.
      void MakeEmpty(); // Initializes list to empty state.
      void RetrieveItem(ItemType& item, bool& found);
      // Retrieves list element whose key matches item's key
```

Operations on List ADT

```
void InsertItem(ItemType item);
// Adds item to list.
// Pre: List is not full.
// Post: item is in list.

void DeleteItem(ItemType item);
// Deletes the element whose key matches item's key.
// Pre: Key member of item is initialized.
// Post: No element in list has a key matching item's key.
```

Operations on List ADT

```
void ResetList();
   // Initializes current position for an iteration through the list.
   // Post: Current position is prior to first item in list.

void GetNextItem(ItemType& item);
   // Gets the next element in list.
   // Pre: Current position is defined.
   // Element at current position is not last in list.
   // Post: Current position is updated to next position.
   // item is a copy of element at current position.

private:
   NodeType<ItemType>* listData;
   int length;
   NodeType<ItemType>* currentPos;};
```

Operations on List ADT

```
template<class ItemType>
struct NodeType
{
   ItemType info;
   NodeType* next;
};
```

- To initialize an empty list, we set **listData** (the external pointer to the linked list) to **NULL** and set length to **0**.

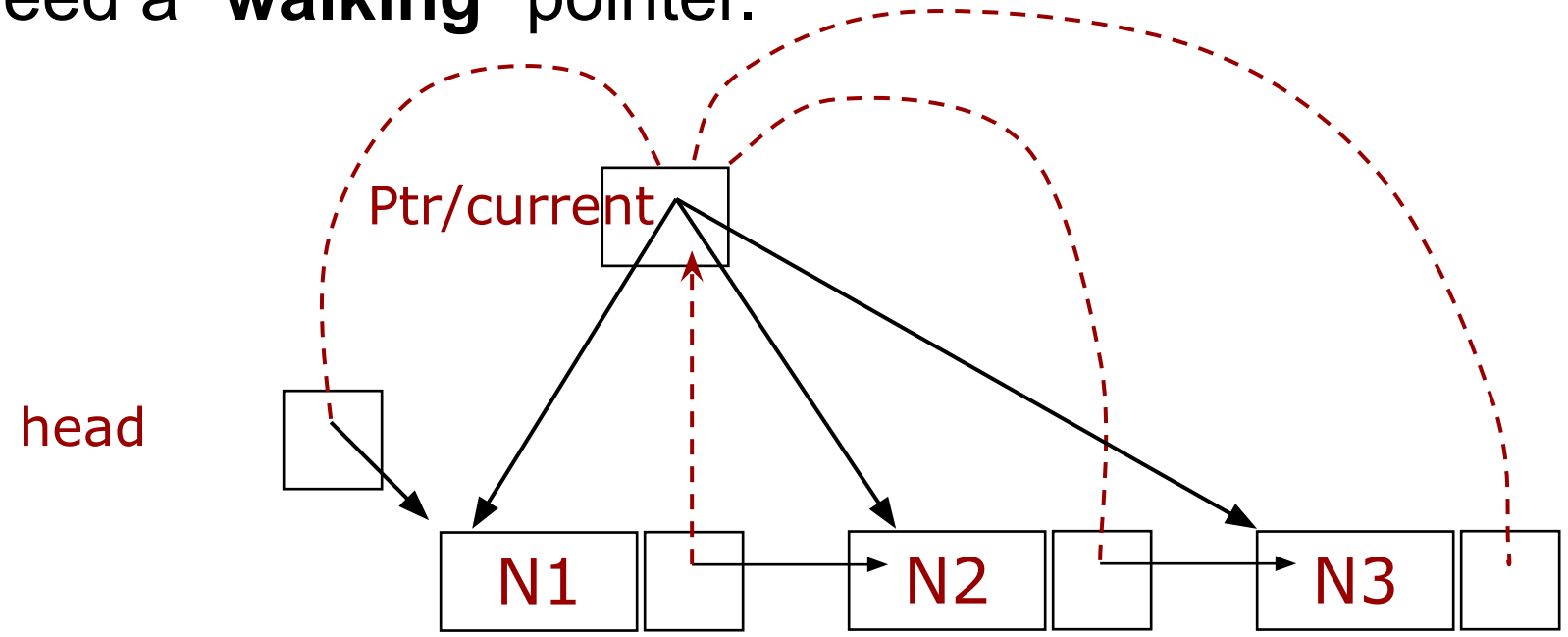
Operations on List ADT

- Here is the class constructor to implement this operation:

```
template <class ItemType>
UnsortedType<ItemType>::UnsortedType()
{
   length = 0;
   listData = NULL;
}
```

Traversing

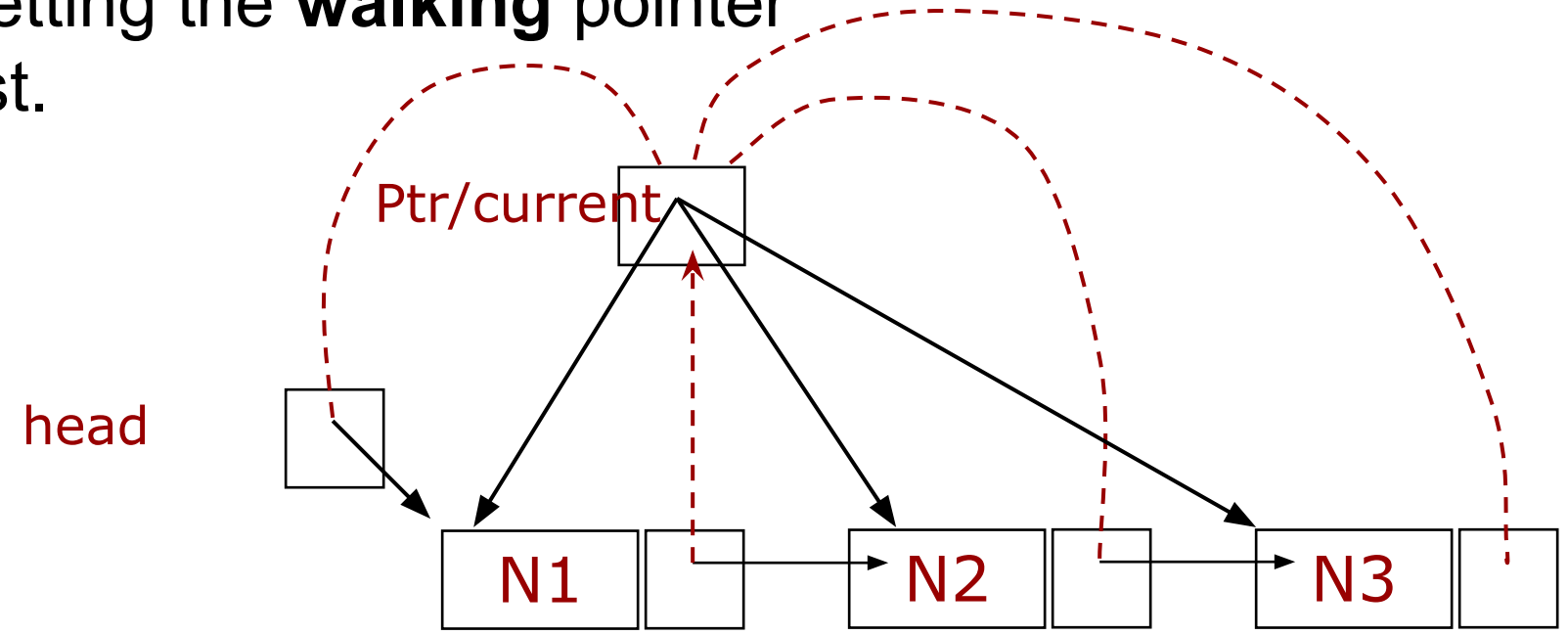
- To traverse the list, we need a "**walking**" pointer.



  - Walker is a pointer that moves from node to node as each element is processed.

Traversing

- We start traversing by setting the **walking** pointer to the first node in the list.



  - Each iteration of the loop processes the current node, then advances the walking pointer to the next node.
  - When the last node has been processed, the walking pointer becomes null and the loop terminates.

IsFull()

- We use the operator new to get a node within a **try block**.
- If more space is available, we return an indicator that the list is not full.
- If no more space is available, an exception is thrown and we return an indicator that there is no more space.

IsFull()

```
template<class ItemType>
bool UnsortedType<ItemType>::IsFull() const
// Returns true if there is no room for another NodeType object
// on the free store and false otherwise.
{
   NodeType<ItemType>* location;
   try{
      location = new NodeType<ItemType>;
      delete location;
      return false;
      }
   catch(std::bad_alloc exception){
      return true;}
   }
```

MakeEmpty()

```
template <class ItemType>
void UnsortedType<ItemType>::MakeEmpty()
// Post: List is empty; all items have been deallocated.
{
   NodeType<ItemType>* tempPtr;
   while (listData != NULL)
   {
      tempPtr = listData;
      listData = listData->next;
      delete tempPtr;
   }
   length = 0;
   }
```

Insertion

- Inserting a new node involves:
  - Creating a **new node**
  - **Linking** this node to its logical **predecessor** and **successor** node
- There can be three scenarios to insert a new node
  - Insertion at Start
  - Insertion at End
  - Insertion at Middle

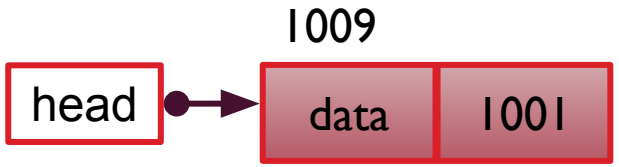
- Insertion at middle operations need searching the linked list.

Insertion At Start

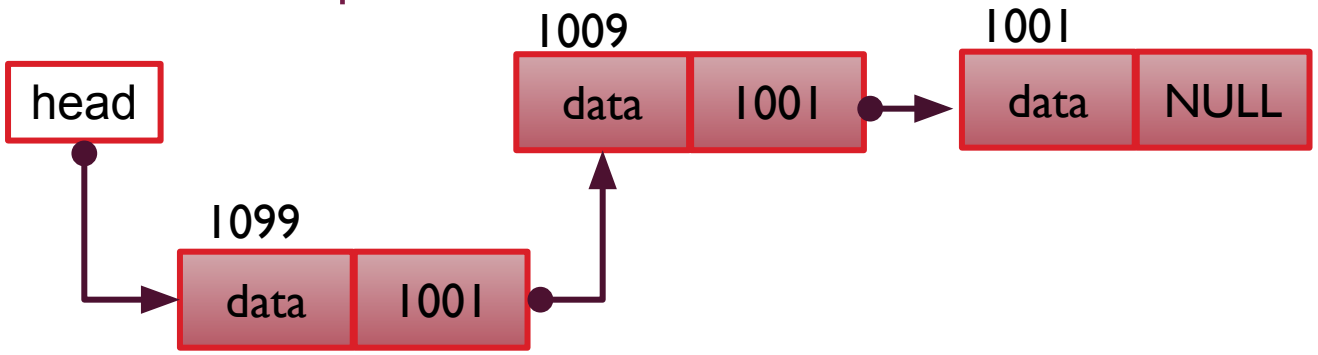
- Empty List

- Non-empty List

  - Create a Node and Update Head

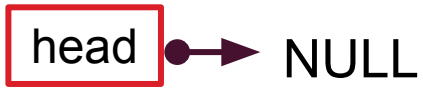


  - Create new node, and update head pointer

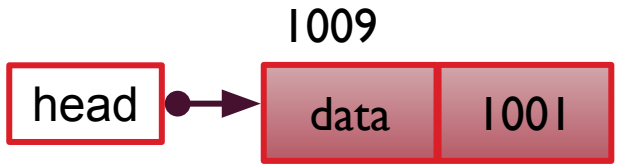


Insertion At End

- Empty List

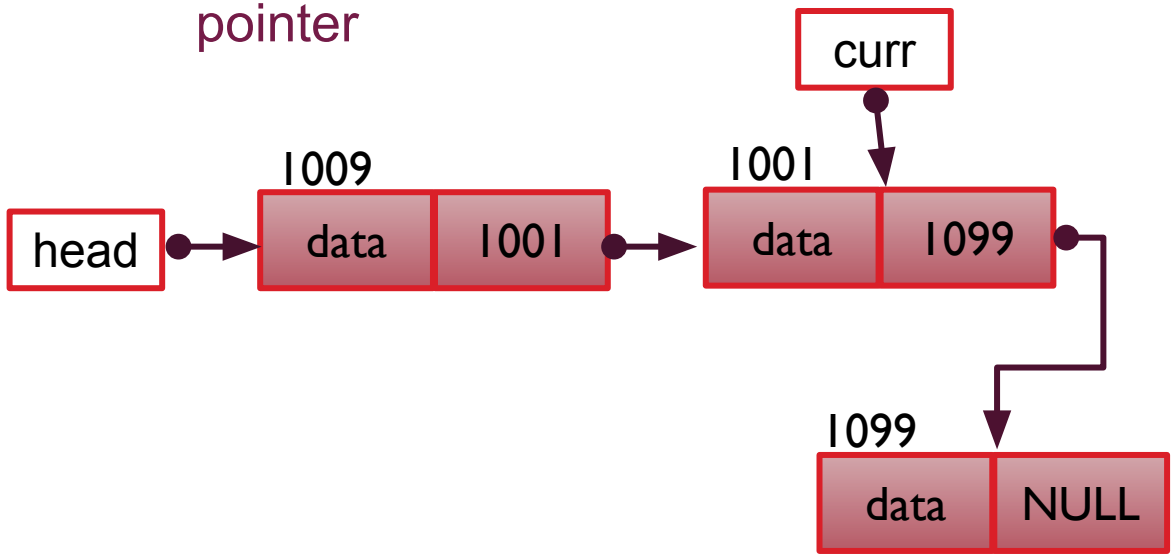


  - Create a Node and Update Head



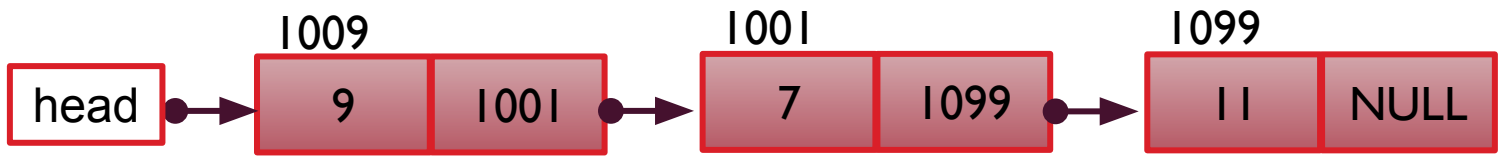
- Non-empty List

  - Create new node, and update head pointer

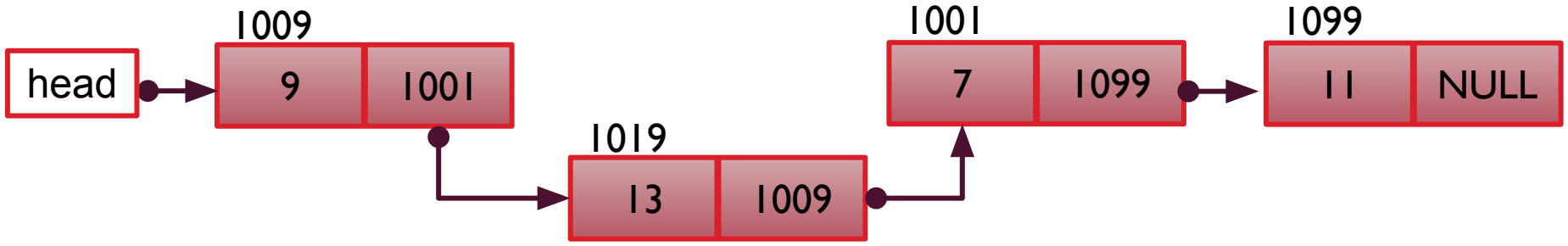


Insertion At Middle

- Let say we want to insert 13 in following list, at position 2.

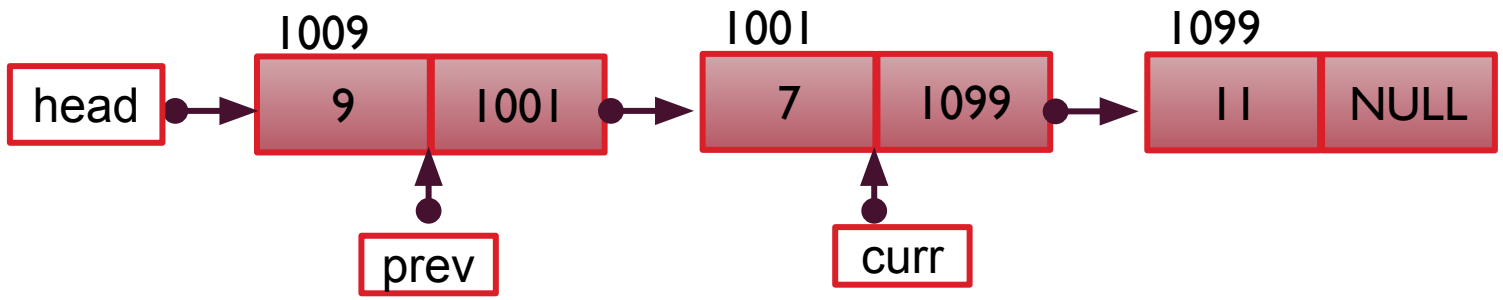


  - In this case, first we need to locate the 2nd node.
  - That is node with **data=7**.
  - Now this will become 3rd node and new node will be inserted before this node.

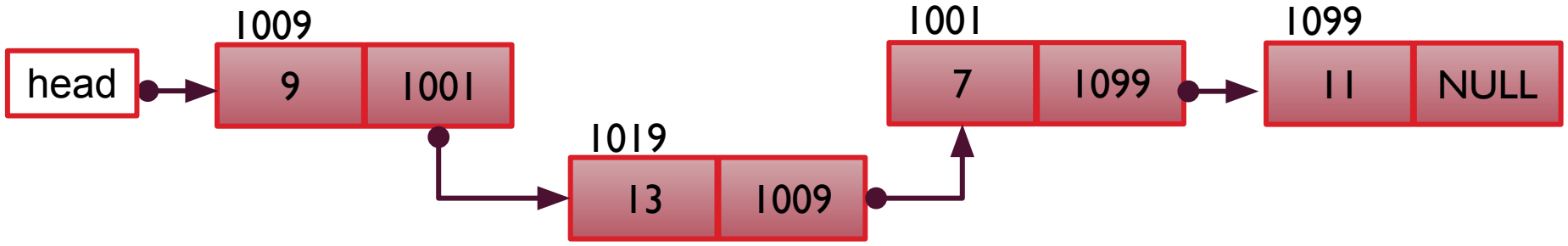


Insertion At Middle

- We need to maintain two pointers: **current** and **previous.**



  - So when new node is inserted, we can easily change **next links** of new node and previous node.



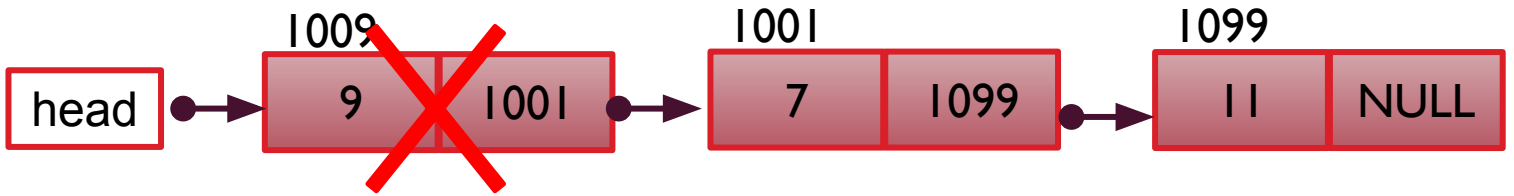
Deletion

- Deleting a new node involves two things:
  - Unlinking the node in a way that its **logical predecessor** gets connected to **next node** of list to maintain linking
  - Deleting the node

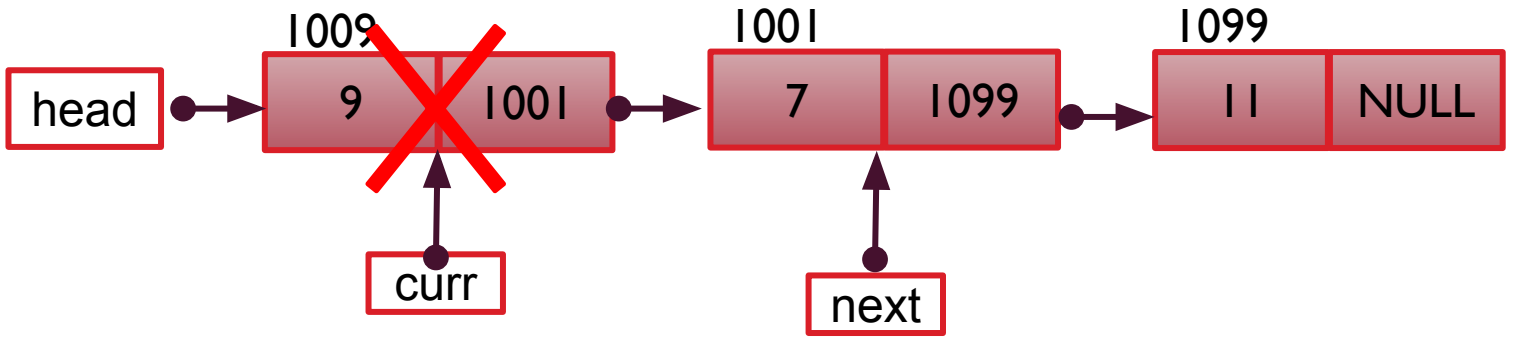
- There can be three scenarios to delete node
  - Deletion at Start
  - Deletion at End
  - Deletion at Middle

- Deletion at middle operations need searching the linked list.

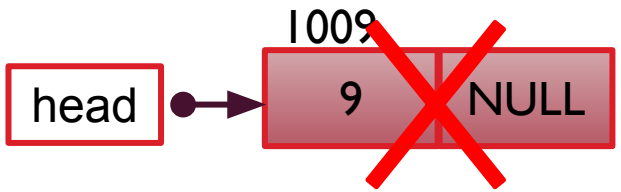
Deletion At Start



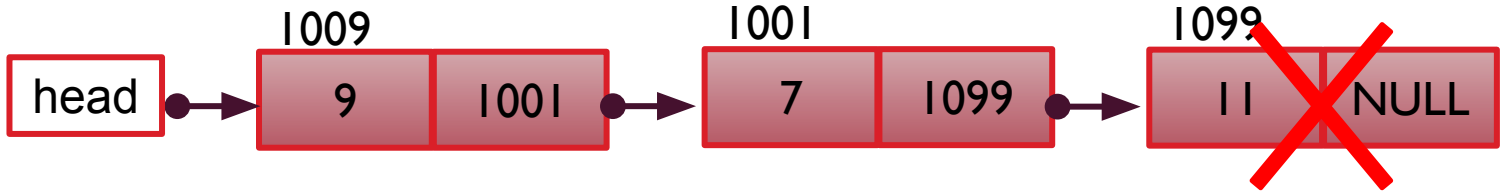
Deletion At Start



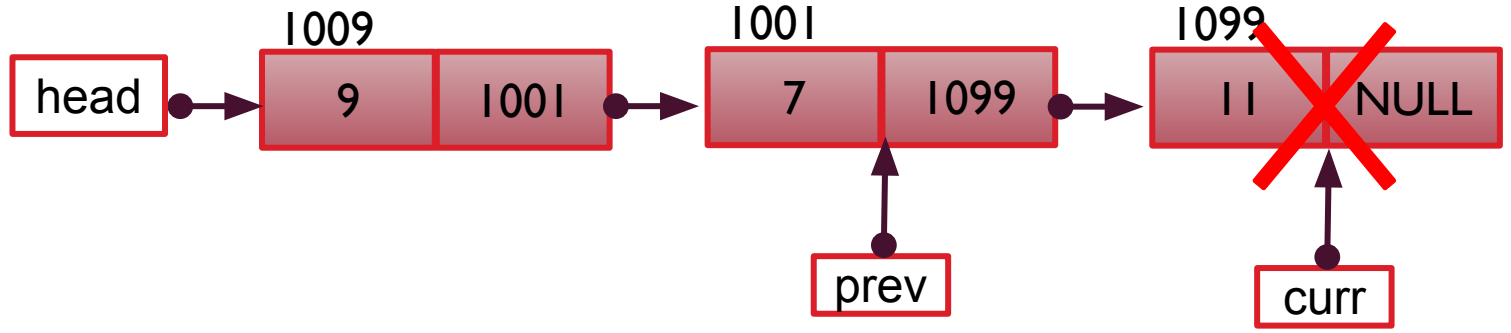
Deletion At Start or End



Deletion At End

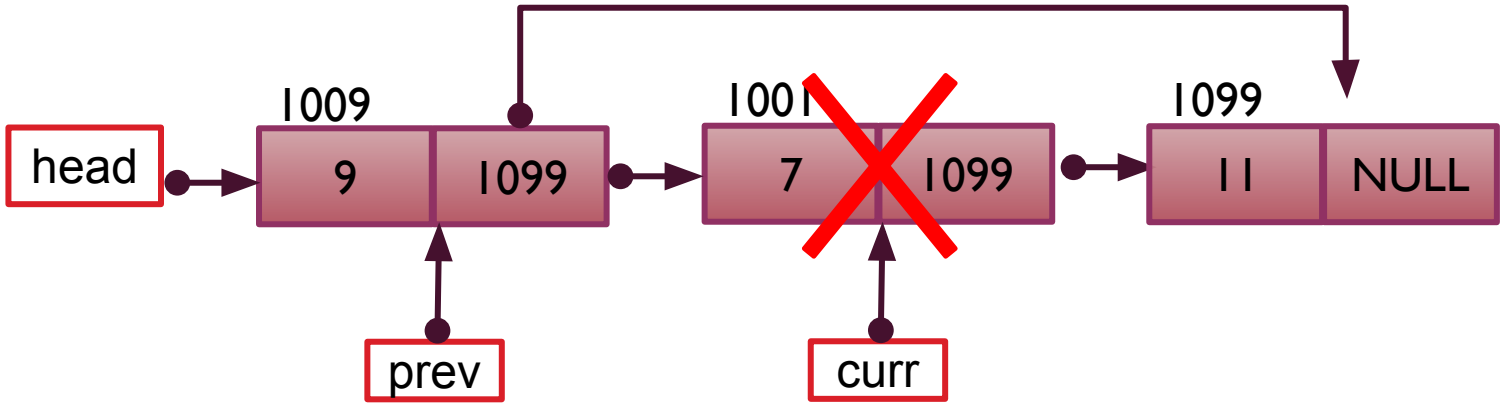


Deletion At End



Deletion At Middle

- We need to maintain two pointers: **current** and **previous.**



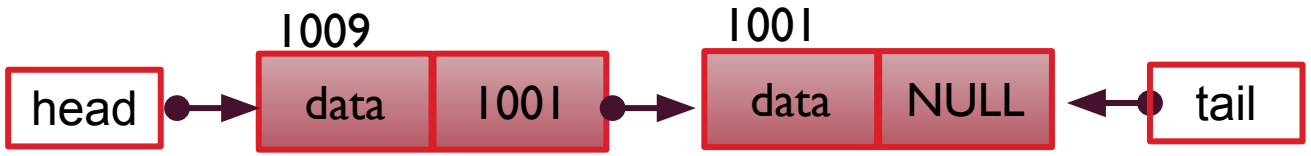
  - So when this node is deleted, we have to change **next link** of previous node to connect it with the new next successor.

Variation in Delete

- How Delete_Position will change?
  - If a data value is given instead of location.?
  - If a node is given instead of location?

Last Node

- Rather than searching the last node each time, we can maintain a reference to last node just like we do for first node.
- It will save time



Search

- Loop Termination Conditions
  - We reach at the end and key not found
  - We found the key and return the position

Time Complexity

Example of algorithm	Complexity
Traversal	O(n)
Retrieval / Access	O(n)
Insertion At End/Start (with tail)	O(1)
Insertion At middle	O(n)
Deletion At End/Start (with tail)	O(1)
Deletion At middle	O(n)
Linear Search	O(n)

Types

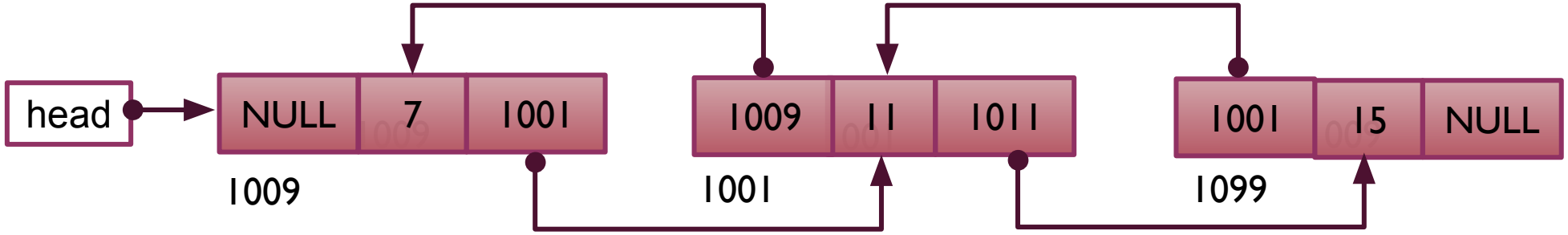
- **Nodes Linkage**
  - Single/ Singly lists (Discussed till yet)
    - Each node contains a link only to the next node.
    - Only one direction to move.
  - Double/ Doubly lists
    - Each node contains two links - to the previous and to the next node.
    - Can move forward and backward.
- **Last Node Ending**
  - Circular/ Circulary lists
    - The tail is linked to the head node.
  - Grounded lists
    - A tail is ended with NULL terminator.

	Array Implementation	Linked Implementation
Class constructor	O(1)	O(1)
Destructor	NA	O(*N*)
`MakeEmpty`	O(1)	O(*N*)
`IsFull`	O(1)	O(1)
`LengthIs`	O(1)	O(1)
`ResetList`	O(1)	O(1)
`GetNextItem`	O(1)	O(1)
`RetrieveItem`		
Find	O(*N*)	O(*N*)
Process	O(1)	O(1)
Combined	O(*N*)	O(*N*)
`InsertItem`		
Find	O(1)	O(1)
Insert	O(1)	O(1)
Combined	O(1)	O(1)
`DeleteItem`		
Find	O(*N*)	O(*N*)
Delete	O(1)	O(1)
Combined	O(*N*)	O(*N*)

Doubly List

Double Linked List

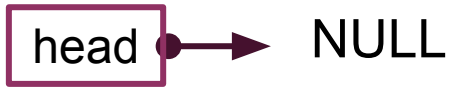
- Every node contains **two links**, next which points to next node and previous which points to previous node in list



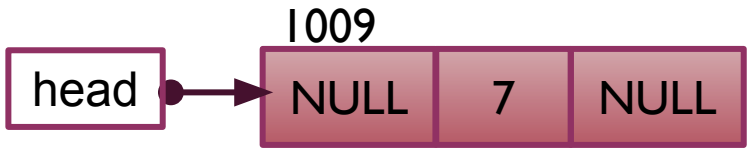
  - Note that prev and next links hold address of nodes. So, prev link of node located at 1001 points to 1009 and next link points to 1099.
    - Previous link of first node is NULL
    - Next link of last node is NULL
- Doubly linked list can be traversed from start to end and from end to start.
  - If we have tail node

Insertion At Start

- Empty List

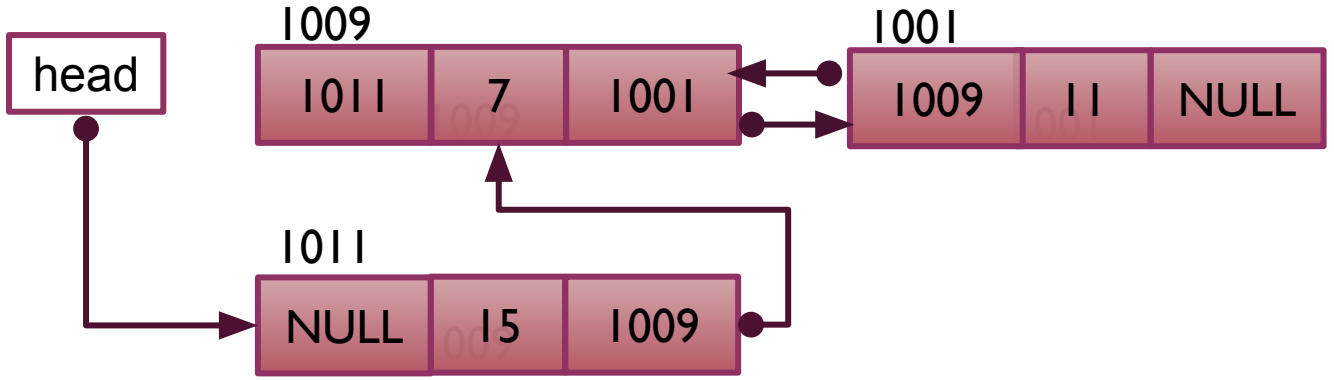


  - Create a Node and Update Head



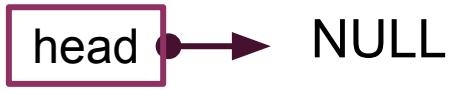
- Non-empty List

  - Create new node, update head & next pointers



Insertion At End

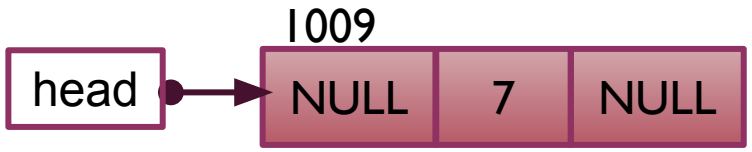
- Empty List

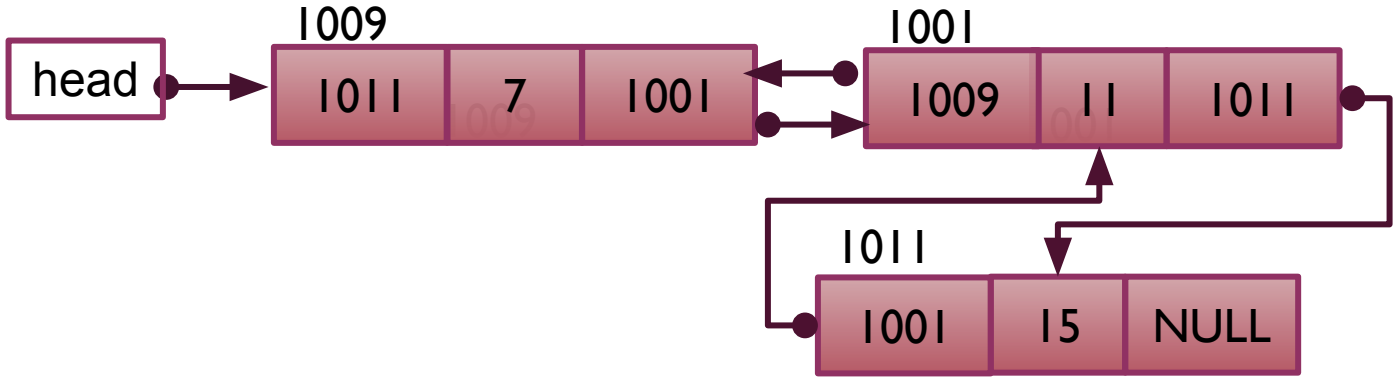


- Non-empty List

  - Create new node, update head & next pointers

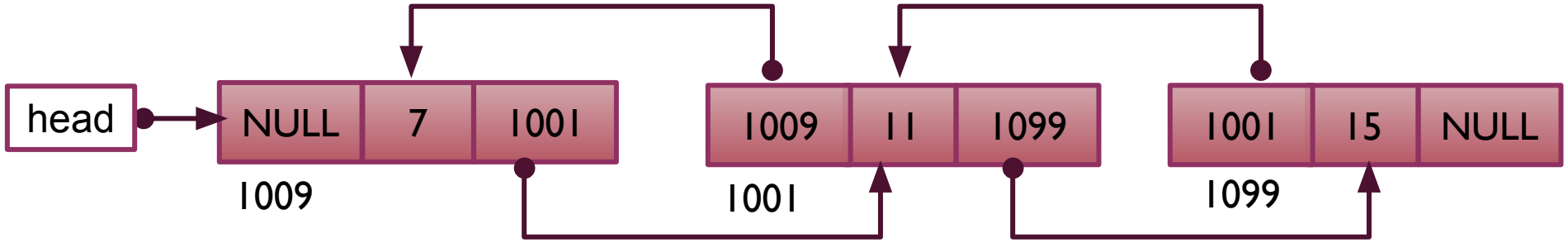
  - Create a Node and Update Head



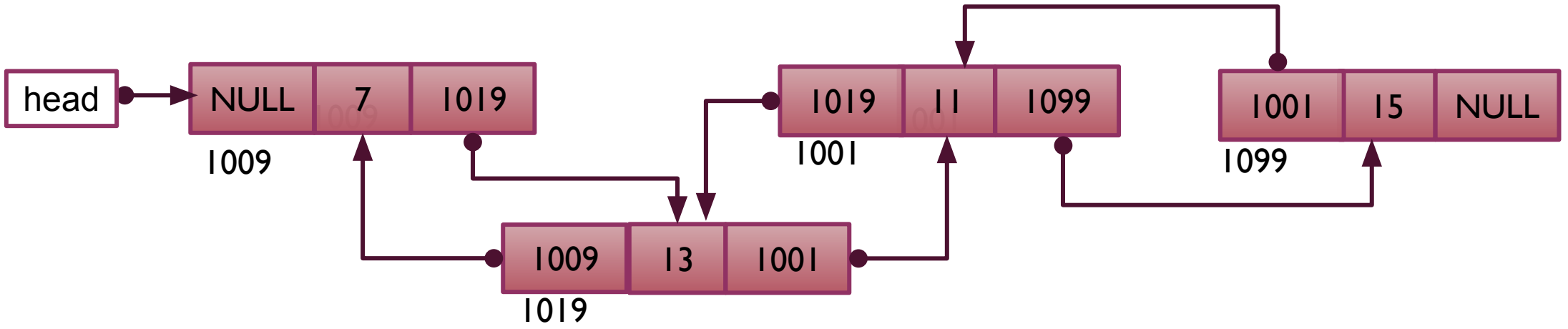


Insertion At Middle

- Let say we want to insert 13 in following list, at position 2.

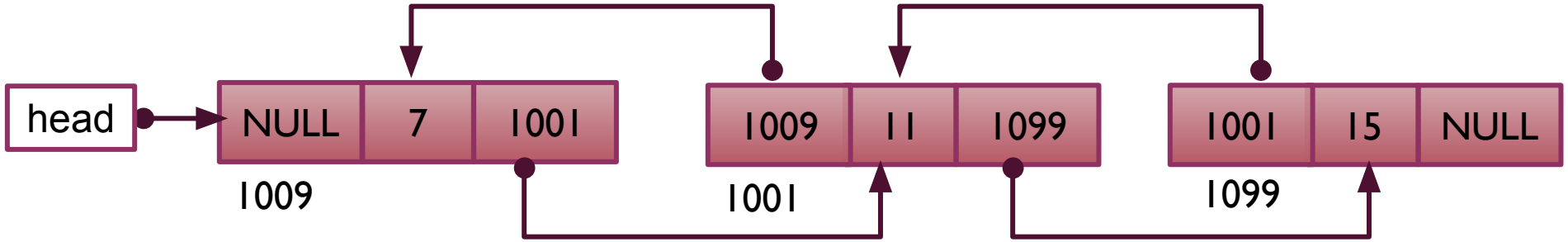


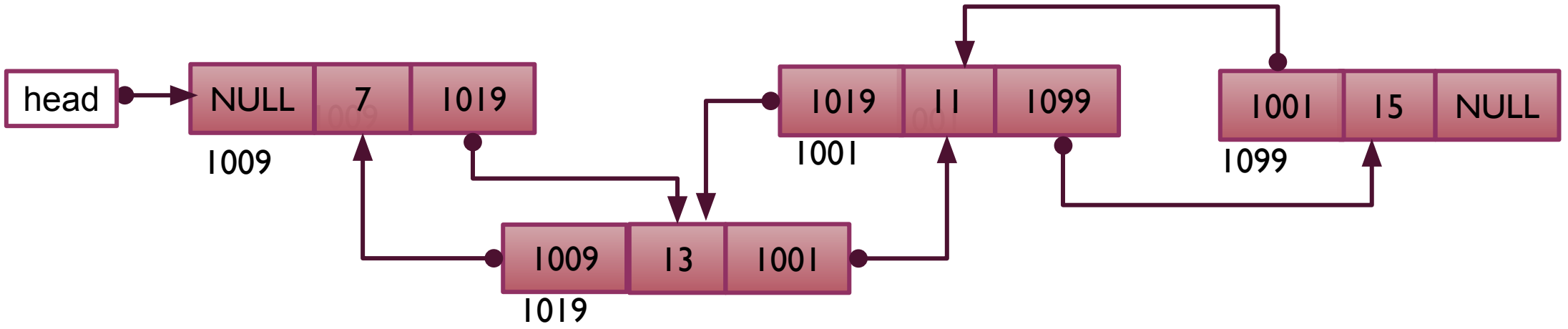
  - In this case, first we need to locate the 2nd node.
  - That is node with **data=11**.



Insertion At Middle

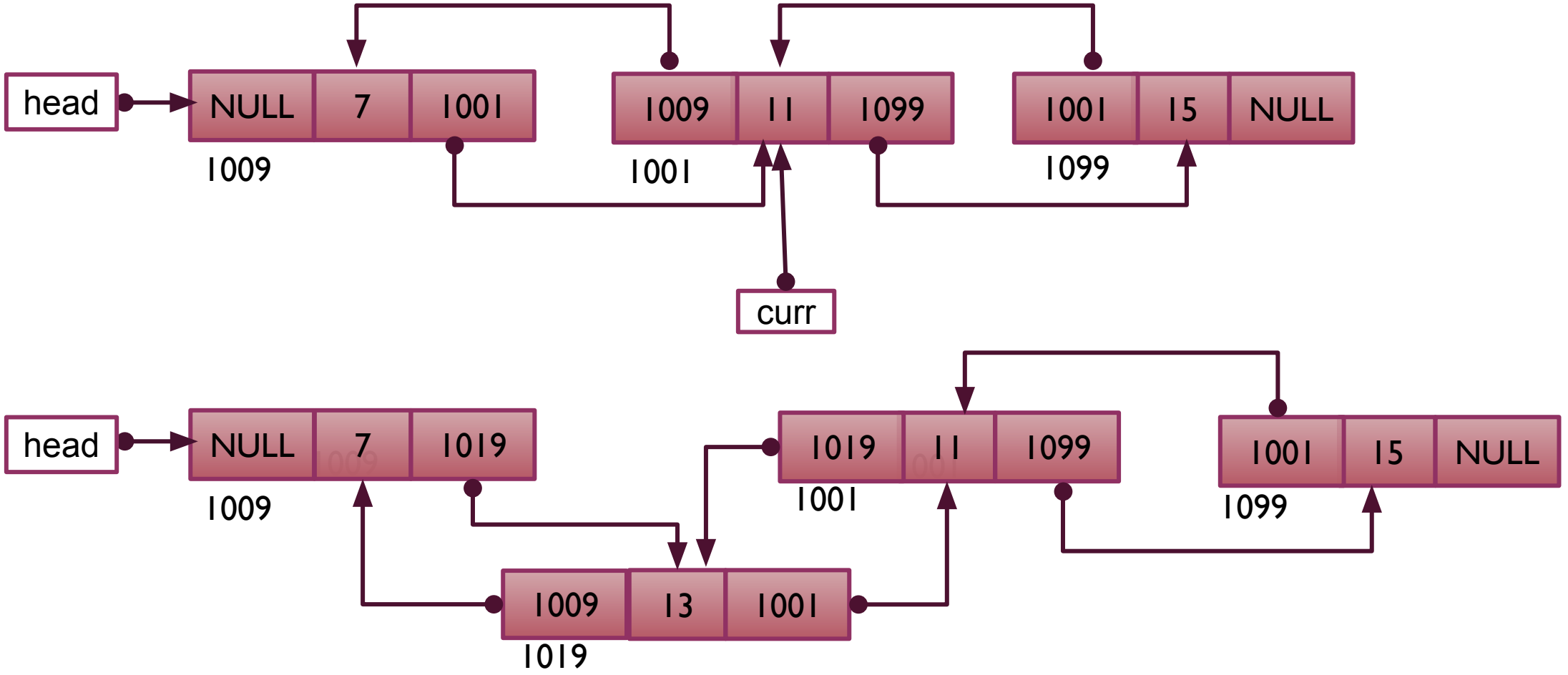
- How many pointers we need to maintain?



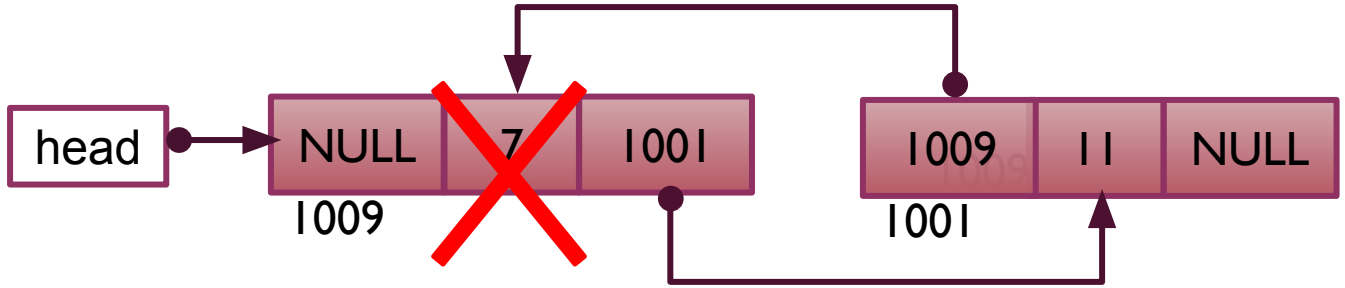


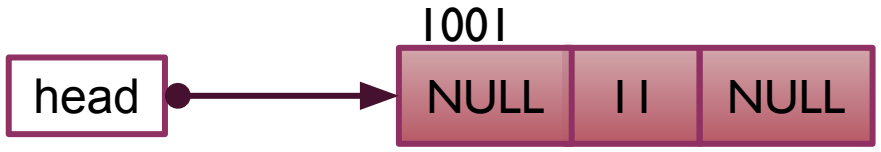
Insertion At Middle

- How many pointers we need to maintain?

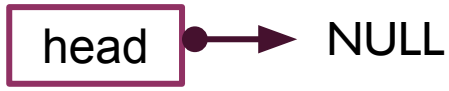


Deletion At Start

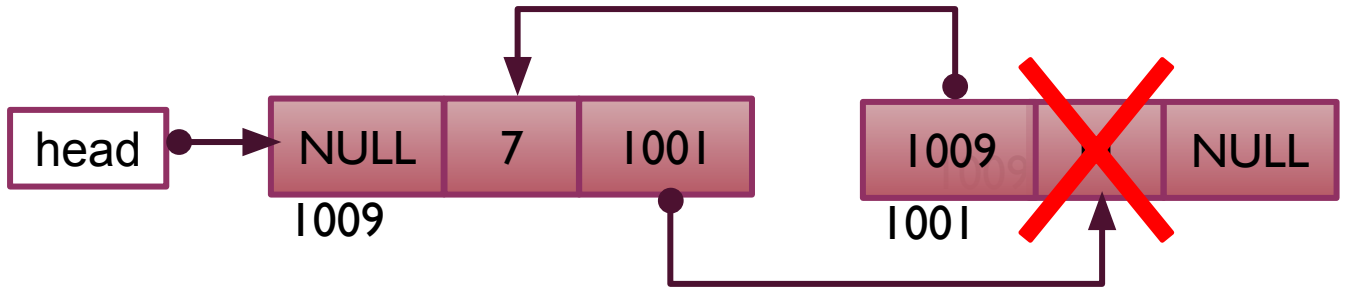


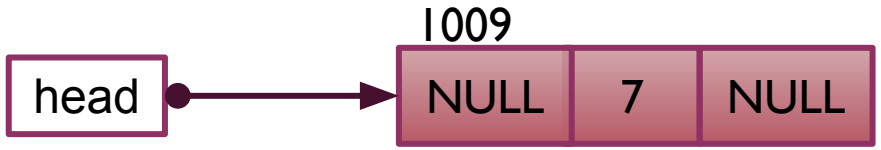


Deletion At Start or End

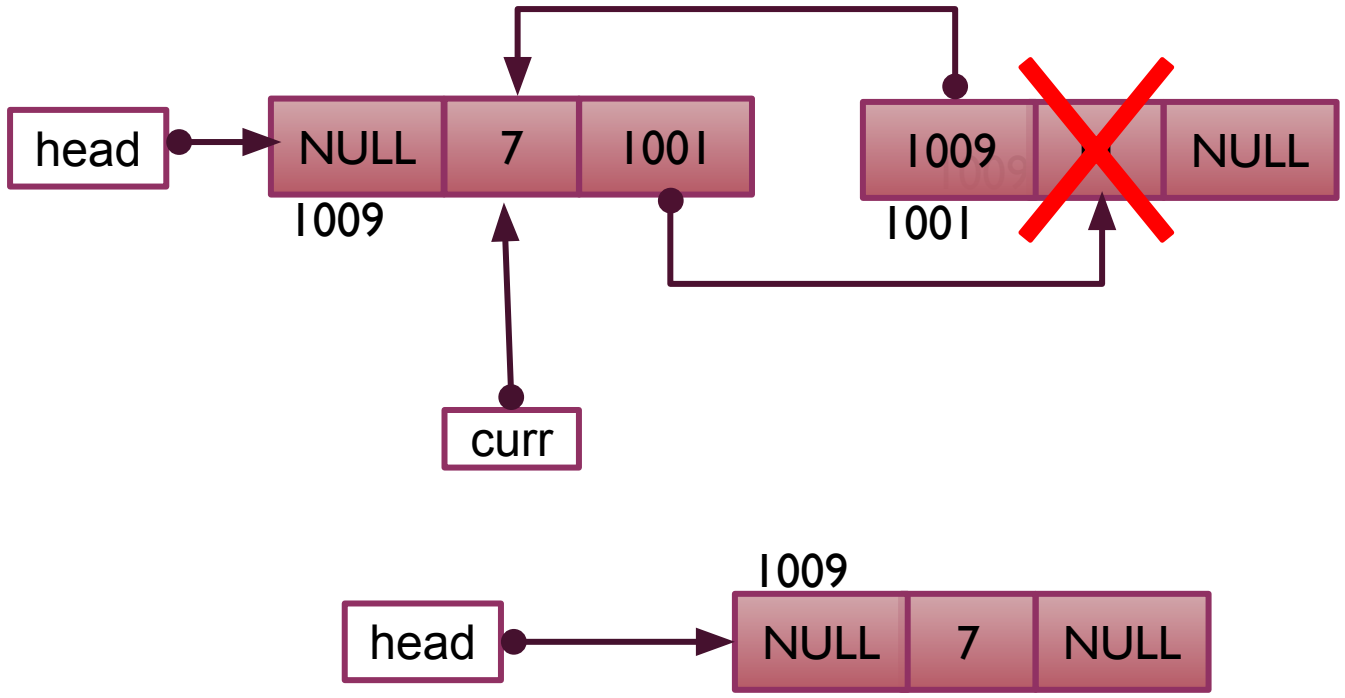


Deletion At End



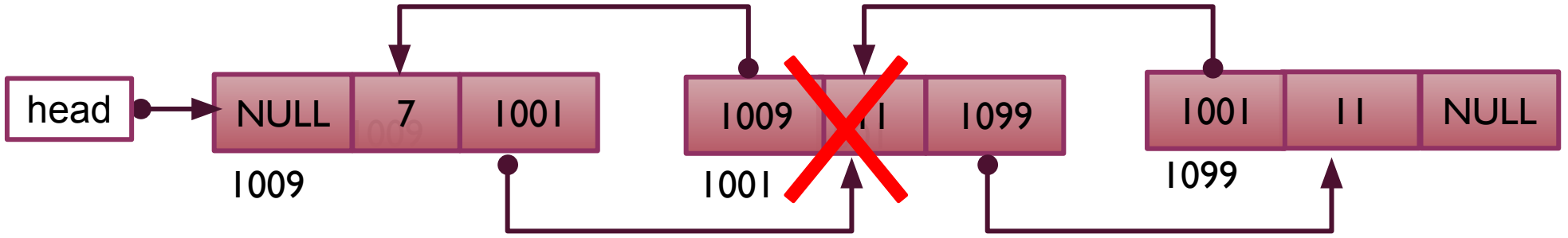


Deletion At End

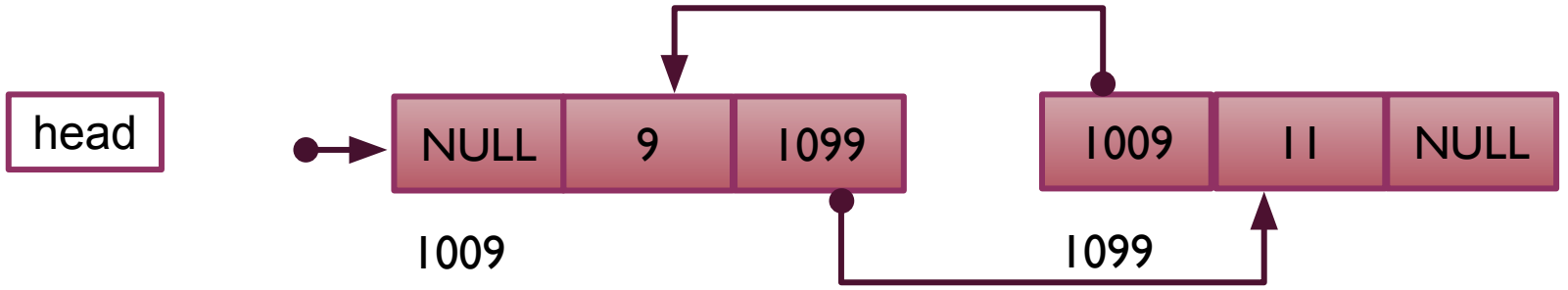


Deletion At Middle

- We want to delete the 2nd node.

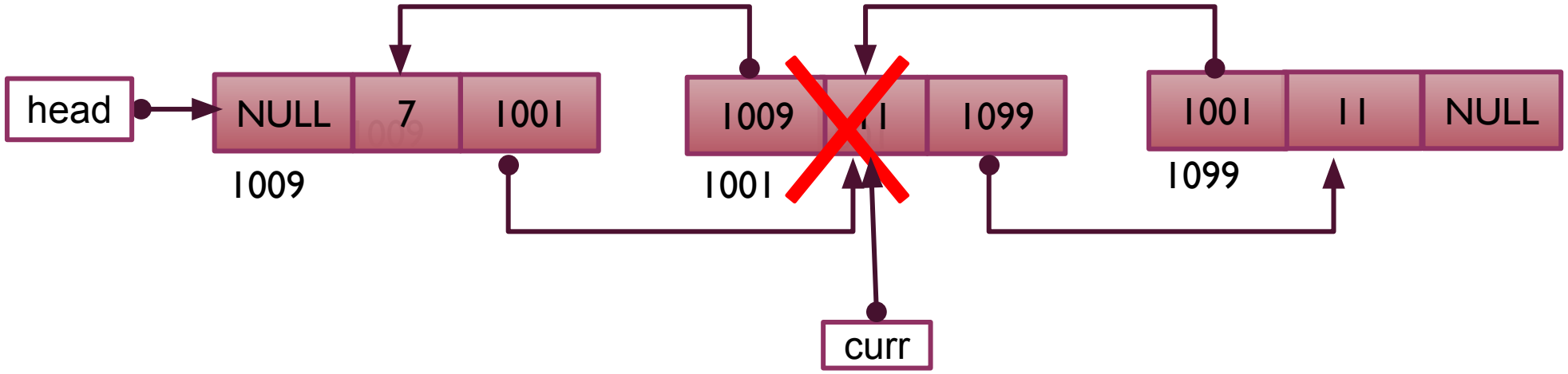


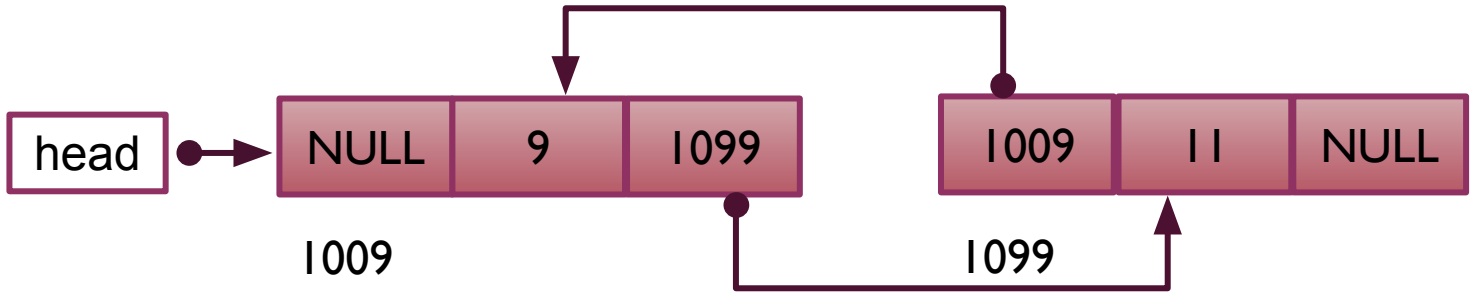
  - So when this node is deleted, we have to change **links** to connect previous node with the new next successor.



Deletion At Middle

- How many **pointers** do we need?

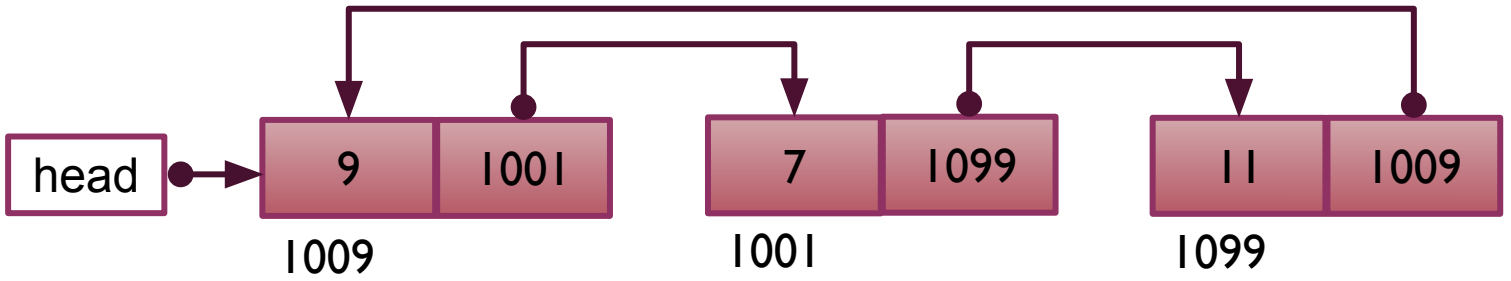




Circular List

Circular Singly List

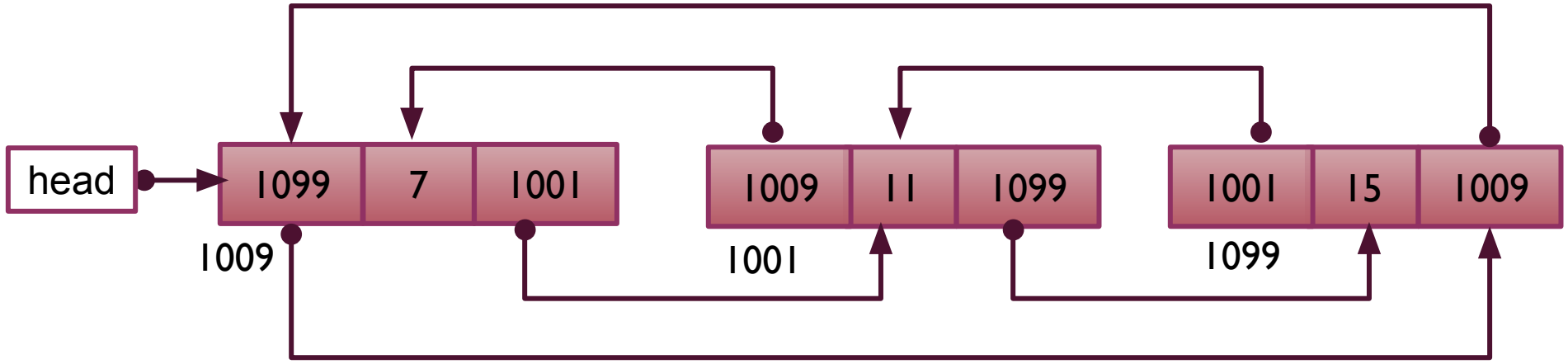
- Every node contains only one next link which points to next node in list.



- The **Last node** points to the **First node** of list

Circular Doubly List

- Doubly linked list, with last node pointing to the first node and the first node pointing to the last node.



  - Previous link of the first node is the last node
  - Next link of the last node is the first node

Circular List

- What change will be required in following algorithms of both single and double linked list:
  - Insert
  - Delete
  - Search
- When loop will terminate?

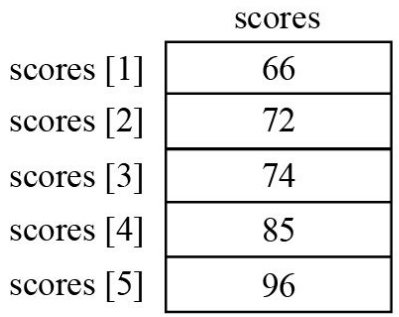
- What change needs to be done in singly linked list algorithms?

- How to know that node is the **last node** in list?

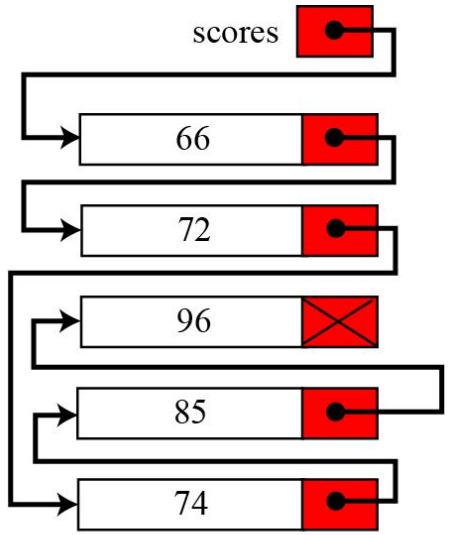
Difference of Implementation

Array–based vs. Linked–based

- Both an array and a linked list are representations of a list of items in memory.
  - The only difference is the way in which the items are linked together.
  - Linked list allows addition or deletion of items in the middle of collection with only a **constant amount** of data movement. **Contrast** this with array.



a. Array representation



b. Linked list representation

Array–based vs. Linked–based

- Arrays can be static or dynamic

- Getting position
  - random vs sequential
  - Access 3rd element in array vs in linked list

- Add/Delete
  - Shifting vs. changing links

Array–based vs. Linked–based

- Searching
  - If data is unordered
  - Search until found or end
  - If data is ordered
    - Array-based list: Middle, Lower, Upper Bound calculation is straightforward
    - LinkedList-based list:Can we do binary search over linked list

Summary

- In this lecture we have been discussed:
  - List ADT with linked based implementation
  - Fundamental operations on List ADT
  - Difference of implementations
  - Concept of Big-O notation