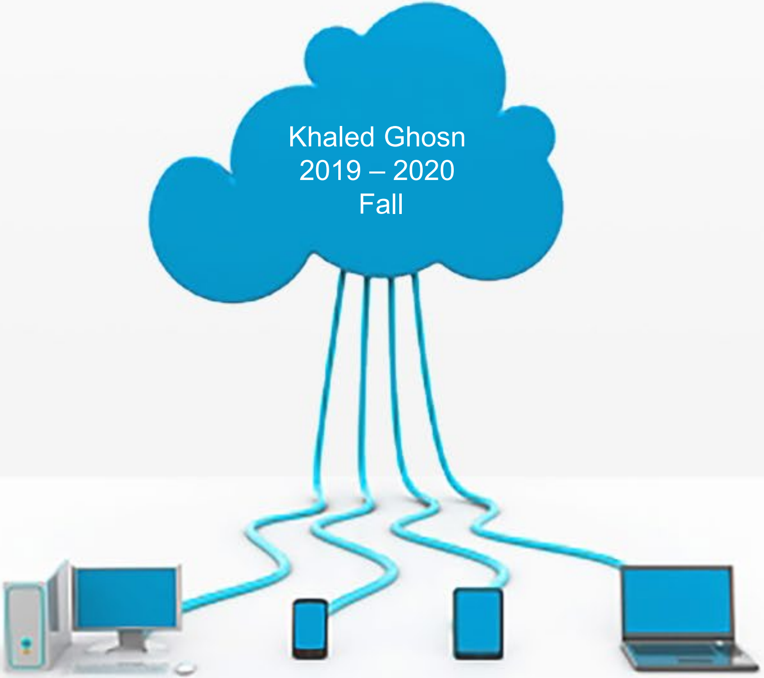
ARTS, SCIENCES & TECHNOLOGY
UNIVERSITY IN LEBANON
AUL

Types of Linked Lists

Singly, Doubly, and Circular Linked Lists



Khaled Ghosn
2019 – 2020
Fall

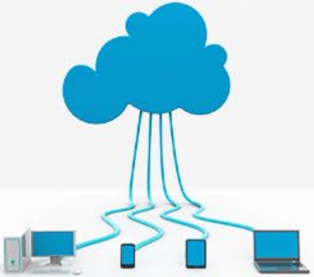
Linked List

Definition

- A linked list is a dynamic data structure where each element is a separate element
  - Dynamically allocate space for each element as needed
    - ✓ **Flexible space use**
  - Each element of a list is called a **Node**
  - Nodes are NOT contiguous but are scattered in the memory

- Linked list consists of one or more nodes
  - The number of nodes in a list is not fixed and can grow and shrink
  - Any application which has to deal with an unknown number of objects will need to use a linked list

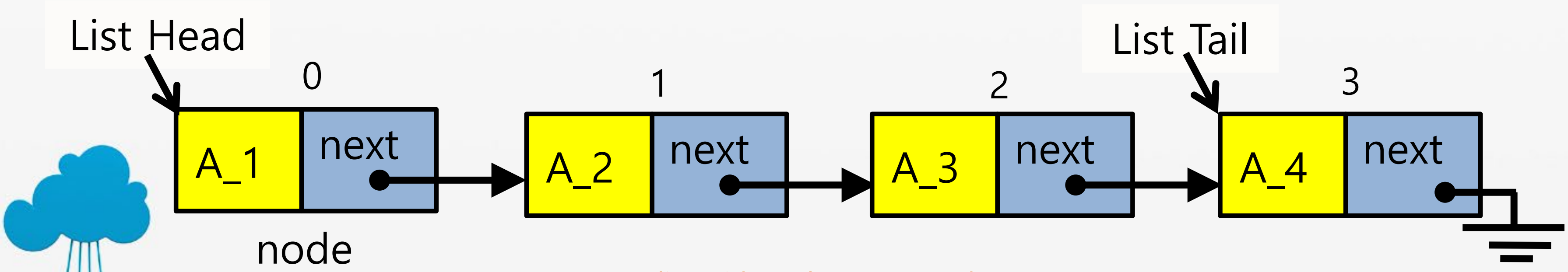
- Linked Lists addresses some of the limitations of arrays



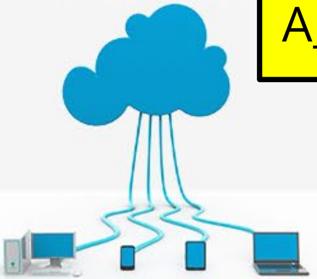
Linked List

Nodes

- Each node contains some data
- A Node class must store reference/s to other nodes (pointer/s to another nodes)



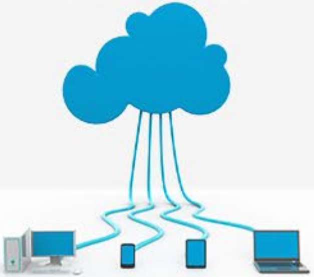
A list with 4 elements (nodes)



Linked List

Head & Tail Nodes

- Need to know the location of the first node
  - the entry point into a linked list
- The first node is called the **Head** (**Front / First-Node**) of the list
  - The head is not a separate node, just the reference to the first node
  - If you want to access a particular item then you have to start at the head and follow the references until you get to that item
  - If the list is empty then the head is a null reference
  - Initially NULL
- The last node is known as **Tail** (**Back / Last-Node**) of the list
  - has a reference to NULL

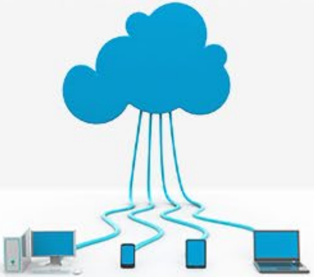


Linked List

Disadvantages

- One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements.

- Another disadvantage is that a linked list uses more memory compare with an array



Linked List

Types

1. **Singly Linked List**
2. **Doubly Linked List**

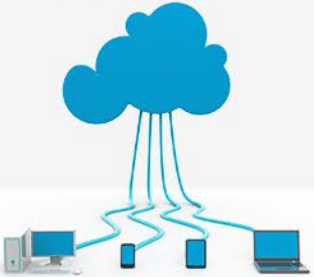
   is a list that has two references:

   a) **next node**

   b) **previous node**
3. **Singly Circular Linked List**

   where last node of the list points back to the first node (head)
4. **Doubly Circular Linked List**

   where last node of the list points back to the first node (head), and the first node of the list points to the last node (tail)

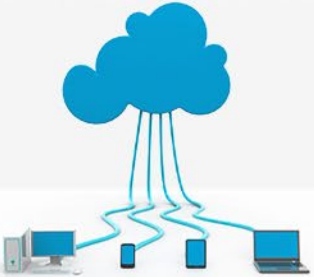


Array-List vs. Linked List

Types

- Accessing elements are faster with Array-List (because it is index based)
- Accessing is difficult with Linked List. It is slow access. This is to access any element, you need to navigate through the elements one by one
- Insertion and deletion is much faster with Linked List, because if you know the node, just change the pointers before or after nodes
- Insertion and deletion is slow with Array-List, this is because, during these operations Array-List need to adjust the indexes according to deletion or insertion if you are performing on middle indexes.

  Means, an Array-List having 10 elements, if you are inserting at index 5, then you need to shift the indexes above 5 to one more



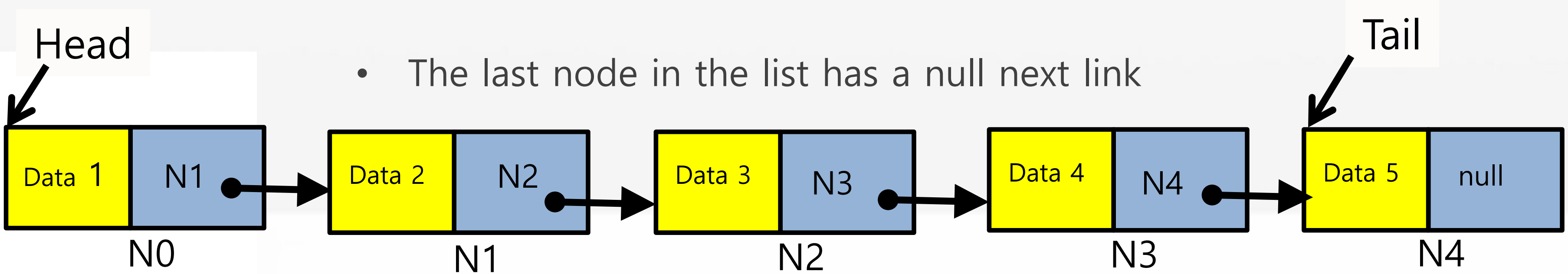
Singly Linked List

SLL

- The basic linked list consists of a collection of connected dynamically allocated nodes

- In a *singly linked list,* each node consists of:
  1. Data element (value)
  2. One Link (reference) to the next node in the list

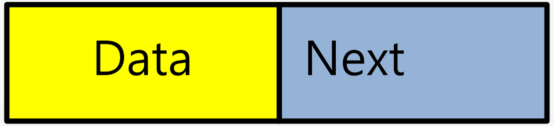
- The last node in the list has a null next link



Singly Linked List

SLL-Node operations

SLL Node

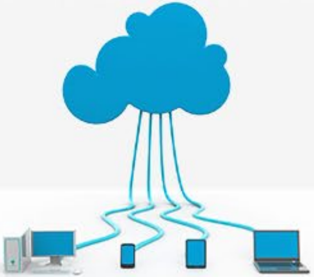


SLLNode

- *Anytype* **Data**
- *SLLNode* **Next**

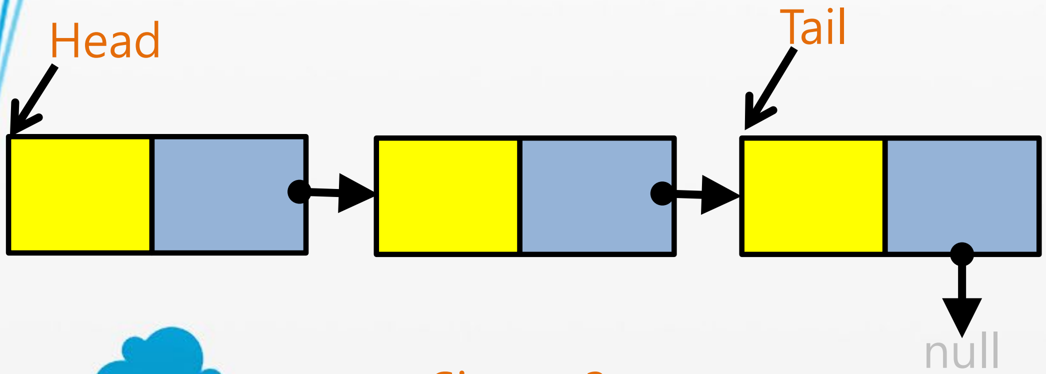
+ **SLLNode** ( )
+ **SLLNode** (*Anytype* Data)
+ **SLLNode** (*Anytype* Data, *SLLNode* next)

+ *Anytype* **getData** ( )
+ *void* **setData** (*Anytype* Data)
+ *SLLNode* **getNext** ( )
+ *void* **setNext** (*SLLNode* next)



Singly Linked List

SLL operations



Size = 3

SLL

- *SLLNode* **Head**
- *SLLNode* **Tail**
- *int* **Size**

+ **SLL** ( )
+ **SLL** (*Anytype* value)
+ **SLL** (*SLLNode* FirstNode)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *SLLNode* **getHead** ( )
+ *SLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )

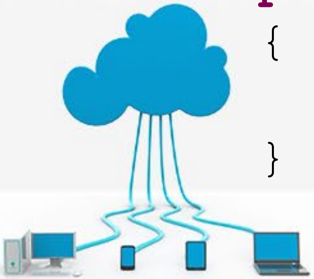
Singly Linked List

isEmpty ( )

- ✓ Check if Head is null

Head → null

```
public boolean isEmpty()
{
    return this.Head == null;
}
```



SLL

- *SLLNode* **Head**
- ~~*SLLNode* **Tail**~~
- ~~*int* **Size**~~

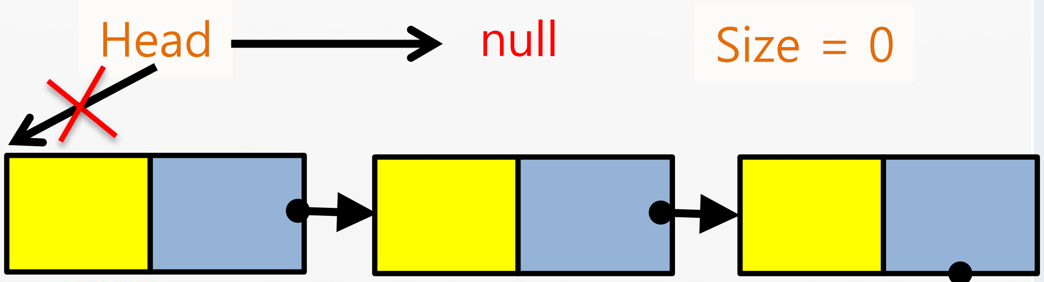
+ **SLL** ( )
+ **SLL** (*Anytype* value)
+ **SLL** (*SLLNode* FirstNode)

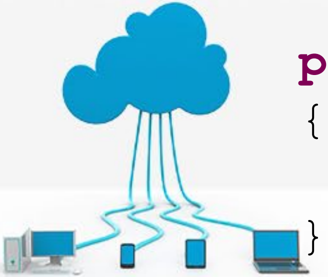
+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *SLLNode* **getHead** ( )
+ *SLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )

Singly Linked List

makeEmpty ( )

- ✓ Let Head be null





```
public void makeEmpty()
{
    this.Head = null;
}
```

SLL

- *SLLNode* **Head**
- ~~*SLLNode* **Tail**~~
- ~~*int* **Size**~~

+ **SLL** ( )
+ **SLL** (*Anytype* value)
+ **SLL** (*SLLNode* FirstNode)

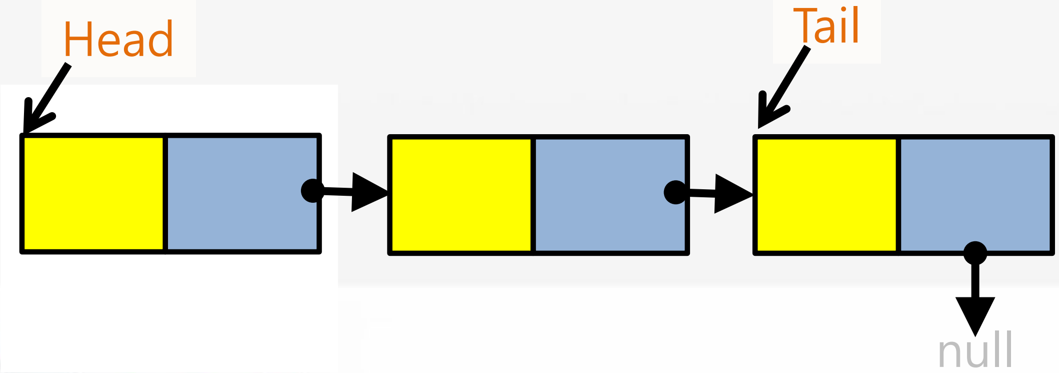
+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *SLLNode* **getHead** ( )
+ *SLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )

Singly Linked List

Length ( )

✓ Count Nodes

1. Start from Head (check if null to stop)
2. Count and move to next node
3. Stop when reaching the tail (its "next node" is null)

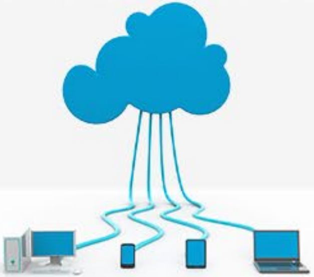


SLL
- *SLLNode* **Head** - ~~*SLLNode* **Tail**~~ - ~~*int* **Size**~~
+ **SLL** ( ) + **SLL** (*Anytype* value) + **SLL** (*SLLNode* FirstNode) + *boolean* **isEmpty** ( ) + *void* **makeEmpty** ( ) + *int* **Length** ( ) + *SLLNode* **getHead** ( ) + *SLLNode* **getTail** ( ) + *void* **addFirst** (*Anytype* value) + *void* **addLast** (*Anytype* value) + *Anytype* **removeFirst** ( ) + *Anytype* **removeLast** ( ) + *void* **Print** ( )

Singly Linked List

Length ( ) - Iterative

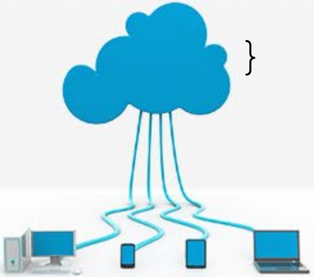
```
public int Length()
{
  if (isEmpty())
     return 0;
  SLLNode currentNode = this.Head;
  int counter=1;
  while(currentNode.getNext()!=null)
  {
     currentNode=currentNode.getNext();
     counter++;
  }
  return counter;
}
```



Singly Linked List

Length ( ) - Recursive

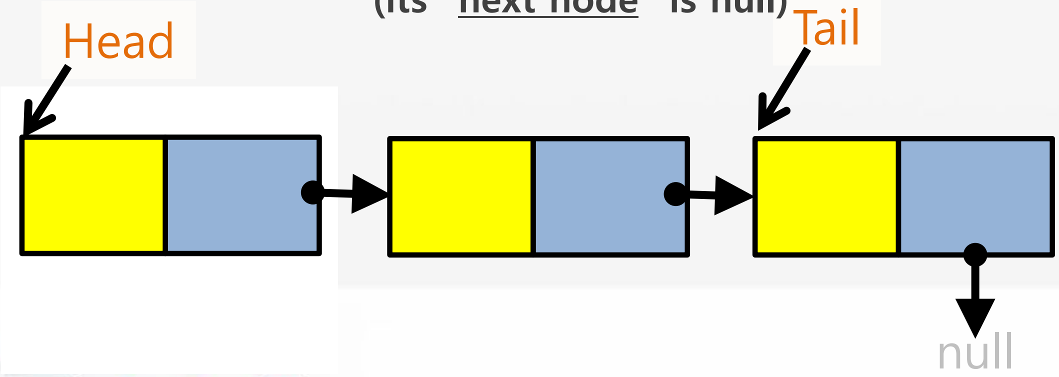
```
public int Length(SLLNode start)
{
    if (start == null)
        return 0;
    else
        return 1 + Length(start.getNext());
}
```



Singly Linked List

getTail ( )

- ✓ Find Last Node -Tail-
  1. Start from Head (if List is not empty)
  2. Move to next node
  3. Stop when reaching the node having a NULL next node (its "next node" is null)

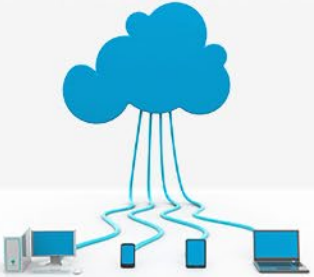


SLL
- *SLLNode* **Head** - ~~*SLLNode* **Tail**~~ - ~~*int* **Size**~~
+ **SLL** ( ) + **SLL** (*Anytype* value) + **SLL** (*SLLNode* FirstNode) + *boolean* **isEmpty** ( ) + *void* **makeEmpty** ( ) + *int* **Length** ( ) + *SLLNode* **getHead** ( ) + *SLLNode* **getTail** ( ) + *void* **addFirst** (*Anytype* value) + *void* **addLast** (*Anytype* value) + *Anytype* **removeFirst** ( ) + *Anytype* **removeLast** ( ) + *void* **Print** ( )

Singly Linked List

getTail ( ) - Iterative

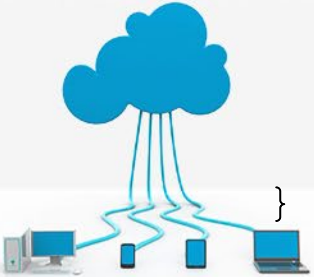
```
public SLLNode getTail()
{
  if (this.isEmpty())
     return null;

  SLLNode currentNode = this.Head;

  while (currentNode.getNext() != null)
     currentNode = currentNode.getNext();

  return currentNode;
}
```



Singly Linked List

getTail ( ) - Recursive

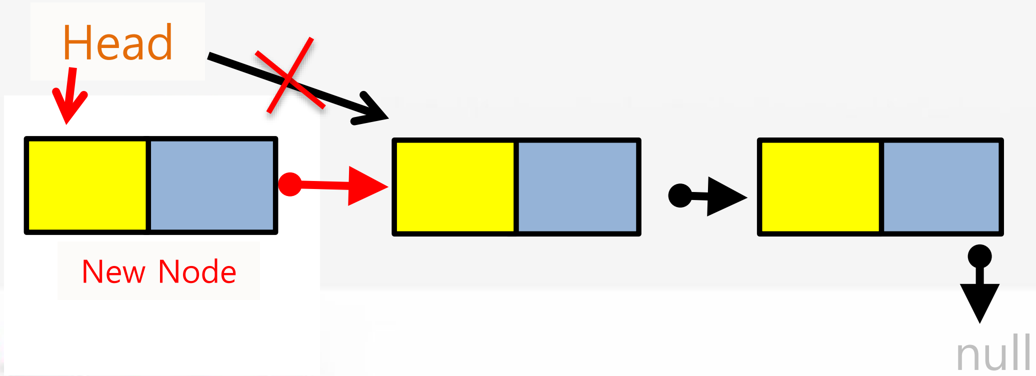
```
public SLLNode getTail(SLLNode start)
{
    if (start == null)
        return null;

    if (start.getNext() == null)
        return start;
    else
        return getTail(start.getNext());
}
```



Singly Linked List

addFirst (value)

1. Create a new node
2. Let the "<u>next node</u>" of the new node be the "Head" node (unless if the list is empty)
3. Let "Head" refer to the new node



SLL

- *SLLNode* **Head**
- ~~*SLLNode* **Tail**~~
- ~~*int* **Size**~~

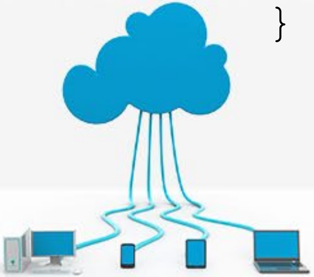
+ **SLL** ( )
+ **SLL** (*Anytype* value)
+ **SLL** (*SLLNode* FirstNode)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *SLLNode* **getHead** ( )
+ *SLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )

Singly Linked List

addFirst (value)

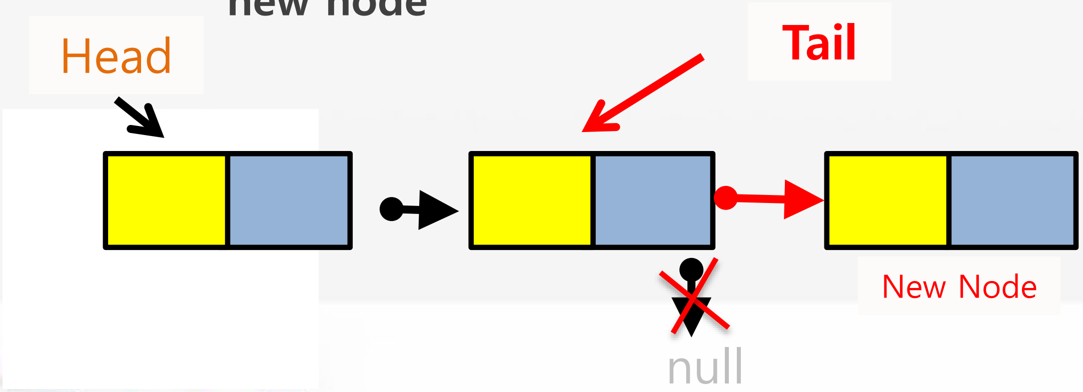
```
public void addFirst(Anytype value)
{
  //if ( isEmpty() )
  //    this.Head = new SLLNode(value);
  // else
       this.Head = new SLLNode (value, this.Head);
}
```



Singly Linked List

addLast (value)

1. Check if the List is not empty
   (if empty, just set the Head to be the new node)
2. Create a new node
3. Find the last node (getTail( ))
4. Set the Tail's "next node" to refer to the new node

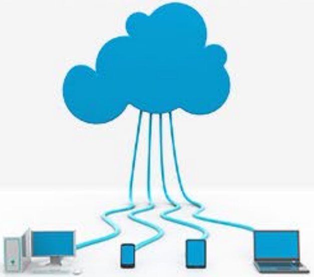


SLL
- *SLLNode* **Head** - ~~*SLLNode* **Tail**~~ - ~~*int* **Size**~~
+ **SLL** ( ) + **SLL** (*Anytype* value) + **SLL** (*SLLNode* FirstNode) + *boolean* **isEmpty** ( ) + *void* **makeEmpty** ( ) + *int* **Length** ( ) + *SLLNode* **getHead** ( ) + *SLLNode* **getTail** ( ) + *void* **addFirst** (*Anytype* value) + *void* **addLast** (*Anytype* value) + *Anytype* **removeFirst** ( ) + *Anytype* **removeLast** ( ) + *void* **Print** ( )

Singly Linked List

addLast (value)

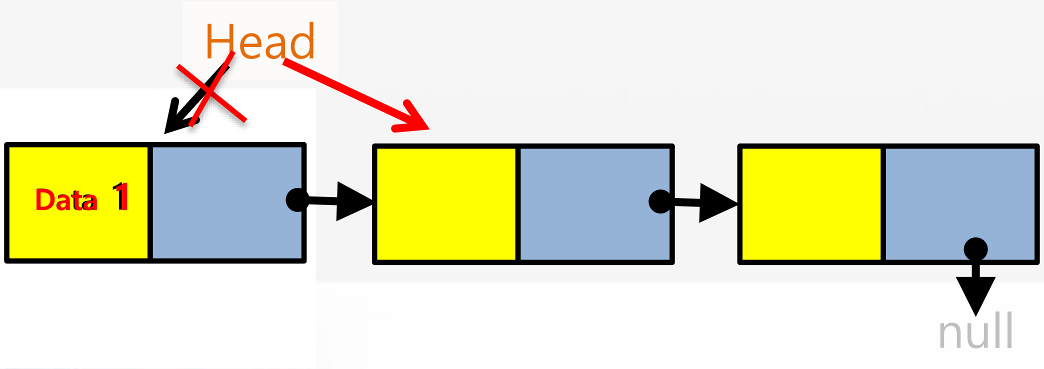
```
public void addLast (value)
{
    if (this.isEmpty())
      this.addFirst(value);
    else
    {
      SLLNode newNode = new SLLNode (value);
      SLLNode currentNode = this.getTail();
       currentNode.setNext(newNode);
    }
}
```



Singly Linked List

removeFirst ( )

1. Check if the List is not empty
   (if empty, stop here)
2. Get the data stored in the Head
3. Let the Head refers to the "<u>next node</u>" of the Head node
4. Return the stored data



SLL

- *SLLNode* **Head**
- ~~*SLLNode* **Tail**~~
- ~~*int* **Size**~~

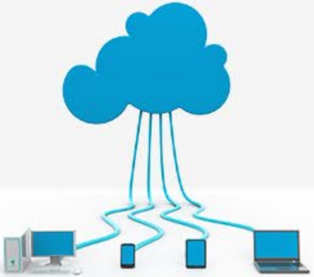
+ **SLL** ( )
+ **SLL** (*Anytype* value)
+ **SLL** (*SLLNode* FirstNode)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *SLLNode* **getHead** ( )
+ *SLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )

Singly Linked List

removeFirst ( )

```
public Anytype removeFirst()
 {
     if (this.isEmpty())
         return null;

     Anytype removedValue = this.Head.getData();
     this.Head = this.Head.getNext();

     return removedValue;
  }
```



Singly Linked List

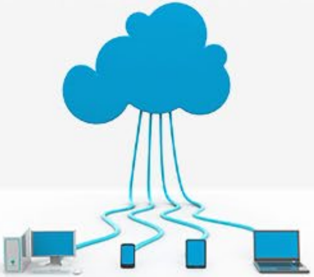
removeLast ( )

SLL

- *SLLNode* **Head**
- ~~*SLLNode* **Tail**~~
- ~~*int* **Size**~~

+ **SLL** ( )
+ **SLL** (*Anytype* value)
+ **SLL** (*SLLNode* FirstNode)

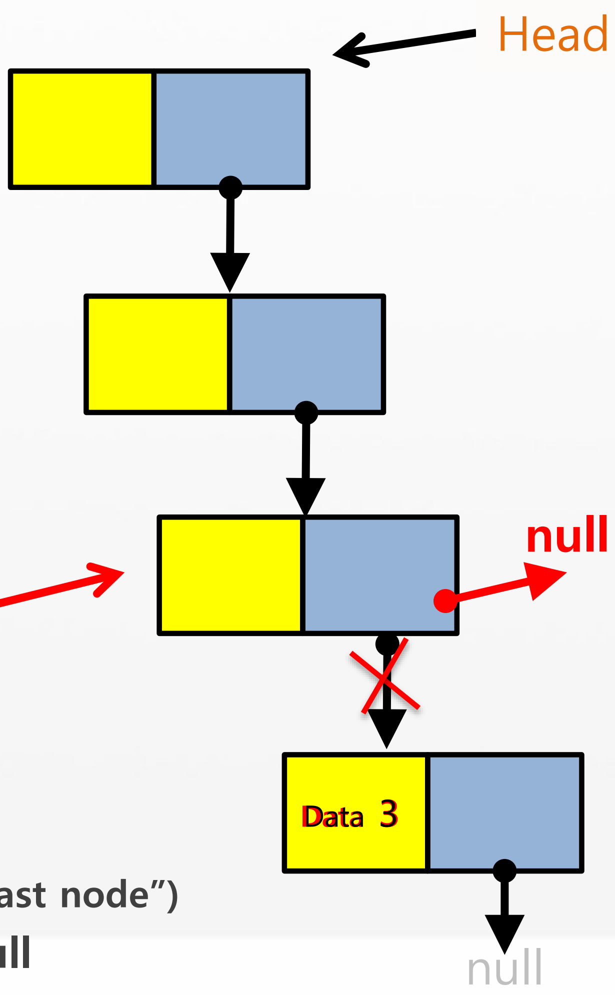
+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *SLLNode* **getHead** ( )
+ *SLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )



Singly Linked List

removeLast ( )

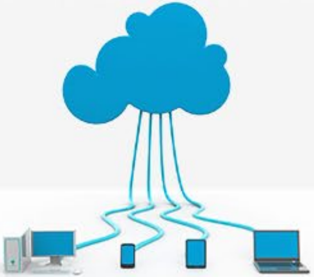
1. Check if the List is not empty
   (if empty, stop here)
2. Check if the List has only one node, if so:
   a) Get the data stored in the Head
   b) Let the Head refers to null
   c) Return the stored data (& stop here)
3. If more than a node exits:
   a) Find the "before last node": the node having its "next node" is the Tail itself (i.e. the node having its "next node" has a null "next node")
   b) Get the data stored in the Tail (next of "before last node")
   c) Set the "before last node" next node to be null
   d) Return the stored data



Singly Linked List

removeLast ( )

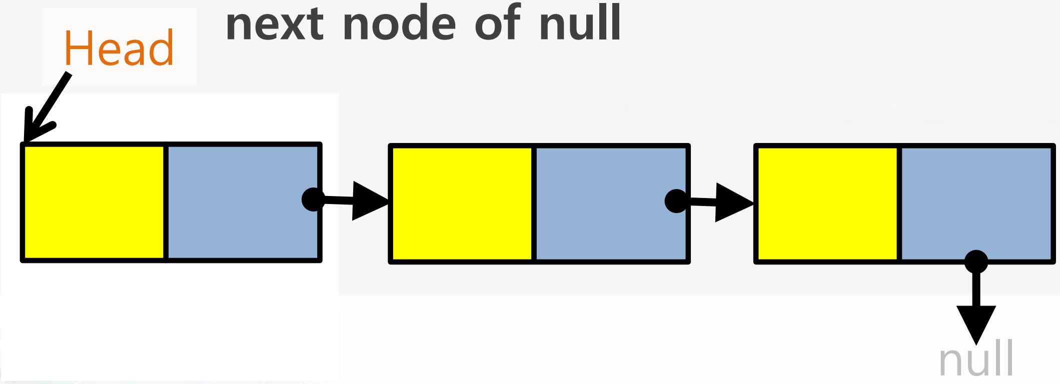
```
public Anytype removeLast() {
   if (this.isEmpty())                          // Size = 0
       return null;

   Anytype removedValue;
   if (Head.getNext()==null) {                  // Size = 1
       removedValue = Head.getData();
       Head = null; }
   else {                                       // Size > 1
       SLLNode<Anytype> currentNode=this.Head;
       while (currentNode.getNext().getNext() != null)
              currentNode = currentNode.getNext();

       removedValue = currentNode.getNext().getData();
       currentNode.setNext(null); }
   return removedValue; }
```



Singly Linked List

Print ( )

1. Check if List is not empty
   (if empty, stop here)
2. Starting from Head node:
3. Print the Data stored in the node
4. Go to next node
5. Repeat step 3 & 4 until you reach a next node of null



SLL

- *SLLNode* **Head**
- ~~*SLLNode* **Tail**~~
- ~~*int* **Size**~~

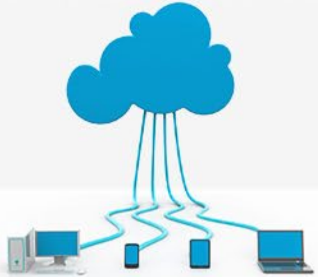
+ **SLL** ( )
+ **SLL** (*Anytype* value)
+ **SLL** (*SLLNode* FirstNode)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *SLLNode* **getHead** ( )
+ *SLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )

Singly Linked List

Print ( )

```
public void Print() {
  if (this.isEmpty())
     System.out.println("The list is empty.");
  else
  {
     SLLNode currentNode = this.Head;

     while (currentNode != null)
     {
        System.out.print(currentNode.getData() + " --> ");
        currentNode = currentNode.getNext();
     }

     System.out.println("");
  }
}
```



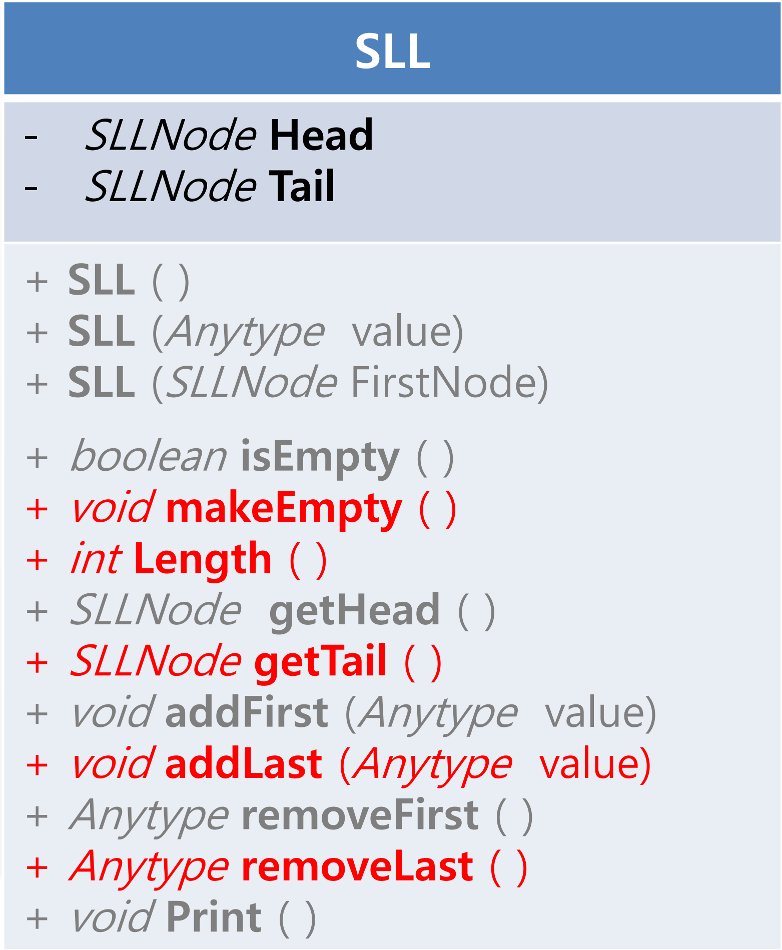
Singly Linked List

Exercise 1

a) Write the appropriate implementations for all of the class operations marked in ***red*** taking into consideration that the class has only 2 attributes:

- SLLNode **Head**;
- SLLNode **Tail**;

b) What is the complexity (in the form of Big-*Oh* notation) for each operation

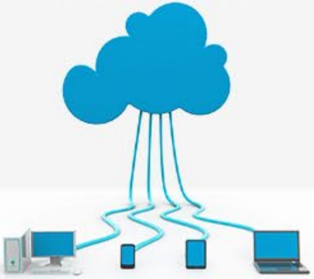


SLL

- *SLLNode* **Head**
- *SLLNode* **Tail**

+ **SLL** ( )
+ **SLL** (*Anytype* value)
+ **SLL** (*SLLNode* FirstNode)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *SLLNode* **getHead** ( )
+ *SLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )



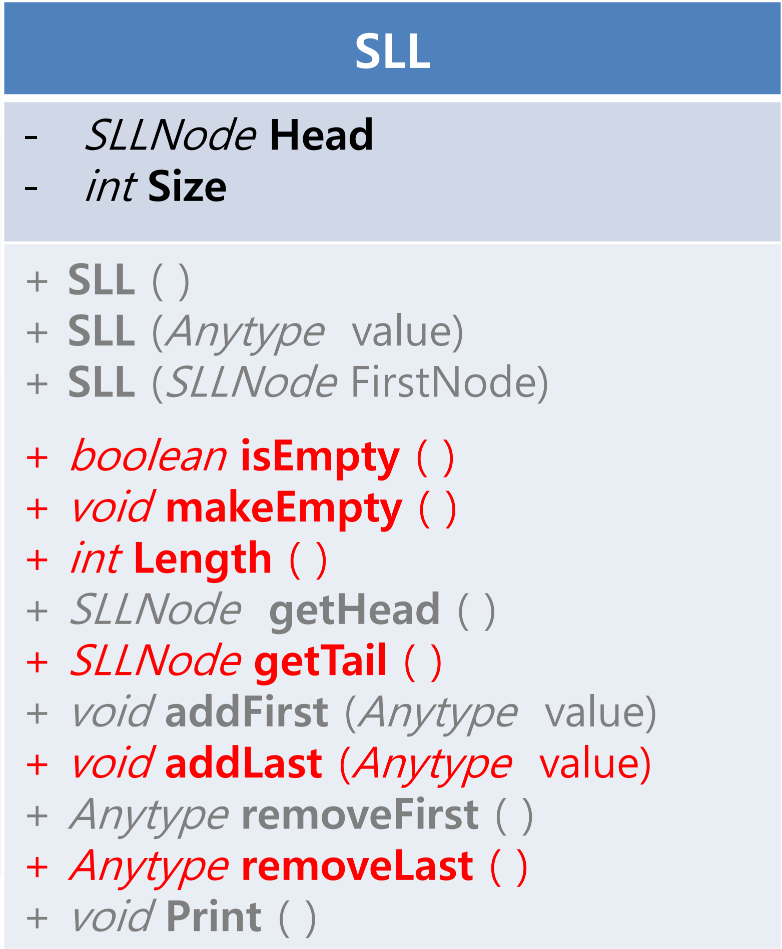
Singly Linked List

Exercise 2

a) Write the appropriate implementations for all of the class operations marked in ***red*** taking into consideration that the class has only 2 attributes:

- SLLNode **Head**;
- int **Size**;

b) What is the complexity (in the form of Big-*Oh* notation) for each operation

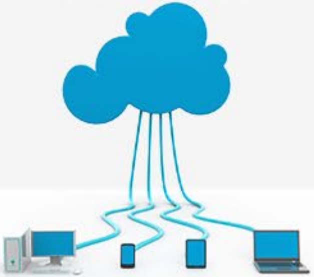


SLL

- *SLLNode* **Head**
- *int* **Size**

+ **SLL** ( )
+ **SLL** (*Anytype* value)
+ **SLL** (*SLLNode* FirstNode)

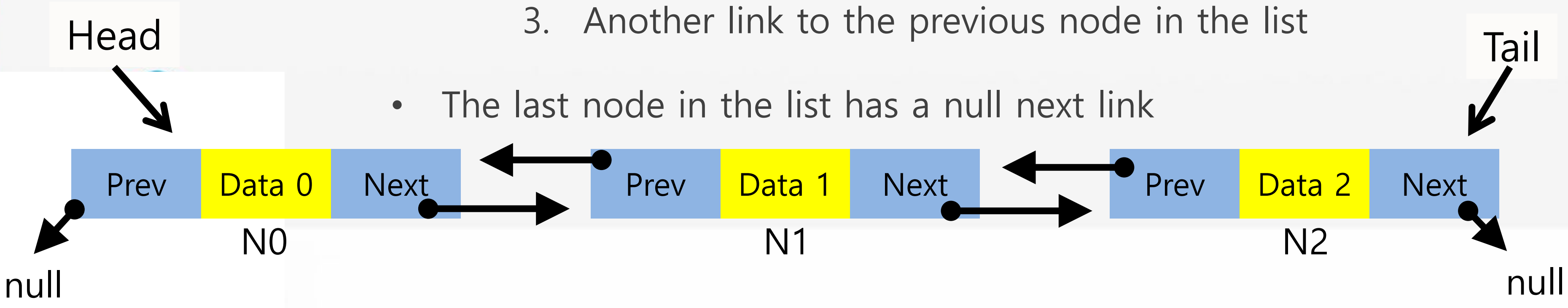
+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *SLLNode* **getHead** ( )
+ *SLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )



Doubly Linked List

DLL

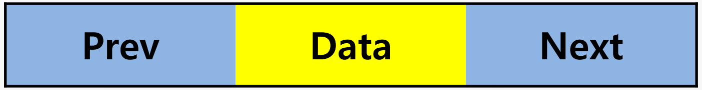
- The basic linked list consists of a collection of connected dynamically allocated nodes
- In a *doubly linked list*, each node consists of:
    1. Data element (value)
    2. One Link (reference) to the next node in the list
    3. Another link to the previous node in the list
- The last node in the list has a null next link



Doubly Linked List

DLL-Node operations

DLL Node



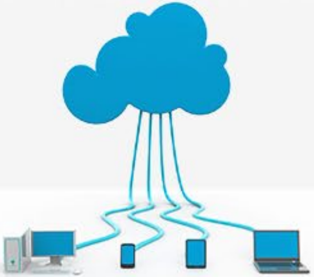
Prev	Data	Next

DLLNode

- *Anytype* **Data**
- *DLLNode* **Prev**
- *DLLNode* **Next**

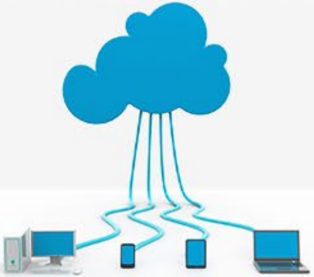
+ **DLLNode** ( )
+ **DLLNode** (*Anytype* Data)

+ *Anytype* **getData** ( )
+ *void* **setData** (*Anytype* Data)
+ *DLLNode* **getPrev** ( )
+ *void* **setPrev** (*DLLNode* prevNode)
+ *DLLNode* **getNext** ( )
+ *void* **setNext** (*DLLNode* nextNode)



Doubly Linked List

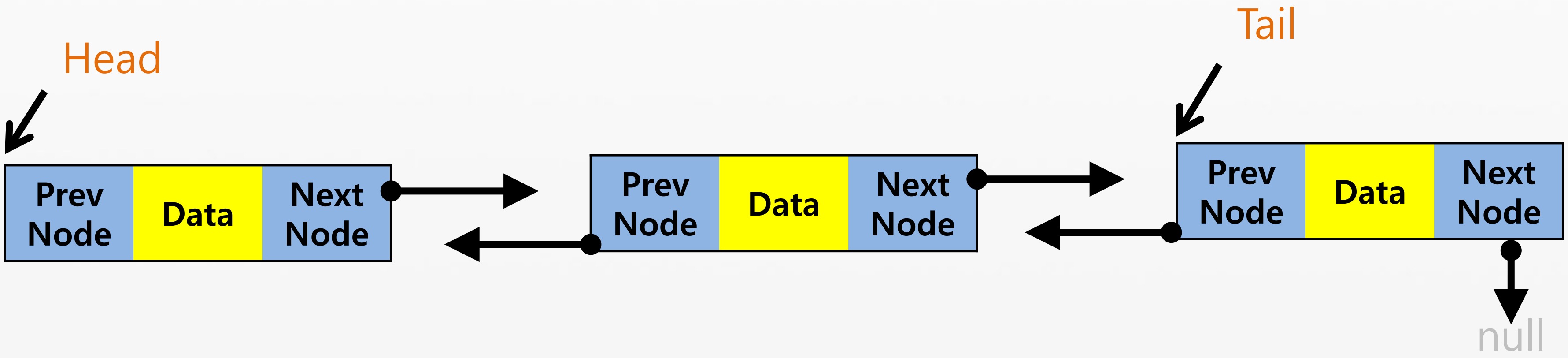
DLL operations



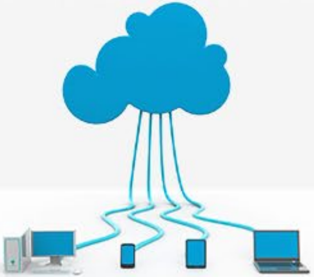
DLL
- *DLLNode* **Head** - *DLLNode* **Tail** - *int* **Size**
+ **DLL** ( ) + **DLL** (*Anytype* value) + **DLL** (*DLLNode* FirstNode) + *boolean* **isEmpty** ( ) + *void* **makeEmpty** ( ) + *int* **Length** ( ) + *DLLNode* **getHead** ( ) + *DLLNode* **getTail** ( ) + *void* **addFirst** (*Anytype* value) + *void* **addLast** (*Anytype* value) + *Anytype* **removeFirst** ( ) + *Anytype* **removeLast** ( ) + *void* **Print** ( )

Doubly Linked List

DLL operations



Size = 3

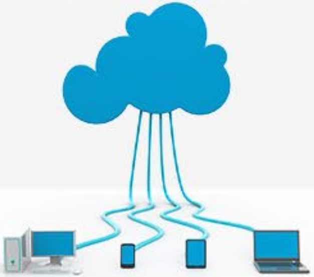


Doubly Linked List

isEmpty ( )

- ✓ Check if Head is null

Same as SLL



DLL

- *DLLNode* **Head**
- ~~*DLLNode* **Tail**~~
- ~~*int* **Size**~~

+ **DLL** ( )
+ **DLL** (*Anytype* value)
+ **DLL** (*SLLNode* FirstNode)

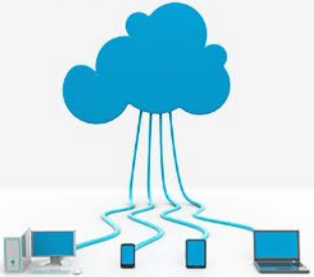
+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *DLLNode* **getHead** ( )
+ *DLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )

Doubly Linked List

makeEmpty ( )

- ✓ Let Head be null

Same as SLL



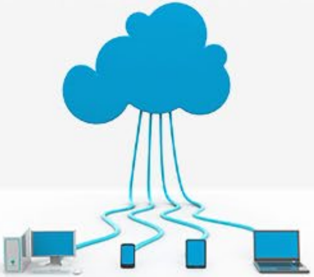
DLL
- *DLLNode* **Head** - ~~*DLLNode* **Tail**~~ - ~~*int* **Size**~~
+ **DLL** ( ) + **DLL** (*Anytype* value) + **DLL** (*SLLNode* FirstNode) + *boolean* **isEmpty** ( ) + *void* **makeEmpty** ( ) + *int* **Length** ( ) + *DLLNode* **getHead** ( ) + *DLLNode* **getTail** ( ) + *void* **addFirst** (*Anytype* value) + *void* **addLast** (*Anytype* value) + *Anytype* **removeFirst** ( ) + *Anytype* **removeLast** ( ) + *void* **Print** ( )

Doubly Linked List

Length ( )

- ✓ Count Nodes

Same as SLL



null

DLL

- *DLLNode* **Head**
- ~~*DLLNode* **Tail**~~
- ~~*int* **Size**~~

+ **DLL** ( )
+ **DLL** (*Anytype* value)
+ **DLL** (*SLLNode* FirstNode)

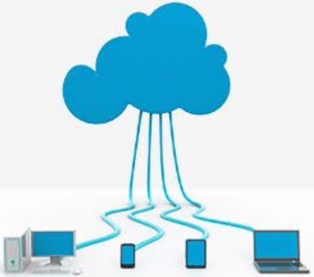
+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *DLLNode* **getHead** ( )
+ *DLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )

Doubly Linked List

getTail ( )

- ✓ Find Last Node -Tail-

Same as SLL



DLL

- *DLLNode* **Head**
- ~~*DLLNode* **Tail**~~
- ~~*int* **Size**~~

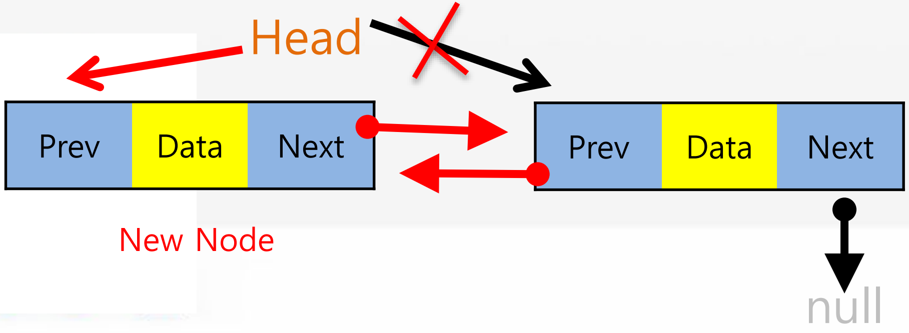
+ **DLL** ( )
+ **DLL** (*Anytype* value)
+ **DLL** (*SLLNode* FirstNode)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *DLLNode* **getHead** ( )
+ *DLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )

Doubly Linked List

addFirst (value)

1. Create a new node
2. Check if the List is not empty
   (if empty, just set the Head to be the new node)
3. Let the "next node" of the new node be the "Head" node
4. Let the "prev node" of the Head be the new node
5. Let "Head" refer to the new node



DLL

- *DLLNode* **Head**
- ~~*DLLNode* **Tail**~~
- ~~*int* **Size**~~

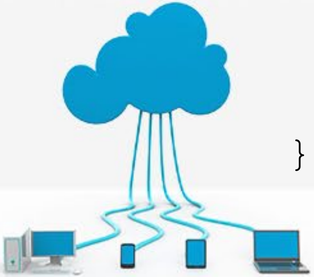
+ **DLL** ( )
+ **DLL** (*Anytype* value)
+ **DLL** (*SLLNode* FirstNode)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *DLLNode* **getHead** ( )
+ *DLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )

Doubly Linked List

addFirst (value)

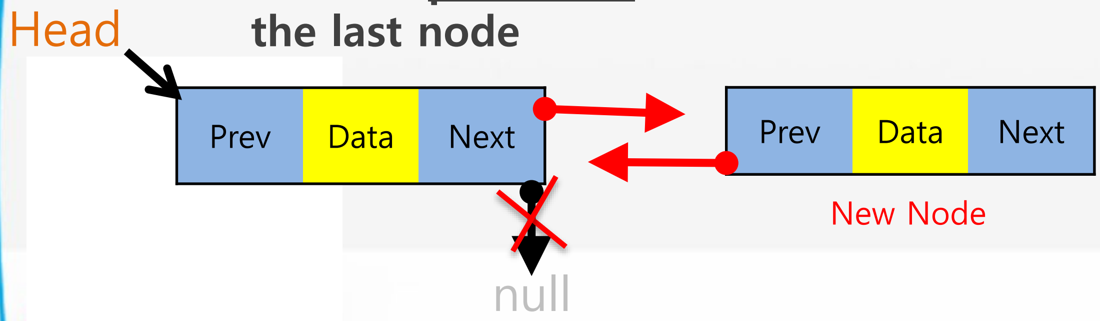
```
public void addFirst(Anytype value)
{
    if ( isEmpty() )
          this.Head=new DLLNode<Anytype>(value);
    else
    {
        DLLNode newNode = new DLLNode(value, null, this.Head);
        this.Head.setPrev(newNode);
        this.Head=newNode;
    }
 }
```



Doubly Linked List

addLast (value)

1. Create a new node
2. Check if the List is not empty
   (if empty, just set the Head to be the new node)
3. Find the last node (getTail)
4. Let the "<u>next node</u>" of the last node be the new node
5. Let the "<u>prev node</u>" of the new node be the last node



DLL

- *DLLNode* **Head**
- ~~*DLLNode* **Tail**~~
- ~~*int* **Size**~~

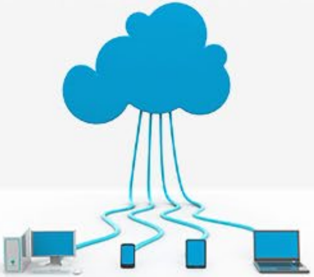
+ **DLL** ( )
+ **DLL** (*Anytype* value)
+ **DLL** (*SLLNode* FirstNode)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *DLLNode* **getHead** ( )
+ *DLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )

Doubly Linked List

addLast (value)

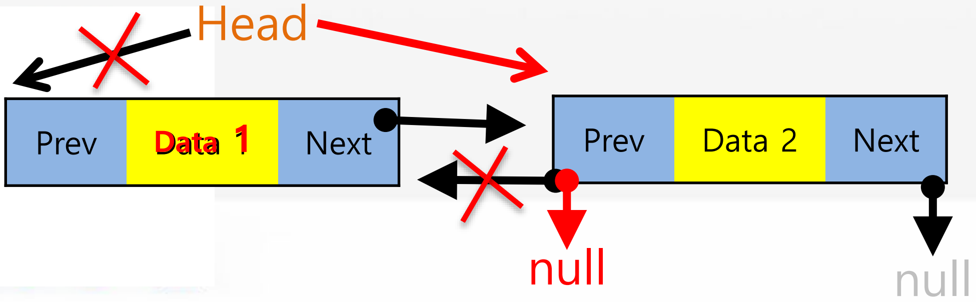
```
public void addLast(Anytype value)
{
   if (this.isEmpty())
     this.addFirst(value);
   else
   {
   DLLNode<Anytype> newNode = new DLLNode<Anytype>(value);
      DLLNode<Anytype> Tail = this.getTail();
      Tail.setNext(newNode);
      newNode.setPrev(Tail);
   }
 }
```



Doubly Linked List

removeFirst ( )

1. Check if the List is not empty
   (if empty, stop here)
2. Get the data stored in the Head
3. Let the Head refers to the "next node" of the Head node
4. Set the previous node of the new Head to be null
5. Return the stored data



DLL

- *DLLNode* **Head**
- ~~*DLLNode* **Tail**~~
- ~~*int* **Size**~~

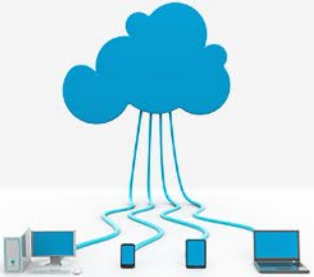
+ **DLL** ( )
+ **DLL** (*Anytype* value)
+ **DLL** (*SLLNode* FirstNode)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *DLLNode* **getHead** ( )
+ *DLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )

Doubly Linked List

removeFirst ( )

```
public Anytype removeFirst()
{
   if (this.isEmpty())
       return null;

   Anytype removedValue = this.Head.getData();
   if (Head.getNext()==null)
      Head = null;
else {
      this.Head = this.Head.getNext();
      this.Head.setPrev(null);
  }
   return removedValue;
 }
```



Doubly Linked List

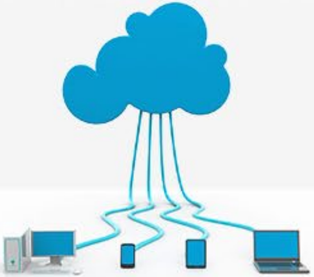
removeLast ( )

DLL

- *DLLNode* **Head**
- ~~*DLLNode* **Tail**~~
- ~~*int* **Size**~~

+ **DLL** ( )
+ **DLL** (*Anytype* value)
+ **DLL** (*SLLNode* FirstNode)

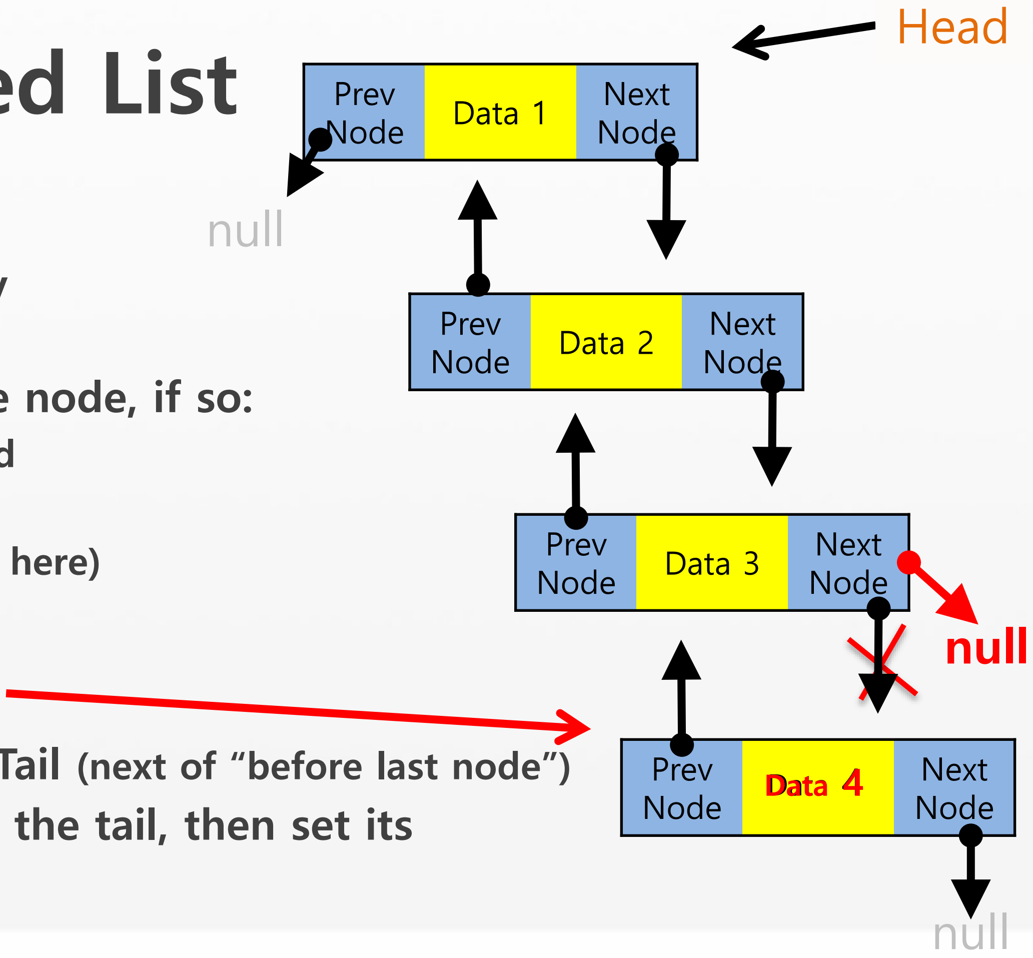
+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *DLLNode* **getHead** ( )
+ *DLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )



Doubly Linked List

removeLast ( )

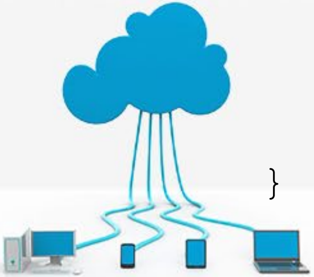
1. Check if the List is not empty
   (if empty, stop here)
2. Check if the List has only one node, if so:
   a) Get the data stored in the Head
   b) Let the Head refers to null
   c) Return the stored data (& stop here)
3. If more than a node exits:
   a) Get the "<u>last node</u>" (Tail)
   b) Get the data stored in the Tail (next of "before last node")
   c) Catch the previous node of the tail, then set its
      "Next Node" to be null
   d) Return the stored data



Doubly Linked List

removeLast ( )

```
public Anytype removeLast() {
   if (this.isEmpty())                      // Size = 0
       return null;

   Anytype removedValue;
   if (Head.getNext()==null) {              // Size = 1
       removedValue = Head.getData();
       Head = null; }
   else {                                   // Size > 1
       DLLNode<Anytype> currentNode=getTail();
       removedValue = currentNode.getData();
       currentNode.getPrev().setNext(null);
   }
   return removedValue;
}
```



Doubly Linked List

Print ( )

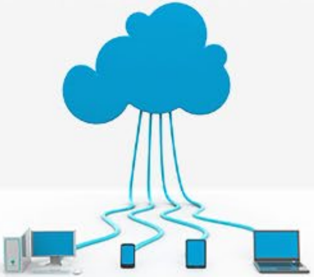
Same as SLL

DLL

- *DLLNode* **Head**
- ~~*DLLNode* **Tail**~~
- ~~*int* **Size**~~

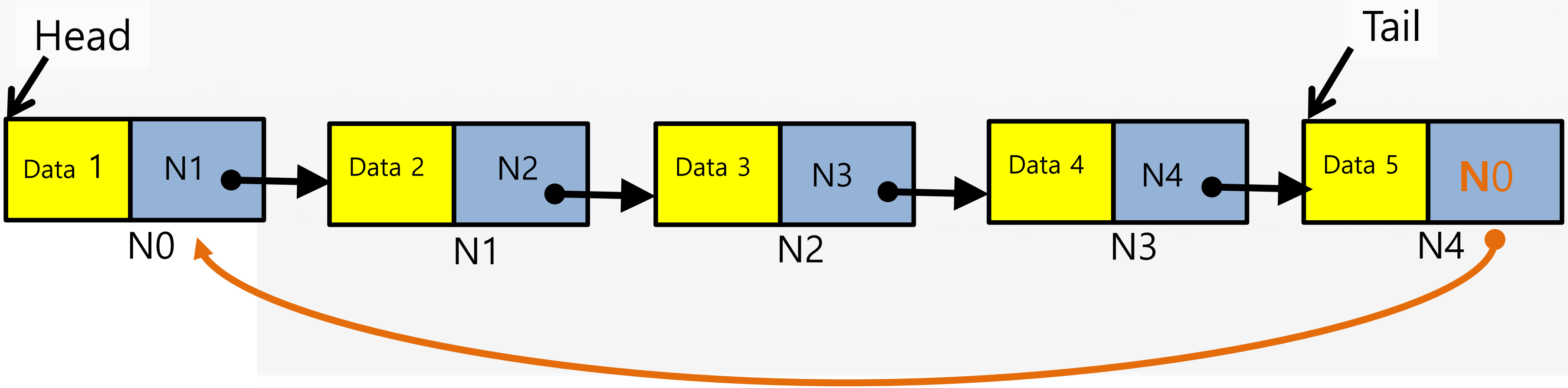
+ **DLL** ( )
+ **DLL** (*Anytype* value)
+ **DLL** (*SLLNode* FirstNode)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *DLLNode* **getHead** ( )
+ *DLLNode* **getTail** ( )
+ *void* **addFirst** (*Anytype* value)
+ *void* **addLast** (*Anytype* value)
+ *Anytype* **removeFirst** ( )
+ *Anytype* **removeLast** ( )
+ *void* **Print** ( )



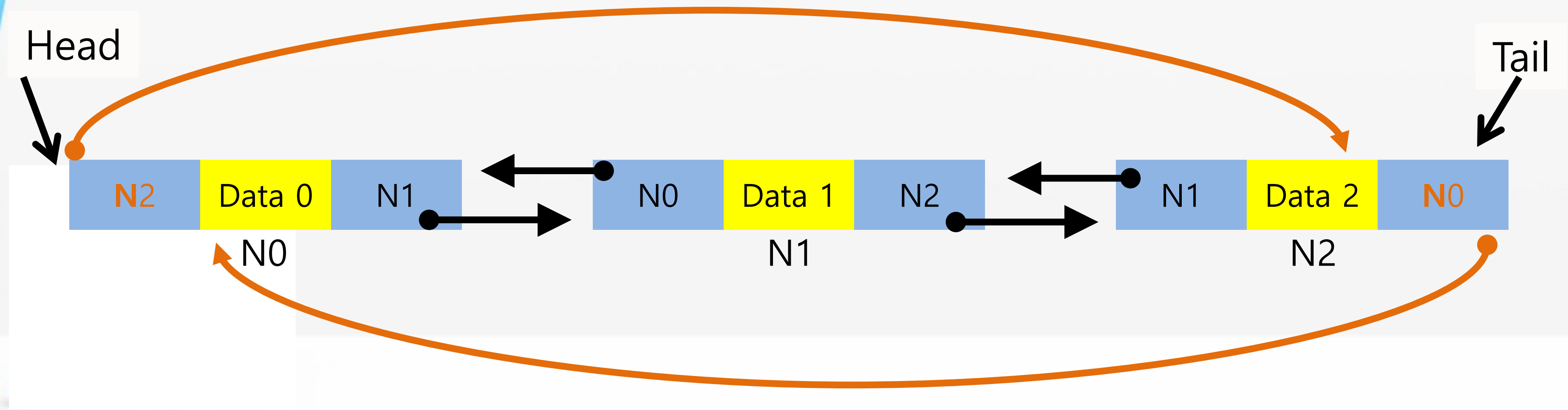
Singly Circular Linked List

- It's a Singly Linked List
- The last node has a reference to the first node



Doubly Circular Linked List

- It’s a Doubly Linked List
- The last node has a reference to the first node
- The first node has a reference to the last node


Head
Tail
N2
Data 0
N1
N0
Data 1
N2
N1
Data 2
N0
N0
N1
N2