Lecture No.5

Data Structures & Algorithms

Linked Lists

Introduction

The sequential storage of array is not either possible or efficient in larger computer systems because:

1. Where a number of users share main memory there may not be enough adjacent memory locations left to hold an array. But there could be enough memory in the shape of small free blocks.
2. Array is a static data structure. The size of array cannot be changed during the program execution.

To overcome these limitations, linked lists are used.

In linked lists, data is arranged into records. Each record is called a node.

A data item may be stored anywhere in the memory. A pointer, is added to each record. It is used to contain a link to the next record. Thus an item in the list is linked logically with the next item by assigning to it the memory address of next item;

There are three types of linked lists.

- Single or One-way lists
- Double-linked lists or two-way lists
- Circular-linked list

Single Linked List

A single linked list is a linear collection of data elements. The elements in a single linked list can be visited starting from beginning and towards its end, i.e. only in one direction. Therefore, the single linked list is also called one-way list or one-way chain.

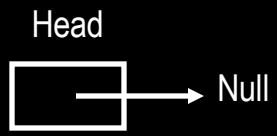
Each node in a single linked list consists of at least two fields.

1. The first field contains the data or value field.
2. The second field contains the pointer or link to the next node in the list. This field is called the linked field or pointer field.

The linked field of the last node contains a NULL value.

A NULL value in the pointer field indicates that pointer does not point to any other node. The single linked list that has no node is called empty list It has a value NULL

Single Linked List



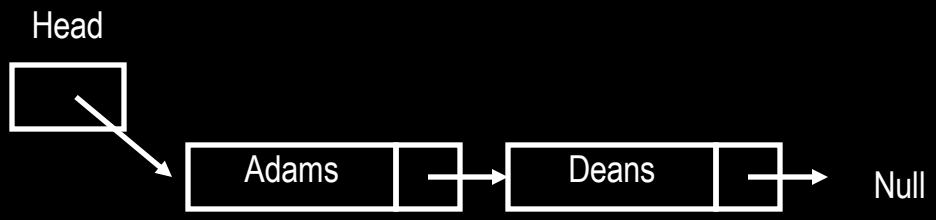
Bed	Patient	Link
01		
02		
03		
04		
05		
06		
07		
08		
09		
10		
11		
12		

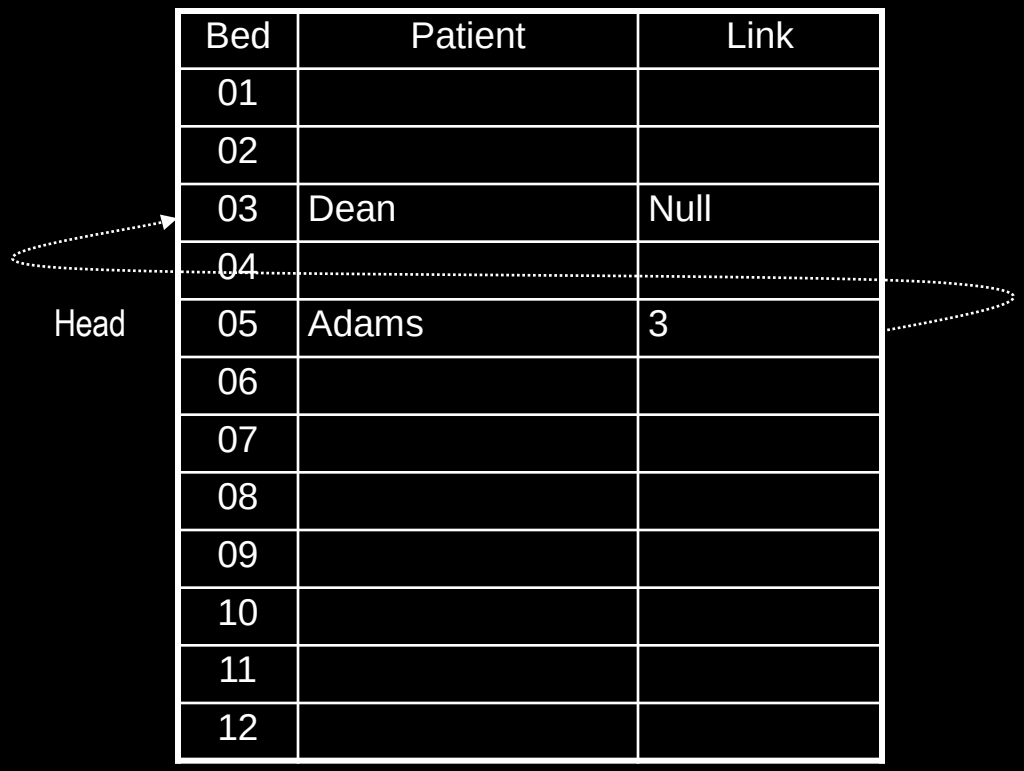
Single Linked List

Head

Bed	Patient	Link
01		
02		
03		
04		
05	Adams	Null
06		
07		
08		
09		
10		
11		
12		

Single Linked List





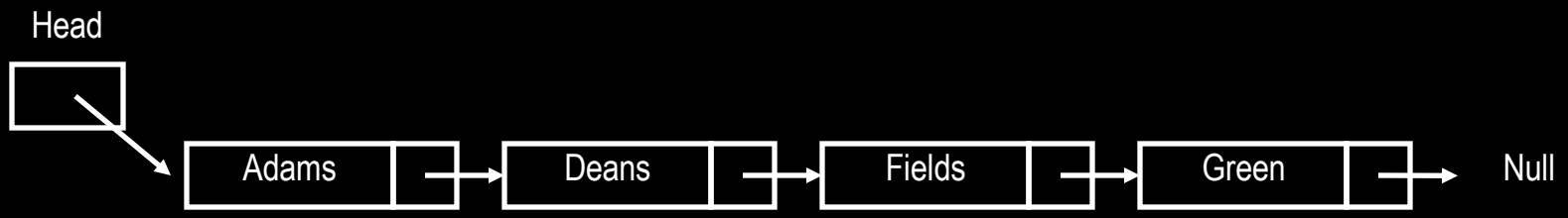
Bed	Patient	Link
01		
02		
03	Dean	Null
04		
05	Adams	3
06		
07		
08		
09		
10		
11		
12		

Single Linked List

Head

Bed	Patient	Link
01		
02		
03	Dean	11
04		
05	Adams	3
06		
07		
08		
09		
10		
11	Fields	Null
12		

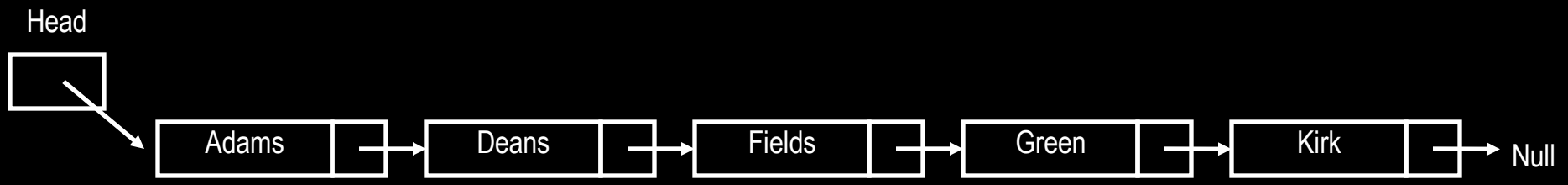
Single Linked List



Head

Bed	Patient	Link
01		
02		
03	Dean	11
04		
05	Adams	3
06		
07		
08	Green	Null
09		
10		
11	Fields	8
12		

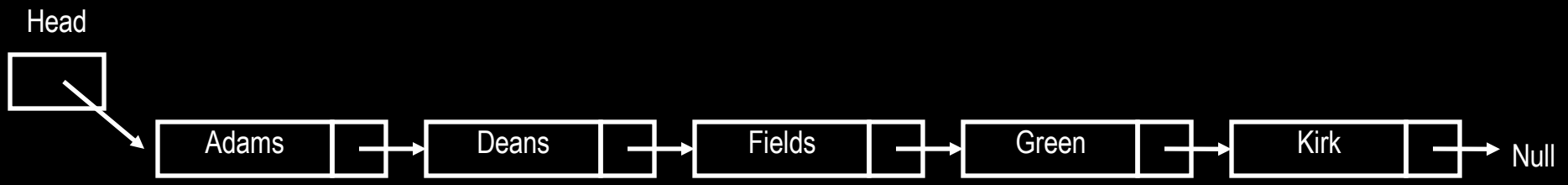
Single Linked List

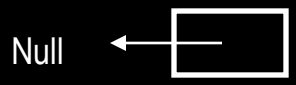


Head

Bed	Patient	Link
01	Kirk	Null
02		
03	Dean	11
04		
05	Adams	3
06		
07		
08	Green	1
09		
10		
11	Fields	8
12		

Single Linked List



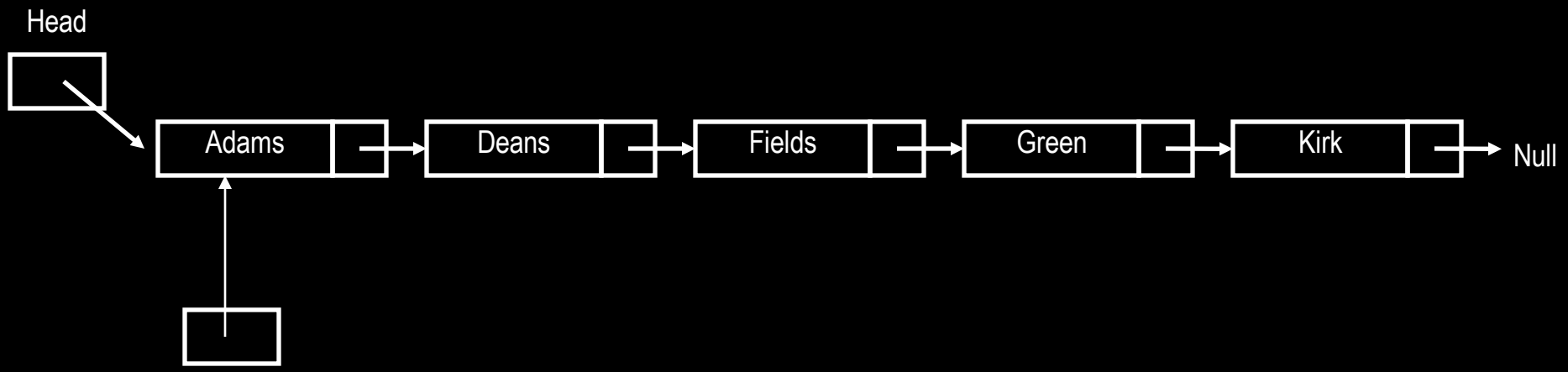


Current

Head

Bed	Patient	Link
01	Kirk	Null
02		
03	Dean	11
04		
05	Adams	3
06		
07		
08	Green	1
09		
10		
11	Fields	8
12		

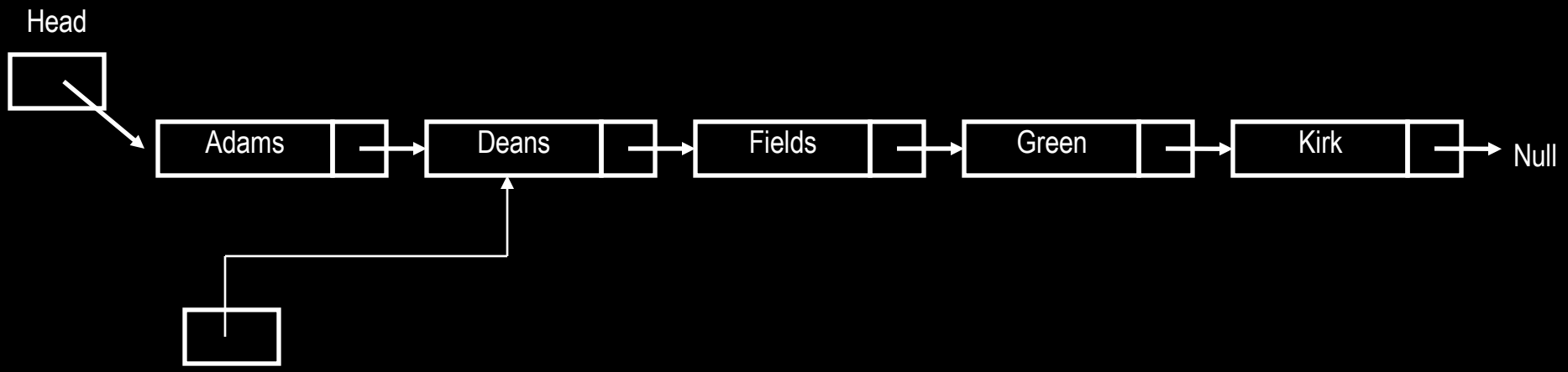
Single Linked List



Head

Bed	Patient	Link
01	Kirk	Null
02		
03	Dean	11
04		
05	Adams	3
06		
07		
08	Green	1
09		
10		
11	Fields	8
12		

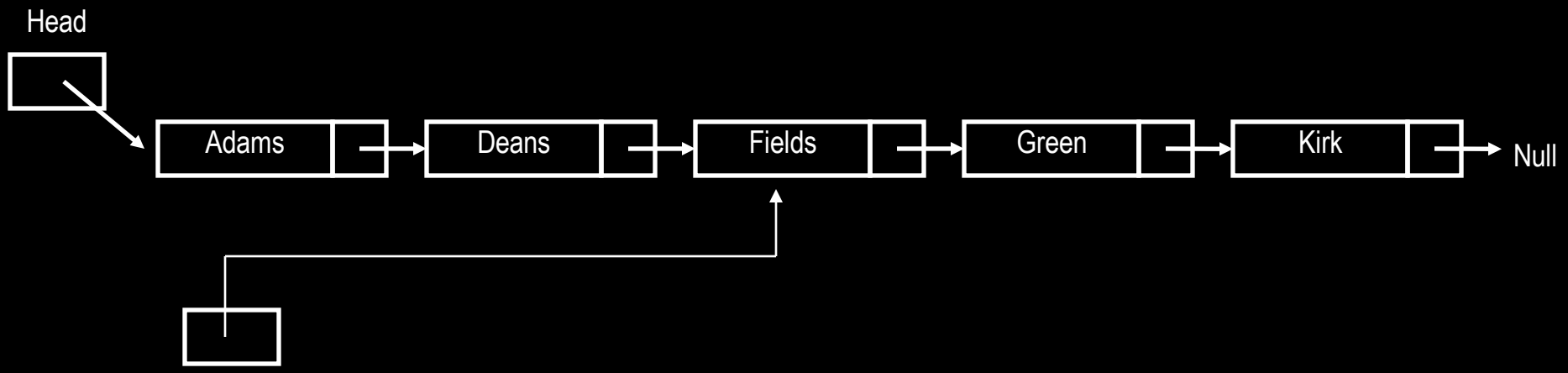
Single Linked List



Head

Bed	Patient	Link
01	Kirk	Null
02		
03	Dean	11
04		
05	Adams	3
06		
07		
08	Green	1
09		
10		
11	Fields	8
12		

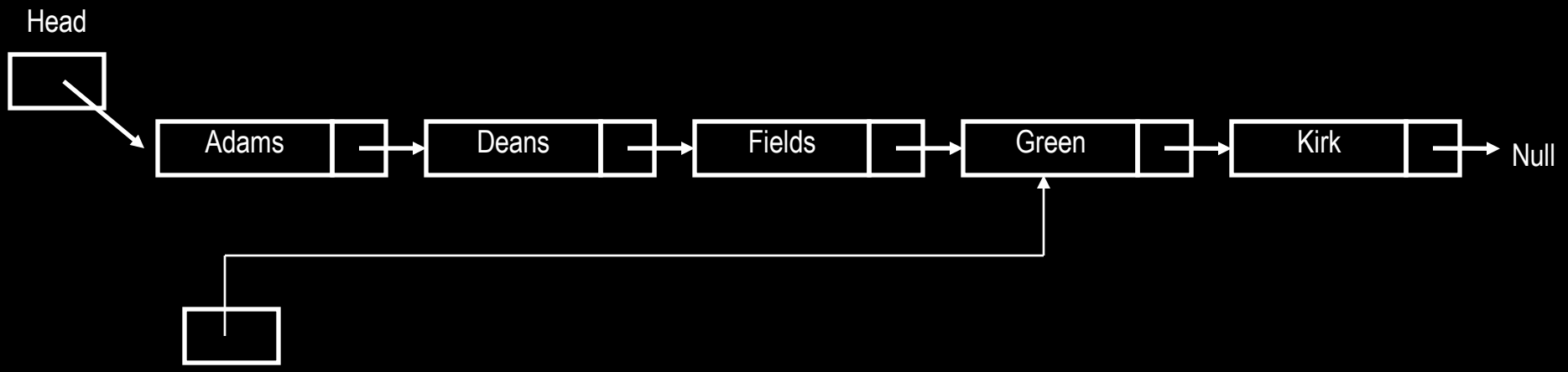
Single Linked List



Head

Bed	Patient	Link
01	Kirk	Null
02		
03	Dean	11
04		
05	Adams	3
06		
07		
08	Green	1
09		
10		
11	Fields	8
12		

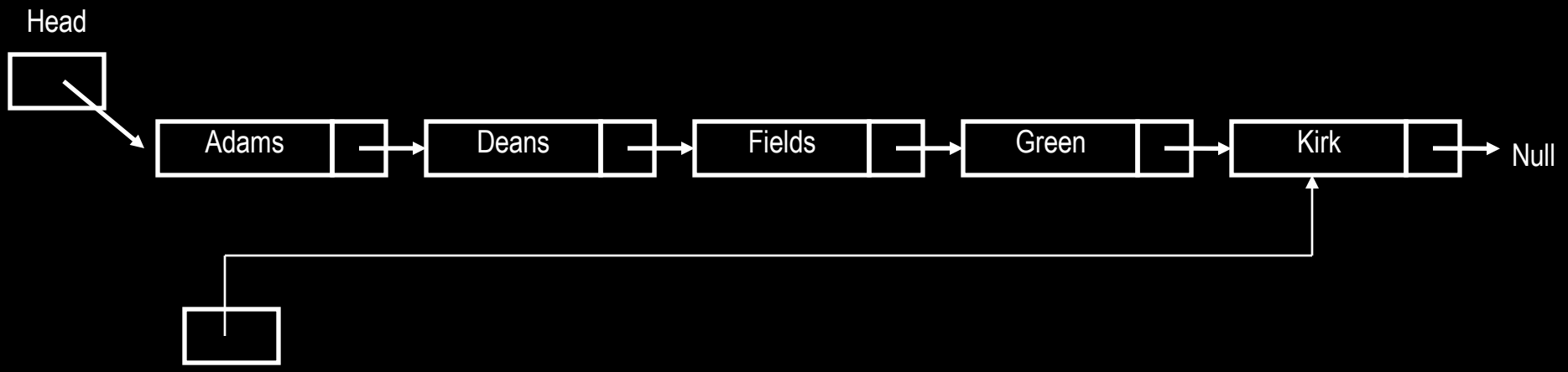
Single Linked List



Bed	Patient	Link
01	Kirk	Null
02		
03	Dean	11
04		
05	Adams	3
06		
07		
08	Green	1
09		
10		
11	Fields	8
12		

Head

Single Linked List



Bed	Patient	Link
01	Kirk	Null
02		
03	Dean	11
04		
05	Adams	3
06		
07		
08	Green	1
09		
10		
11	Fields	8
12		

Head

Linked List

Note some features of the list:

- Need a *head* to point to the first node of the list. Otherwise we won't know where the start of the list is.
- The *current* here is a pointer, not an index, used as a cursor which can be used for insertion and deletion.
- The next field in the last node points to *nothing*. We will place the memory address NULL which is guaranteed to be inaccessible.

The LIST Data Structure

- The List is among the most generic of data structures.
- Real life:

  a. shopping list,
  b. groceries list,
  c. list of people to invite to dinner
  d. List of presents to get

Lists

- A list is collection of items that are all of the same type (grocery items, integers, names)
- The items, or elements of the list, are stored in some particular order
- It is possible to insert new elements into various positions in the list and remove any element of the list

Lists

- List is a set of elements in a linear order. For example, data values a_1, a_2, a_3, a_4 can be arranged in a list:

  (a_3, a_1, a_2, a_4)

  In this list, a_3, is the first element, a_1 is the second element, and so on

- The order is important here; this is not just a random collection of elements, it is an *ordered* collection

List Operations

Useful operations

- createList(): create a new list (presumably empty)
- copy(): set one list to be a copy of another
- clear(); clear a list (remove all elments)
- insert(X, ?): Insert element X at a particular position in the list
- remove(?): Remove element at some position in the list
- get(?): Get element at a given position
- update(X, ?): replace the element at a given position with X
- find(X): determine if the element X is in the list
- length(): return the length of the list.

List Operations

- We need to decide what is meant by “particular position”; we have used “?” for this.

- There are two possibilities:

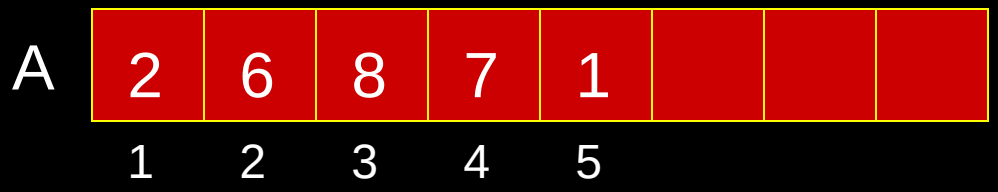
1. Use the actual index of element: insert after element 3, get element number 6. This approach is taken by arrays
2. Use a “current” marker or pointer to refer to a particular position in the list.

List Operations

- If we use the “current” marker, the following four methods would be useful:
  - **start()**: moves to “current” pointer to the very first element.
  - **tail()**: moves to “current” pointer to the very last element.
  - **next**(): move the current position forward one element
  - **back**(): move the current position backward one element

Implementing Lists

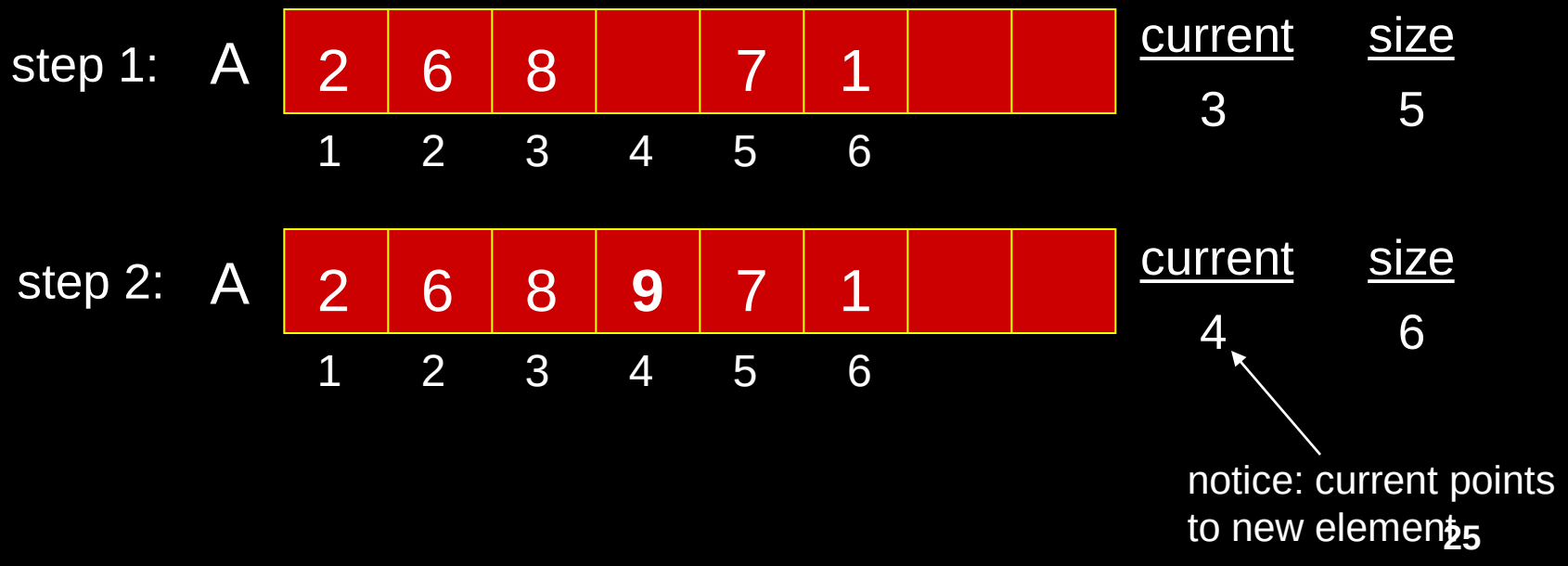
- We have designed the interface for the List; we now must consider how to implement that interface.
- Implementing Lists using an array: for example, the list of integers (2, 6, 8, 7, 1) could be represented as:



current	size
3	5

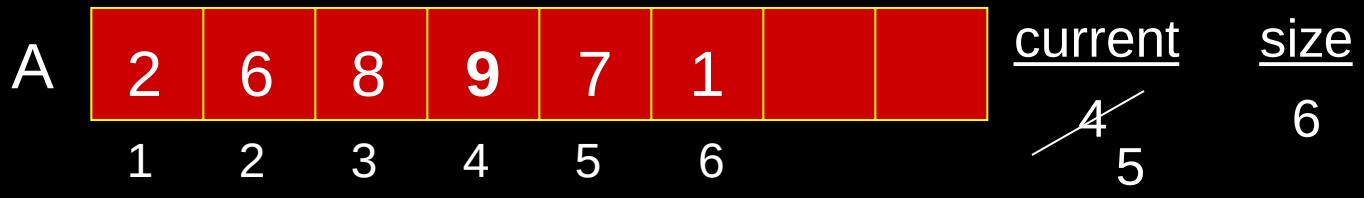
List Implementation

- add(9); current position is 3. The new list would thus be: (2, 6, 8, 9, 7, 1)
- We will need to *shift* everything to the right of 8 one place to the right to make place for the new element ‘9’.



Implementing Lists

- next():



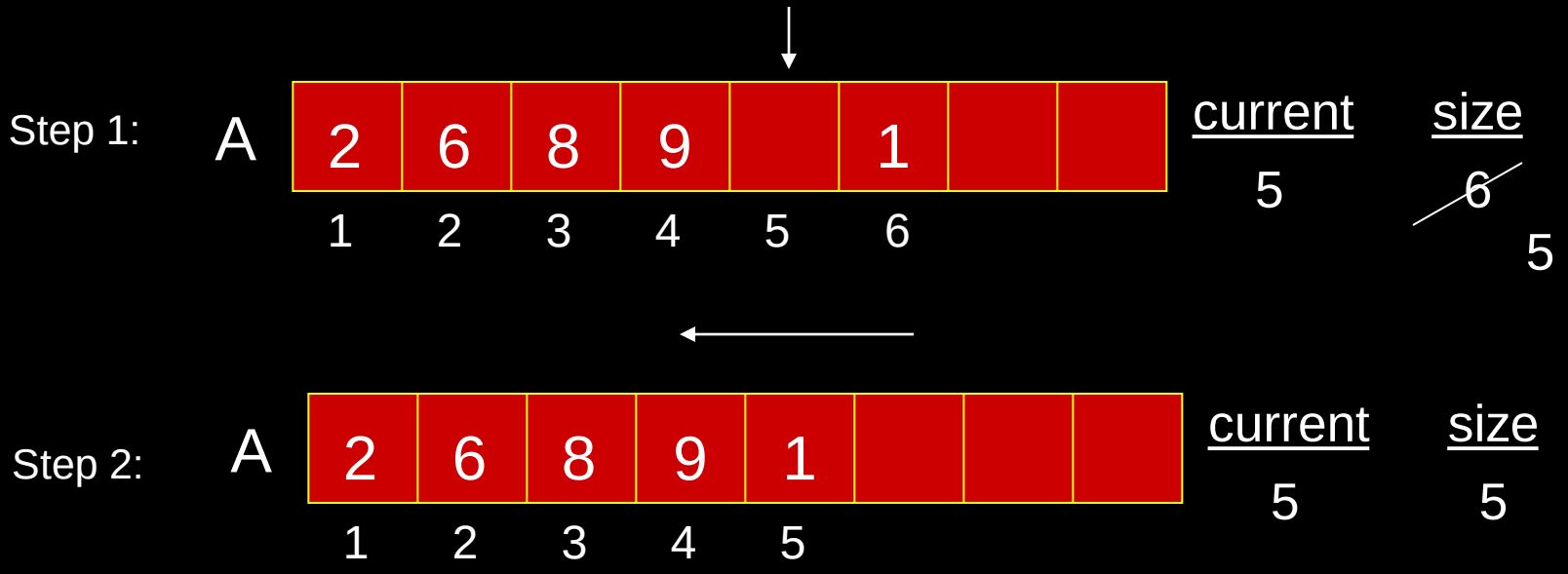
Implementing Lists

- There are special cases for positioning the current pointer:
  a. past the last array cell
  b. before the first cell

- We will have to worry about these when we write the actual code.

Implementing Lists

- remove(): removes the element at the current index



- We fill the blank spot left by the removal of 7 by shifting the values to the right of position 5 over to the left one space.

Implementing Lists

find(X): traverse the array until X is located.

```
int find(int X)
{
    int j;
    for(j=1; j < size+1; j++ )
        if( A[j] == X ) break;

    if( j < size+1 ) {        // found X
        current = j;        // current points to where X found
        return 1;   // 1 for true
    }
    return 0;  // 0 (false) indicates not found
}
```

Implementing Lists

- Other operations:

get()	→ return A[current];
update(X)	→ A[current] = X;
length()	→ return size;
back()	→ current--;
start()	→ current = 1;
end()	→ current = size;

Analysis of Array Lists

- add
  - we have to move every element to the right of current to make space for the new element.
  - Worst-case is when we insert at the beginning; we have to move every element right one place.
  - Average-case: on average we may have to move half of the elements

Analysis of Array Lists

- remove
  - Worst-case: remove at the beginning, must shift all remaining elements to the left.
  - Average-case: expect to move half of the elements.
- find
  - Worst-case: may have to search the entire array
  - Average-case: search at most half the array.
- Other operations are one-step.

List Using Linked Memory

- Various cells of memory are not allocated consecutively in memory.
- Not enough to store the elements of the list.
- With arrays, the second element was right next to the first element.
- Now the first element must explicitly tell us where to look for the second element.
- Do this by holding the memory address of the second element

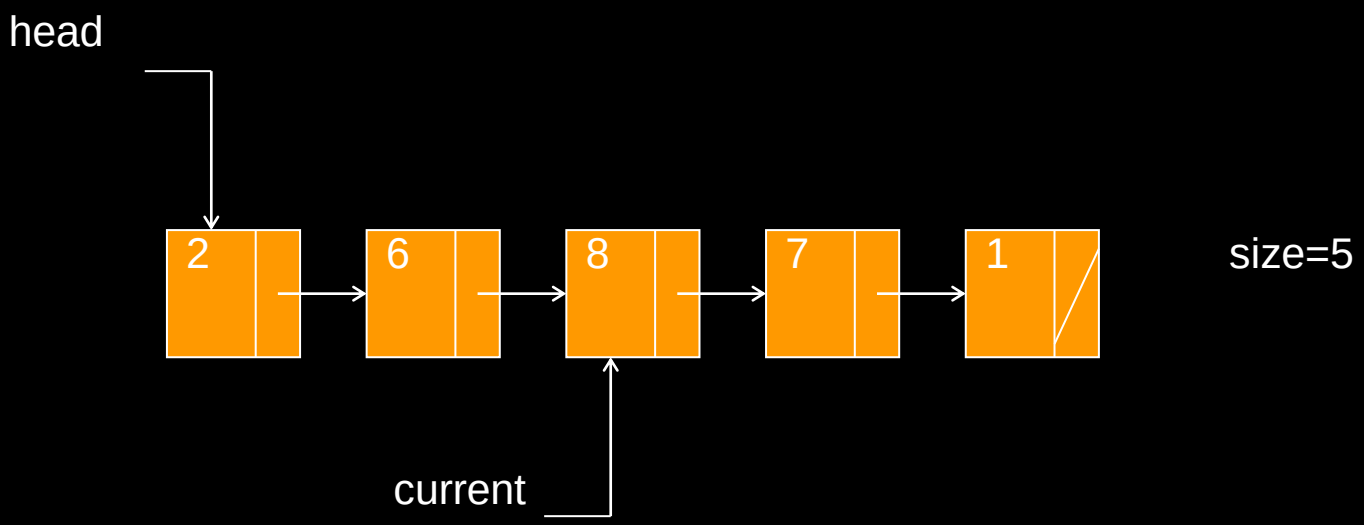
Linked List

- Create a structure called a *Node*.

- The *object* field will hold the actual list element.
- The *next* field in the structure will hold the starting location of the next node.
- Chain the nodes together to form a *linked* list.

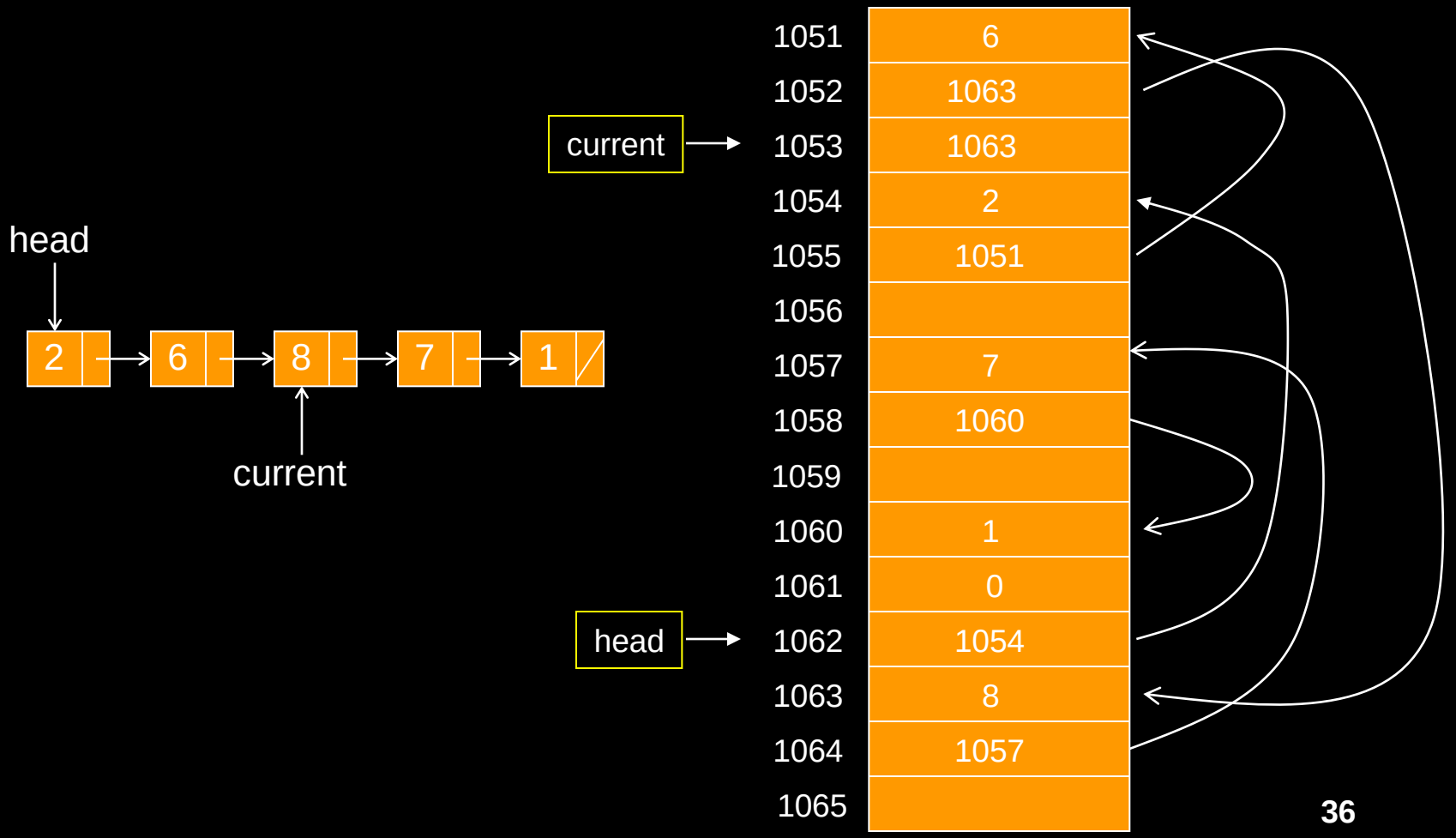
Linked List

- Picture of our list (2, 6, 7, 8, 1) stored as a linked list:



Linked List

- Actual picture in memory:



Linked List Operations

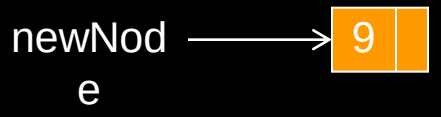
- add(9): Create a new node in memory to hold ‘9’

```
Node* newNode = new Node(9);
```

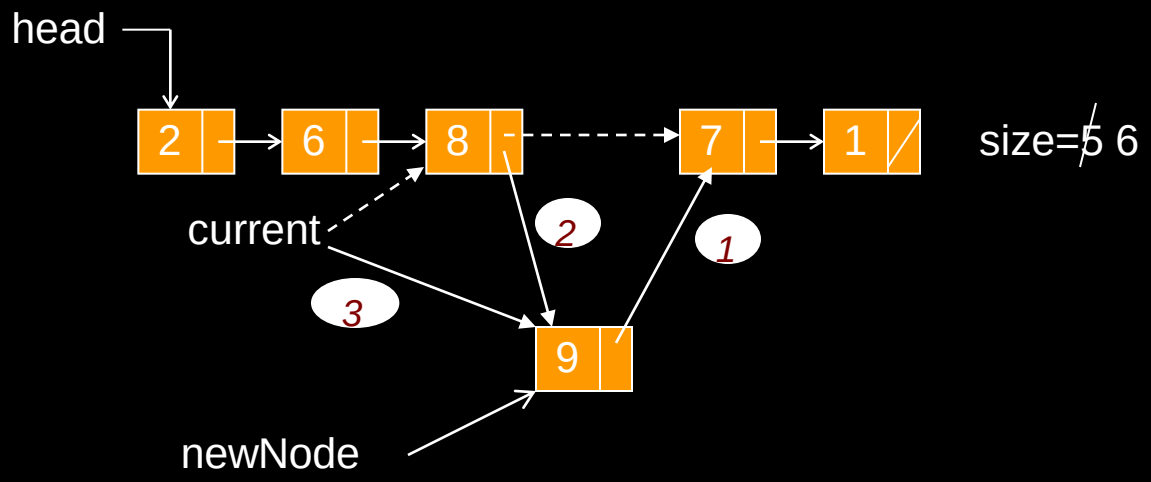
Linked List Operations

- add(9): Create a new node in memory to hold ‘9’

```
Node* newNode = new Node(9);
```



- Link the new node into the list



C++ Code for Linked List

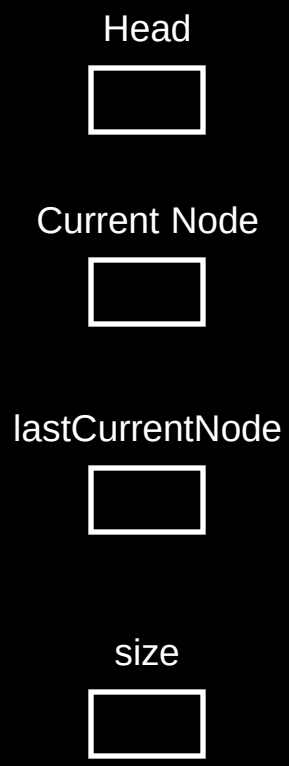
The Node class

```
class Node {
public:
    int get() { return object; };
    void set(int object) { this->object = object; };

    Node *getNext() { return nextNode; };
    void setNext(Node *nextNode)
           { this->nextNode = nextNode; };
private:
    int object;
    Node *nextNode;
};
```

Declaring a class for list of Nodes

b. Declaring constructor to start the list

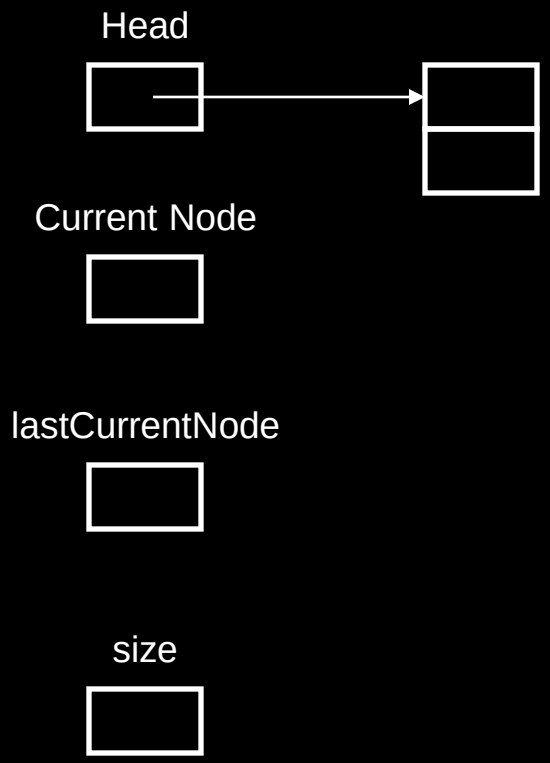
```
#include <iostream>

class List {

private:
        int size;
        Node *headNode;
        Node *currentNode, *lastCurrentNode;

public:
        // Constructor
        List() {
                headNode = new Node();
                headNode->setNext(NULL);
                currentNode = NULL;
                size = 0;
        };
```



Declaring a class for list of Nodes

b. Declaring constructor to start the list

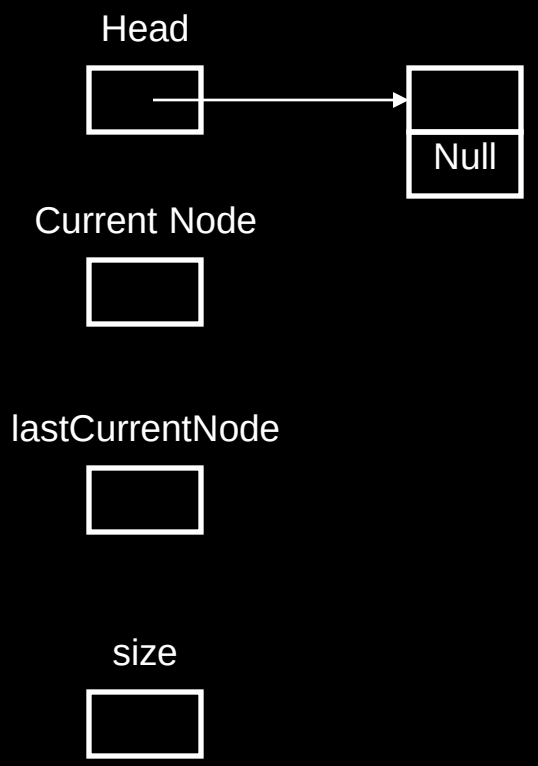
```
class List {

private:
        int size;
        Node *headNode;
        Node *currentNode, *lastCurrentNode;

public:
        // Constructor
        List() {
                headNode = new Node();
                headNode->setNext(NULL);
                currentNode = NULL;
                size = 0;
        };
```



Declaring a class for list of Nodes

b. Declaring constructor to start the list

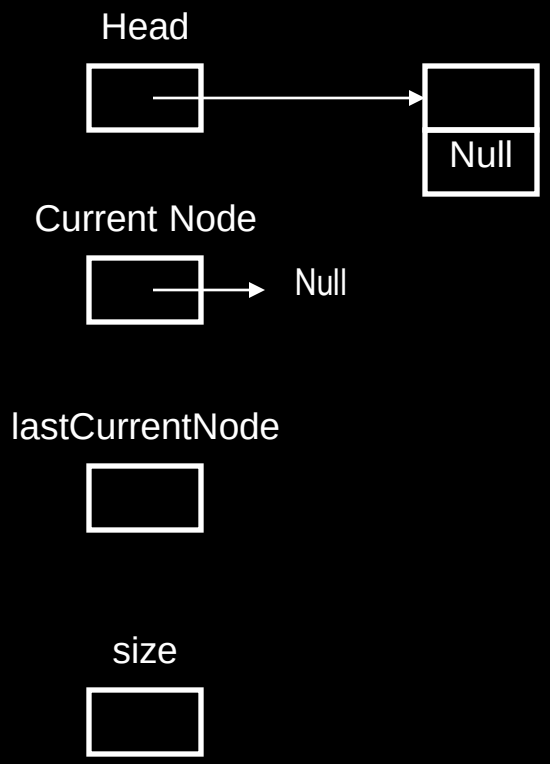
```
class List {

private:
        int size;
        Node *headNode;
        Node *currentNode, *lastCurrentNode;

public:
        // Constructor
        List() {
                headNode = new Node();
                headNode->setNext(NULL);
                currentNode = NULL;
                size = 0;
        };
```



Declaring a class for list of Nodes

b. Declaring constructor to start the list

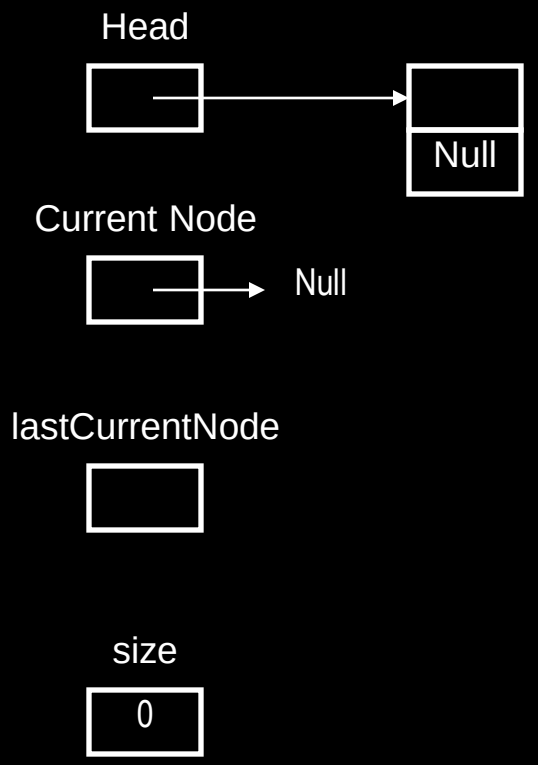
```
class List {

private:
        int size;
        Node *headNode;
        Node *currentNode, *lastCurrentNode;

public:
        // Constructor
        List() {
                headNode = new Node();
                headNode->setNext(NULL);
                currentNode = NULL;
                size = 0;
        };
```



Declaring a class for list of Nodes

b. Declaring constructor to start the list (construct empty list)

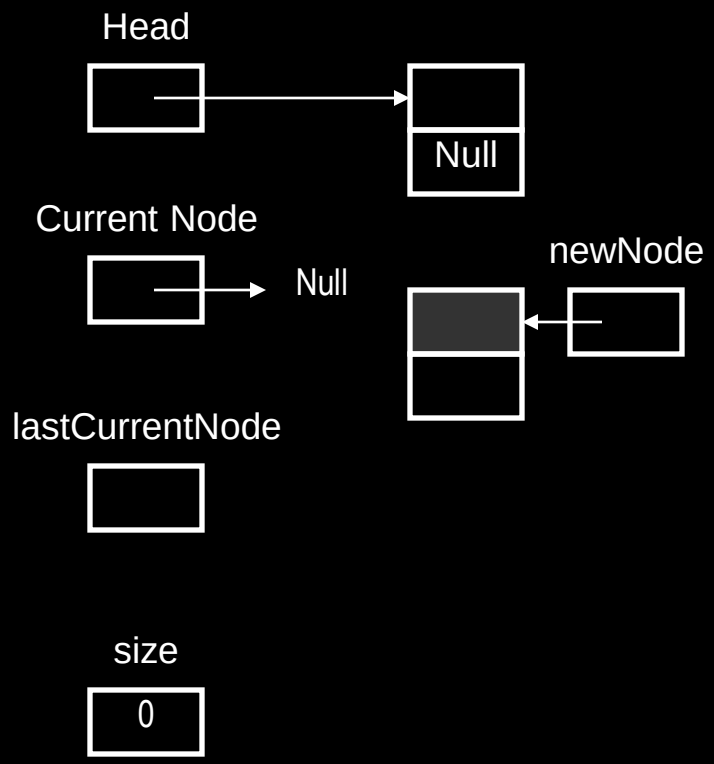
```
class List {

private:
        int size;
        Node *headNode;
        Node *currentNode, *lastCurrentNode;

public:
        // Constructor
        List() {
                headNode = new Node();
                headNode->setNext(NULL);
                currentNode = NULL;
                size = 0;
        };
```



Adding or Appending data in Linked List

a. Create a new node in heap and store the data in data field

```
void add(int addObject) {
   Node* newNode = new Node();
   newNode->set(addObject);
   }
```

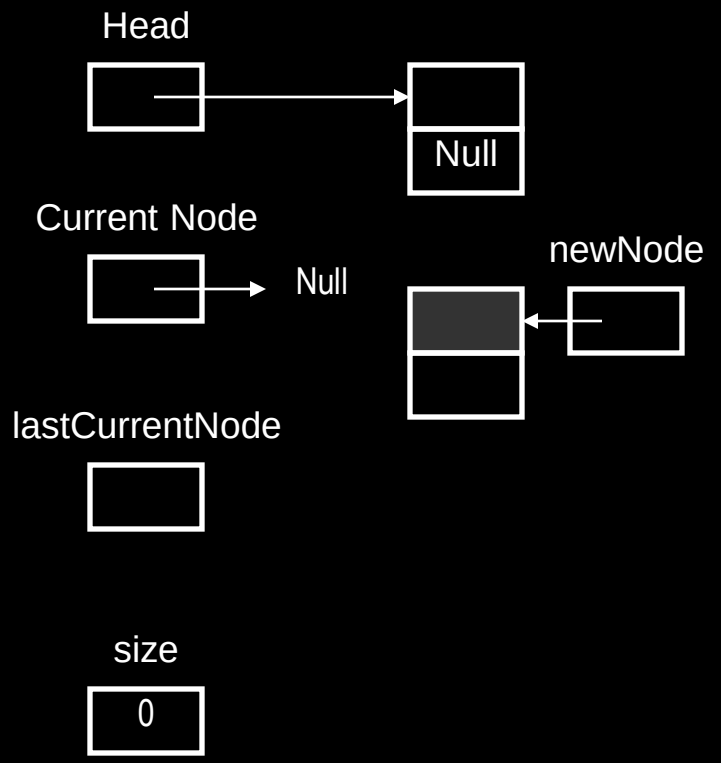


3. Adding or Appending data in Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
newNode -> setNext( NULL );
headNode -> setNext( newNode );
lastCurrentNode = headNode;
currentNode =  newNode;
}
size++;
```

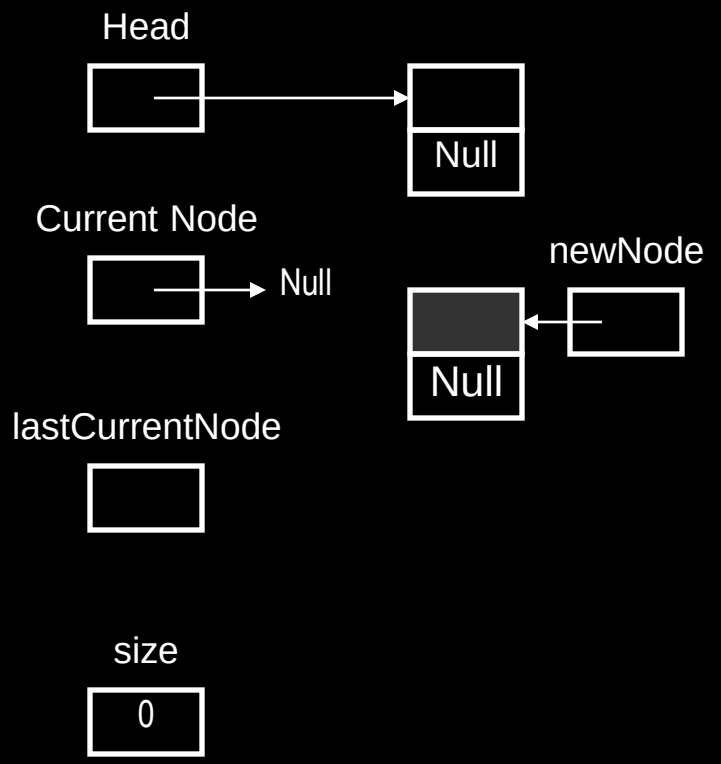


3. Adding or Appending data in Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
newNode -> setNext( NULL );
headNode -> setNext( newNode );
lastCurrentNode = headNode;
currentNode =  newNode;
}
size++;
```

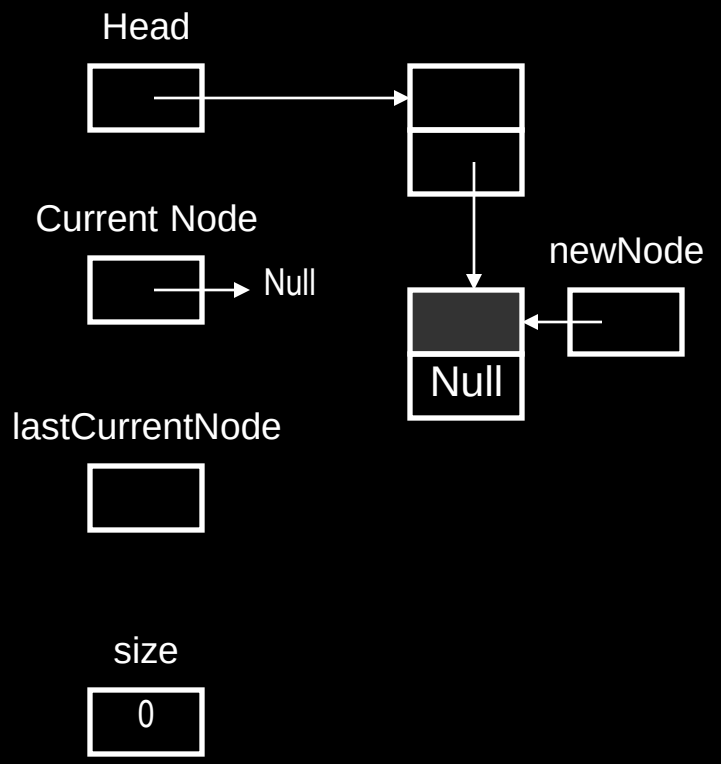


3. Adding or Appending data in Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
newNode -> setNext( NULL );
headNode -> setNext( newNode );
lastCurrentNode = headNode;
currentNode =  newNode;
}
size++;
```

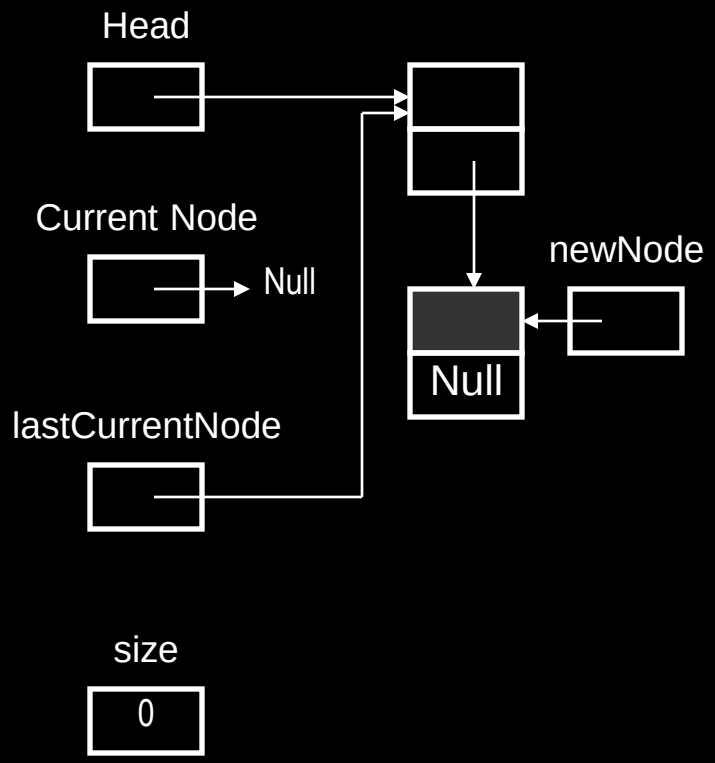


3. Adding or Appending data in Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
newNode -> setNext( NULL );
headNode -> setNext( newNode );
lastCurrentNode = headNode;
currentNode =  newNode;
}
size++;
```

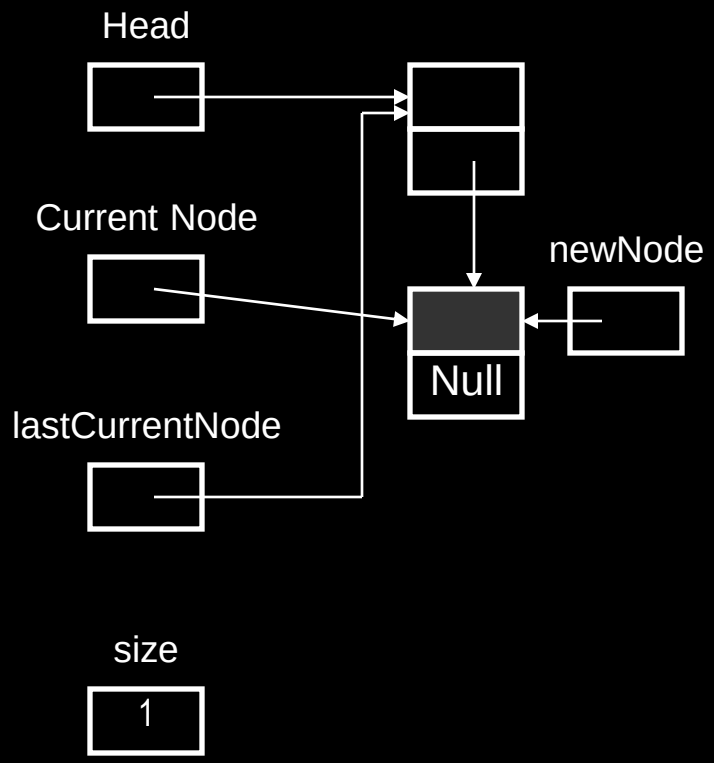


3. Adding or Appending data in Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
newNode -> setNext( NULL );
headNode -> setNext( newNode );
lastCurrentNode = headNode;
currentNode =  newNode;
}
size++;
```

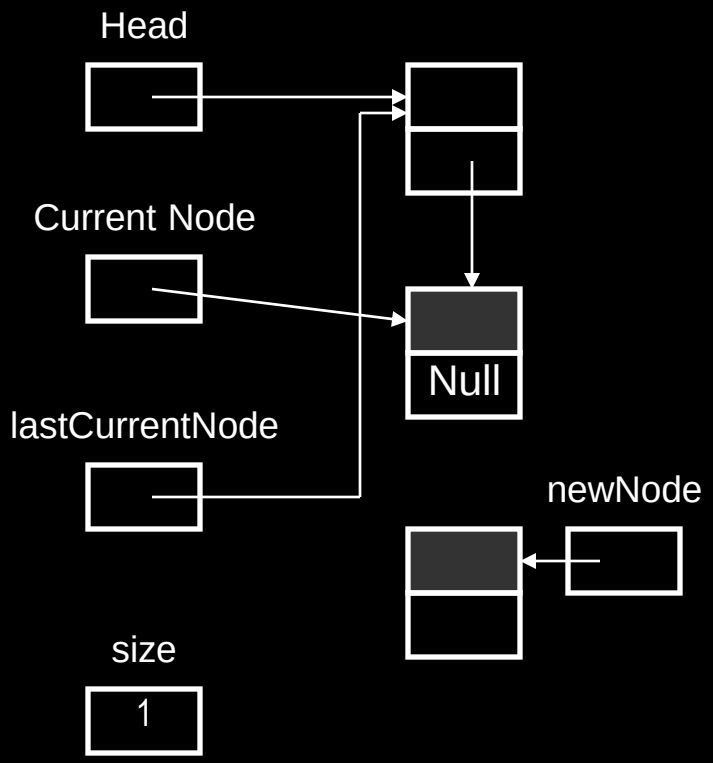


3. Adding or Appending data in Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());

currentNode->setNext( newNode );

lastCurrentNode = currentNode;

currentNode = newNode;

size++;
```

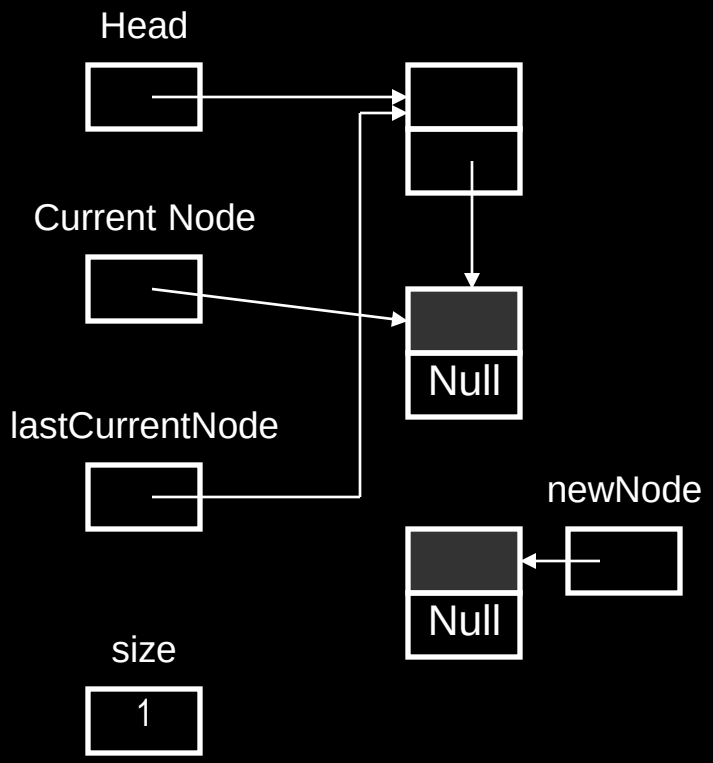


3. Adding or Appending data in Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());

currentNode->setNext( newNode );

lastCurrentNode = currentNode;

currentNode = newNode;

size++;
```

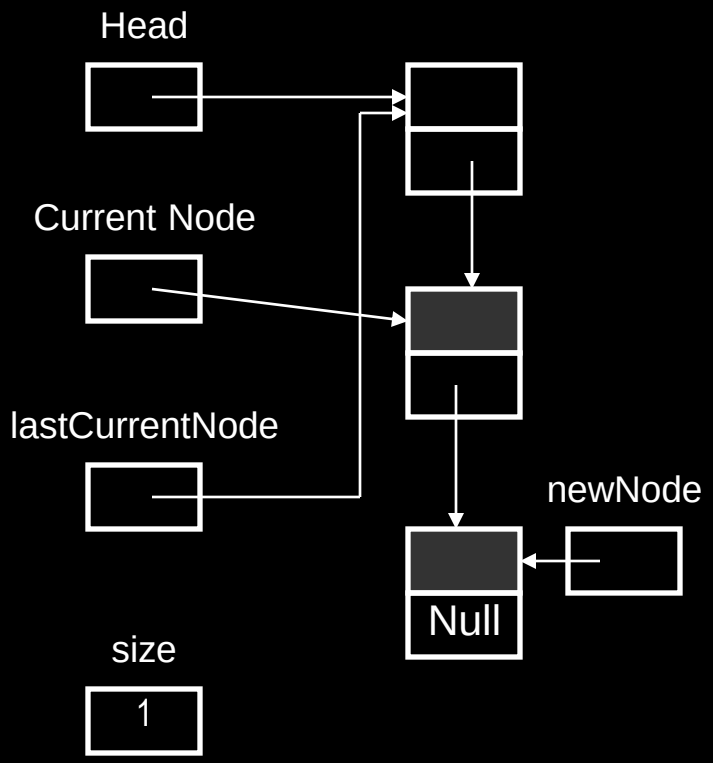


3. Adding or Appending data in Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());

currentNode->setNext( newNode );

lastCurrentNode = currentNode;


currentNode = newNode;

size++;
```

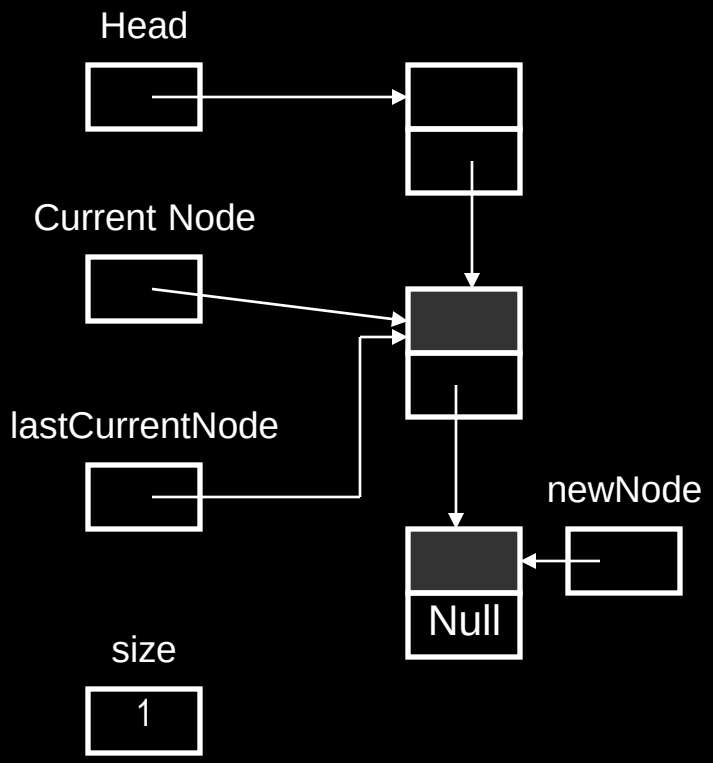


3. Adding or Appending data in Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());

currentNode->setNext( newNode );

lastCurrentNode = currentNode;


currentNode = newNode;

size++;
```

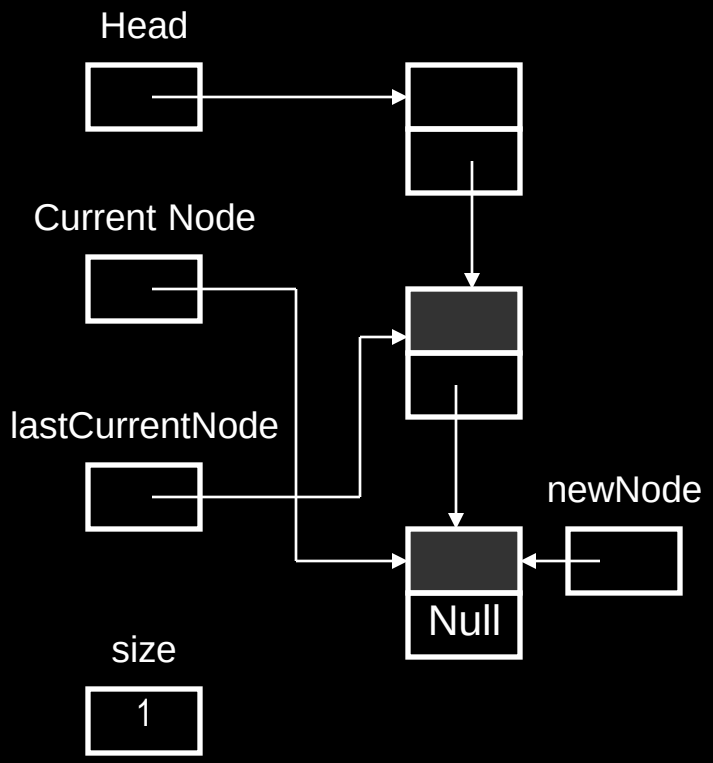


3. Adding or Appending data in Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());

currentNode->setNext( newNode );

lastCurrentNode = currentNode;

currentNode = newNode;

size++;
```

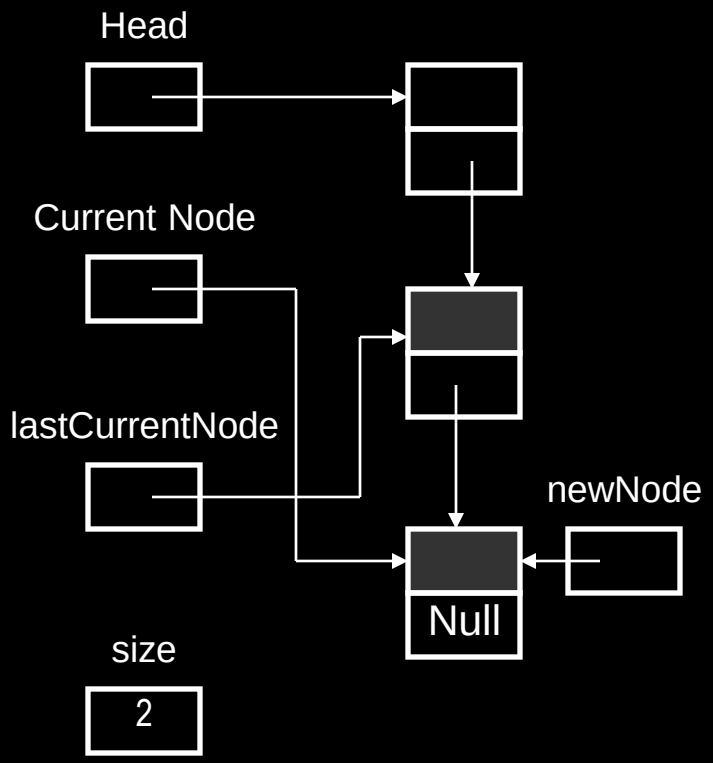


3. Adding or Appending data in Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());

currentNode->setNext( newNode );

lastCurrentNode = currentNode;

currentNode = newNode;

size++;
```



3. Adding or Appending data in Linked List

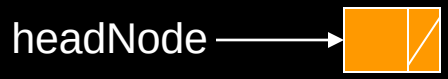
```
void add(int addObject) {
   Node* newNode = new Node();
   newNode->set(addObject);
   if( currentNode != NULL ){
        newNode->setNext(currentNode->getNext());
        currentNode->setNext( newNode );
        lastCurrentNode = currentNode;
        currentNode = newNode;
        }
   else {
        newNode->setNext(NULL);
        headNode->setNext(newNode);
        lastCurrentNode = headNode;
        currentNode =  newNode;
        }
    size++;
};
```

Building a Linked List

```
List list;
```

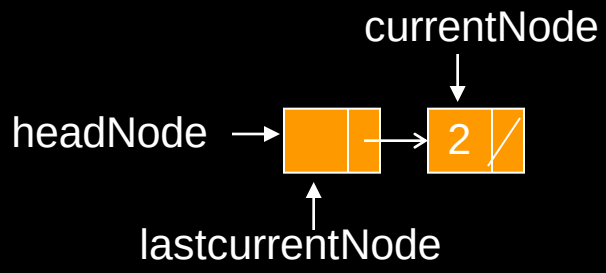
Building a Linked List

```
List list;
```



size=0

```
list.add(2);
```



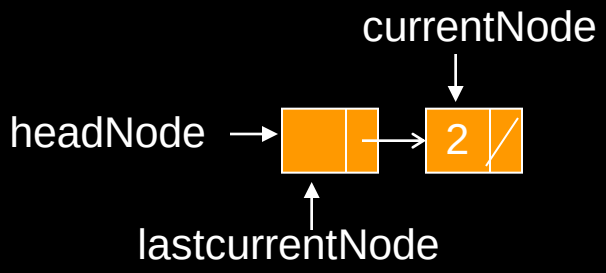
size=1

Building a Linked List

```
List list;
```

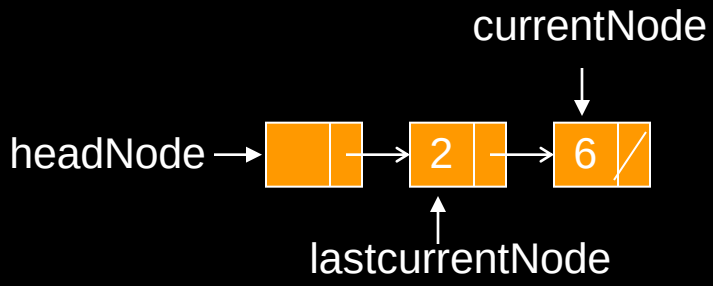
size=0

```
list.add(2);
```



size=1

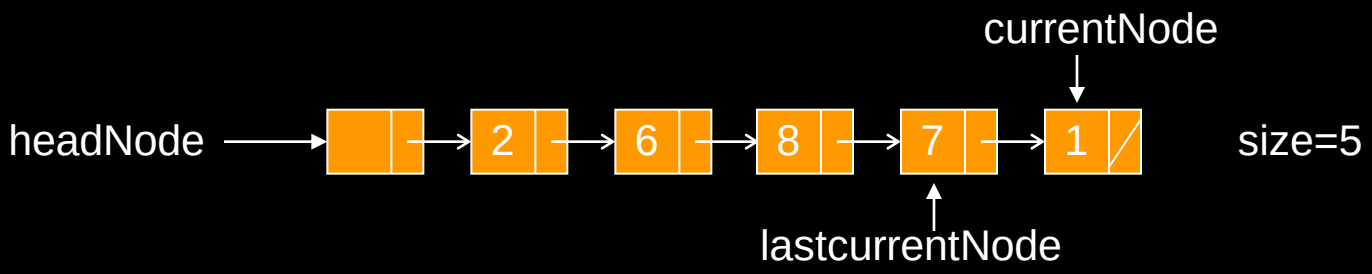
```
list.add(6);
```



size=2

Building a Linked List

```
List.add(8); list.add(7); list.add(1);
```



C++ Code for Linked List

```
int get() {
    if (currentNode != NULL)
        return currentNode->get();
};
```

C++ Code for Linked List

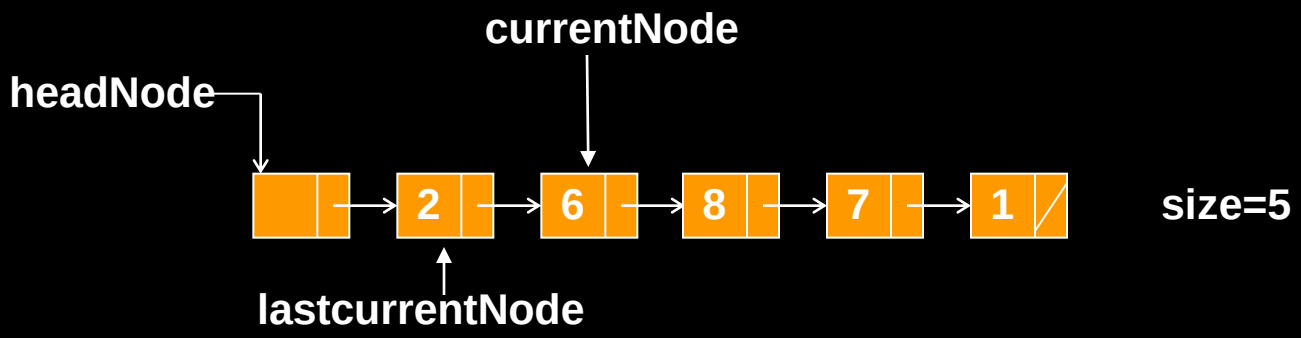
```
bool next() {
    if (currentNode == NULL) return false;

    lastCurrentNode = currentNode;
    currentNode = currentNode->getNext();
    if (currentNode == NULL || size == 0)
      return false;
    else
      return true;
};
```

C++ Code for Linked List

```
// position current before the first
// list element
void start() {
     lastCurrentNode = headNode;
     currentNode = headNode;
};
```

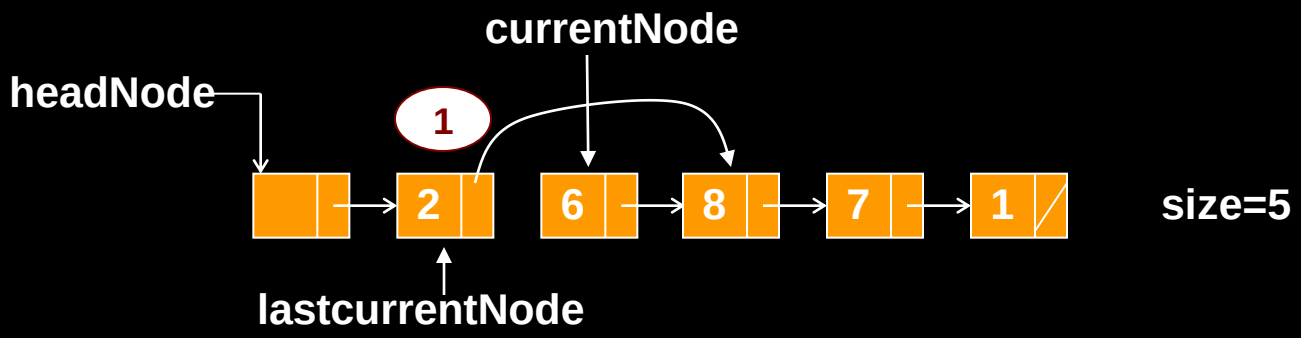
C++ Code for Linked List

```
void remove() {
  if( currentNode != NULL &&
      currentNode != headNode) {
       lastCurrentNode->setNext(currentNode->getNext());
       delete currentNode;
       currentNode = lastCurrentNode->getNext();
       size--;
  }
};
```



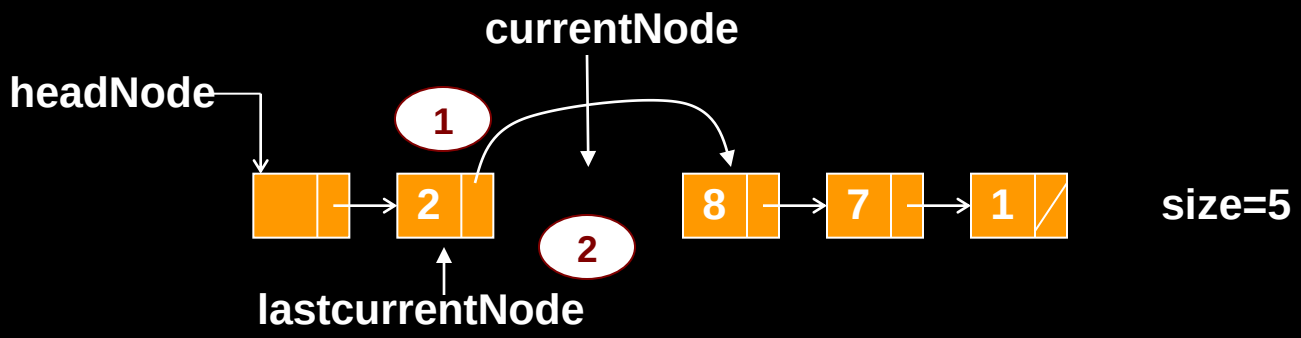
C++ Code for Linked List

```
void remove() {
  if( currentNode != NULL &&
      currentNode != headNode) {
    1 lastCurrentNode->setNext(currentNode->getNext());
      delete currentNode;
      currentNode = lastCurrentNode->getNext();
      size--;
  }
};
```



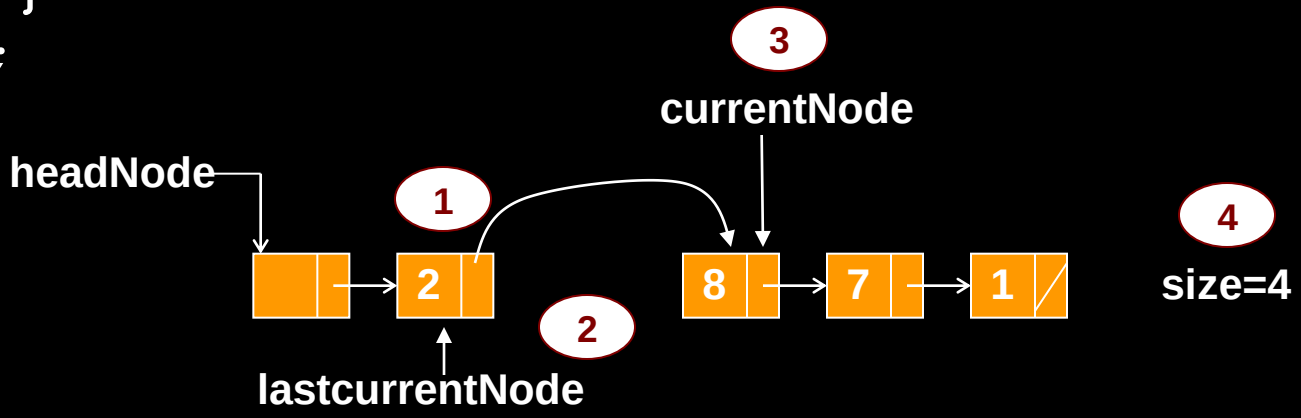
C++ Code for Linked List

```
void remove() {
  if( currentNode != NULL &&
      currentNode != headNode) {
    1 lastCurrentNode->setNext(currentNode->getNext());
    2 delete currentNode;
      currentNode = lastCurrentNode->getNext();
      size--;
  }
};
```



C++ Code for Linked List

```
void remove() {
  if( currentNode != NULL &&
      currentNode != headNode) {
    1 lastCurrentNode->setNext(currentNode->getNext());
    2 delete currentNode;
    3 currentNode = lastCurrentNode->getNext();
    4 size--;
  }
};
```



C++ Code for Linked List

```
int length()
        {
        int count=0;
        lastCurrentNode = headNode;
        currentNode=headNode->getNext();
        if (currentNode==NULL) return 0;
        while (currentNode!=NULL)
                {
                lastCurrentNode = currentNode;
                currentNode = currentNode->getNext();
                count++;
                }
        return count;
        };
```

C++ Code for Linked List

```
void positionAt(int pos)
    {
    lastCurrentNode = headNode;
    currentNode=headNode->getNext();
    for (int i=1; i<pos; i++)
        {
        lastCurrentNode = currentNode;
        currentNode = currentNode->getNext();
        }
}
```

Example of List Usage

```
#include <iostream>
int main()
{
      List list;
list.add(5);list.add(13);list.add(4);list.add(8);
list.add(24); list.add(48); list.add(12);
cout << "\n\nThe length of list = "<< list.length();
      list.start();
       cout << "\nList Element: ";
      while (list.next() == true)
         cout << list.get() << ",";
      cout << "\n\nDeleting Element 4 ";
      list.positionAt(4);
     list.remove();
}
```

Example of List Usage

```
cout << "\n\nThe lenght of list = "<< list.length();
      list.start();
      cout << "\nList Element: ";
while (list.next() ==true)cout << list.get() << ",";

      cout << "\n\nInserting 99 as Element 5 ";
      list.positionAt(5);
       list.add(99);
cout << "\n\nThe lenght of list = "<< list.length();
      list.start();
       cout << "\nList Element: ";
      while (list.next() == true)
        cout << list.get() << ",";
```

Analysis of Linked List

- **add**
  - **we simply insert the new node after the current node. So add is a one-step operation.**
- **remove**
  - **remove is also a one-step operation**
- **find**
  - **worst-case: may have to search the entire list**
- **back**
  - **moving the current pointer back one node requires traversing the list from the start until the node whose next pointer points to current node.**

Doubly-linked List

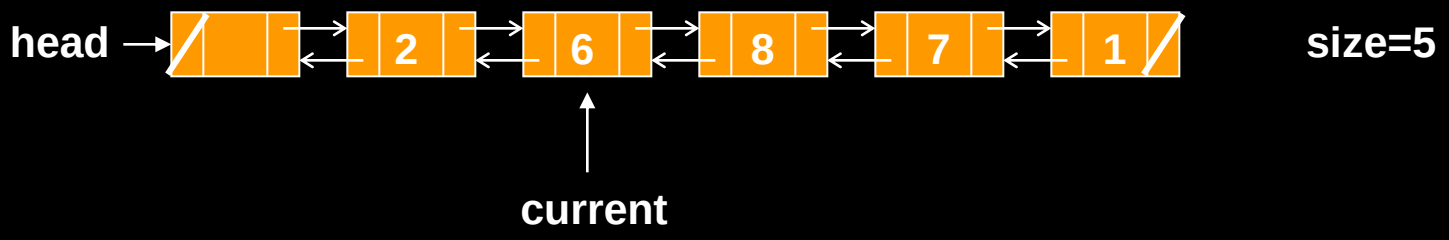
- Moving forward in a singly-linked list is easy; moving backwards is not so easy.
- To move back one node, we have to start at the head of the singly-linked list and move forward until the node before the current.
- To avoid this we can use *two* pointers in a node: one to point to next node and another to point to the previous node:

Doubly-Linked List Node

```
class Node {
public:
    int get() { return object; };
    void set(int object) { this->object = object; };

    Node* getNext() { return nextNode; };
    void setNext(Node* nextNode)
          { this->nextNode = nextNode; };
    Node* getPrev() { return prevNode; };
    void setPrev(Node* prevNode)
          { this->prevNode = prevNode; };
private:
    int object;
    Node* nextNode;
    Node* prevNode;
};
```

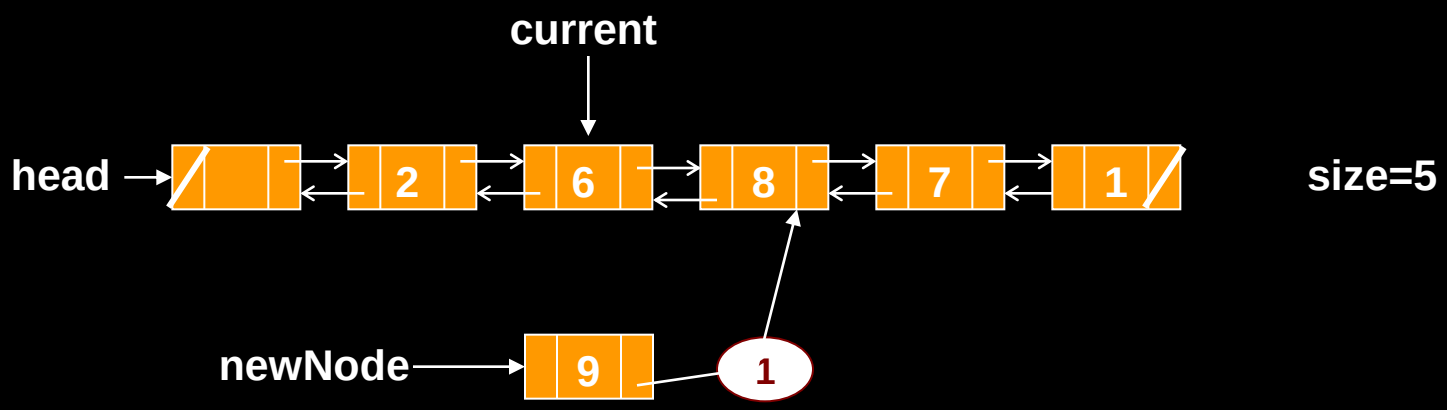
Doubly-linked List

- Need to be more careful when adding or removing a node.
- Consider add: the order in which pointers are reorganized is important:



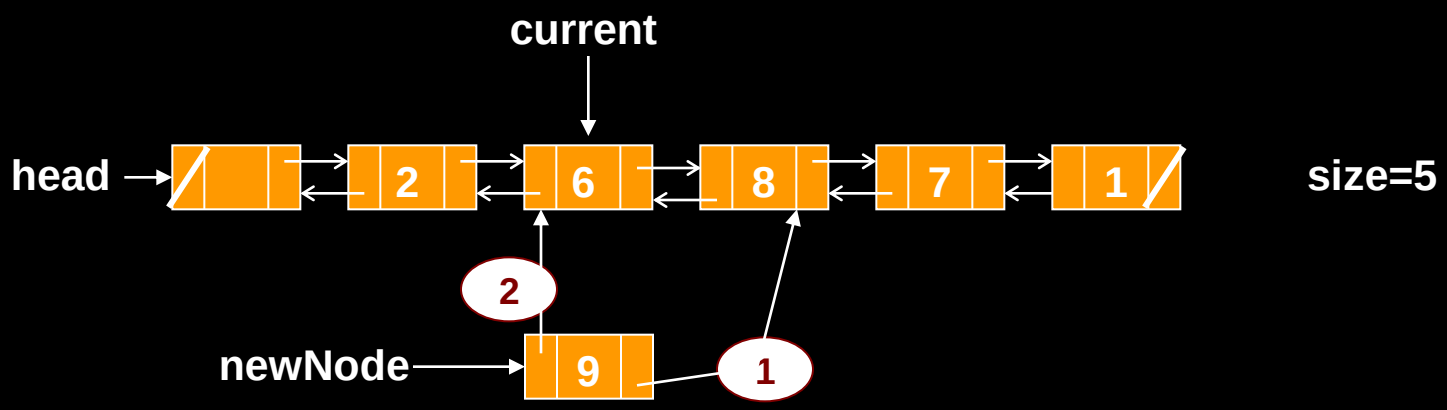
Doubly-linked List

```
1.   newNode->setNext( current->getNext() );
```



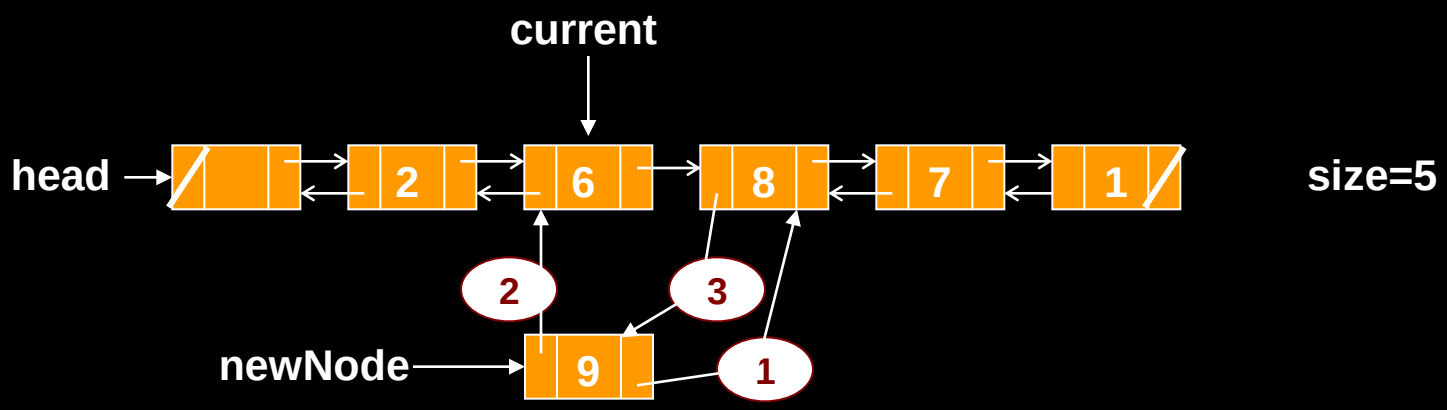
Doubly-linked List

```
1.   newNode->setNext( current->getNext() );
2.   newNode->setprev( current );
```



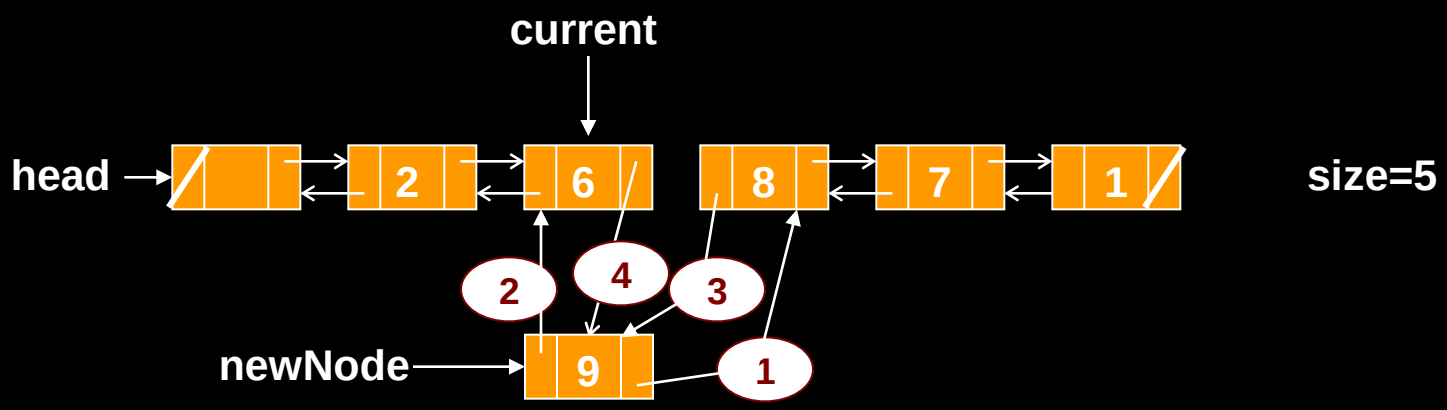
Doubly-linked List

```
1.   newNode->setNext( current->getNext() );
2.   newNode->setprev( current );
3.   (current->getNext())->setPrev(newNode);
```



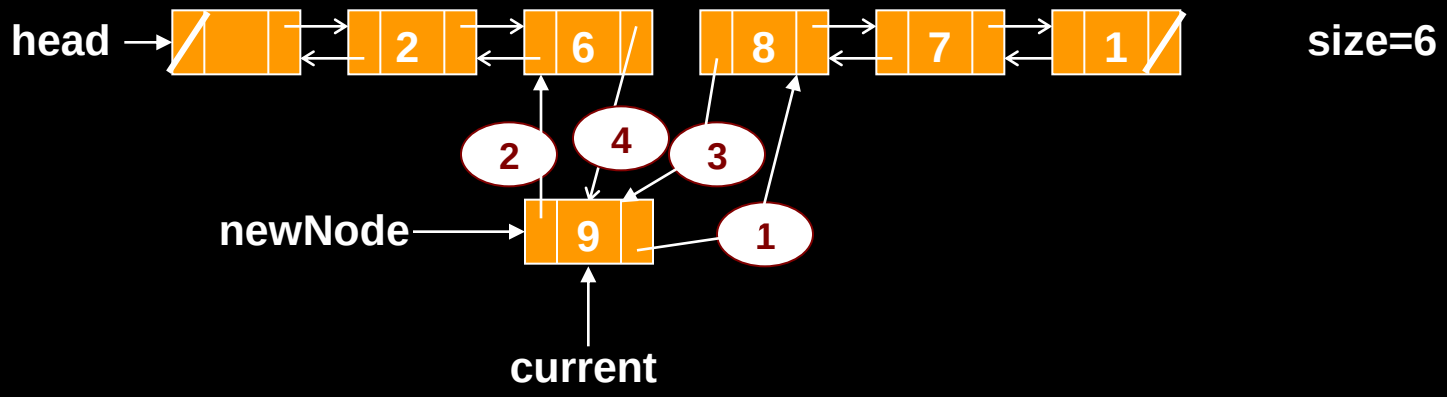
Doubly-linked List

```
1.   newNode->setNext( current->getNext() );
2.   newNode->setprev( current );
3.   (current->getNext())->setPrev(newNode);
4.   current->setNext( newNode );
```



Doubly-linked List

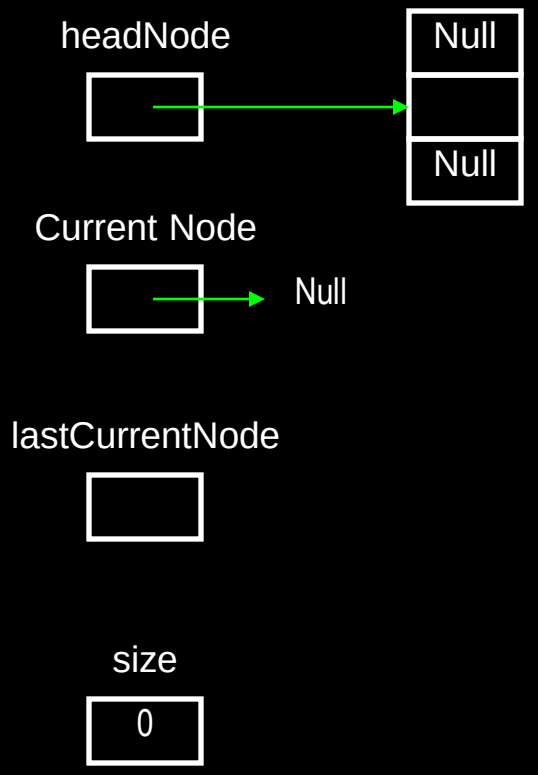
```
1.   newNode->setNext( current->getNext() );
2.   newNode->setprev( current );
3.   (current->getNext())->setPrev(newNode);
4.   current->setNext( newNode );
5.   current = newNode;
6.   size++;
```



2. Declaring a class for list of Nodes for double linked list

b. Declaring constructor to start the list (construct empty list)

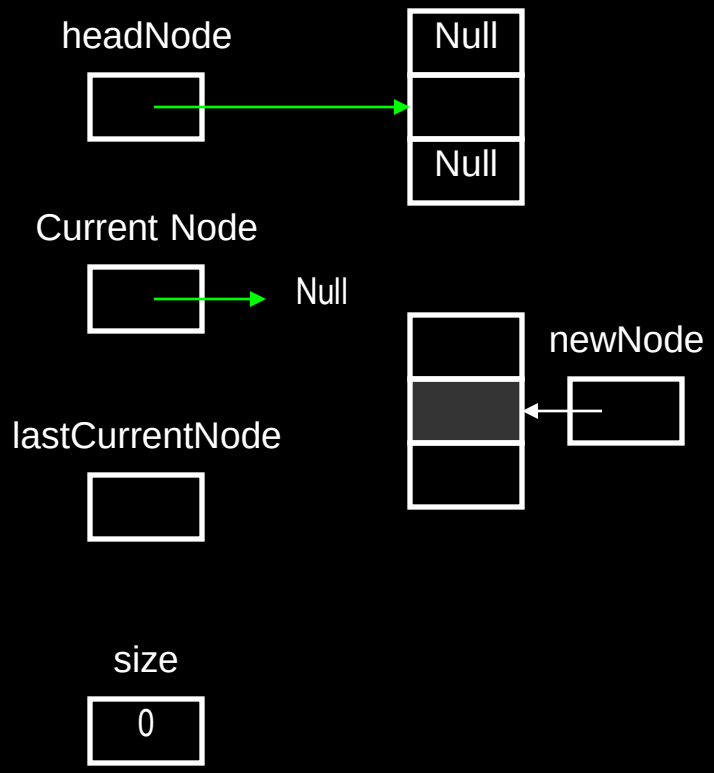
```
class List {

private:
        int size;
        Node *headNode;
        Node *currentNode, *lastCurrentNode;

public:
        // Constructor
        List() {
                headNode = new Node();
                headNode->setNext(NULL);
                headNode->setPrev(NULL);
                currentNode = NULL;
                size = 0;
        };
```



3. Adding or Appending data in Double Linked List

a. Create a new node in heap and store the data in data field

```
void add(int addObject) {
   Node* newNode = new Node();
   newNode->set(addObject);
   }
```

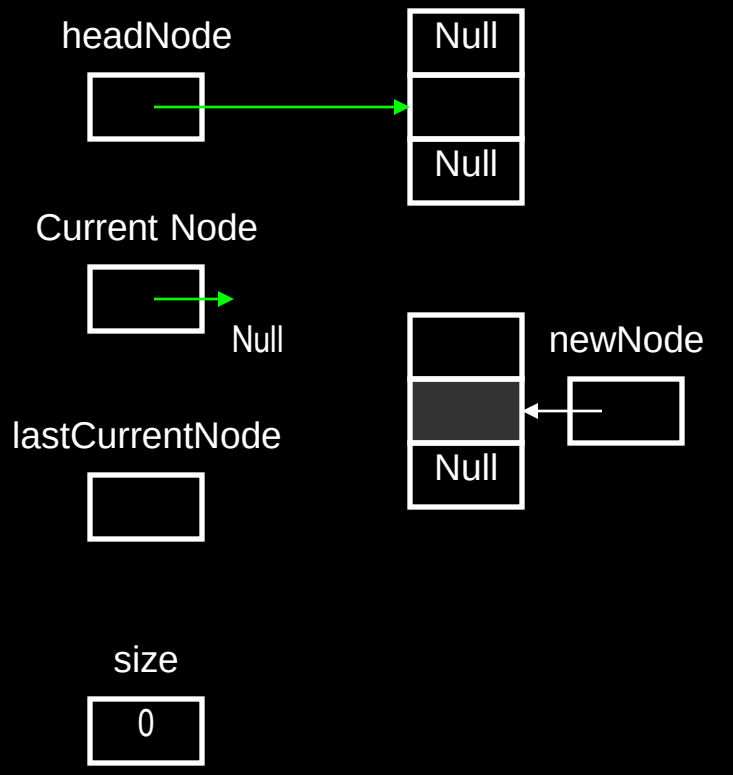


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
newNode -> setNext(NULL);
headNode -> setNext( newNode );
newNode -> setPrev(headNode);
lastCurrentNode = headNode;
currentNode =  newNode;
}
size++;
```

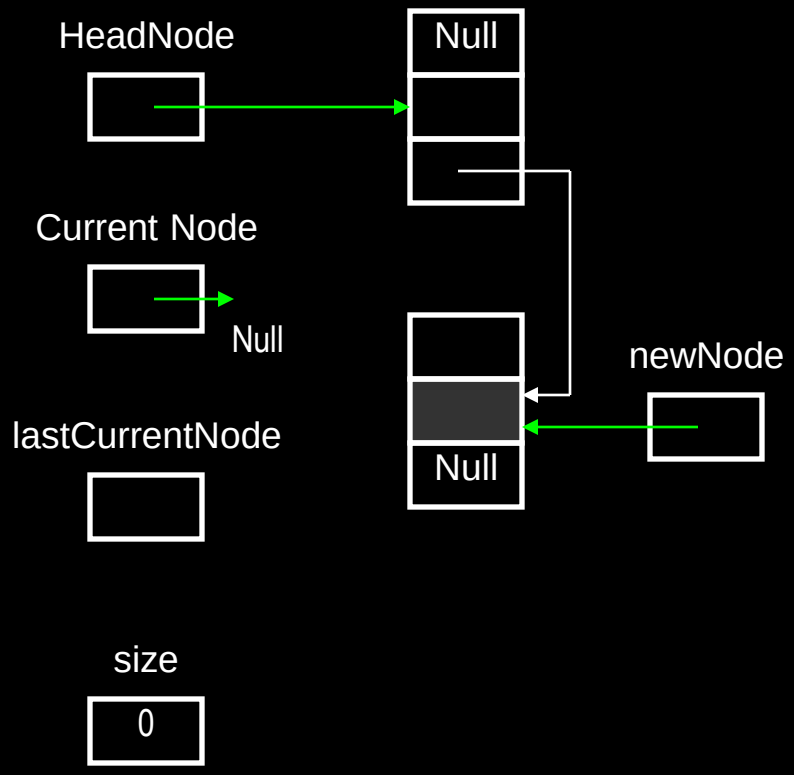


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
newNode -> setNext(NULL);
headNode -> setNext( newNode );
newNode -> setPrev(headNode);
lastCurrentNode = headNode;
currentNode =  newNode;
}
size++;
```

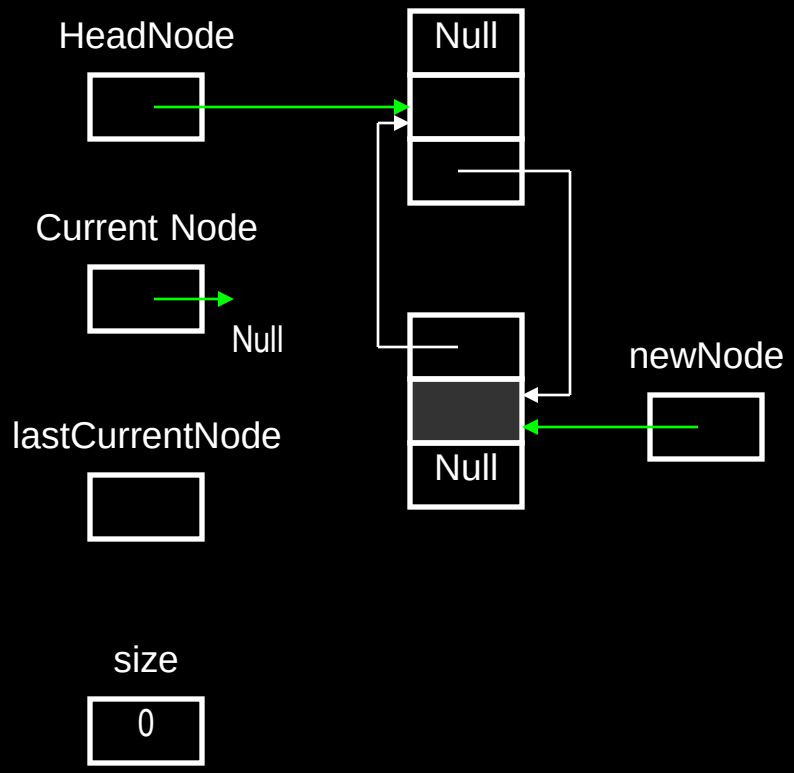


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
newNode -> setNext(NULL);
headNode -> setNext( newNode );
newNode -> setPrev(headNode);
lastCurrentNode = headNode;
currentNode =  newNode;
}
size++;
```

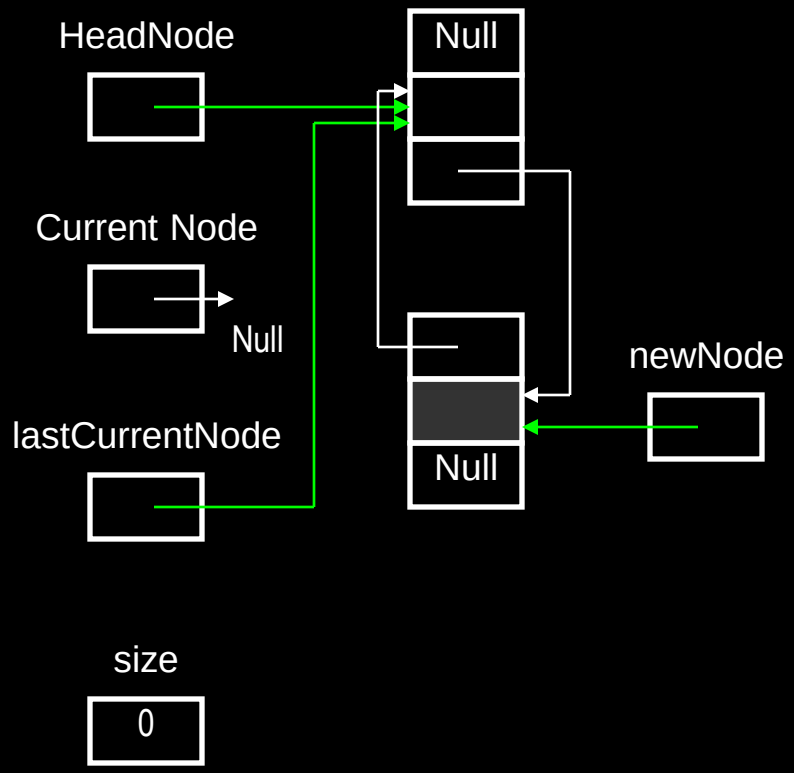


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
newNode -> setNext(NULL);
headNode -> setNext( newNode );
newNode -> setPrev(headNode);
lastCurrentNode = headNode;
currentNode =  newNode;
}
size++;
```

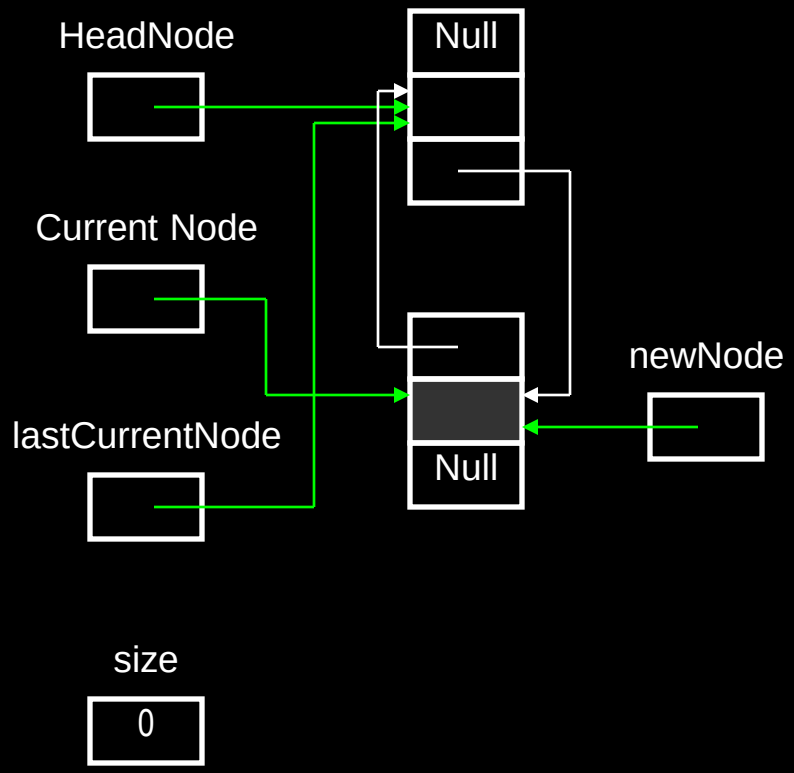


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
newNode -> setNext(NULL);
headNode -> setNext( newNode );
newNode -> setPrev(headNode);
lastCurrentNode = headNode;
currentNode =  newNode;
}
size++;
```

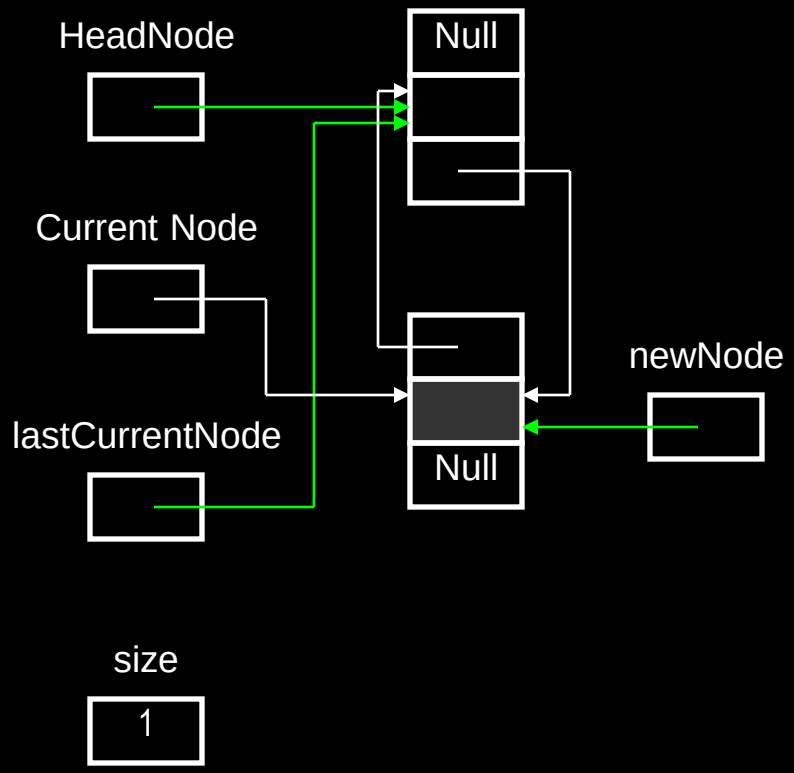


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
newNode -> setNext(NULL);
headNode -> setNext( newNode );
newNode -> setPrev(headNode);
lastCurrentNode = headNode;
currentNode =  newNode;
}
size++;
```

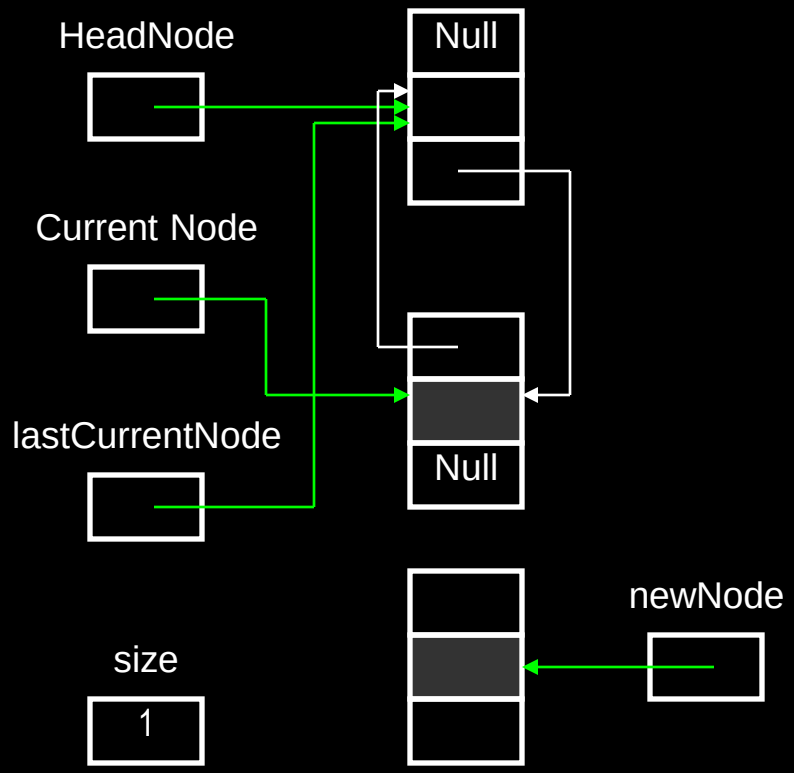


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());
newNode->setPrev(currentNode);
currentNode->setNext( newNode );
lastCurrentNode = currentNode;
currentNode = newNode;
size++;
```

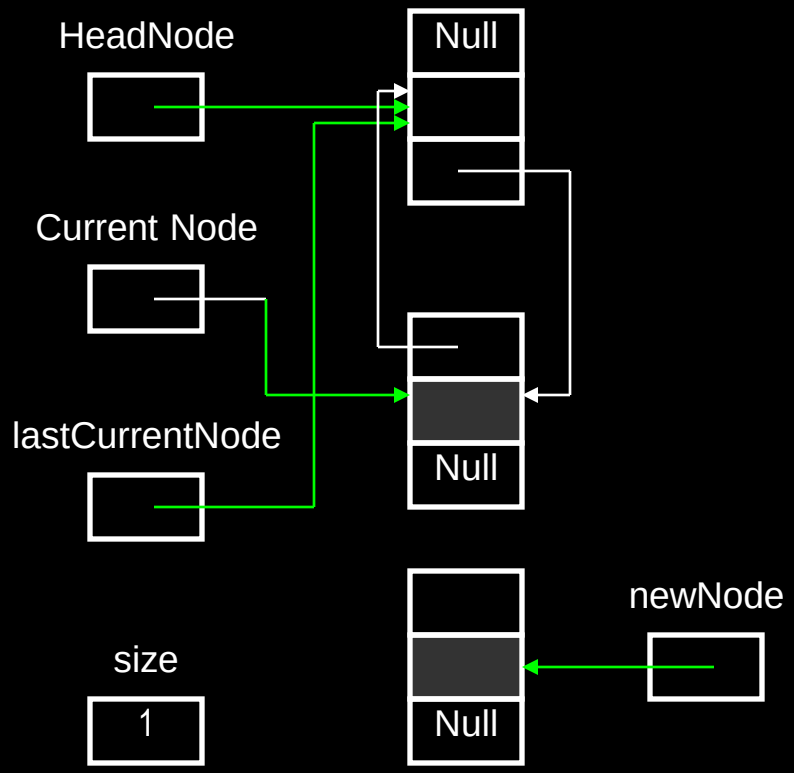


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());
newNode->setPrev(currentNode);
currentNode->setNext( newNode );
lastCurrentNode = currentNode;
currentNode = newNode;
size++;
```

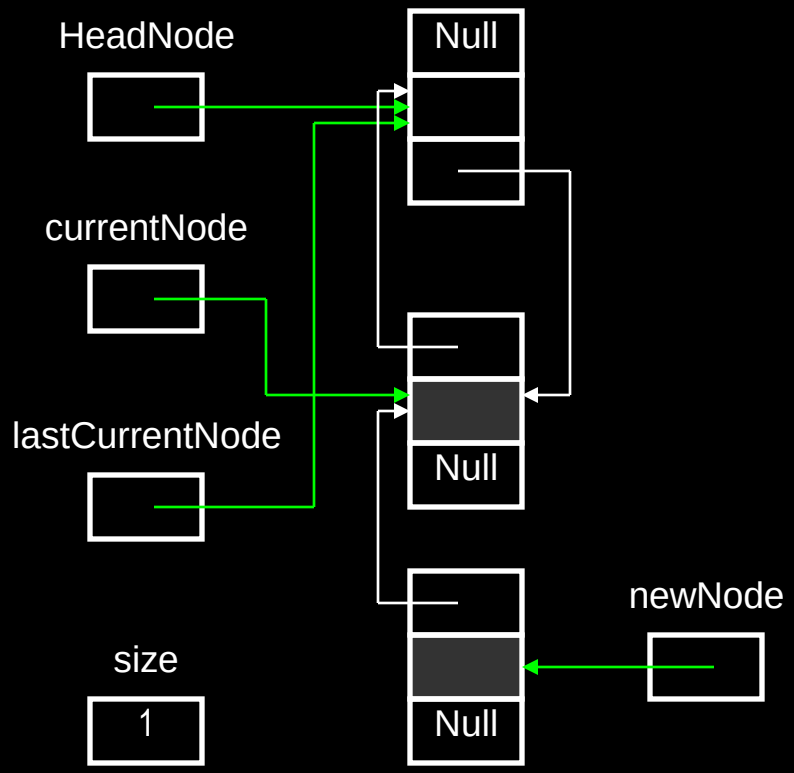


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());
newNode->setPrev(currentNode);
currentNode->setNext( newNode );
lastCurrentNode = currentNode;
currentNode = newNode;
size++;
```

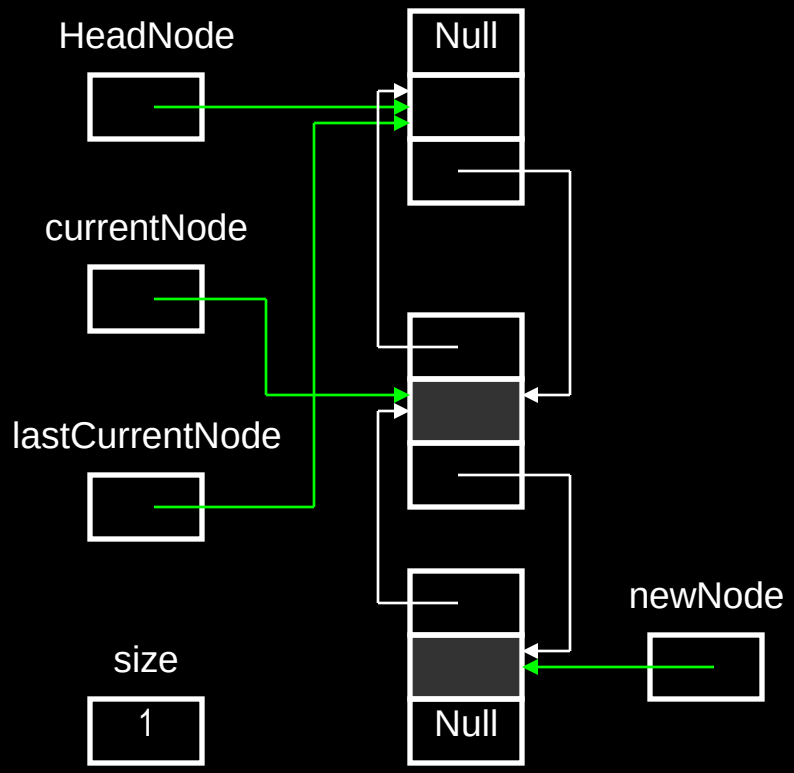


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());
newNode->setPrev(currentNode);
currentNode->setNext( newNode );
lastCurrentNode = currentNode;
currentNode = newNode;
size++;
```

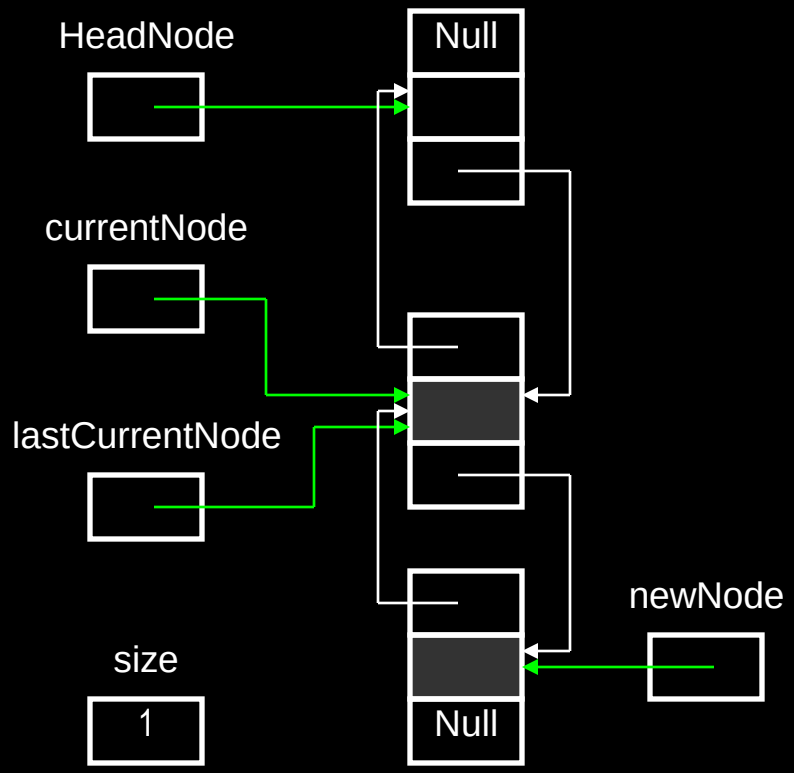


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());
newNode->setPrev(currentNode);
currentNode->setNext( newNode );
lastCurrentNode = currentNode;
currentNode = newNode;
size++;
```

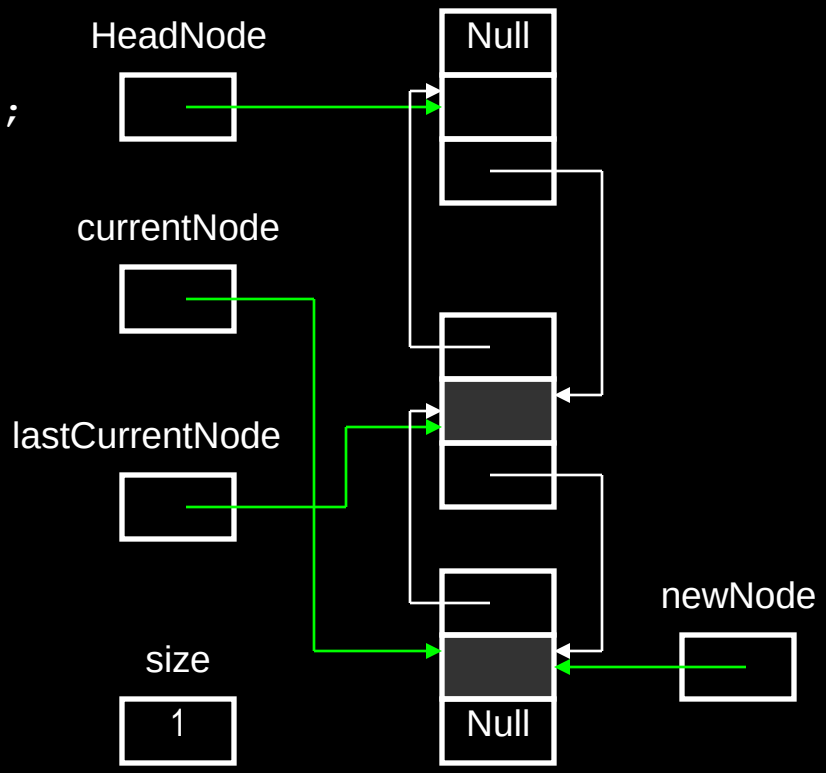


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());
newNode->setPrev(currentNode);
currentNode->setNext( newNode );
lastCurrentNode = currentNode;
currentNode = newNode;
size++;
```

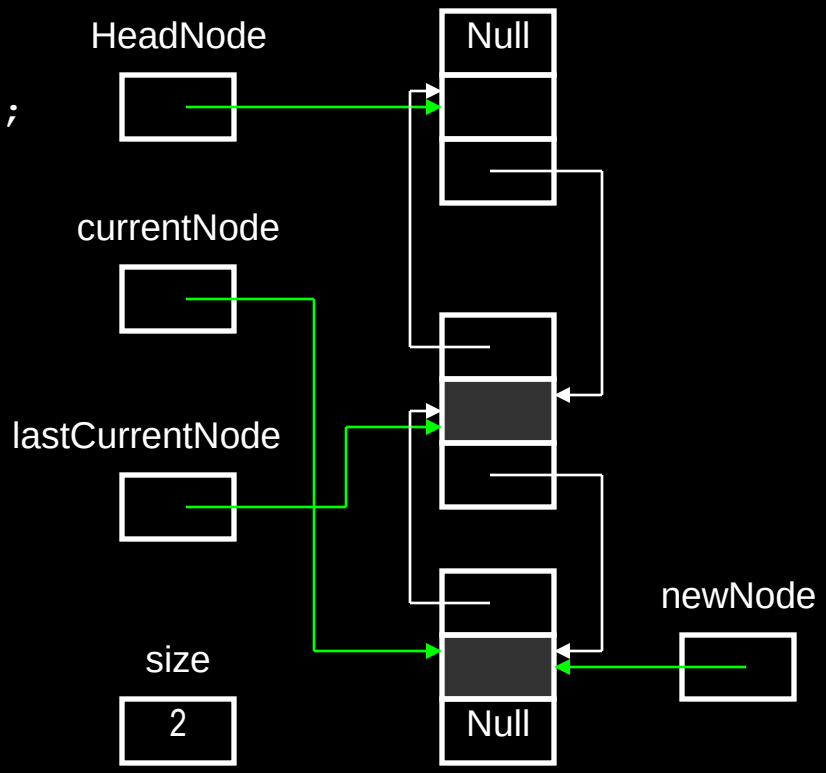


3. Adding or Appending data in Double Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(currentNode->getNext());
newNode->setPrev(currentNode);
currentNode->setNext( newNode );
lastCurrentNode = currentNode;
currentNode = newNode;
size++;
```

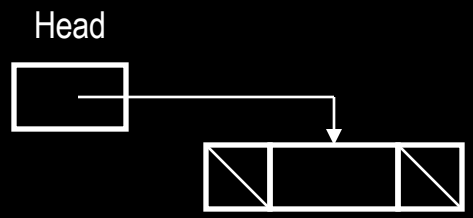


3. Adding or Appending data in Double Linked List

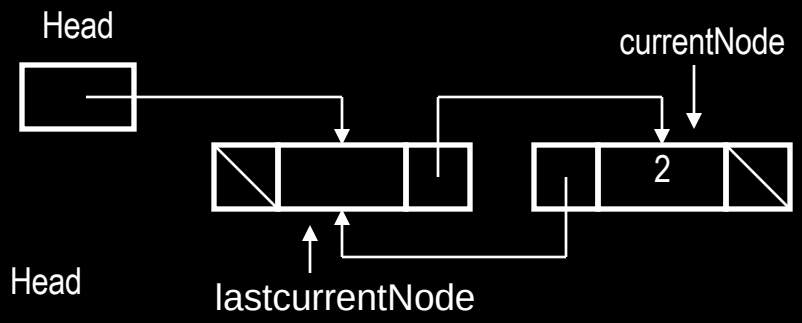
```
void add(int addObject) {
   Node* newNode = new Node();
   newNode->set(addObject);
   if( currentNode != NULL ){
         newNode->setNext(currentNode->getNext());
         newNode->setPrev(currentNode);
         currentNode->setNext( newNode );
         lastCurrentNode = currentNode;
         currentNode = newNode;
         }
   else {
         newNode -> setNext(NULL);
         headNode -> setNext( newNode );
         newNode -> setPrev(headNode);
         lastCurrentNode = headNode;
         currentNode =  newNode;
         }
   size++;
};
```

Building a doubly linked list

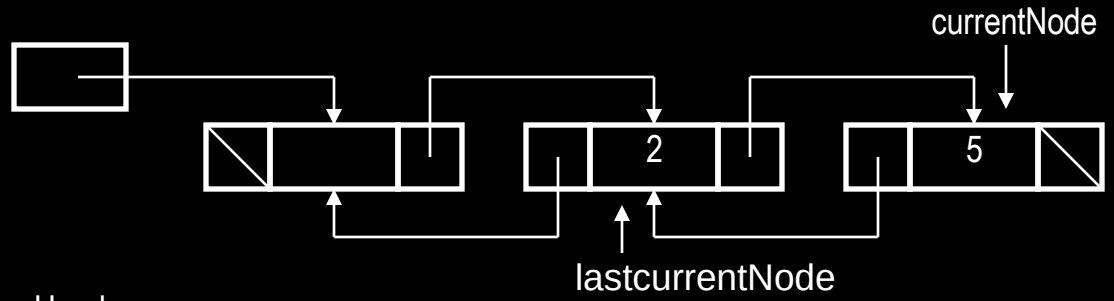
```
List list;
```



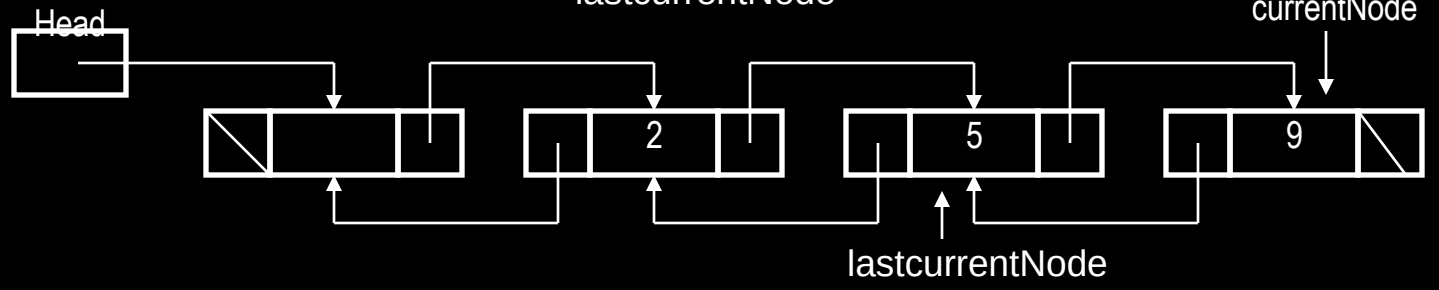

```
listd.add(2);
```




```
listd.add(5);
```




```
listd.add(9);
```



Doubly Linked List Basic Operations
Moving current to next node

```
bool next() {
   if (currentNode == NULL)
  return fale;

   lastCurrentNode = currentNode;
   currentNode = currentNode->getNext();

   if (currentNode == NULL || size == 0)
   return false;
   else
   return true;
 };
```

Doubly Linked List Basic Operations
Moving current to previous node (new)

```
bool back() {
   if (currentNode == NULL)
  return false;

   currentNode = lastCurrentNode;
   lastCurrentNode = lastCurrentNode->getPrev();

   if (lastCurrentNode == NULL || size == 0)
   return false;
   else
   return true;
 };
```

Doubly Linked List Basic Operations
deleting a node at current position

```
void remove() {
  if( currentNode != NULL && currentNode != headNode)
    {
    Node* temp = currentNode;
    lastCurrentNode->setNext(currentNode->getNext());
    currentNode = lastCurrentNode->getNext();
   currentNode->setPrev(temp->getPrev());
    delete temp;
    size--;
   }
};
```

Example of building & displaying Doubly Linked List

```
#include <iostream.h>
void main()
{
        List list;
        list.add(5); list.add(13); list.add(4);
        list.add(8); list.add(24); list.add(48); list.add(12);

        cout << "\n\nThe length of list = " << list.length();
        list.positionAt(1); cout << "\nList Element: ";
        for (int i=1; i<=list.length(); i++)
             { cout << list.get() << ","; list.next(); }

        cout << "\n\nList Element in reverse order: ";
        list.positionAt(list.length());
        for (i=list.length(); i>=1; i--)
             { cout << list.get() << ","; list.back();}
```

Example of building & displaying Doubly Linked List

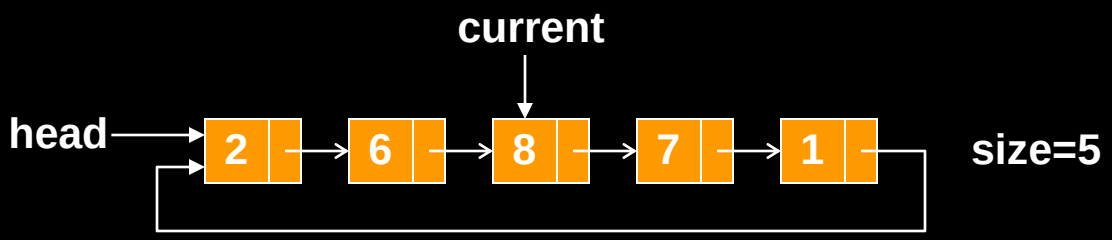
```
 cout << "\n\nDeleting Element 4 ";
 list.positionAt(4);
  list.remove();
 cout << "\n\nThe lenght of list = " << list.length();
 list.start();
  cout << "\nList Element: ";
 while (list.next() == true)
cout << list.get() << ",";
}
```

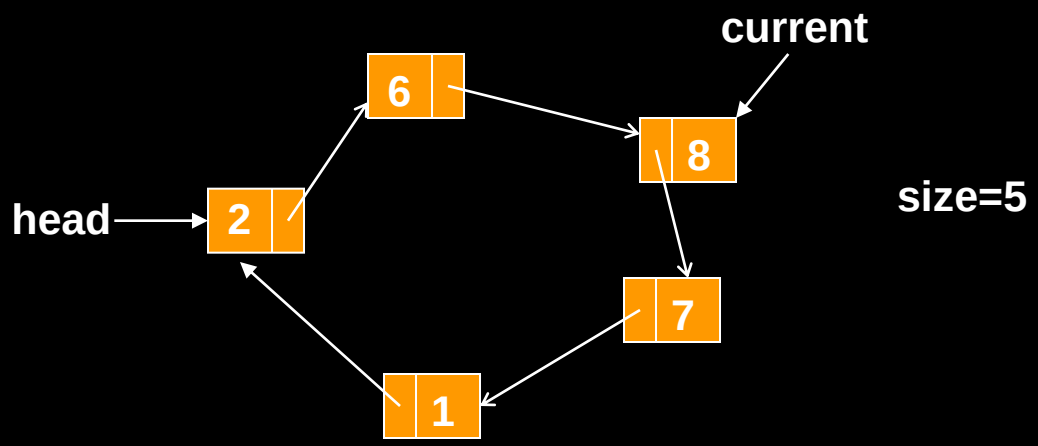
Circularly-linked lists

- The next field in the last node in a singly-linked list is set to NULL.
- Moving along a singly-linked list has to be done in a watchful manner.
- Doubly-linked lists have two NULL pointers: prev in the first node and next in the last node.
- A way around this potential hazard is to link the last node with the first node in the list to create a *circularly-linked list*.

Cicularly Linked List

- Two views of a circularly linked list:



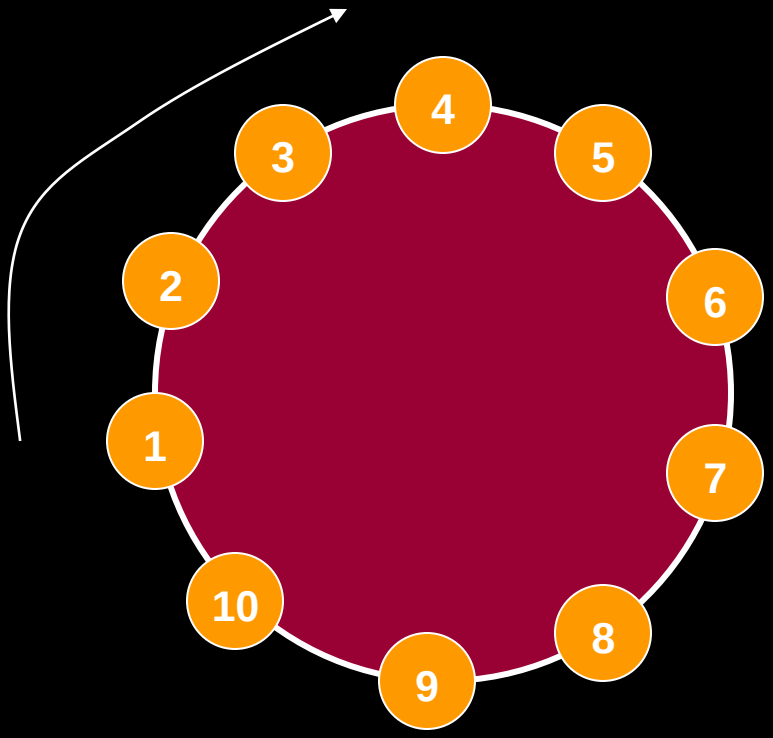


Josephus Problem

- A case where circularly linked list comes in handy is the solution of the *Josephus Problem*.
- Consider there are 10 persons. They would like to choose a leader.
- The way they decide is that all 10 sit in a circle.
- They start a count with person 1 and go in clockwise direction and skip 3. Person 4 reached is eliminated.
- The count starts with the fifth and the next person to go is the fourth in count.
- Eventually, a single person remains.

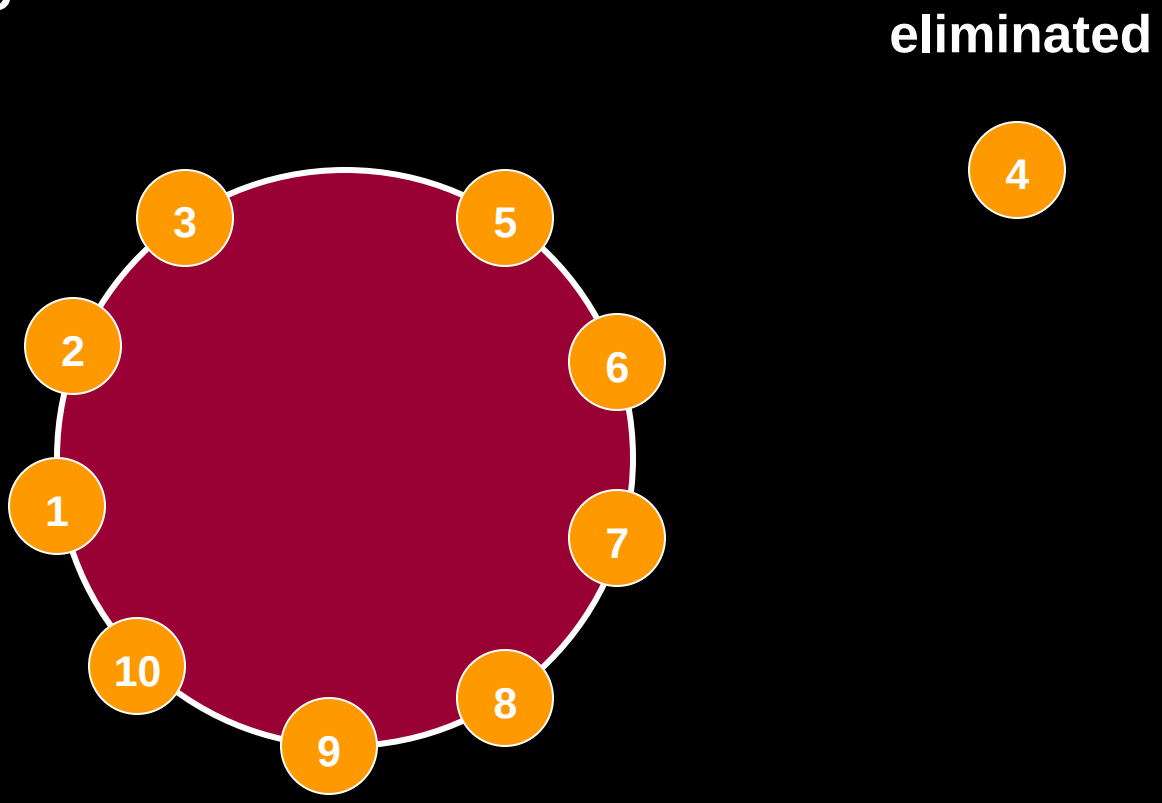
Josephus Problem

- N=10, M=3



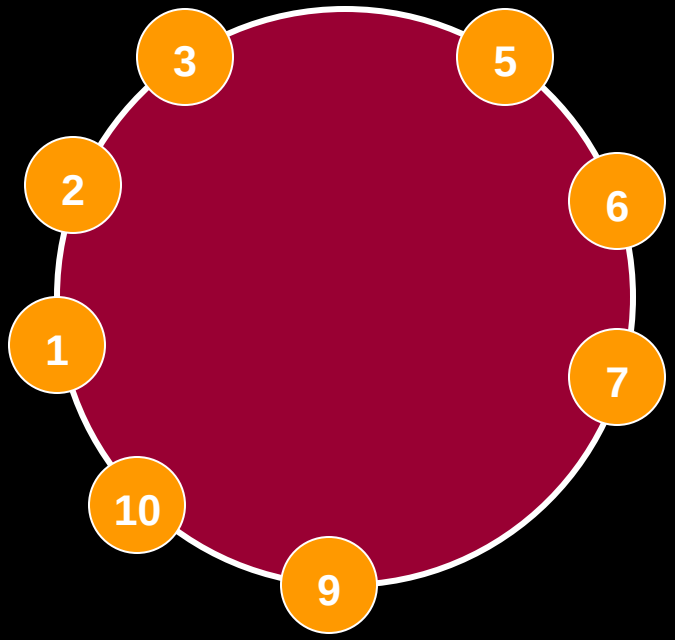
Josephus Problem

- N=10, M=3



Josephus Problem

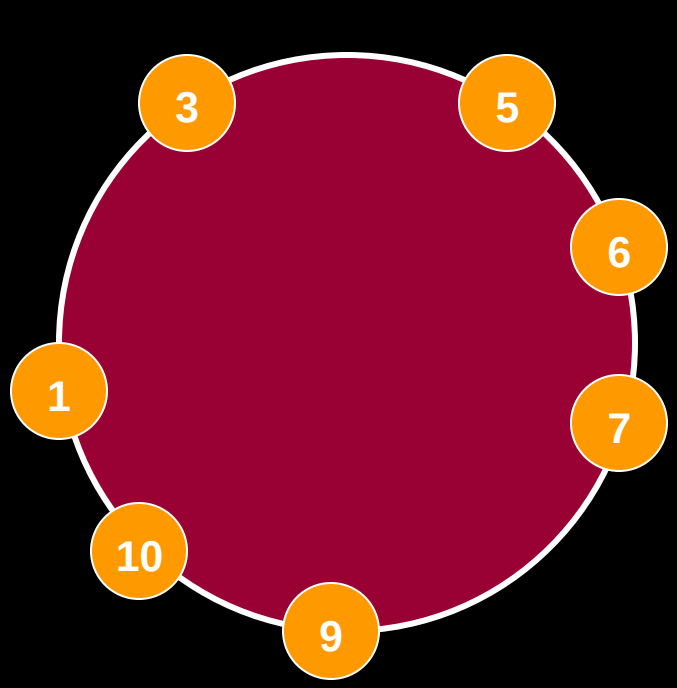
- N=10, M=3



eliminated

Josephus Problem

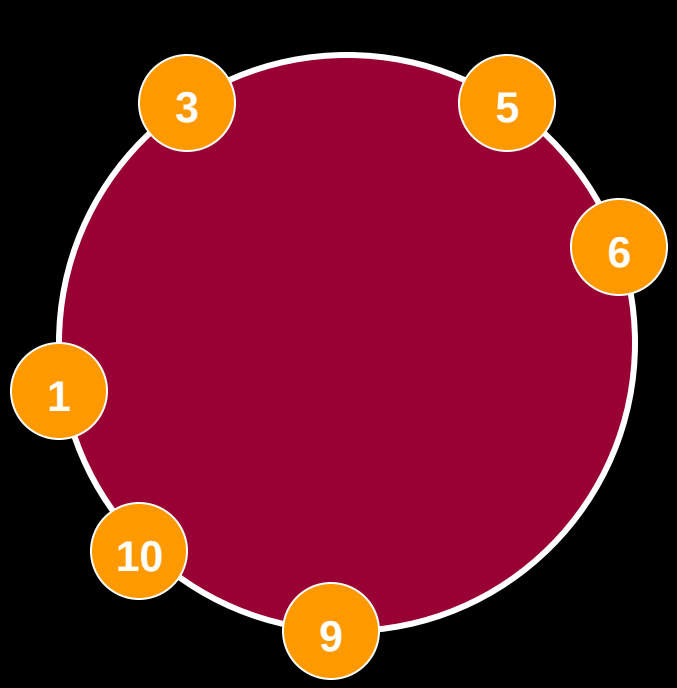
- N=10, M=3



eliminated

Josephus Problem

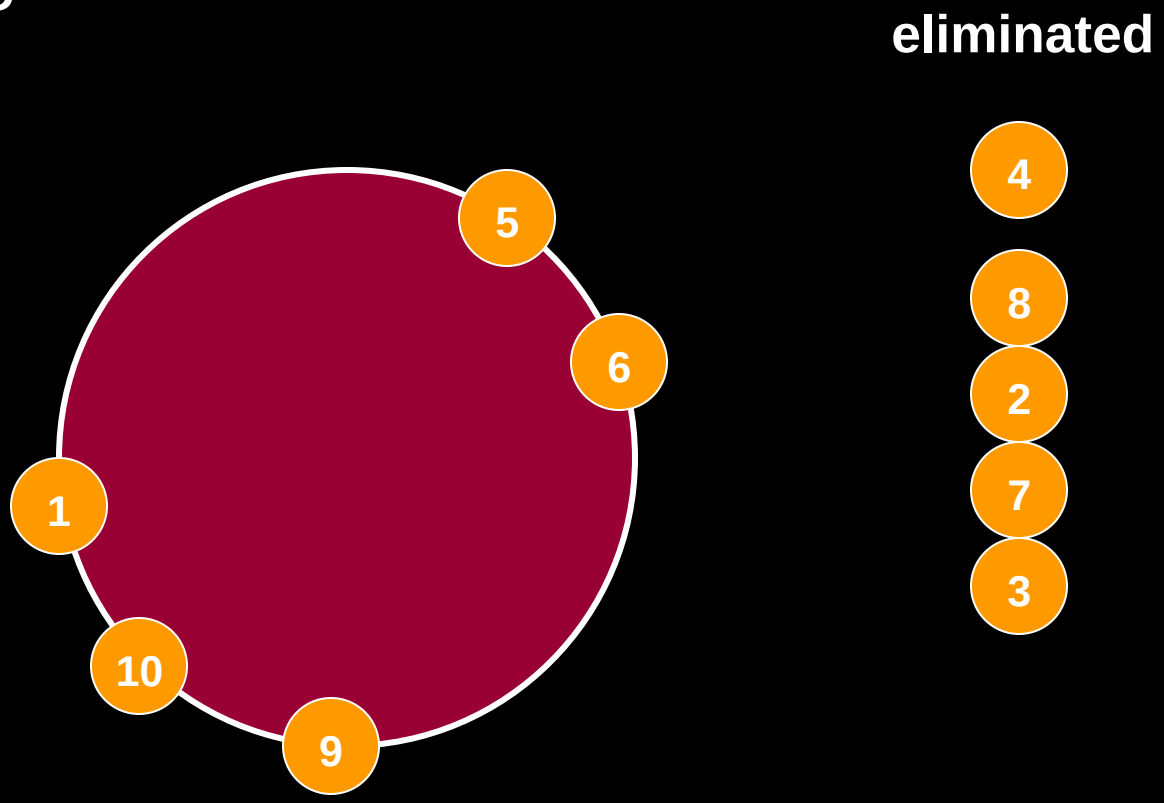
- N=10, M=3



eliminated

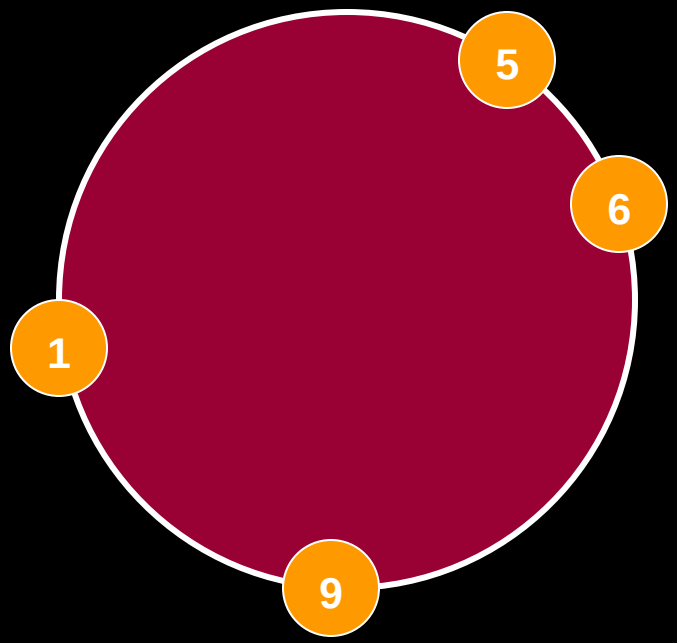
Josephus Problem

- N=10, M=3



Josephus Problem

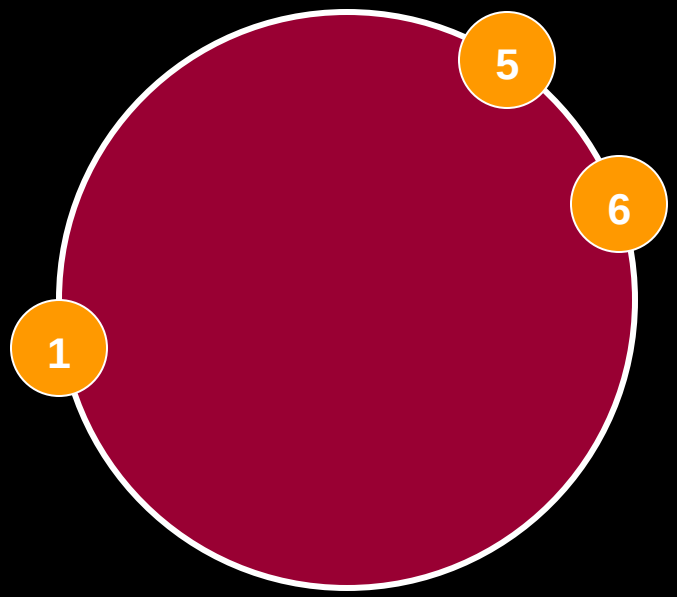
- N=10, M=3



eliminated

Josephus Problem

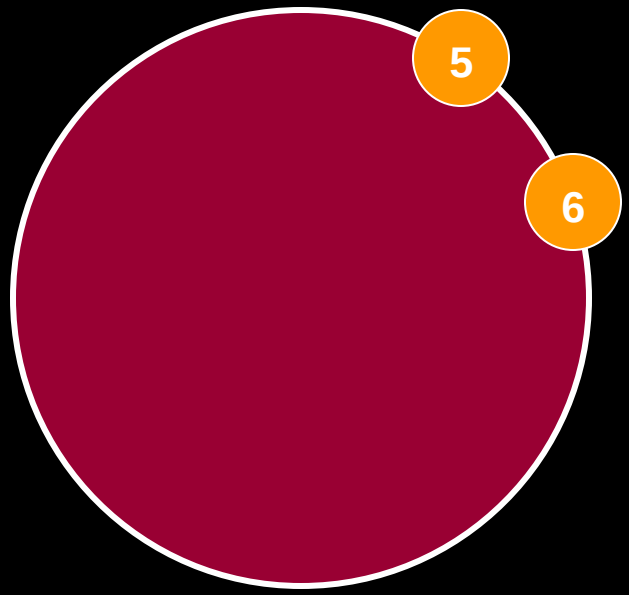
- N=10, M=3



eliminated

Josephus Problem

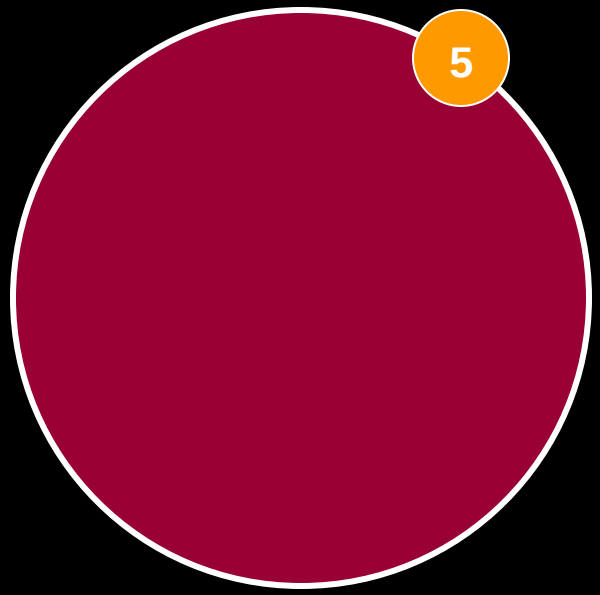
- N=10, M=3



eliminated

Josephus Problem

- N=10, M=3



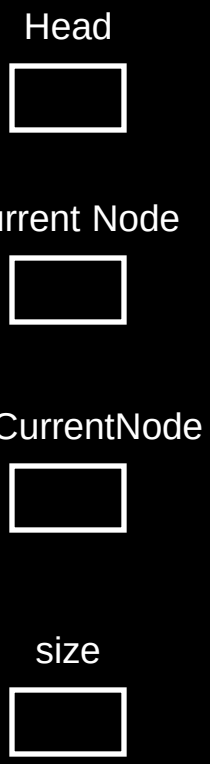
1. Declaring a class of Nodes
b. Declaring accessor functions for Node object

```
class CNode {
private:
        int data;
        CNode *nextNode;

public:
        int get() { return data; };
        void set(int data) { this->data = data; };

        CNode *getNext() { return nextNode; };
        void setNext(CNode *nextNode)
                        { this->nextNode = nextNode; };
};
```

2. Declaring a class for list of Nodes

b. Declaring constructor to start the list

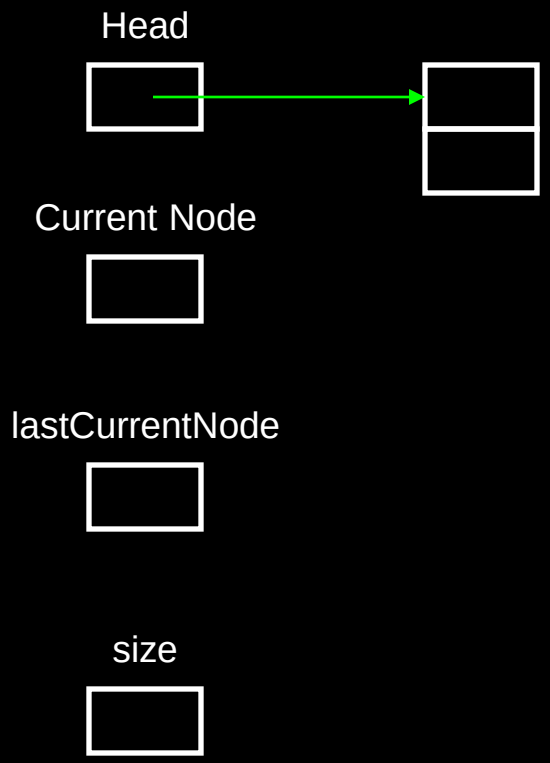
```
#include <stdlib.h>
#include "\tc\linklist\CNode.cpp"

class CList {

private:
        int size;
        CNode *headNode;
        CNode *currentNode, *lastCurrentNode;

public:
        // Constructor
        CList() {
                headNode = new CNode;
                headNode->setNext(headNode);
                currentNode = NULL;
                size = 0;
        };
```



2. Declaring a class for list of Nodes

b. Declaring constructor to start the list

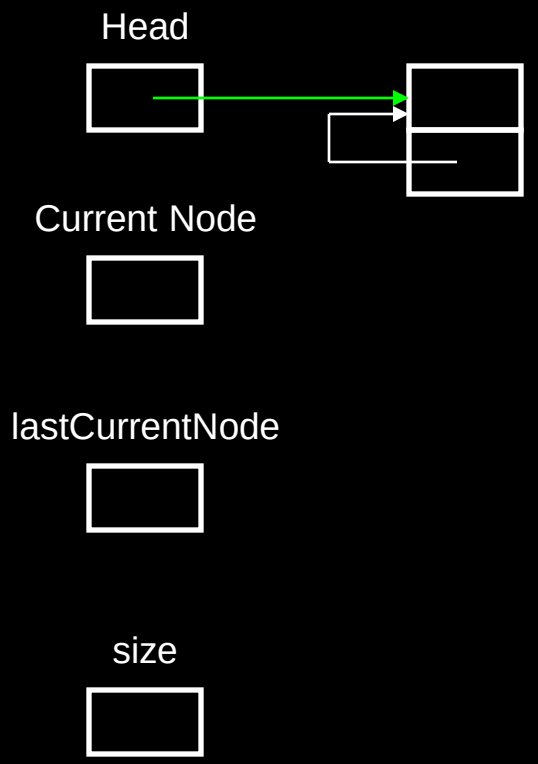
```
#include <stdlib.h>
#include "\tc\linklist\CNode.cpp"

class CList {

private:
        int size;
        CNode *headNode;
        CNode *currentNode, *lastCurrentNode;

public:
        // Constructor
        CList() {
                headNode = new CNode;
                headNode->setNext(headNode);
                currentNode = NULL;
                size = 0;
        };
```



2. Declaring a class for list of Nodes

b. Declaring constructor to start the list

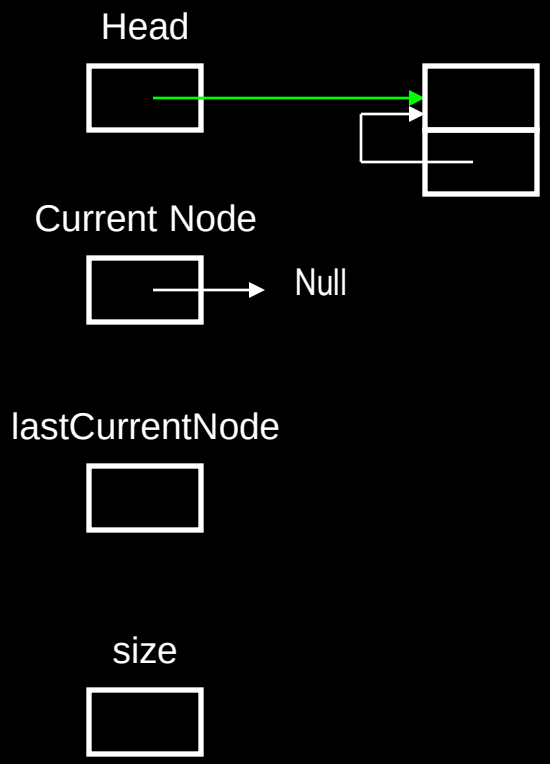
```
#include <stdlib.h>
#include "\tc\linklist\CNode.cpp"

class CList {

private:
        int size;
        CNode *headNode;
        CNode *currentNode, *lastCurrentNode;

public:
        // Constructor
        CList() {
                headNode = new CNode;
                headNode->setNext(headNode);
                currentNode = NULL;
                size = 0;
        };
```



2. Declaring a class for list of Nodes

b. Declaring constructor to start the list

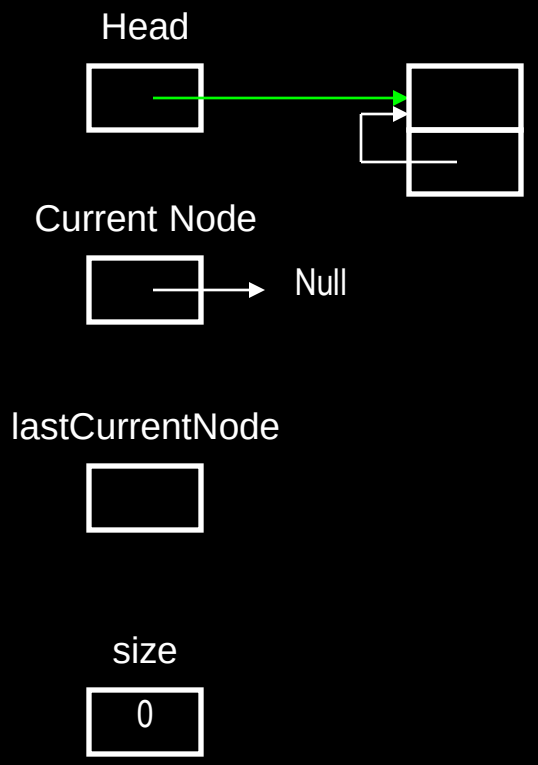
```
#include <stdlib.h>
#include "\tc\linklist\CNode.cpp"

class CList {

private:
        int size;
        CNode *headNode;
        CNode *currentNode, *lastCurrentNode;

public:
        // Constructor
        CList() {
                headNode = new CNode;
                headNode->setNext(headNode);
                currentNode = NULL;
                size = 0;
        };
```



2. Declaring a class for list of Nodes

b. Declaring constructor to start the Circular list (construct empty list)

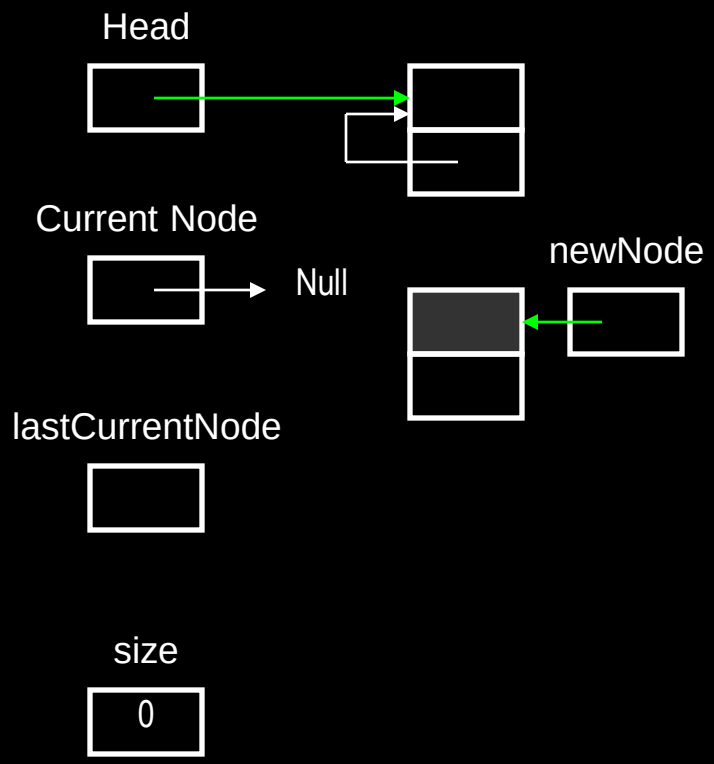
```
#include <stdlib.h>
#include "\tc\linklist\CNode.cpp"

class CList {

private:
        int size;
        CNode *headNode;
        CNode *currentNode, *lastCurrentNode;

public:
        // Constructor
        CList() {
                headNode = new CNode;
                headNode->setNext(headNode);
                currentNode = NULL;
                size = 0;
        };
```



3. Adding or Appending data in Circularly Linked List

a. Create a new node in heap and store the data in data field

```
void add(int addObject) {
   CNode *newNode = new CNode;
   newNode->set(addObject);
   }
```

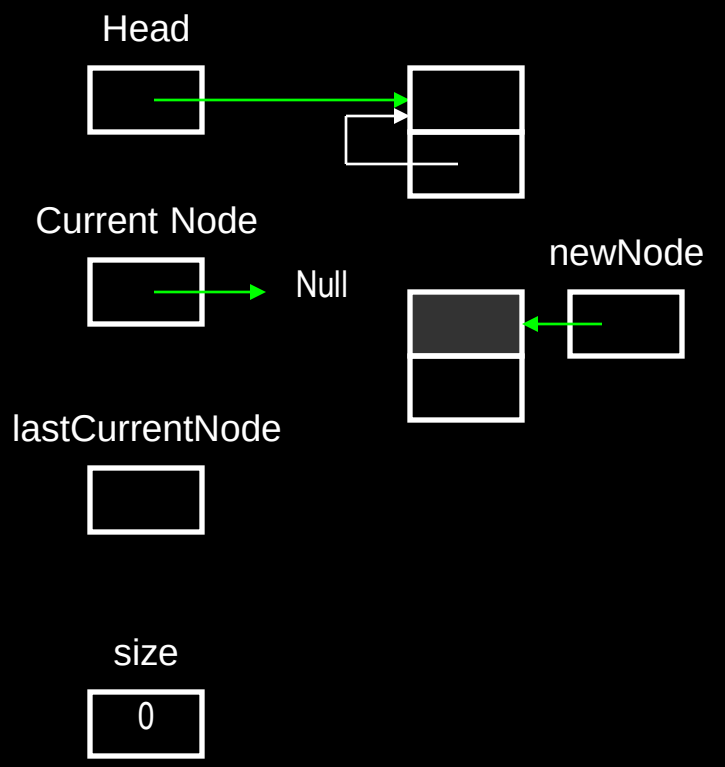


3. Adding or Appending data in Circularly Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
headNode -> setNext( newNode );
newNode -> setNext( headNode );
lastCurrentNode = headNode;
currentNode =  newNode;
size++;
```

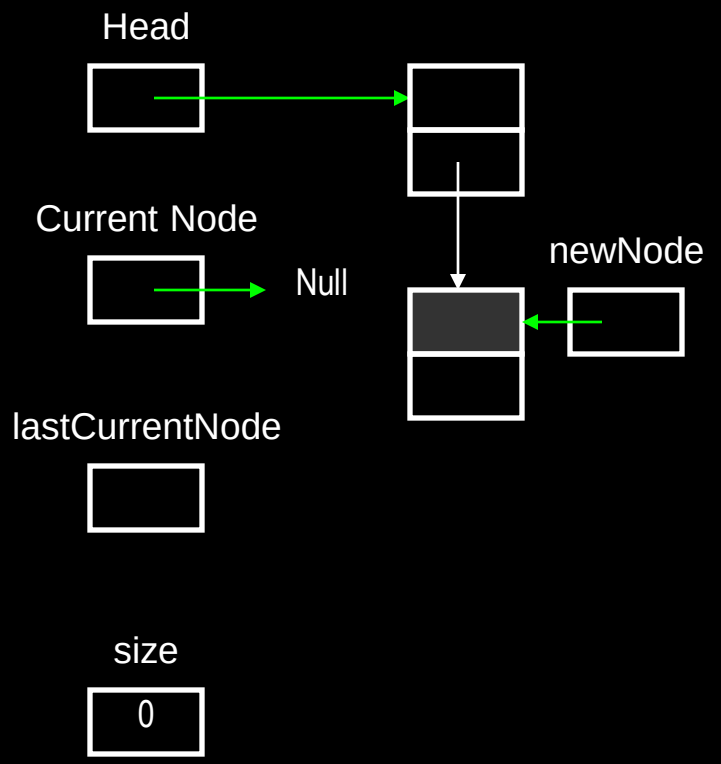


3. Adding or Appending data in Circularly Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
headNode -> setNext( newNode );
newNode -> setNext( headNode );
lastCurrentNode = headNode;
currentNode =  newNode;
size++;
```

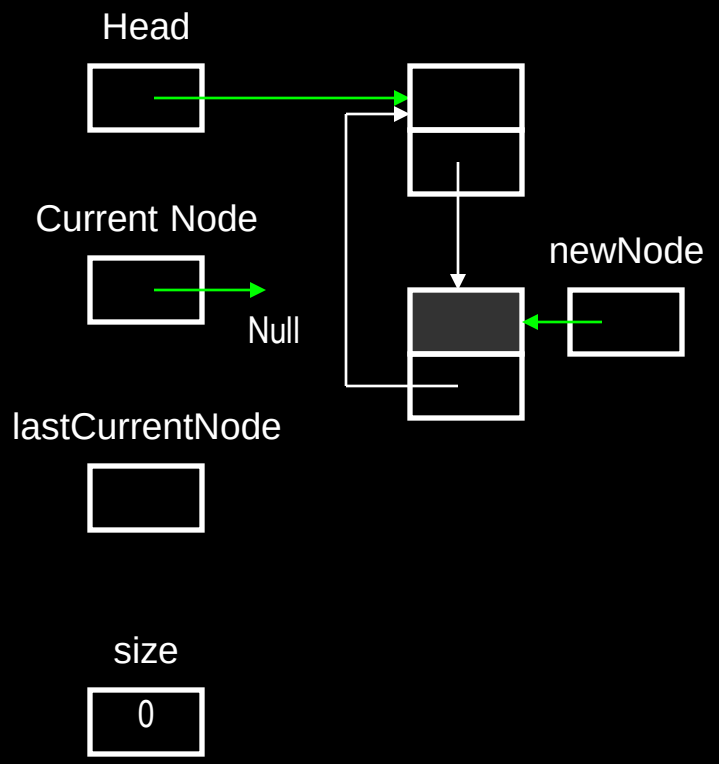


3. Adding or Appending data in Circularly Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
headNode -> setNext( newNode );
newNode -> setNext( headNode );
lastCurrentNode = headNode;
currentNode =  newNode;
size++;
```

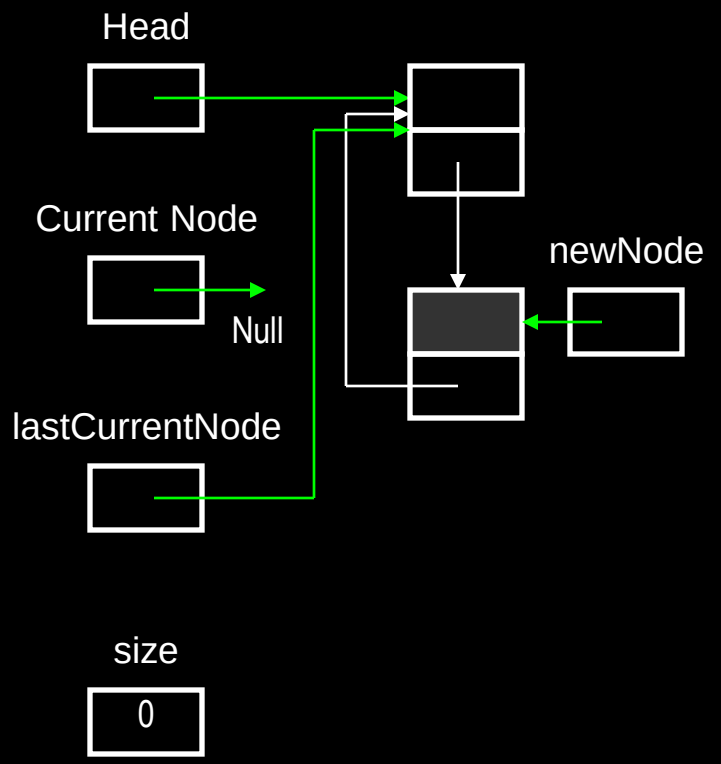


3. Adding or Appending data in Circularly Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
headNode -> setNext( newNode );
newNode -> setNext( headNode );
lastCurrentNode = headNode;
currentNode =  newNode;
size++;
```

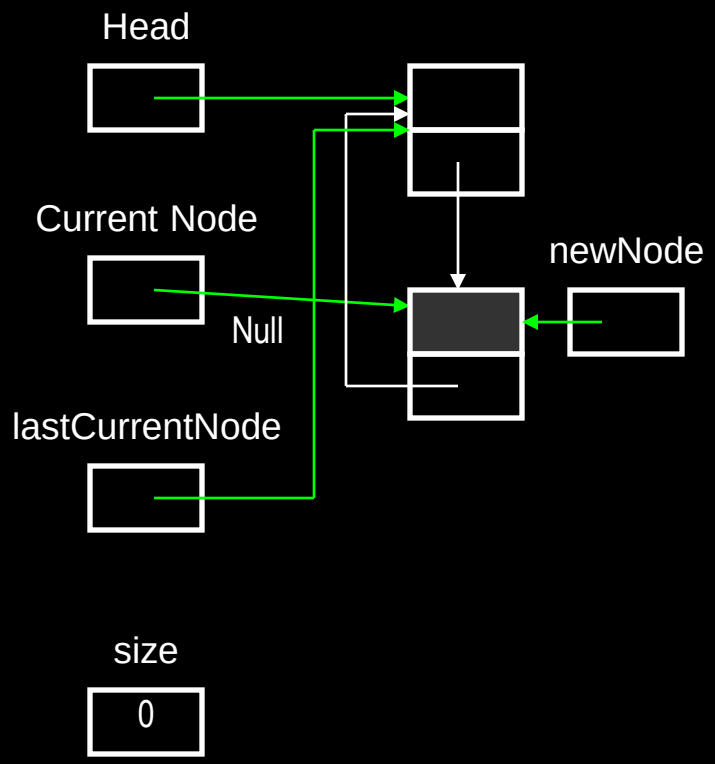


3. Adding or Appending data in Circularly Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
headNode -> setNext( newNode );
newNode -> setNext( headNode );
lastCurrentNode = headNode;
currentNode =  newNode;
size++;
```

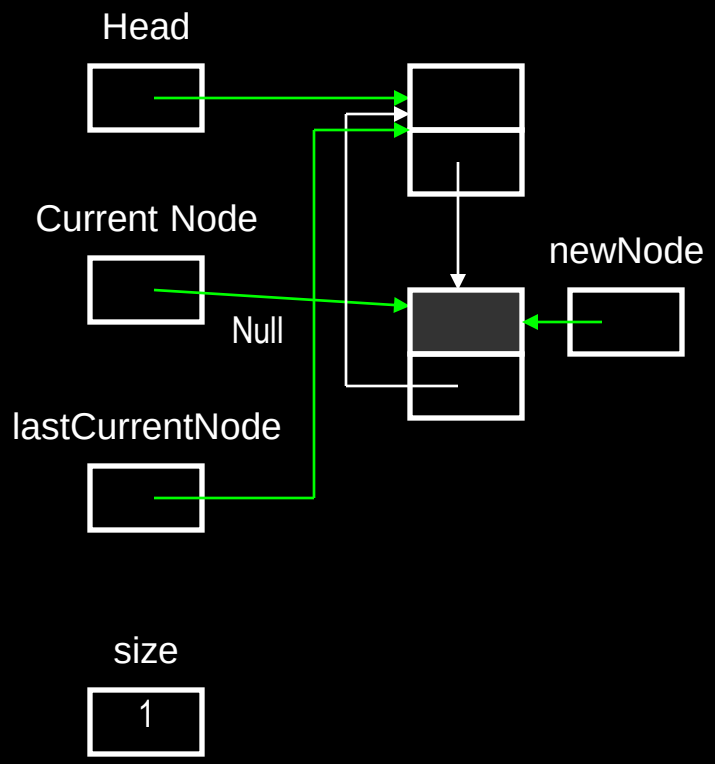


3. Adding or Appending data in Circularly Linked List

b. Connect the newNode with the list

i. If List is empty (`currentNode==Null`)

```
headNode -> setNext( newNode );
newNode -> setNext( headNode );
lastCurrentNode = headNode;
currentNode =  newNode;
size++;
```

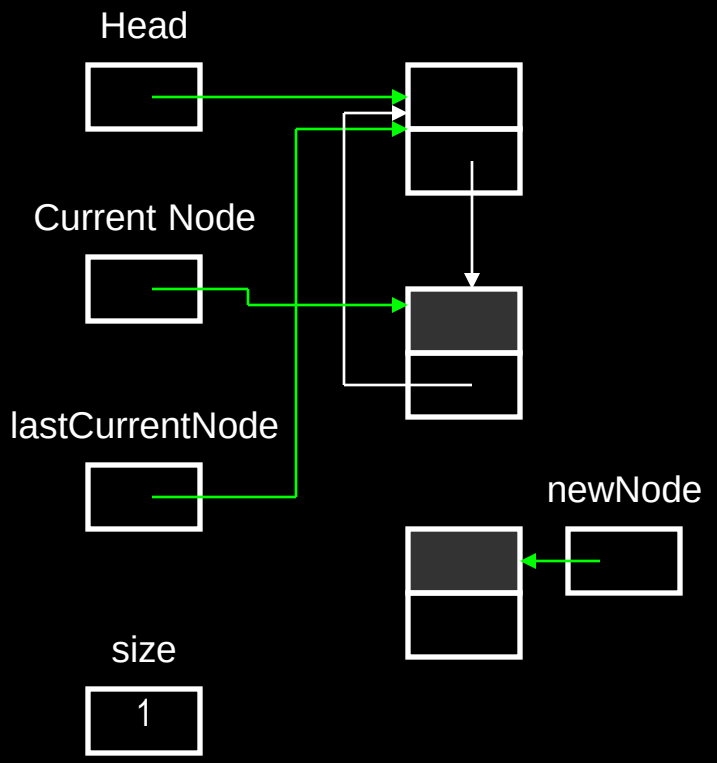


3. Adding or Appending data in Circularly Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(headNode);

currentNode->setNext( newNode );

lastCurrentNode = currentNode;

currentNode = newNode;

size++;
```

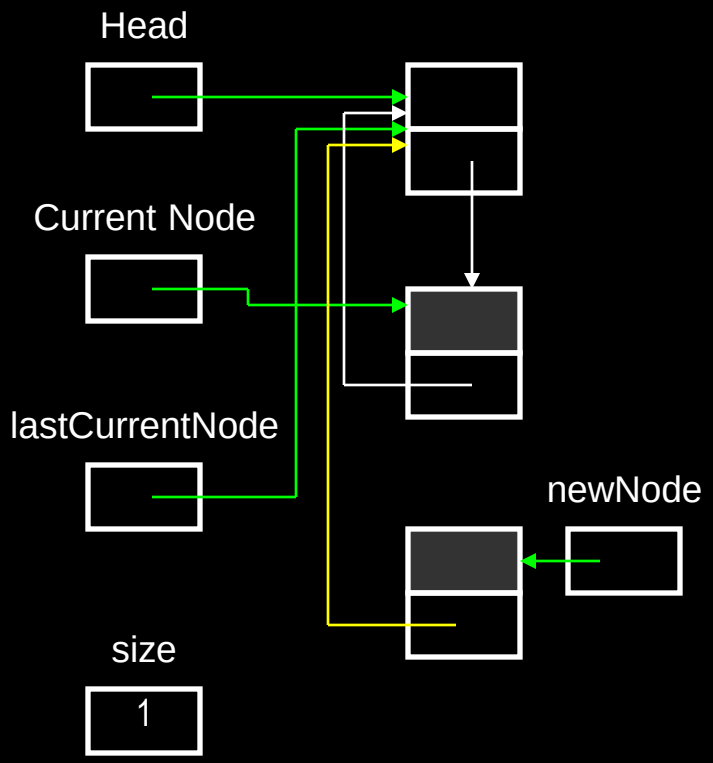


3. Adding or Appending data in Circularly Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(headNode);

currentNode->setNext( newNode );

lastCurrentNode = currentNode;

currentNode = newNode;

size++;
```

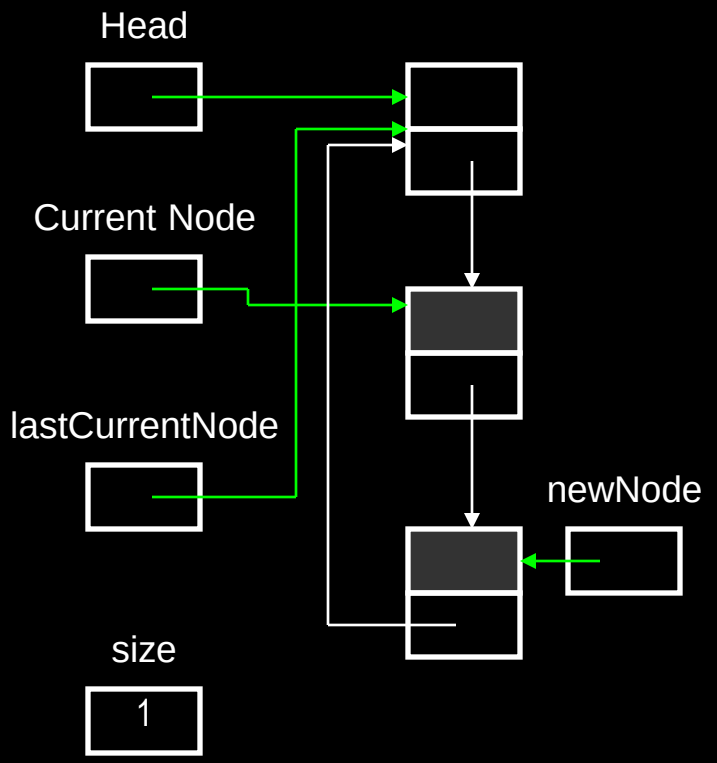


3. Adding or Appending data in Circularly Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(headNode);

currentNode->setNext( newNode );

lastCurrentNode = currentNode;

currentNode = newNode;

size++;
```

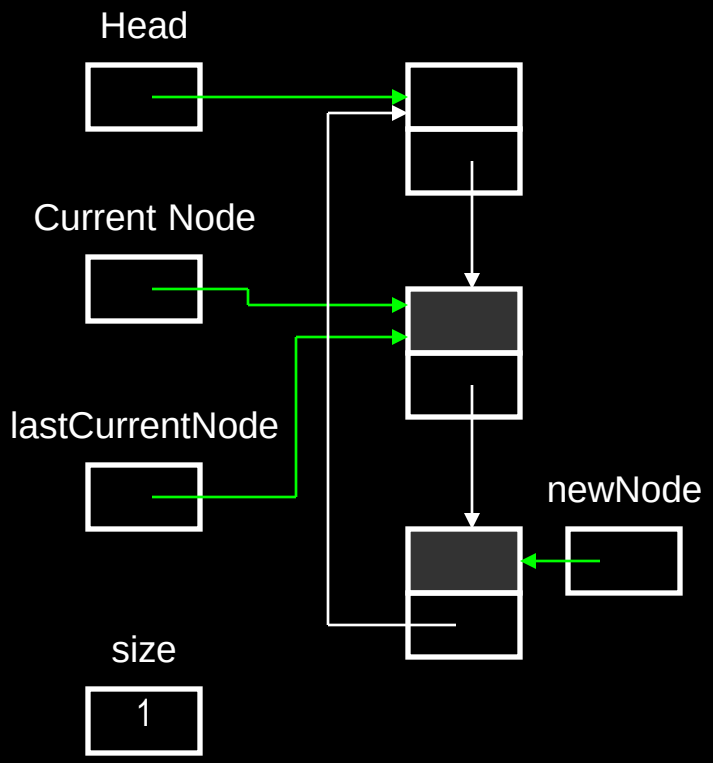


3. Adding or Appending data in Circularly Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(headNode);

currentNode->setNext( newNode );

lastCurrentNode = currentNode;

currentNode = newNode;

size++;
```

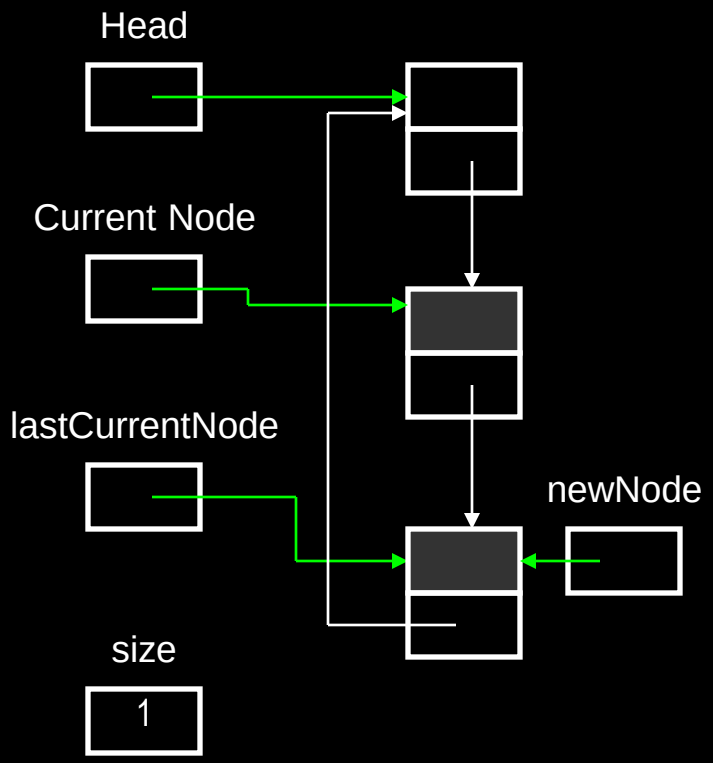


3. Adding or Appending data in Circularly Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(headNode);

currentNode->setNext( newNode );

lastCurrentNode = currentNode;

currentNode = newNode;

size++;
```

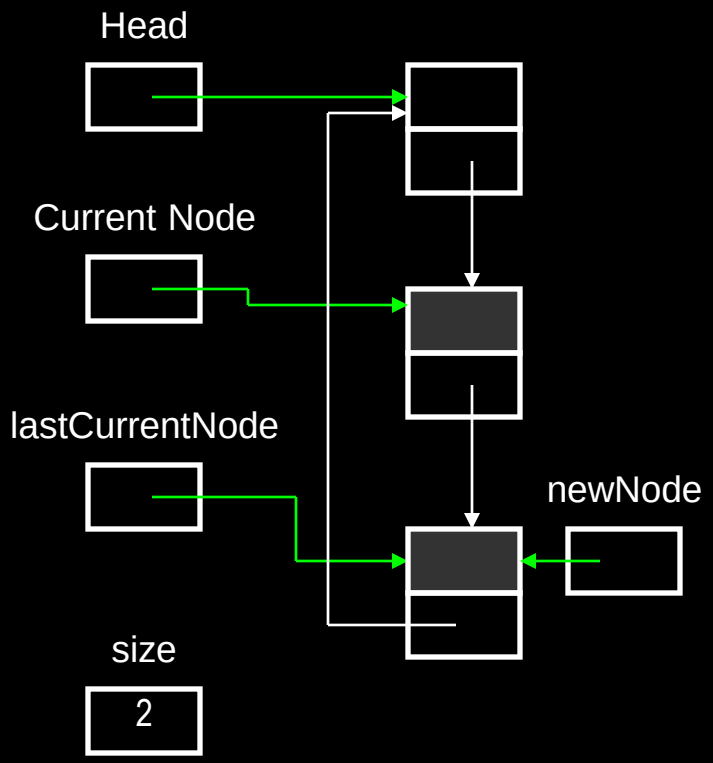


3. Adding or Appending data in Circularly Linked List

b. Connect the newNode with the list

ii. If List is not empty (`currentNode != Null`)

```
newNode->setNext(headNode);

currentNode->setNext( newNode );

lastCurrentNode = currentNode;

currentNode = newNode;

size++;
```

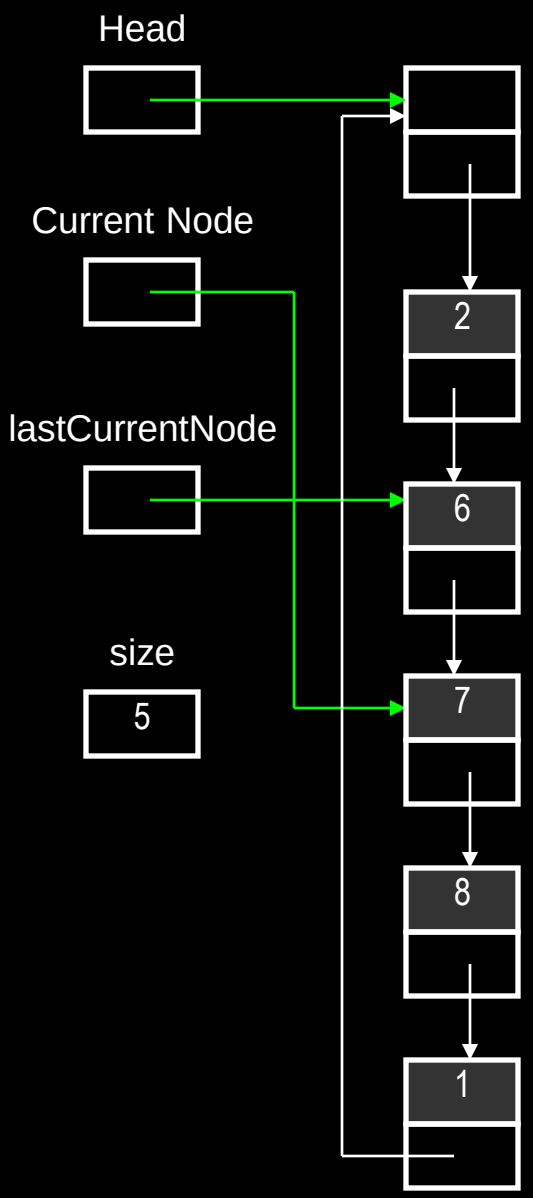


3. Adding or Appending data in Circularly Linked List

```
void add(int addObject) {
   CNode* newNode = new CNode;
   newNode->set(addObject);
   if  (currentNode != NULL ){
        newNode->setNext(headNode);
        currentNode->setNext( newNode );
        lastCurrentNode = currentNode;
        currentNode = newNode;
        }
   else
       {headNode -> setNext( newNode );
        newNode -> setNext( headNode );
        lastCurrentNode = headNode;
        currentNode =  newNode;
        }
   size++;
};
```

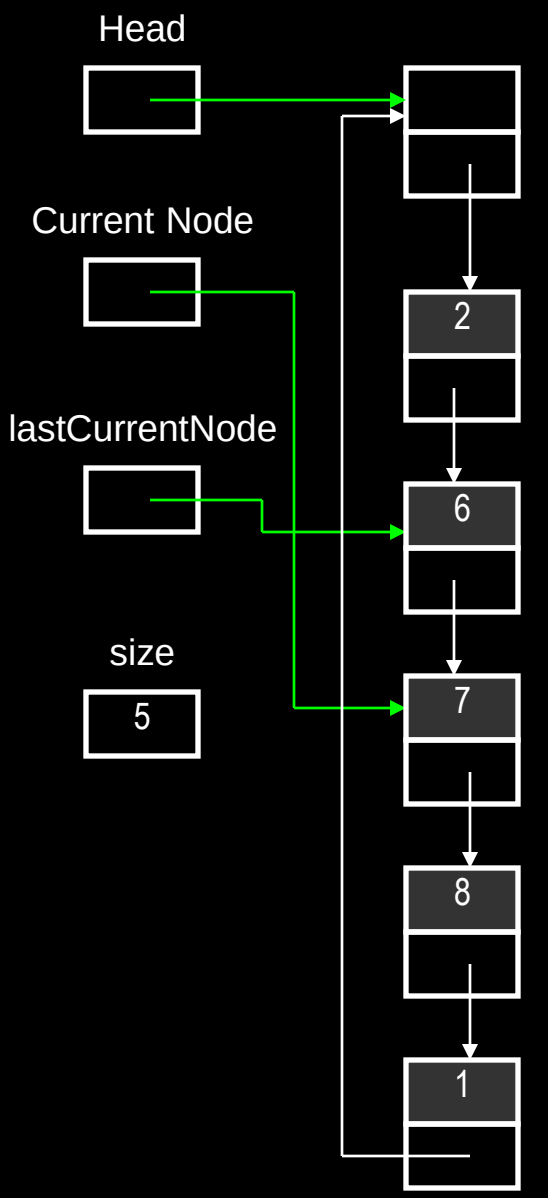
Circularly Linked List Basic Operations
Moving current to next node

```
void next() {
   lastCurrentNode = currentNode;
   currentNode = currentNode->getNext();

   if (currentNode==headNode)
     {
      lastCurrentNode = currentNode;
      currentNode = currentNode->getNext();
     }
};
```



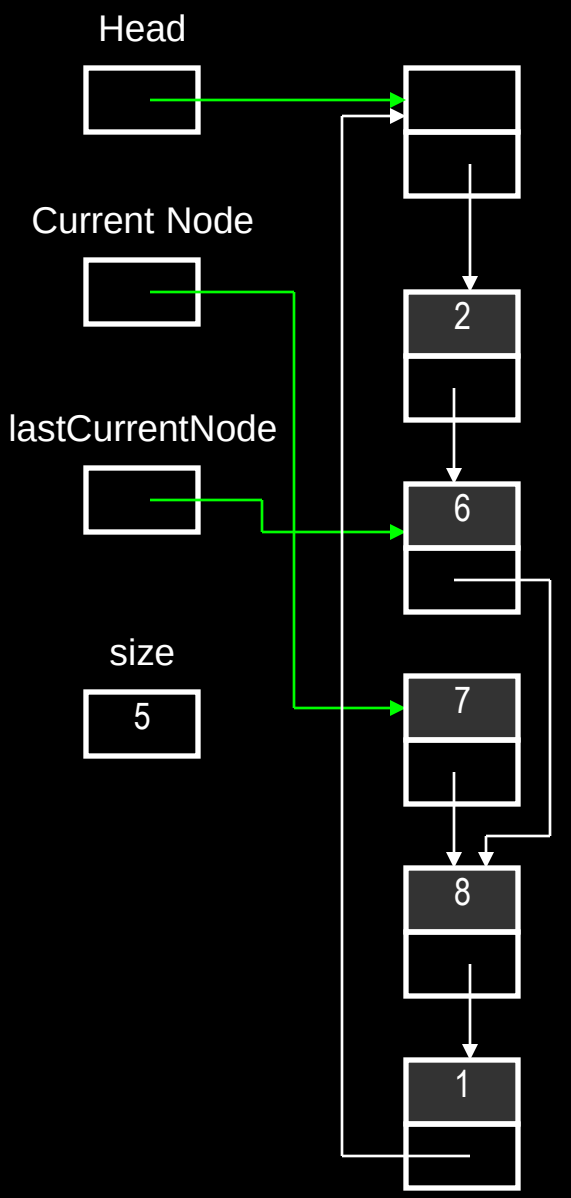
Circularly Linked List Basic Operations
deleting a node at current position - 1

```
void remove() {
  if( size != 0)
    {
    lastCurrentNode->setNext(currentNode->getNext());
    delete currentNode;
    currentNode = lastCurrentNode->getNext();
    size--;
   }
};
```



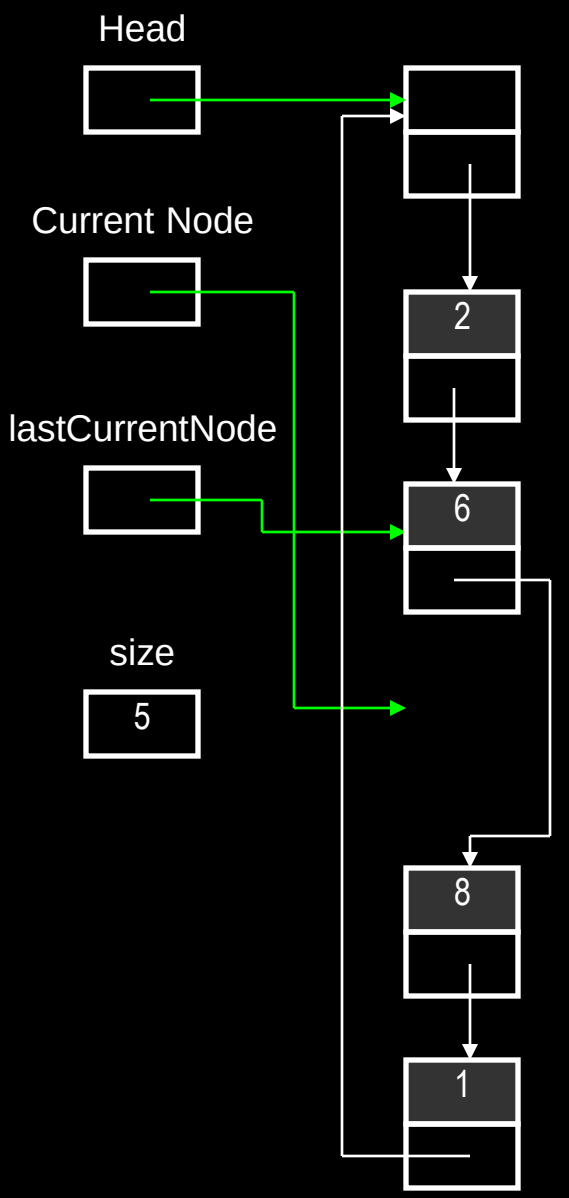
Circularly Linked List Basic Operations
deleting a node at current position -2

```
void remove() {
  if( size != 0)
    {
    lastCurrentNode->setNext(currentNode->getNext());
    delete currentNode;
    currentNode = lastCurrentNode->getNext();
    size--;
   }
};
```



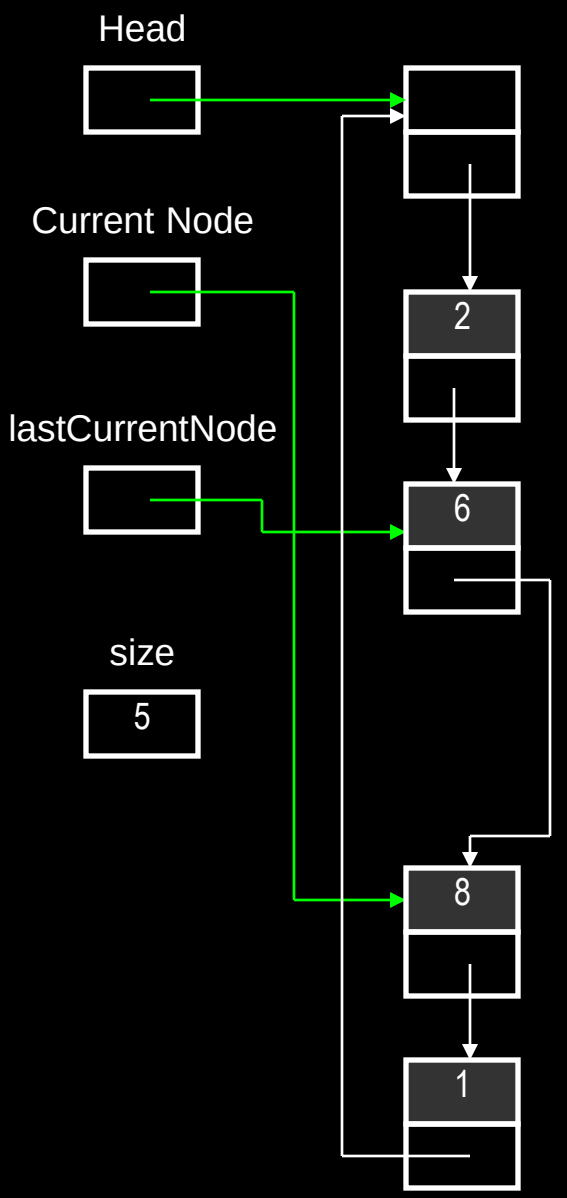
Circularly Linked List Basic Operations
deleting a node at current position -3

```
void remove() {
  if( size != 0)
    {
    lastCurrentNode->setNext(currentNode->getNext());
    delete currentNode;
    currentNode = lastCurrentNode->getNext();
    size--;
   }
};
```



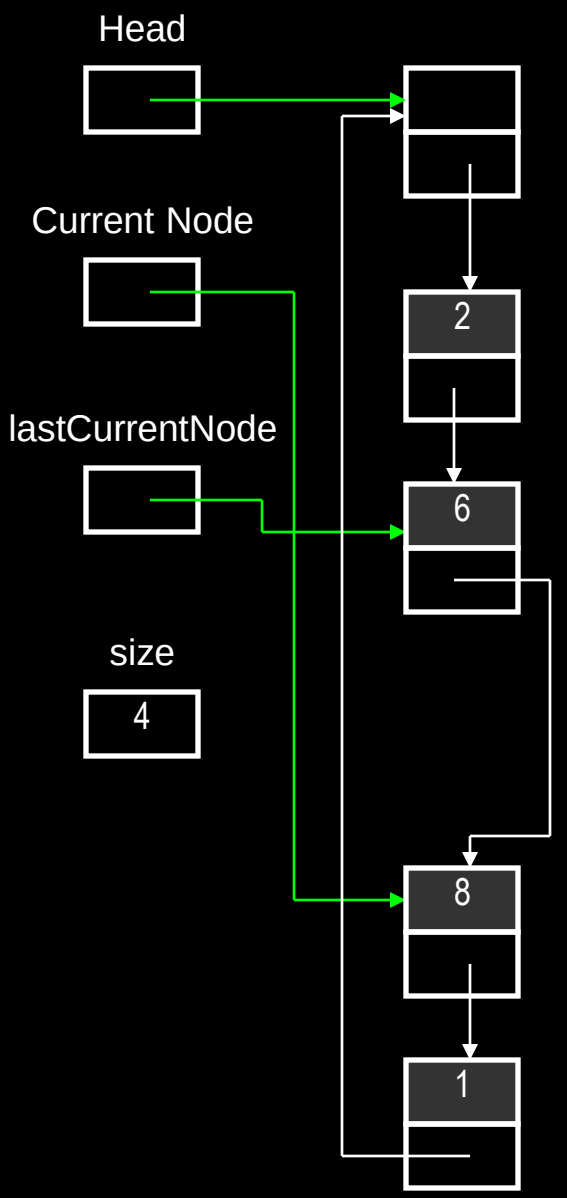
Circularly Linked List Basic Operations
deleting a node at current position -4

```
void remove() {
  if( size != 0)
    {
    lastCurrentNode->setNext(currentNode->getNext());
    delete currentNode;
    currentNode = lastCurrentNode->getNext();
    size--;
   }
};
```



Circularly Linked List Basic Operations
deleting a node at current position -4

```
void remove() {
  if( size != 0)
    {
    lastCurrentNode->setNext(currentNode->getNext());
    delete currentNode;
    currentNode = lastCurrentNode->getNext();
    size--;
   }
};
```



Circularly Linked List Summary

josephus.cpp

```
#include <iostream.h>
void main()
{
        CList list;
        int i, N=10, M=3;
        for(i=1; i <= N; i++ )
        list.add(i);
        list.start();
        while( list.length() > 1 ) {
             for(i=1; i <= M; i++ )
               list.next();
             cout << "remove: " << list.get() << endl;
             list.remove();
         }
         cout << "leader is: " << list.get() << endl;
}
```

Josephus Problem

- Using a circularly-linked list made the solution trivial.
- The solution would have been more difficult if an array had been used.
- This illustrates the fact that the choice of the appropriate data structures can significantly simplify an algorithm. It can make the algorithm much faster and efficient.
- Later we will see how some elegant data structures lie at the heart of major algorithms.
- An entire CS course “Design and Analysis of Algorithms” is devoted to this topic.

Abstract Data Type

- We have looked at four different implementations of the List data structures:
  - Using arrays
  - Singly linked list
  - Doubly linked list
  - Circularly linked list.
- The interface to the List stayed the same, i.e., add(), get(), next(), start(), remove() etc.
- The list is thus an abstract data type; we use it without being concerned with how it is implemented.

Abstract Data Type

- What we care about is the methods that are available for use with the List ADT.
- We will follow this theme when we develop other ADT.
- We will publish the interface and keep the freedom to change the implementation of ADT without effecting users of the ADT.
- The C++ classes provide us the ability to create such ADTs.