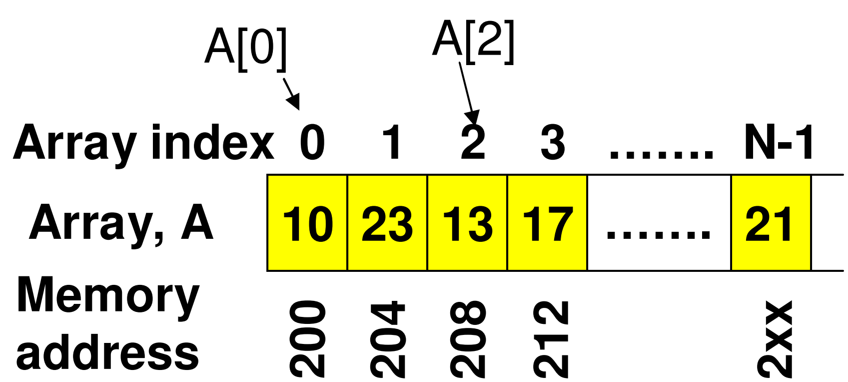
Module 2:
List ADT

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

List ADT

- A collection of entities of the same data type
- List ADT (static)
  - Functionalities (logical view)
    - Store a given number of elements of a given data type
    - Write/modify an element at a particular position
    - Read an element at a particular position
- Implementation:
  - Arrays: A contiguous block of memory of a certain size, allocated at the time of creation/initialization
    - Time complexity to read and write/modify are Θ(1) each



Code 1(C++): Static List Implementation using Arrays

```
#include <iostream>
using namespace std;

class List{
   private:
        int *array;

   public:
        List(int size){
                array = new int[size];
        }

        void write(int index, int data){
                array[index] = data;
        }

        int read(int index){
                return array[index];
        }
 };
```

```
int main(){

    int listSize;

    cout << "Enter list size: ";
    cin >> listSize;

    List integerList(listSize);

    for (int i = 0; i < listSize; i++){
            int value;
            cout << "Enter element # " << i << " : ";
            cin >> value;
            integerList.write(i, value);
    }

return 0;
}
```

Code 1(Java): Static List Implementation using Arrays

```
class List{

        private  int array[];

        public List(int size){
                array = new int[size];
        }

        public void write(int index, int data){
                array[index] = data;
        }

        public int read(int index){
                return array[index];
        }

}
```

```
import java.util.*;

class StaticListArray{

	public static void main(String[] args){

	Scanner input = new Scanner(System.in);

	int listSize;

	System.out.print("Enter list size: ");
	listSize = input.nextInt();

	List integerList = new List(listSize);

	for (int i = 0; i < listSize; i++){

		int value;
		System.out.print("Enter element # "+ i+" : ");
		value = input.nextInt();

		integerList.write(i, value);
	}

}
```

Dynamic List ADT

- **Limitations with Static List**
  - The list size is fixed (during initialization); cannot be increased or decreased.
  - With a static list, the array is filled at the time of initialization and can be later only read or modified. A new element cannot be “inserted” after the initialization phase.
- **Key Features of a Dynamic List**
  - Be able to resize (increase or decrease) the list at run time. The list size need not be decided at the time of initialization. We could even start with an empty list and populate it as elements are to be added.
  - Be able to insert or delete an element at a particular index at any time.
- **Performance Bottleneck**
  - When we increase the size of the list (i.e., increase the size of the array that stores the elements), the contents of the array need to be copied to a new memory block, element by element. ➔ O(n) time.
  - Hence, even though, we could increase the array size by one element at a time, the ‘copy’ operation is a performance bottleneck and the standard procedure is to double the size of the array (list) whenever the list gets full.
  - A delete operation also takes O(n) time as elements are to be shifted one cell to the left.

Code 2: Code for Dynamic List ADT Implementation using Arrays

Variables and Constructor (C++)

```
private:
  int *array;
  int maxSize;
  int endOfArray;

public:
   List(int size){
         maxSize = size;
         array = new int[maxSize];
         endOfArray = -1;
   }
```

Variables and Constructor (Java)

```
private  int array[];
private  int maxSize;
private  int endOfArray;

public List(int size){
         maxSize = size;
         array = new int[maxSize];
         endOfArray = -1;
}
```

isEmpty (C++)

```
bool isEmpty(){
   if (endOfArray = = -1)
         return true;
   return false;
}
```

isEmpty (Java)

```
public boolean isEmpty(){
   if (endOfArray == -1)
         return true;
   return false;
}
```

Code 2: Insert Function (C++ and Java)

```
void insertAtIndex(int insertIndex, int data){

        // if the user enters an invalid insertIndex, the element is
        // appended to the array, after the current last element
        if (insertIndex > endOfArray+1)
                insertIndex = endOfArray+1;

        if (endOfArray == maxSize-1)
                resize(2*maxSize);

        for (int index = endOfArray; index >= insertIndex; index--)
                array[index+1] = array[index];

        array[insertIndex] = data;
        endOfArray++;

}
```

Will take O(n) time each, where n = maxSize + 1

```
void insert(int data){
        if (endOfArray == maxSize-1)
                resize(2*maxSize);

        array[++endOfArray] = data;
}
```

Code 2: Resize Function (C++)

```
void resize(int s){
        // in addition to increasing, the resize function
        // also provides the flexibility to reduce the size
        // of the array
```

Have another pointer (a temporary ptr) to refer to the starting address of the memory represented by the original array

```
        int *tempArray = array;
```

Allocating a new set of memory blocks to the 'array' variable

```
        array = new int[s];
```

Copying back the contents pointed to by the temporary array pointer to the original array

```
        for (int index = 0; index < min(s, endOfArray+1); index++){
                array[index] = tempArray[index];
        }
```

If the array size is reduced from maxSize to s, only the first 's' elements are copied. Otherwise, all the maxElements are copied

```
        maxSize = s;
}
```

new value of maxSize

Code 2: Resize Function (Java)

```
public void resize(int s){

    int tempArray[] = array;

    array = new int[s];

    for (int index = 0; index < Math.min(s, endOfArray+1); index++){
        array[index] = tempArray[index];
    }

    maxSize = s;

}
```

Have another reference (a temporary ref) to refer to the starting address of the memory represented by the original array

Allocating a new set of memory blocks to the ‘array’ variable

Copying back the contents pointed to by the temporary array reference to the original array

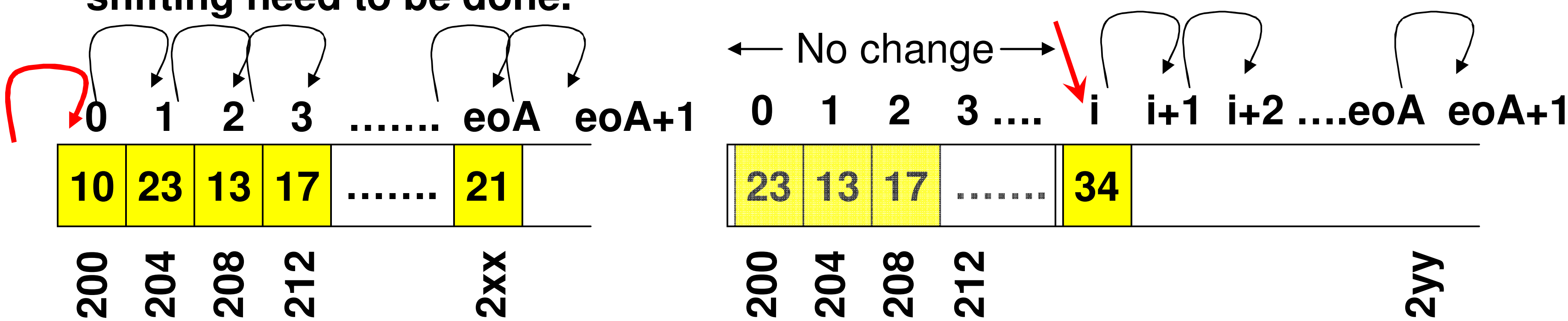
If the array size is reduced from maxSize to s, only the first ‘s’ elements are copied. Otherwise, all the maxElements are copied

new value of maxSize

Time complexity analysis for ‘Insert’: Dynamic List ADT as an Array

Insert operation

(i) Worst case: If the element is to be inserted as the first element in the array, then elements from index endOfArray(eoA) to index ‘0’ have to be shifted one position to the right. If eoA = n-1, then ‘n’ (indexes 0 to n-1) such shifting need to be done.



(ii) Best case: If the element is to be inserted at the end of the array, no shifting is needed.

(iii) In general, if the element is to be inserted at index i, then the elements from index endOfArray(eoA) to index ‘i’ need to be shifted one cell to the right.

Time complexity for insert operation: O(n)

Code 2: Other Auxiliary Functions

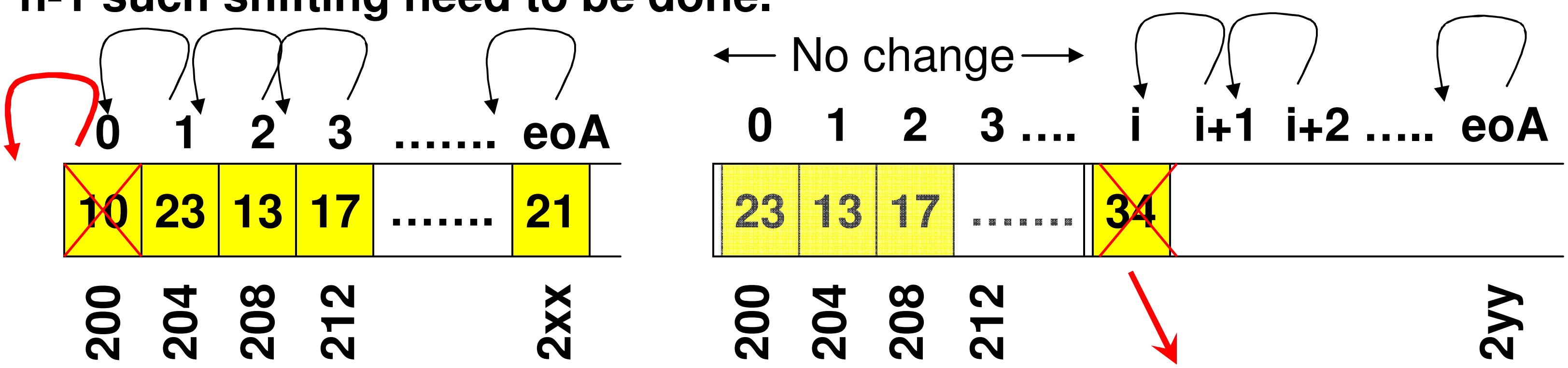
(for both C++ and Java)

```
int read(int index){
        return array[index];
}

void modifyElement(int index, int data){
        array[index] = data;
}

void deleteElement(int deleteIndex){
        // shift elements one cell to the left starting from
        // deleteIndex+1 to endOfArray-1
        // i.e., move element at deleteIndex + 1 to deleteIndex and so on

        for (int index = deleteIndex; index < endOfArray; index++)
                array[index] = array[index+1];

        endOfArray--;
}

int countList(){
        int count = 0;
        for (int index = 0; index <= endOfArray; index++)
                count++;

        return count;
}
```

Time complexity analysis for ‘Delete’: Dynamic List ADT as an Array

Delete operation

(i) Worst case: If the element to be deleted is the first element (at index 0) in the array, then the subsequent elements have to be shifted one position to the left, starting from index 1 to endOfArray (eoA) . If eoA = n-1, then n-1 such shifting need to be done.



(ii) Best case: If the element to be deleted is at the end of the array, no shifting is needed.

(iii) In general, if the element to be deleted is at index i, then the elements from index i+1 to endOfArray need to be shifted one cell to the left.

Time complexity for delete operation: O(n)

Code 2: C++ main function

```
int main(){

    int listSize;

    cout << "Enter list size: ";
    cin >> listSize;

    List integerList(1);

    for (int i = 0; i < listSize; i++){

        int value;
        cout << "Enter element # " << i << " : ";
        cin >> value;

        integerList.insert(i, value);
}
```

We will set the maximum size of the list to 1 and double it as and when needed

Code 2: Java main function

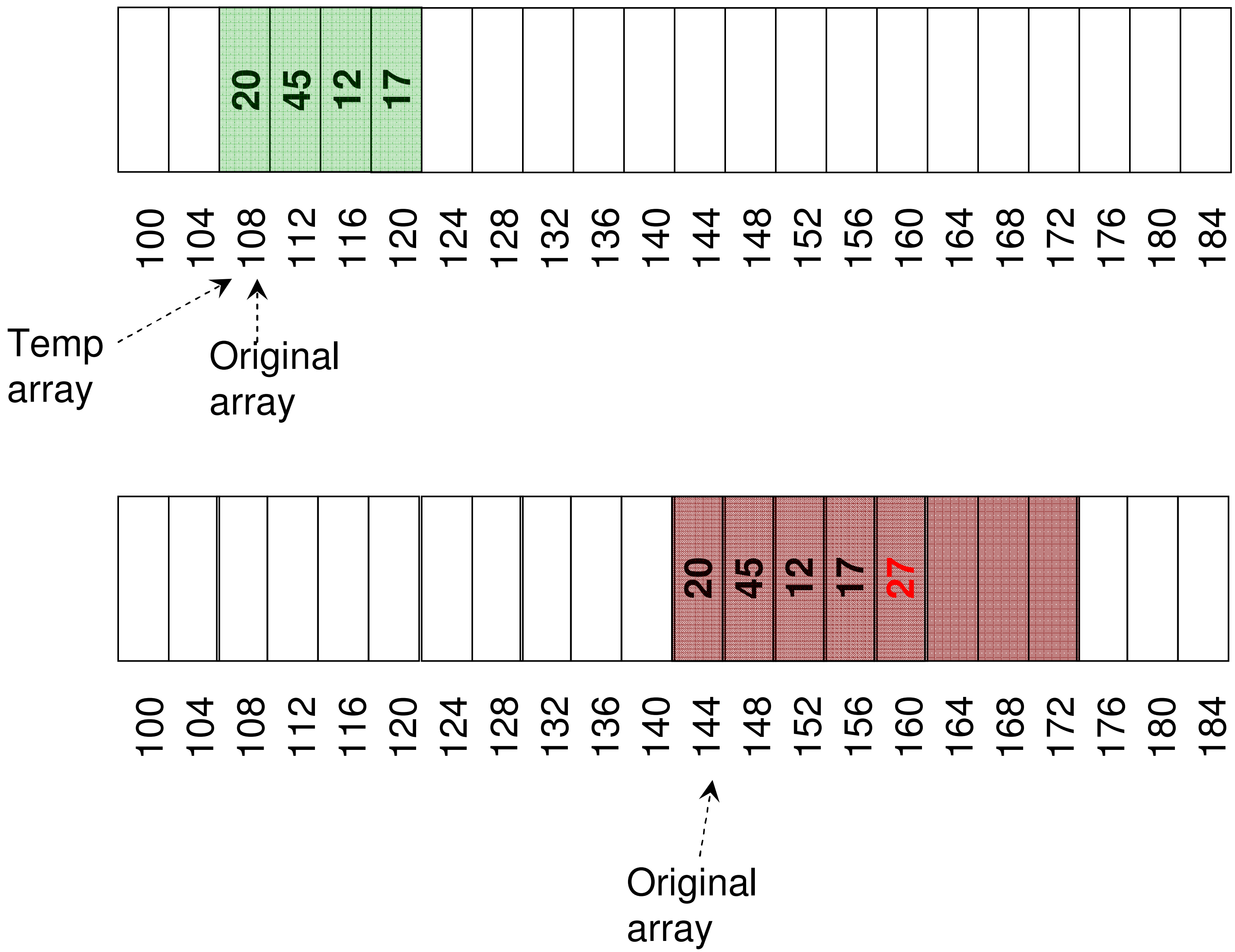
```
class DynamicListArray{

        public static void main(String[] args){

        int listSize;

        Scanner input = new Scanner(System.in);

        System.out.print("Enter list size: ");
        listSize = input.nextInt();

        List integerList = new List(1);

        for (int i = 0; i < listSize; i++){

                int value;

                System.out.print("Enter element # " + i + " : ");
                value = input.nextInt();

                integerList.insert(i, value);
        }

    }

}
```

We will set the maximum size of the list to 1 and double it as and when needed

Pros and Cons of Implementing Dynamic List using Array

- Pros: $\Theta(1)$ time to read or modify an element at a particular index
- Cons: $O(n)$ time to insert or delete an element (at any arbitrary position)
- Note: Array is a contiguous block of memory
- When we double the array size (to handle the need for more space), the memory management system of the OS needs to search for contiguous blocks of memory that is double the previous array size.
  - Sometimes, it becomes difficult to allocate a contiguous block of memory, if the requested array size is larger.
- After we double the size (say from 50,000 to 100,000 to insert just one more element), the rest of the array remains unused. However, increasing the size of the array one element at a time is time consuming too.
  - The copy operation involved during resizing the array is also time consuming

Insert Operation
(incl. Relocation and Doubling the Size of the Array)



Linked List

- A Linked List stores the elements of the 'List' in separate memory locations and we keep track of the memory locations as part of the information stored with an element (called a node).
  - A 'node' in a Linked List contains the data value as well as the address of the next node.
- Singly Linked List: Each node contains the address of the node with the subsequent value in the list. There is also a head node that points to the first node in the list.

  Data **With singly linked list – we can traverse only in one direction**

  nextNodePtr
- Doubly Linked List: Each node contains the address of the node with the subsequent value as well as the address of the node with the preceding value. There is also a head node pointing to the first node in the list and a tail node pointing to the last node in the list.

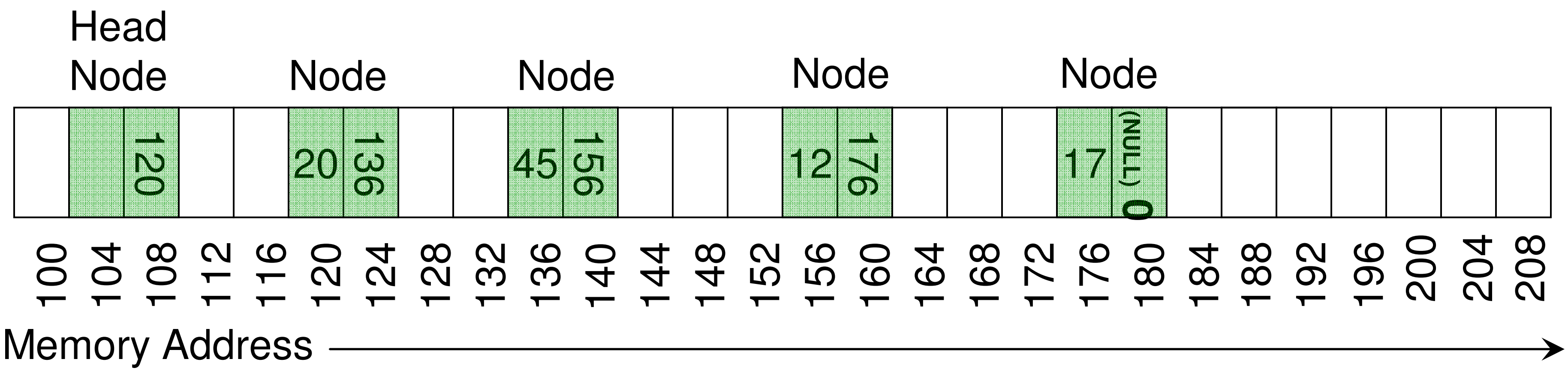
  prevNodePtr

  Data **With doubly linked list – we can traverse in both directions**

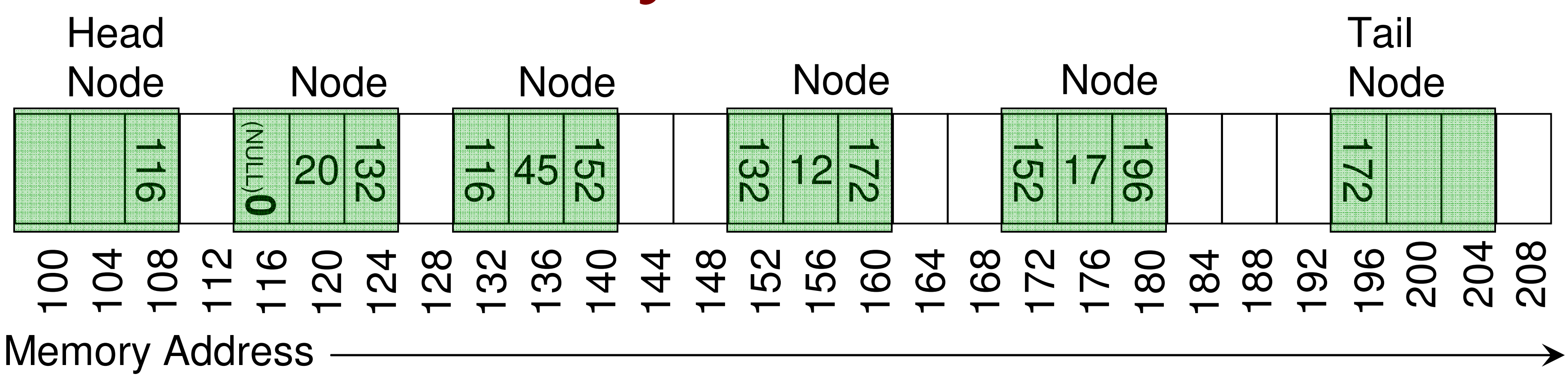
  nextNodePtr

- Note: Memory address can be represented in 4 bytes. Hence, each pointer or reference to a memory will take 4 bytes of space.

Singly Linked List



Doubly Linked List



Linked List vs. Arrays: Memory Usage

	Data size	Next Node Ptr	Prev Node Ptr	Node Size
Singly Linked List	4 (int)	4	N/A	8 bytes
Singly Linked List	32	4	N/A	36 bytes
Doubly Linked List	4 (int)	4	4	12 bytes
Doubly Linked List	32	4	4	40 bytes

An array is usually considered to take space that is twice the number of elements in it. Still, it looks like the Linked Lists will take a larger memory compared to an array. But, it is not always the case.

Consider a scenario wherein 64,000 objects (each of size 32 bytes) are to be stored in a List.

If we were to stored the objects in an array, there would need to be space for 128,000 objects. Hence, a dynamic array-based implementation will now hold up 128,000 * 32 bytes = 40,96,000 bytes in memory.

A singly linked list based implementation will hold (64,000 + 1 head node) * 36 bytes = 23,04,036 bytes in memory.

A doubly linked list based implementation will hold (64,000 + 1 head node + 1 tail node)* 40 bytes = 25,60,080 bytes in memory.

Linked List vs. Arrays: Memory Usage

On the other hand, Consider a scenario wherein 8,000 integers (each integer is 4 bytes) are to be stored in a List.

An array-based implementation will now hold 8,000 * 4 = 32,000 bytes in memory.

A singly linked list-based implementation will now hold (4,000 + 1 head node) * 8 = 32,008 bytes in memory.

A doubly linked list-based implementation will now hold (4,000 + 1 head node + 1 tail node) * 12 = 48,024 bytes in memory.

Linked List vs. Arrays: Time Complexity

	Array	Singly Linked List	Doubly Linked List
Read/Modify	**Θ(1)**	**O(n)**	**O(n)**
Insert	**O(n)**	**O(n)**	**O(n)**
Delete	**O(n)**	**O(n)**	**O(n)**
isEmpty	**Θ(1)**	**Θ(1)**	**Θ(1)**
Count	**O(n)**	**O(n)**	**O(n)**

We typically use arrays if there are more frequent read/modify operations compared to Insert/Delete

We typically use Linked Lists if there are more frequent insert/delete operations compared to read/modify

Note: With arrays, Insert operations are more time consuming if need to be done at the smaller indices. With singly linked lists, insert operations are more time consuming if done towards the end of the list. A doubly linked list could be traversed either from the head or the tail, and hence if a priori information is know about the sequence of elements in the list, traversal could be initiated from the head or tail, and the traversal time could be lower than a singly linked list. Still O(n) time though!

Preference: Dynamic Array Vs. Linked List

	Read/Modify Operations	Insert/Delete Operations
Several Data Objects, each of larger size	Linked List: Space Dynamic Array: Time	Linked List: Time & Space
Few Data Objects, each of smaller size	Dynamic Array: Time & Space	Linked List: Time Dynamic Array: Space

For insertion and deletion operations, the element-wise copy operation involved with Dynamic arrays during insertion and deletion is relatively more time consuming than traversing through the Linked List to reach the index for insertion/deletion. Hence, with respect to time complexity, Linked List is preferred for insertion and deletion, irrespective of the size of the data Objects.

For read and modify operations, dynamic arrays work in constant time and are always preferred with respect to time complexity.

For space complexity: linked lists are preferred for storing several data objects, each of larger size; dynamic arrays are preferred for storing fewer data objects, each of smaller size.

Singly Linked List Implementation (Code 3) Class Node

C++

```
private:
        int data;
        Node* nextNodePtr;
```

```
public:
    Node(){}

    void setData(int d){
            data = d;
    }

    int getData(){
            return data;
    }
```

Java

```
private int data;
private Node nextNodePtr;
```

```
public Node(){}

public void setData(int d){
        data = d;
}

public int getData(){
    return data;
}
```

Singly Linked List Implementation
Class Node

C++

```
public:
  void setNextNodePtr(Node* nodePtr){
                  nextNodePtr = nodePtr;
  }

  Node* getNextNodePtr(){
          return nextNodePtr;
  }
```

Java

```
public void setNextNodePtr(Node nodePtr){
                  nextNodePtr = nodePtr;
}

public Node getNextNodePtr(){
          return nextNodePtr;
}
```

Singly Linked List: Class List

Class Node (C++) Overview

```
private:
     int data;
    Node* nextNodePtr;

public:
     Node( )
     void setData(int)
     int getData()
     void setNextNodePtr(Node* )
     Node* getNextNodePtr( )
```

Class Node (Java) Overview

```
private int data;
private  Node nextNodePtr;

public Node( )
public  void setData(int)
public  int getData()
public  void setNextNodePtr(Node* )
public  Node getNextNodePtr( )
```

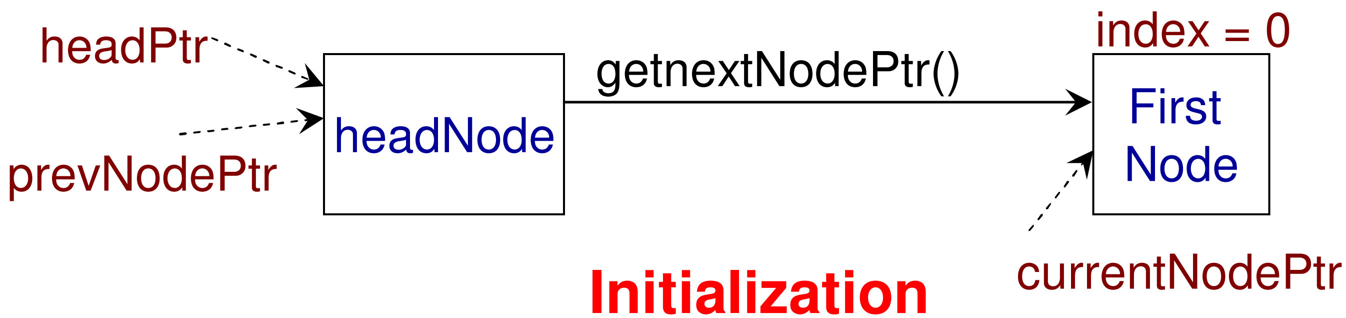
Class List (C++)

```
private:
    Node *headPtr;

public:
    List( ){
            headPtr = new Node();
            headPtr->setNextNodePtr(0);
    }
```

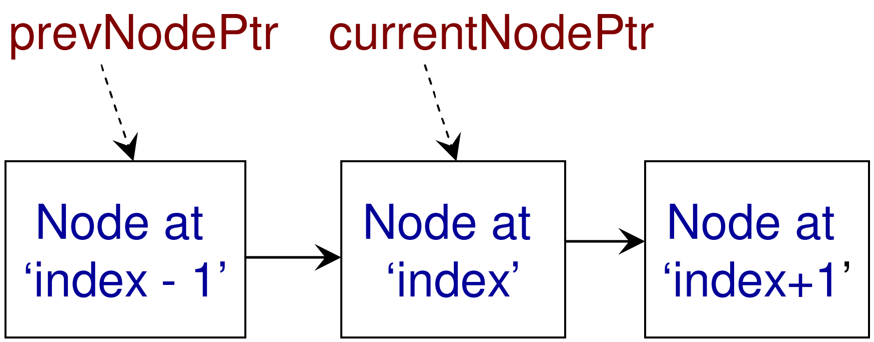
Class List (Java)

```
private Node headPtr;

public List( ){
        headPtr = new Node();
        headPtr.setNextNodePtr(null);
}
```

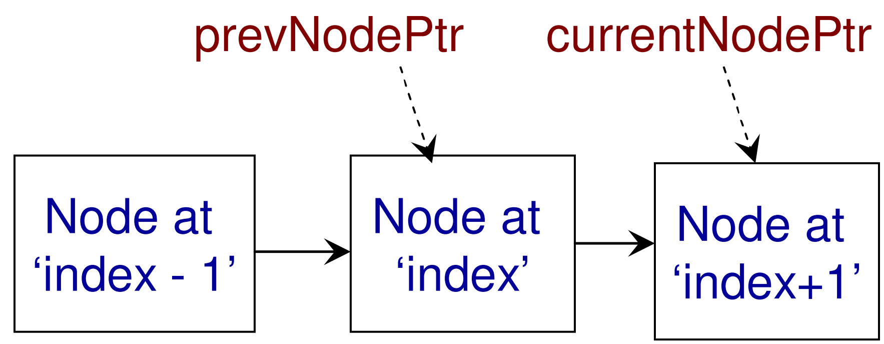
Insert and InsertAtIndex Functions



Initialization

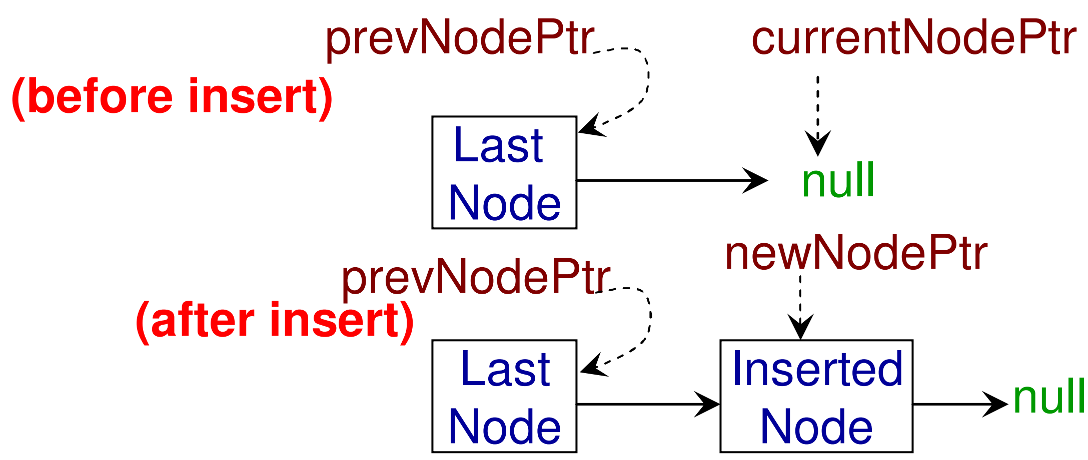


At the beginning of an iteration inside the ‘while’ loop

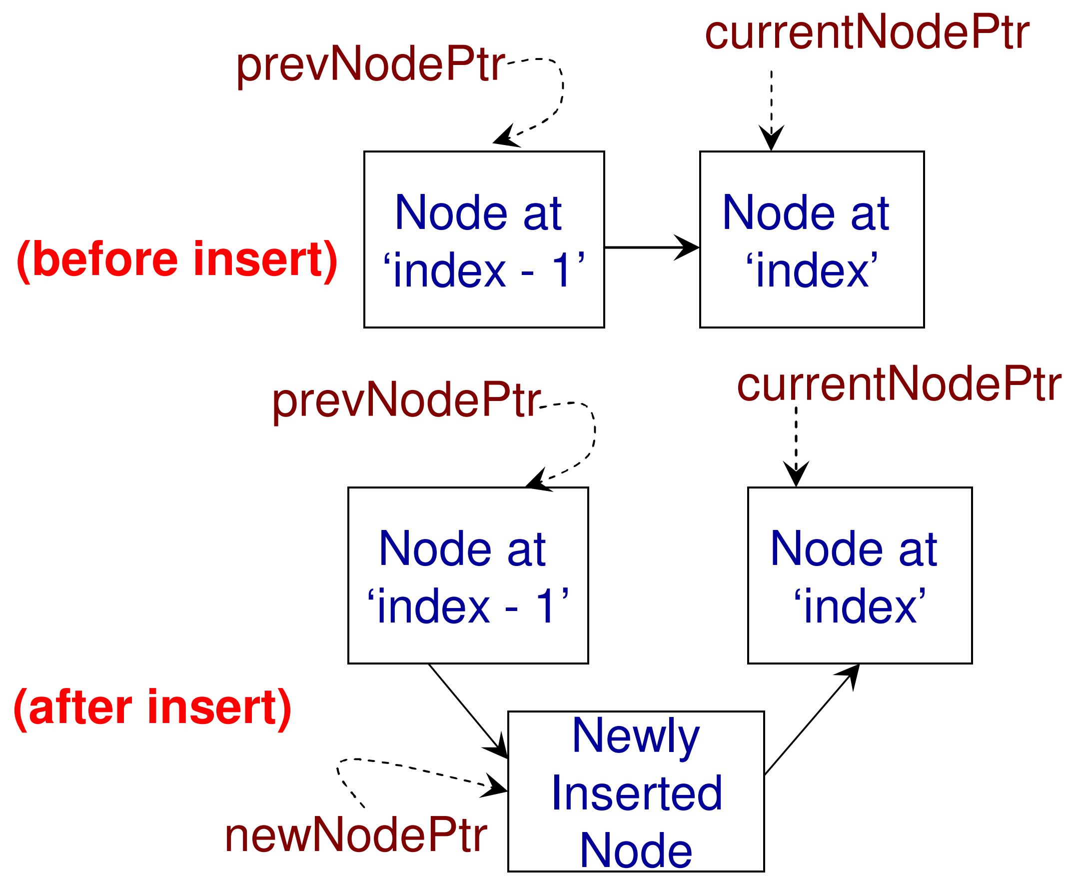


At the end of an iteration inside the ‘while’ loop

Insert Function (at the end of the List)



InsertAtIndex Function



Class List (C++)

```
void insert(int data){
    Node* currentNodePtr = headPtr->getNextNodePtr();
    Node* prevNodePtr = headPtr;

    while (currentNodePtr != 0){
        prevNodePtr = currentNodePtr;
        currentNodePtr = currentNodePtr->getNextNodePtr();
    }

    Node* newNodePtr = new Node();
    newNodePtr->setData(data);
    newNodePtr->setNextNodePtr(0);
    prevNodePtr->setNextNodePtr(newNodePtr);
}
```

Move the currentNode ptr from first node in the list to end of the list. When we come out of the 'while' loop, the prevNode ptr is the last node in the list and currentNode ptr points to null (0).

If the nextNodePtr for the headPtr points to null (0), then the list is empty. Otherwise, the list has at least one node.

```
bool isEmpty(){
    if (headPtr->getNextNodePtr() == 0)
        return true;

    return false;
}
```

Class List (Java)

```
public void insert(int data){
        Node currentNodePtr = headPtr.getNextNodePtr();
        Node prevNodePtr = headPtr;

        while (currentNodePtr != null){
                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr.getNextNodePtr();
        }

        Node newNodePtr = new Node();
        newNodePtr.setData(data);
        newNodePtr.setNextNodePtr(null);
        prevNodePtr.setNextNodePtr(newNodePtr);
}
```

Move the currentNode ptr from first node in the list to end of the list. When we come out of the ‘while’ loop, the prevNode ptr is last node in the list and currentNode ptr points to null (0).

If the nextNodePtr for the headPtr points to null (0), then the list is empty. Otherwise, the list has at least one node.

```
public boolean isEmpty(){
    if (headPtr.getNextNodePtr() == null)
            return true;

    return false;
}
```

Class List (C++)

```
void insertAtIndex(int insertIndex, int data){

    Node* currentNodePtr = headPtr->getNextNodePtr();
    Node* prevNodePtr = headPtr;

    int index = 0;

    while (currentNodePtr != 0){

        if (index == insertIndex)
            break;

        prevNodePtr = currentNodePtr;
        currentNodePtr = currentNodePtr->getNextNodePtr();
        index++;
    }

    Node* newNodePtr = new Node();
    newNodePtr->setData(data);
    newNodePtr->setNextNodePtr(currentNodePtr);
    prevNodePtr->setNextNodePtr(newNodePtr);
}
```

During the beginning and end of the while loop, the value for 'index' corresponds to the Position of the currentNode ptr and prevNode ptr corresponds to index-1.

If index equals insertIndex, we break from the while loop and insert the new node at the index in between prevNode and currentNode.

Class List (Java)

```
public void insertAtIndex(int insertIndex, int data){

        Node currentNodePtr = headPtr.getNextNodePtr();
        Node prevNodePtr = headPtr;

        int index = 0;

        while (currentNodePtr != null){

                if (index == insertIndex)
                        break;

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr.getNextNodePtr();
                index++;
        }

        Node newNodePtr = new Node();
        newNodePtr.setData(data);
        newNodePtr.setNextNodePtr(currentNodePtr);
        prevNodePtr.setNextNodePtr(newNodePtr);
}
```

During the beginning and end of the while loop, the value for 'index' corresponds to the Position of the currentNode ptr and prevNode ptr corresponds to index-1.

If index equals insertIndex, we break from the while loop and insert the new node at the index in between prevNode and currentNode.

Class List (C++)

```
int read(int readIndex){
    Node* currentNodePtr = headPtr->getNextNodePtr();
    Node* prevNodePtr = headPtr;
    int index = 0;

    while (currentNodePtr != 0){

        if (index == readIndex)
            return currentNodePtr->getData();

        prevNodePtr = currentNodePtr;
        currentNodePtr = currentNodePtr->getNextNodePtr();

        index++;

    }
    return -1; // an invalid value indicating index is out of range
}
```

The ‘index’ value corresponds to the Position of the currentNode ptr and index-1 corresponds to prevNode ptr

Class List (Java)

```
public int read(int readIndex){
    Node currentNodePtr = headPtr.getNextNodePtr();
    Node prevNodePtr = headPtr;
    int index = 0;

    while (currentNodePtr != null){

        if (index == readIndex)
            return currentNodePtr.getData();

        prevNodePtr = currentNodePtr;
        currentNodePtr = currentNodePtr.getNextNodePtr();

        index++;

    }

    return -1;
}
```

The ‘index’ value corresponds to the Position of the currentNode ptr and index-1 corresponds to prevNode ptr

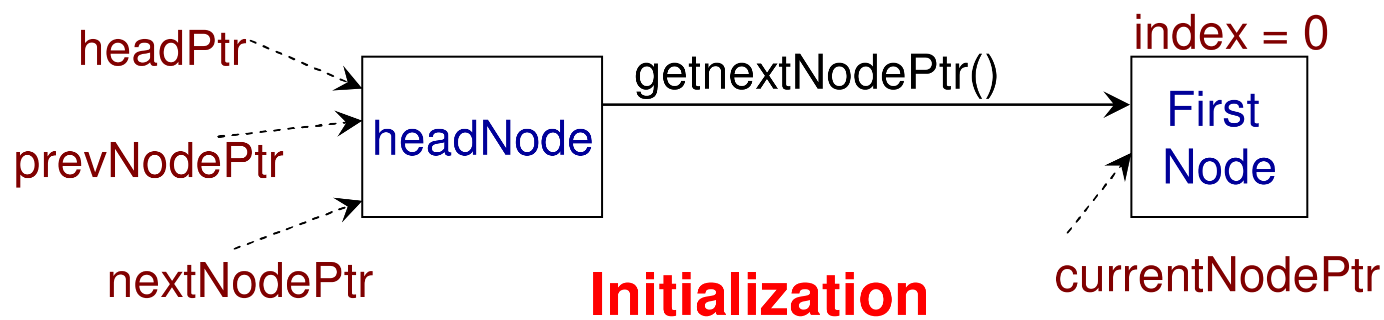
Class List (C++)

```
void modifyElement(int modifyIndex, int data){
        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;
        int index = 0;

        while (currentNodePtr != 0){

                if (index == modifyIndex){
                        currentNodePtr->setData(data);
                        return;
                }

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr->getNextNodePtr();

                index++;
        }
}
```

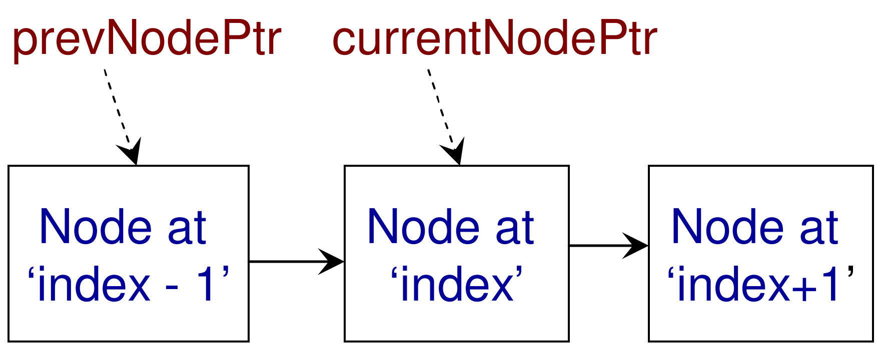
Class List (Java)

```
public void modifyElement(int modifyIndex, int data){
        Node currentNodePtr = headPtr.getNextNodePtr();
        Node prevNodePtr = headPtr;
        int index = 0;

        while (currentNodePtr != null){

                if (index == modifyIndex){
                        currentNodePtr.setData(data);
                        return;
                }

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr.getNextNodePtr();

                index++;
        }
}
```

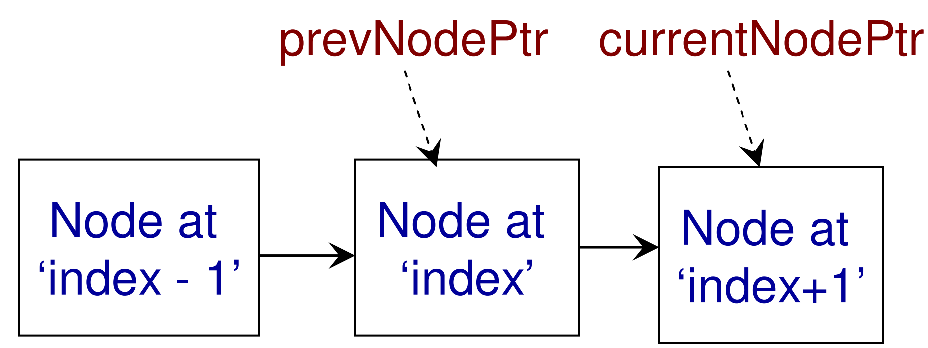
Delete (deleteIndex) Function



Initialization



At the beginning of an iteration inside the 'while' loop

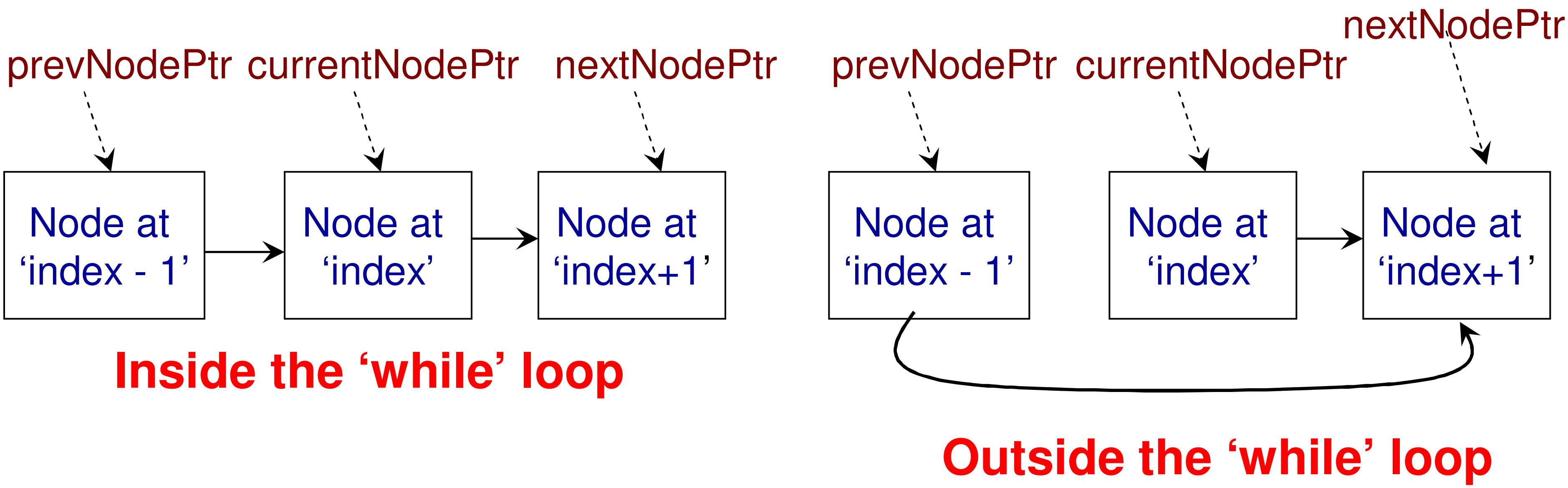


At the end of an iteration inside the 'while' loop

When index != deleteIndex

Delete (deleteIndex) Function

When index == deleteIndex



currentNode at index = deleteIndex is disconnected from the Linked List

Class List (C++)

```
void deleteElement(int deleteIndex){
        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;
        Node* nextNodePtr = headPtr;
        int index = 0;

        while (currentNodePtr != 0){

                if (index == deleteIndex){
                        nextNodePtr = currentNodePtr->getNextNodePtr();
                        break;
                }

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr->getNextNodePtr();


                index++;
        }

        prevNodePtr->setNextNodePtr(nextNodePtr);
}
```

The next node for ‘prevNode’ ptr is now ‘next node’ and not ‘current node’

Class List (Java)

```
public void deleteElement(int deleteIndex){
    Node currentNodePtr = headPtr.getNextNodePtr();
    Node prevNodePtr = headPtr;
    Node nextNodePtr = headPtr;
    int index = 0;

    while (currentNodePtr != null){

        if (index == deleteIndex){
            nextNodePtr = currentNodePtr.getNextNodePtr();
            break;
        }

        prevNodePtr = currentNodePtr;
        currentNodePtr = currentNodePtr.getNextNodePtr();


        index++;
    }
    prevNodePtr.setNextNodePtr(nextNodePtr);
}
```

The next node for ‘prevNode’ ref is now ‘next node’ and not ‘current node’

Iterative Print

Class List (C++)

```
void IterativePrint(){
    Node* currentNodePtr = headPtr->getNextNodePtr();

    while (currentNodePtr != 0){
        cout << currentNodePtr->getData() << " ";
        currentNodePtr = currentNodePtr->getNextNodePtr();
    }
    cout << endl;
}
```

Class List (Java)

```
public void IterativePrint(){
    Node currentNodePtr = headPtr.getNextNodePtr();

    while (currentNodePtr != null){
        System.out.print(currentNodePtr.getData()+" ");
        currentNodePtr = currentNodePtr.getNextNodePtr();
    }
    System.out.println();
}
```

Recursion

- Recursion: A function calling itself.
- Recursions are represented using a recurrence relation (incl. a base or terminating condition)

- Example 1
- Factorial (n) = n * Factorial(n-1) for n > 0
- Factorial (n) = 1 for n = 0

```
Factorial(n)
  if (n == 0)
     return 1;
  else
     return n * Factorial(n-1)
```

Factorial(0) = 1
Factorial(1) = 1 * Factorial(0)
Factorial(2) = 2 * Factorial(1)
Factorial(3) = 3 * Factorial(2)
Factorial(4) = 4 * Factorial(3)
Factorial(5) = 5 * Factorial(4)

Example (Code 4) to Illustrate Recursion and Random Number Generation

C++

```
int main(){

        int arraySize;
        cout << "Enter an array size: ";
        cin >> arraySize;

        int maxValue;
        cout << "Enter the max. value of an element: ";
        cin >> maxValue;

        srand(time(NULL));

        int array[maxValue];

        for (int i = 0; i < arraySize; i++){
                array[i] = rand() % maxValue;
        }

        cout << "IterativePrint: ";
        IterativePrint(array, arraySize);

        cout << "RecursivePrint: ";
        RecursivePrint(array, arraySize, 0);

return 0;
}
```

Initialize the random number generator with a seed that corresponds to the current system time

The random numbers are generated from 0 to maxValue – 1

Headers to be included

```
#include <iostream>
#include <stdlib.h> // random number
#include <time.h> // for time
using namespace std;
```

Code 4: C++

```
void IterativePrint(int* arrayPtr, int size){
    for (int index = 0; index < size; index++)
        cout << arrayPtr[index] << " ";

    cout << endl;
}
```

```
void RecursivePrint(int* arrayPtr, int size, int printIndex){
    if (printIndex == size){
        cout << endl;
        return;
    }

    cout << arrayPtr[printIndex] << " ";

    RecursivePrint(arrayPtr, size, printIndex+1);

    cout << arrayPtr[printIndex] << " ";
}
```

Printing in the forward order

Printing in the reverse order

Recursion

Seq	
	if (printIndex == arraySize){ // 4 == 4
10	cout << endl;
11	return;
	}
8	cout << arrayPtr[3] << “ “;
9	RecursivePrint(arrayPtr, arraySize = 4, printIndex = 4)
12	cout << arrayPtr[3] << “ “;
6	cout << arrayPtr[2] << “ “;
7	RecursivePrint(arrayPtr, arraySize = 4, printIndex = 3)
13	cout << arrayPtr[2] << “ “;
4	cout << arrayPtr[1] << “ “;
5	RecursivePrint(arrayPtr, arraySize = 4, printIndex = 2)
14	cout << arrayPtr[1] << “ “;
2	cout << arrayPtr[0] << “ “;
3	RecursivePrint(arrayPtr, arraySize = 4, printIndex = 1)
15	cout << arrayPtr[0] << “ “;
	@main
1	RecursivePrint(array, arraySize = 4, printIndex = 0)

	0	1	2	3
array	14	21	33	45

Seq	2	4	6	8
	14	21	33	45
	45	33	21	14
Seq	12	14	16	18

Example (Code 4) to Illustrate Recursion and Random Number Generation

Java

```
import java.util.*;

class arrayRecursive{

    public static void IterativePrint(int arrayRef[], int size){

        for (int index = 0; index < size; index++)
            System.out.print(arrayRef[index] + " ");

        System.out.println();

    }

    public static void RecursivePrint(int arrayPtr[], int size, int printIndex){
        if (printIndex == size){
            System.out.println();
            return;
        }
        System.out.print(arrayPtr[printIndex] + " ");

        RecursivePrint(arrayPtr, size, printIndex+1);

        System.out.print(arrayPtr[printIndex] + " ");
    }
```

Java

```
public static void main(String[] args){
    Scanner input = new Scanner(System.in);

    int arraySize;
    System.out.print("Enter an array size: ");
    arraySize = input.nextInt();

    int maxValue;
    System.out.print("Enter the max. value of an element: ");
    maxValue = input.nextInt();

    Random randGen = new Random(System.currentTimeMillis());

    int array[] = new int[arraySize];

    for (int i = 0; i < arraySize; i++){
            array[i] = randGen.nextInt(maxValue);
    }

    System.out.print("IterativePrint: ");
    IterativePrint(array, arraySize);

    System.out.print("RecursivePrint: ");
    RecursivePrint(array, arraySize, 0);
}
```

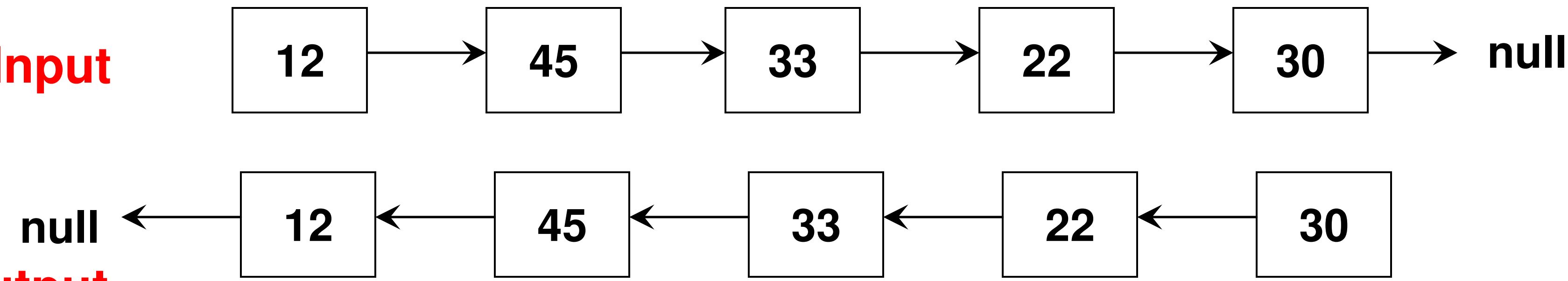
Initialize the random number generator with a seed that corresponds to the current system time

The random numbers are generated from 0 to maxValue – 1

Reversing a Linked List

Input

Output



Logic

Maintain three pointers

currentNodePtr, nextNodePtr, prevNodePtr

Enter the loop if currentNodePtr is not null

After entering the loop,

Step 1: set nextNodePtr = currentNodePtr->getNextNodePtr()

Now that there is a pointer to the next node of currentNode, reverse the direction for the next node of currentNode

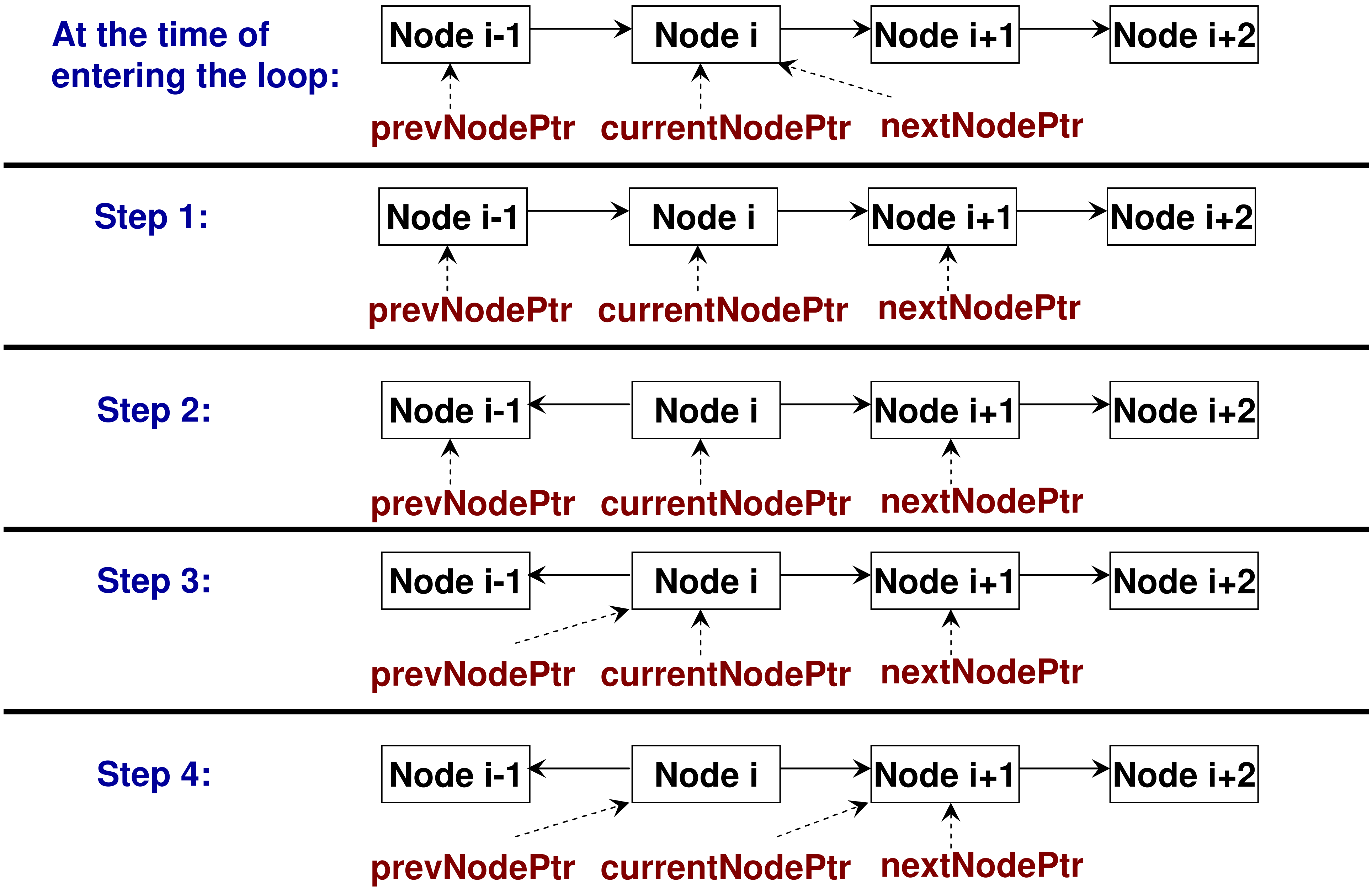
Step 2: currentNodePtr->setNextNodePtr(prevNodePtr)

Now prepare for the next iteration,

Step 3: set prevNodePtr = currentNodePtr

Step 4: set currentNodePtr = nextNodePtr

Reversing a Linked List (logic)



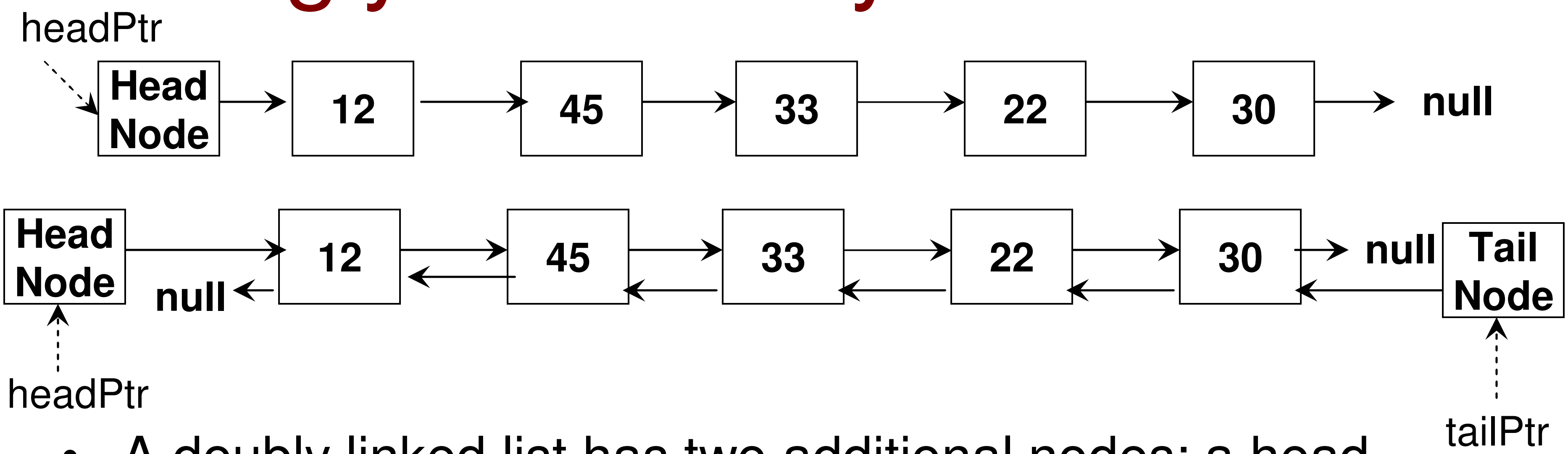
Reversing a Singly Linked List (Code 5): C++

```
void reverseList(){
    Node* currentNodePtr = headPtr->getNextNodePtr();
    Node* prevNodePtr = 0;
    Node* nextNodePtr = currentNodePtr;

    while (currentNodePtr != 0){
        nextNodePtr = currentNodePtr->getNextNodePtr(); // Step 1
        currentNodePtr->setNextNodePtr(prevNodePtr); // Step 2
        prevNodePtr = currentNodePtr;  // Step 3
        currentNodePtr = nextNodePtr;  // Step 4
    }
    headPtr->setNextNodePtr(prevNodePtr);
}
```

Reversing a Singly Linked List (Code 5): Java

```
public void reverseList(){
        Node currentNodePtr = headPtr.getNextNodePtr();
        Node prevNodePtr = null;
        Node nextNodePtr = currentNodePtr;

        while (currentNodePtr != null){
            nextNodePtr = currentNodePtr.getNextNodePtr(); // Step 1
            currentNodePtr.setNextNodePtr(prevNodePtr); // Step 2
            prevNodePtr = currentNodePtr;  // Step 3
            currentNodePtr = nextNodePtr;  // Step 4
        }

        headPtr.setNextNodePtr(prevNodePtr);
  }
```

Singly vs. Doubly Linked List



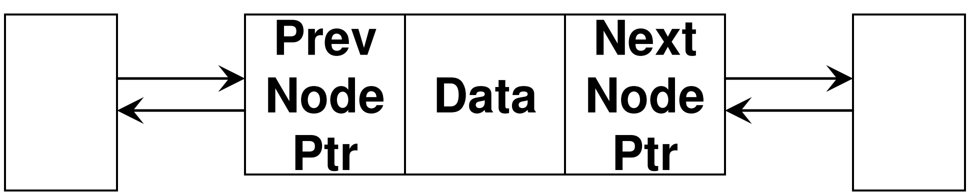
- A doubly linked list has two additional nodes: a head node and tail node (a head ptr points to the head node whose next node is the first node in the list, and a tail ptr points to the tail node whose prev node is the last node in the list).
  - Note the next node for the last node in the list is null (so that the end of the list could be traced) as well as the prev node for the first node in the list is null (so that the beginning of the list could be traced).
- A doubly linked list could be traversed in either direction (from head to tail or from tail to head).

Class Node (Code 6) C++

Singly Linked List

```
private:
        int data;
        Node* nextNodePtr;
```

```
public:
   Node(){}

   void setNextNodePtr(Node* nodePtr){
                    nextNodePtr = nodePtr;
   }

   Node* getNextNodePtr(){
           return nextNodePtr;
   }
```



Doubly Linked List

```
private:
        int data;
        Node* nextNodePtr;
        Node* prevNodePtr;
```

```
public:
        Node(){}

        void setNextNodePtr(Node* nodePtr){
                    nextNodePtr = nodePtr;
        }

        Node* getNextNodePtr(){
              return nextNodePtr;
        }

        void setPrevNodePtr(Node* nodePtr){
                    prevNodePtr = nodePtr;
        }

        Node* getPrevNodePtr(){
              return prevNodePtr;
        }
```

Singly Linked List

```
private:
        Node *headPtr;

public:
        List( ){
                headPtr = new Node();
                headPtr->setNextNodePtr(0);
        }

        Node* getHeadPtr(){
                return headPtr;
        }
```

Doubly Linked List

```
private:
        Node *headPtr;
        Node* tailPtr;

public:
        List(){
                headPtr = new Node();
                tailPtr = new Node();
                headPtr->setNextNodePtr(0);
                tailPtr->setPrevNodePtr(0);
        }

        Node* getHeadPtr(){
                return headPtr;
        }

        Node* getTailPtr(){
                return tailPtr;
        }
```

Code 6: C++

Code 6 (C++)
Insert to the end of a Doubly Linked List

```
void insert(int data){

        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;

        while (currentNodePtr != 0){
                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr->getNextNodePtr();
        }

        Node* newNodePtr = new Node();
        newNodePtr->setData(data);
1       newNodePtr->setNextNodePtr(0);

2       prevNodePtr->setNextNodePtr(newNodePtr);

3       if (prevNodePtr != headPtr)
                newNodePtr->setPrevNodePtr(prevNodePtr);
4       else
                newNodePtr->setPrevNodePtr(0);

5       tailPtr->setPrevNodePtr(newNodePtr);
}
```

The List has at least one element before the insertion and the prevNodePtr points to the last node in the list after which the newNode is inserted (i.e., the newly inserted node is 'NOT THE FIRST NODE' in the list

The List is empty before the insertion and the newNode is 'THE FIRST NODE' to be inserted

For both cases, set the newNode to be the previous node for the tailPtr

Code 6 (Java)
Insert to the end of a Doubly Linked List

```
public void insert(int data){
    Node currentNodePtr = headPtr.getNextNodePtr();
    Node prevNodePtr = headPtr;

    while (currentNodePtr != null){
        prevNodePtr = currentNodePtr;
        currentNodePtr = currentNodePtr.getNextNodePtr();
    }

    Node newNodePtr = new Node();
    newNodePtr.setData(data);
1   newNodePtr.setNextNodePtr(null);

2   prevNodePtr.setNextNodePtr(newNodePtr);

3   if (prevNodePtr != headPtr)
        newNodePtr.setPrevNodePtr(prevNodePtr);
4   else
        newNodePtr.setPrevNodePtr(null);

5   tailPtr.setPrevNodePtr(newNodePtr);
}
```

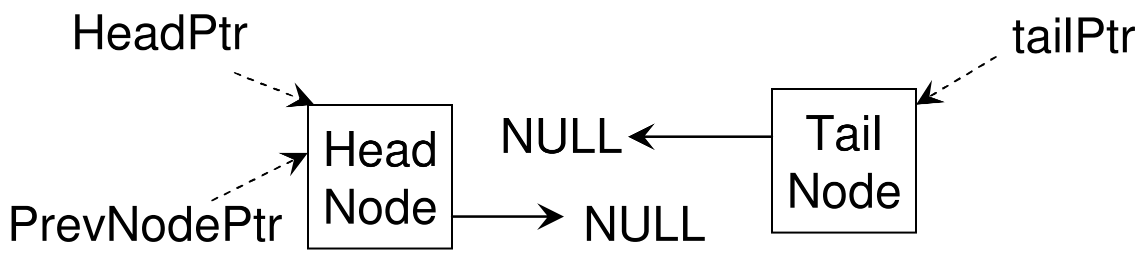
The List has at least one element before the insertion and the prevNodePtr points to the last node in the list after which the newNode is inserted (i.e., the newly inserted node is 'NOT THE FIRST NODE' in the list

The List is empty before the insertion and the newNode is 'THE FIRST NODE' to be inserted

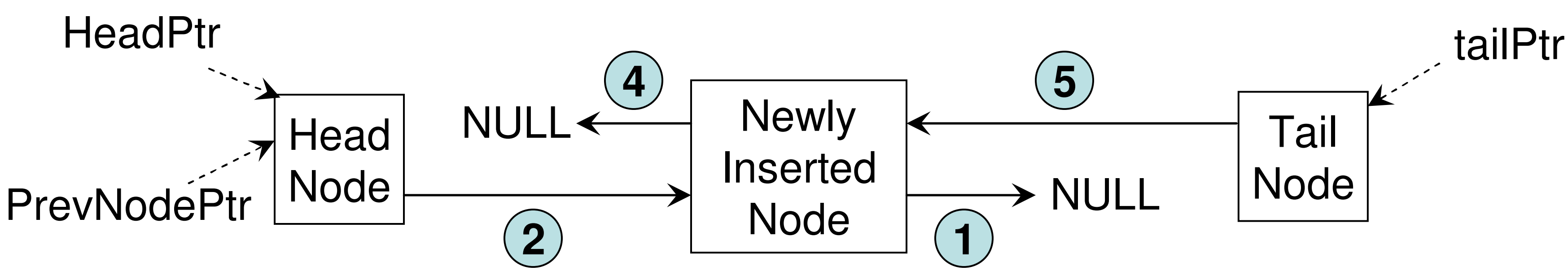
For both cases, set the newNode to be the previous node for the tailPtr

Insert at the End of the List

If the newly inserted node is “THE FIRST NODE” to be inserted in the list



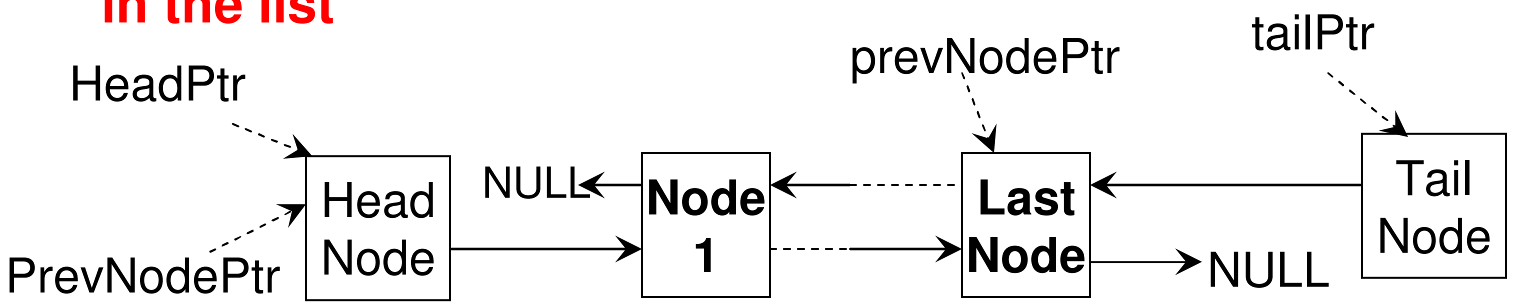
Before Insertion



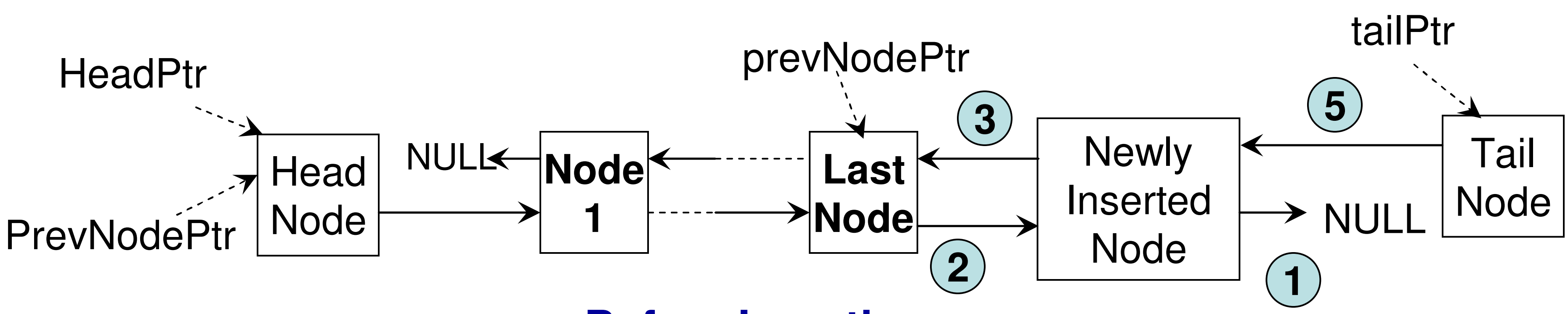
After Insertion

Insert at the End of the List

If the newly inserted node is "NOT THE FIRST NODE" to be inserted in the list



Before Insertion



Before Insertion

Code 6 (C++)
Insert at 'insertIndex' in a Doubly Linked List
(First Part of the Function Code)

```
void insertAtIndex(int insertIndex, int data){

        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;

        int index = 0;

        while (currentNodePtr != 0){

                if (index == insertIndex)
                        break;

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr->getNextNodePtr();
                index++;
        }

        Node* newNodePtr = new Node();
        newNodePtr->setData(data);
        newNodePtr->setNextNodePtr(currentNodePtr); 6
        prevNodePtr->setNextNodePtr(newNodePtr);
```

Note: The first part of the Function code is the same as that of a singly linked list

The second part of the Function code (shown below) is meant to the handle setting the prevNodePtr for the newly inserted node and the current node

The if segment takes care of the scenario wherein the newly inserted node is 'NOT THE FIRST NODE' in the list; the else Segment takes care of the scenario wherein the newly Inserted node is 'THE FIRST NODE' in the list

This part of the code takes care of setting the prevNodePtr for the newly inserted node

```
if (prevNodePtr != headPtr)
        newNodePtr->setPrevNodePtr(prevNodePtr);     3
else
        newNodePtr->setPrevNodePtr(0);               4
```

```
if (currentNodePtr != 0)
        currentNodePtr->setPrevNodePtr(newNodePtr);  7
```

```
if (currentNodePtr == 0)
        tailPtr->setPrevNodePtr(newNodePtr);         5
```

If the currentNodePtr points to a node (i.e., the newly Inserted node is 'NOT THE LAST NODE' in the list, set the newly Inserted node to be the previous node of this currentNode

If the currentNodePtr does not point to any node, it means the end of the list has been reached, and we need to set the newly Inserted node as the previous node for the tailPtr (i.e., the newly inserted node is 'THE LAST NODE' in the list

Code 6 (Java)
Insert at 'insertIndex' in a Doubly Linked List
(First Part of the Function Code)

```
public void insertAtIndex(int insertIndex, int data){

    Node currentNodePtr = headPtr.getNextNodePtr();
    Node prevNodePtr = headPtr;

    int index = 0;

    while (currentNodePtr != null){

        if (index == insertIndex)
            break;

        prevNodePtr = currentNodePtr;
        currentNodePtr = currentNodePtr.getNextNodePtr();
        index++;
    }

    Node newNodePtr = new Node();
    newNodePtr.setData(data);
    newNodePtr.setNextNodePtr(currentNodePtr);
    prevNodePtr.setNextNodePtr(newNodePtr);
```

Note: The first part of the Function code is the same as that of a singly linked list

6

The second part of the Function code (shown below) is meant to the handle setting the prevNodePtr for the newly inserted node and the current node

The if segment takes care of the scenario wherein the newly inserted node is 'NOT THE FIRST NODE' in the list; the else Segment takes care of the scenario wherein the newly Inserted node is 'THE FIRST NODE' in the list

This part of the code takes care of setting the prevNodePtr for the newly inserted node

```
if (prevNodePtr != headPtr)
        newNodePtr.setPrevNodePtr(prevNodePtr);
else
        newNodePtr.setPrevNodePtr(null);
```

3

4

```
if (currentNodePtr != null)
        currentNodePtr.setPrevNodePtr(newNodePtr);
```

7

```
if (currentNodePtr == null)
        tailPtr.setPrevNodePtr(newNodePtr);
```

5

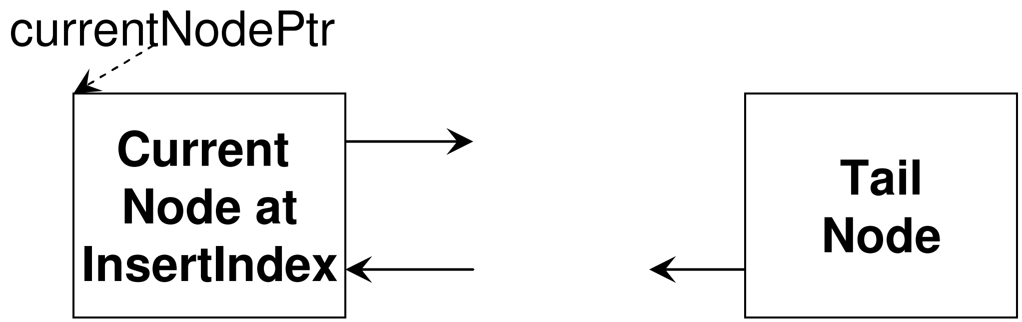
If the currentNodePtr points to a node (i.e., the newly Inserted node is 'NOT THE LAST NODE' in the list, set the newly Inserted node to be the previous node of this currentNode

If the currentNodePtr does not point to any node, it means the end of the list has been reached, and we need to set the newly Inserted node as the previous node for the tailPtr (i.e., the newly inserted node is 'THE LAST NODE' in the list

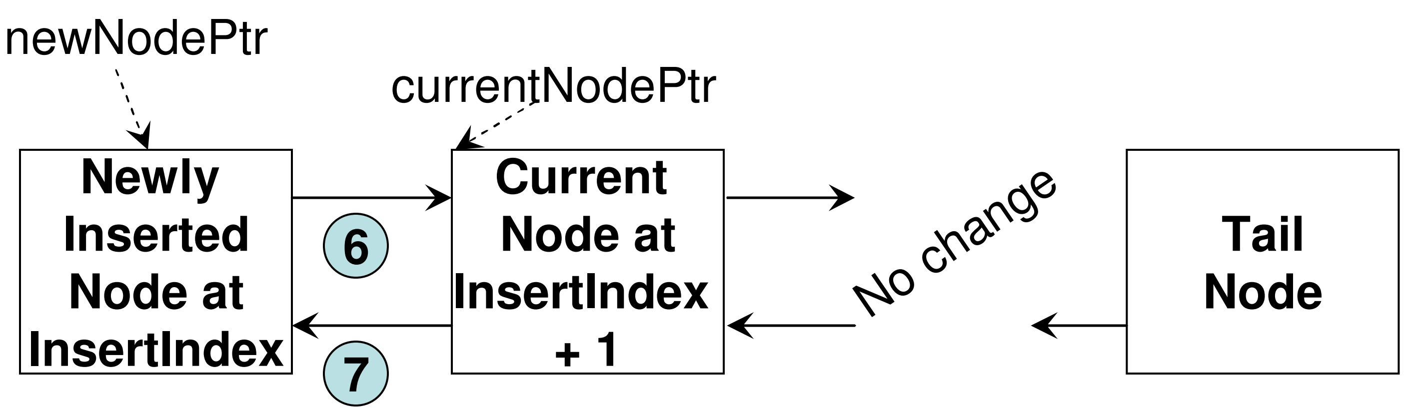
Insert at ‘InsertIndex’

If the newly inserted node is “NOT THE LAST NODE” to be inserted in the list

The prevNodePtr scenraios (3) And (4) are handled as in the Insert at End function



Before Insertion



After Insertion