Module 2:
List ADT

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

Data processed by an Algorithm

- The design and development as well as the time and storage complexities of an algorithm for a problem depend on how we store and process the data on which the algorithm is run.
- For example: if the words in a dictionary are not sorted, it would take a humongously long time to come up with an algorithm to search for a word in the dictionary.
- Sometimes, the data need not be linear (like a dictionary) and need to be hierarchical (like a road map or file system).
- Layman example
  - Abstract view of a car (any user should expect these features for any car): Should be able to start the car, turn steering, press brake to stop and press gas to accelerate, change gear, etc.
  - Implementation (responsibility of the manufacturer and not the user): How each of the above is implemented? Varies with the targeted gas efficiency, usage purpose, etc.

Abstract Data Type (ADT) vs. Data Structures

- Data processed by an algorithm could be represented at two levels:
  - Abstract level (also called logical or user level): merely state the possible values for the data and what operations/functions the algorithm will call to store and access the data
  - Implementation level (also called system level): deals with how the implementation should be done to perform the functions defined for the data at the abstract level.
- The abstract (logical) representation of data is commonly referred to as Abstract Data Type (ADT)
- The term "data structure" is considered to represent the implementation model of an ADT.

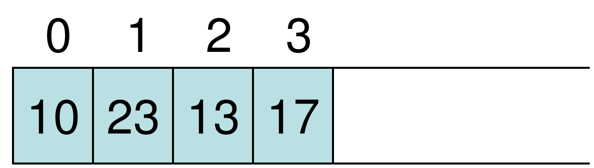
Common ADTs and the Data Structures for their Implementation

- List, Stack, Queue
  - Arrays, Linked List
- Priority Queue
  - Heap
- Dictionary
  - Hash Table, Binary Search Tree
- Graph
  - Adjacency List, Adjacency Matrix

List ADT

10
23
17
19

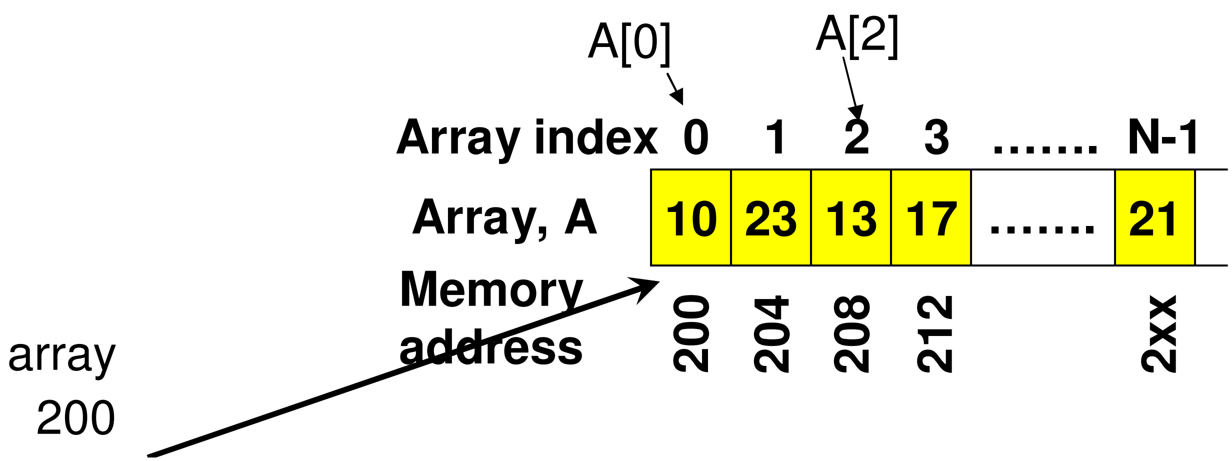
- Data type
  - Store a given number of elements of a particular data type



- Functions/Operations
  - Create an initial empty list
  - Test whether or not a list is empty
  - Read element based on its position in the list.
  - Insert, delete or modify an entity at a specific position in the list

Static 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;
}
```

Dynamic List ADT

- **Limitations with Static List**
  - The list size is fixed (during initialization); cannot be increased or decreased.
  - A new element cannot be inserted (if the list is already full) or an existing element cannot be deleted.
- **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 start with a list of size one 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 for each resize operation.
  - 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;
    }
```

isEmpty (C++)

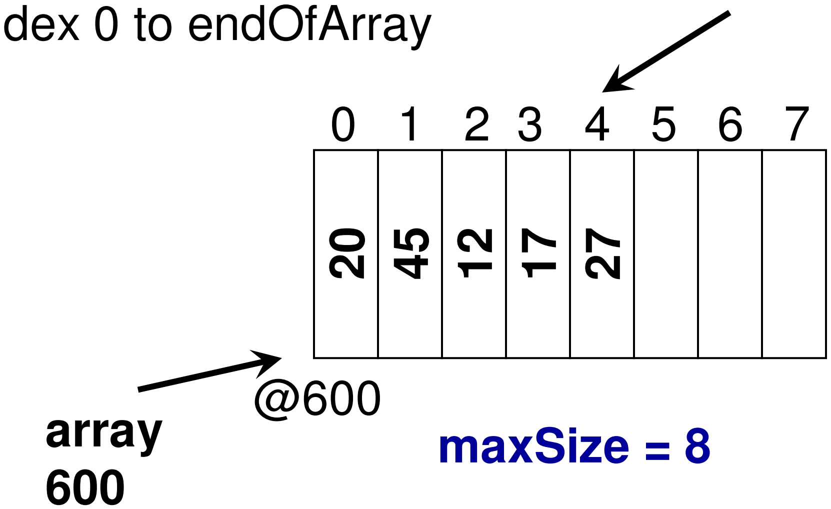
```
bool isEmpty(){

    if (endOfArray = = -1)
            return true;

    return false;
}
```

Function to free the memory (C++)

```
void deleteList(){
    delete[] array;
}
```

Note: The accessible portion of the List/array is from index 0 to endOfArray

endOfArray = 4



Code 2: Insert Function (C++)

```
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++;

}
```

Self-healing code logic

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

        int *tempArray = array;
```

Have another pointer (a temporary ptr) 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

```
        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];
        }
```

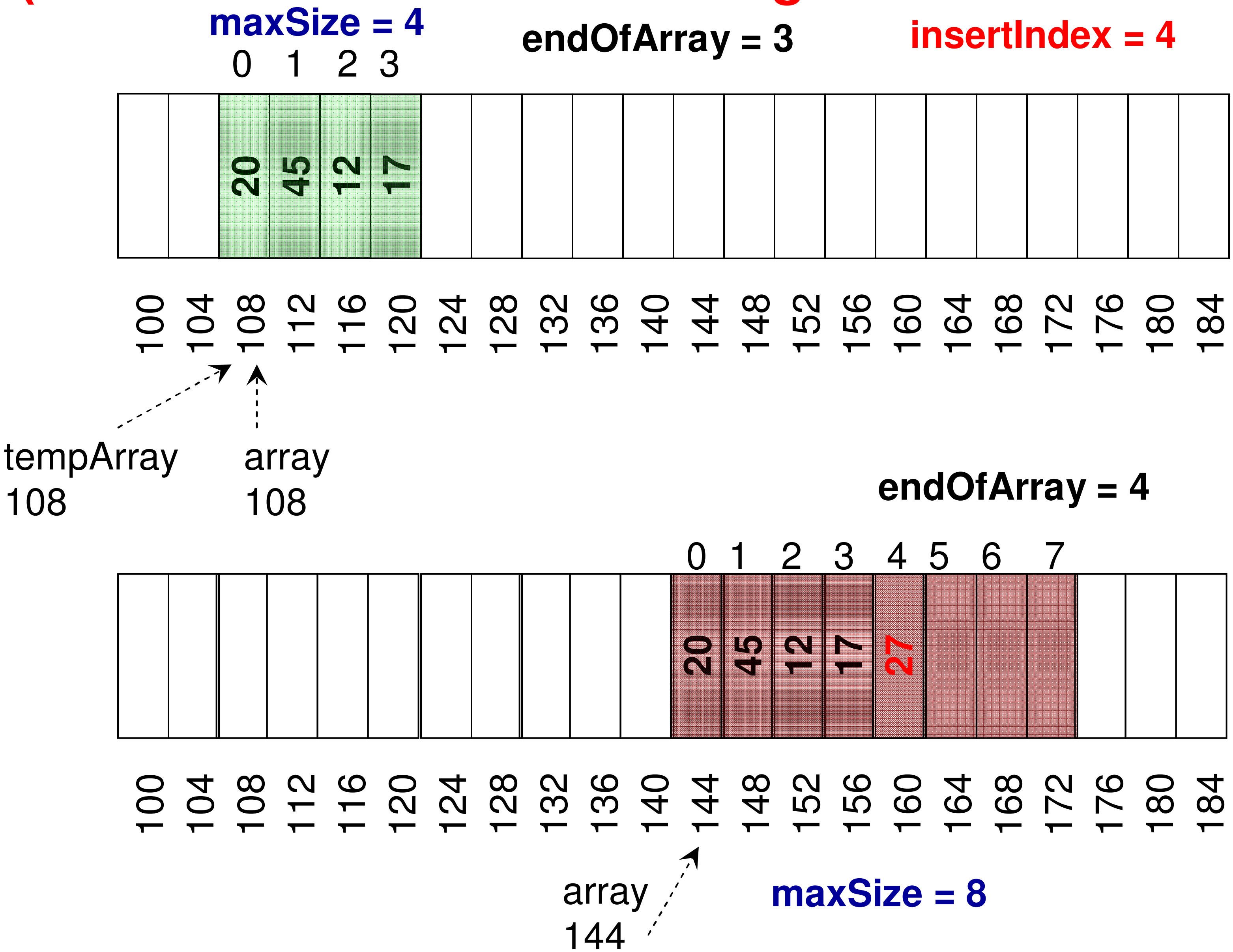
If the array size is reduced from maxSize to s, only the first 's' elements are copied. Otherwise, all the maxElements are copied

```
        maxSize = s;
}
```

new value of maxSize

Note: Include <algorithm> header file if the *min* function is not automatically loaded to your computing environment.

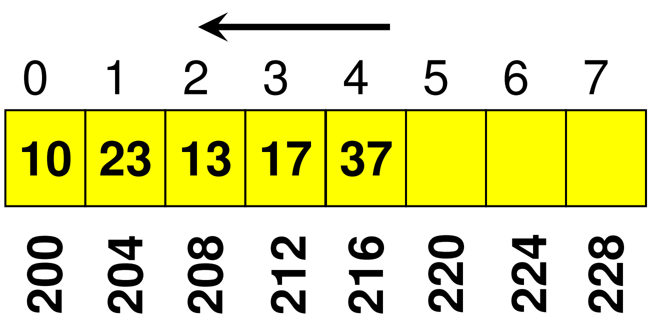
Insert Operation
(incl. Relocation and Doubling the Size of the Array)



Example for Insert Operation (Array-based Dynamic List ADT)

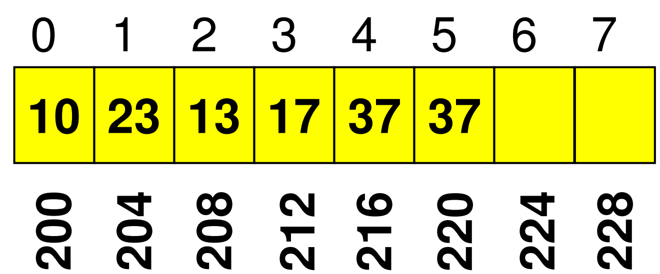
```
for (int index = endOfArray; index >= insertIndex; index--)
        array[index+1] = array[index];

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

Assume insertIndex = 2; data = 30



maxSize = 8; endOfArray = 4

Before entry to the loop



maxSize = 8; endOfArray = 4

index = 4

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

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

Assume insertIndex = 2; data = 30

0	1	2	3	4	5	6	7
10	23	13	17	37	37		
200	204	208	212	216	220	224	228

maxSize = 8; endOfArray = 4

@ end of the index = 4 iteration

0	1	2	3	4	5	6	7
10	23	13	17	17	37		
200	204	208	212	216	220	224	228

maxSize = 8; endOfArray = 4

index = 3

0	1	2	3	4	5	6	7
10	23	13	13	17	37		
200	204	208	212	216	220	224	228

maxSize = 8; endOfArray = 4

index = 2

0	1	2	3	4	5	6	7
10	23	30	13	17	37		
200	204	208	212	216	220	224	228

maxSize = 8; endOfArray = 4

After exiting from the loop; the data is written at insertIndex = 2

Time Complexity Analysis of Insert

Best Case: **If the insert is to be done at index corresponding to endOfArray + 1**

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

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

Assume insertIndex = 5; data = 30

Note: We will not be even entering the loop
as index = 4 is < insertIndex = 5

0	1	2	3	4	5	6	7
10	**23**	**13**	**17**	**37**			
200	**204**	**208**	**212**	**216**	**220**	**224**	**228**

maxSize = 8; endOfArray = 4

Before entry to the loop

0	1	2	3	4	5	6	7
10	**23**	**13**	**17**	**37**	**30**		
200	**204**	**208**	**212**	**216**	**220**	**224**	**228**

maxSize = 8; endOfArray = 4

After the loop

Time Complexity Analysis of Insert

Worst Case: **If the insert is to be done at index 0**

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

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

0	1	2	3	4	5	6	7
10	23	13	17	37			
200	204	208	212	216	220	224	228

maxSize = 8; endOfArray = 4

Before entry to the loop

index = 4

0	1	2	3	4	5	6	7
10	23	13	17	37	37		

index = 3

0	1	2	3	4	5	6	7
10	23	13	17	17	37		

index = 2

0	1	2	3	4	5	6	7
10	23	13	13	17	37		

Worst Case: **If the insert is to be done at index 0**

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

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

0	1	2	3	4	5	6	7
10	23	13	17	37			
200	204	208	212	216	220	224	228

maxSize = 8; endOfArray = 4

Before entry to the loop

index = 2

0	1	2	3	4	5	6	7
10	23	13	13	17	37		

index = 1

0	1	2	3	4	5	6	7
10	23	23	13	17	37		

index = 0

0	1	2	3	4	5	6	7
10	10	23	13	17	37		

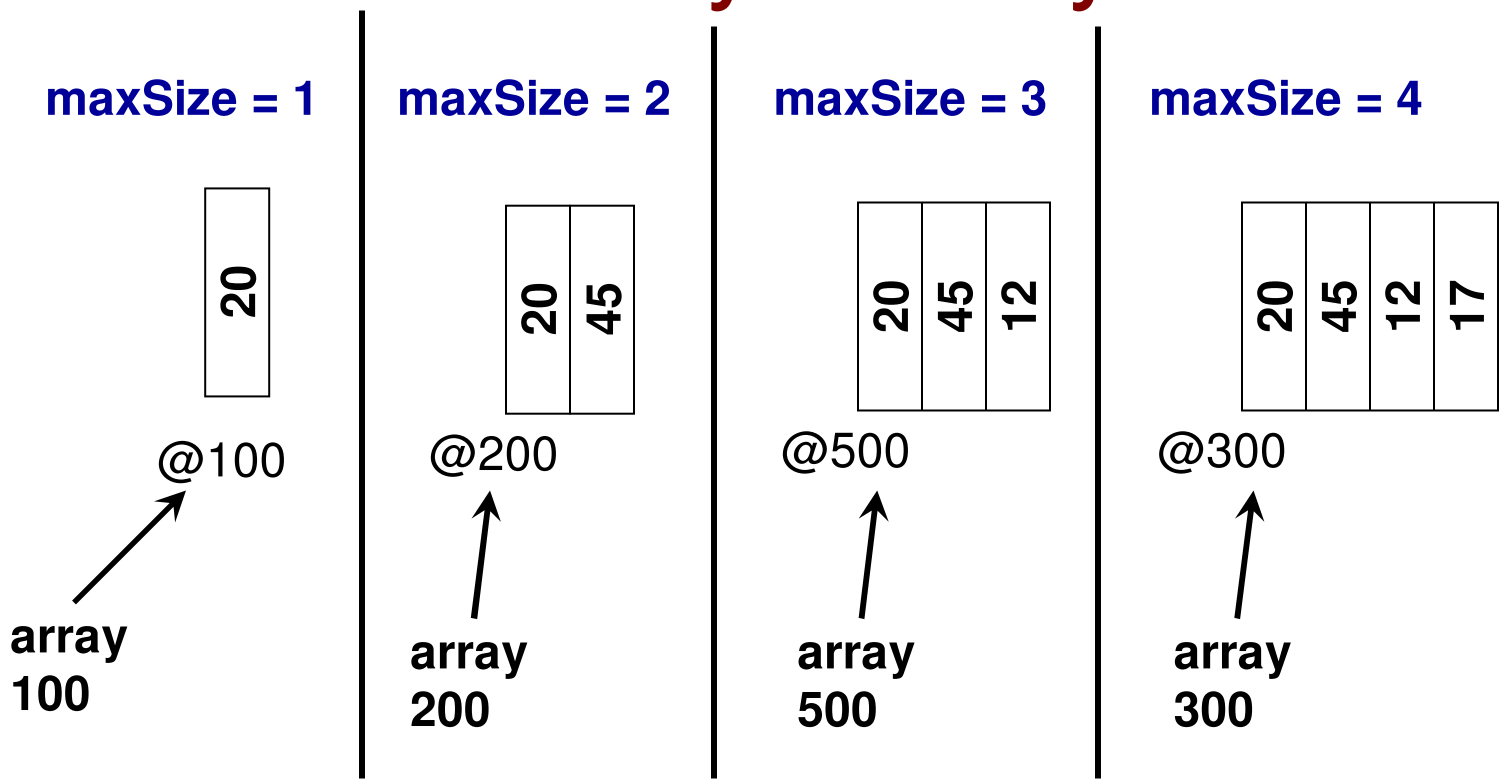
0	1	2	3	4	5	6	7
30	10	23	13	17	37		
200	204	208	212	216	220	224	228

maxSize = 8; endOfArray = 4

After exiting the loop

The number of copy operations is endOfArray + 1, which could be at most maxSize-1. Hence, the Time complexity for the insert operation is O(n), where ‘n’ is the size of the List

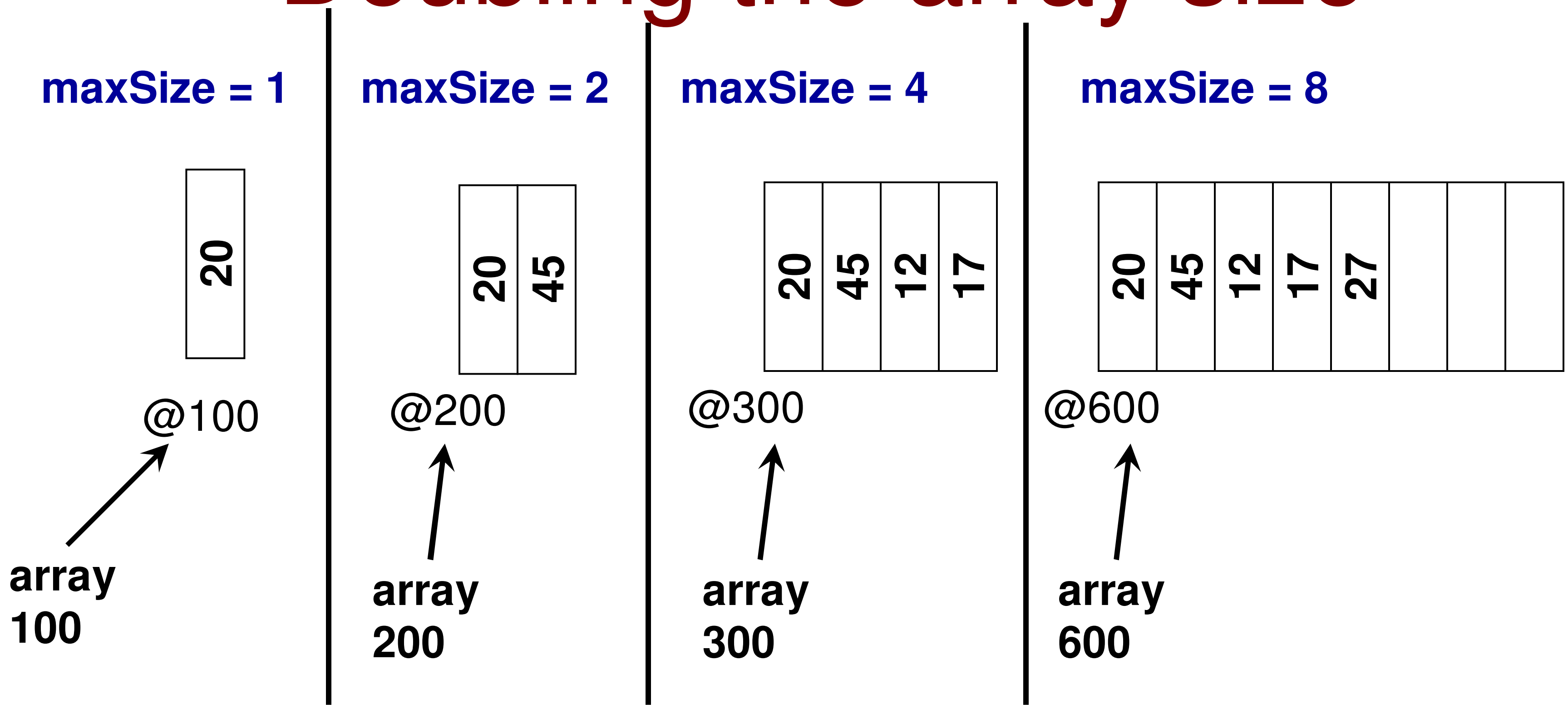
Problem with Incrementing the array size by one



memory cells allocated to eventually store a list of 4 elements is
1 + 2 + 3 + 4 = 10;
For 8 elements: 1 + 2 + 3 + …. + 8 = 8*(8+1)/2 = 36

In general, $1 + 2 + 3 + 4 + \ldots + n = n(n+1)/2 = \Theta(n^2)$

Doubling the array size



 # memory cells allocated to eventually store a list of 8 elements is:
$1 + 2 + 4 + 8 = 15$
In general, the number of cells to store at most 'n' elements, where 'n' is a perfect square of 2; i.e., $n = 2^k$, where k is an integer >= 0; $k = \log_2(n)$
$1 + 2 + 4 + 8 + \ldots + n = 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^k$, where $k = \log_2(n)$
$= 2^{(k+1)} - 1 = 2*2^k - 1 = 2n - 1 = \Theta(n)$

Code 2: Other Auxiliary Functions

(C++)

```
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;
}
```

Example for Delete Operation (Array-based Dynamic List ADT)

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

endOfArray--;
```

Assume deleteIndex = 2

0	1	2	3	4	5	6	7
10	**23**	**13**	**17**	**37**			
200	**204**	**208**	**212**	**216**	**220**	**224**	**228**

maxSize = 8; endOfArray = 4

Before entry to the loop

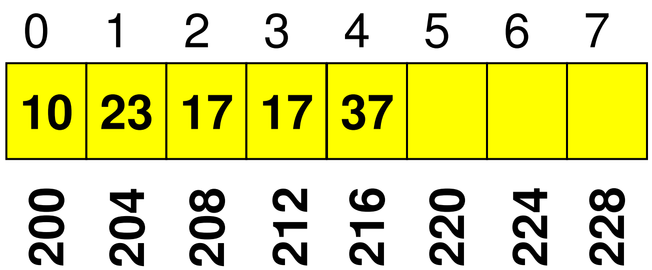
0	1	2	3	4	5	6	7
10	**23**	**17**	**17**	**37**			
200	**204**	**208**	**212**	**216**	**220**	**224**	**228**

maxSize = 8; endOfArray = 4

index = 2

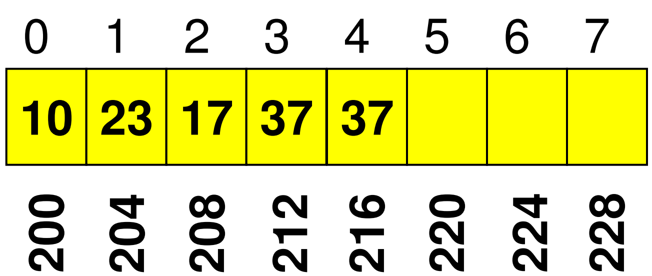
```
for (int index = deleteIndex; index < endOfArray; index++)
        array[index] = array[index+1];

endOfArray--;
```

Assume deleteIndex = 2



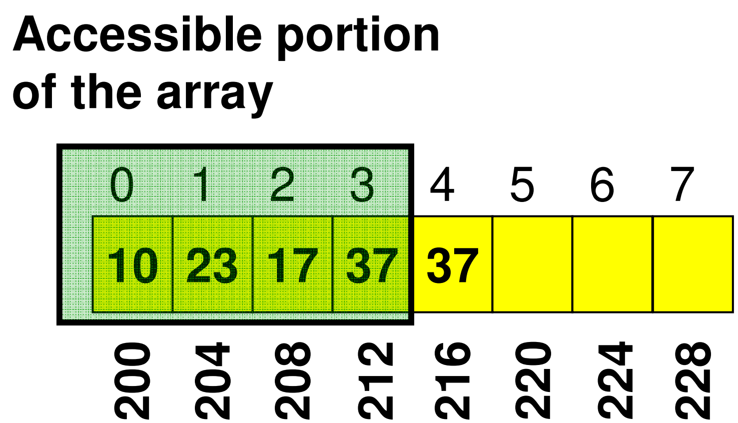
maxSize = 8; endOfArray = 4

@ end of index = 2 iteration



maxSize = 8; endOfArray = 4

index = 3



maxSize = 8; endOfArray = 3

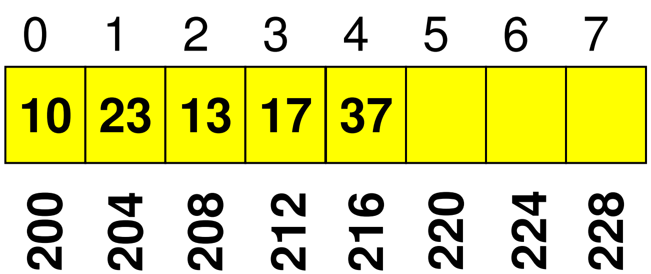
After exiting the loop

Time complexity of Delete

Best Case: **If the delete is to be done at index corresponding to endOfArray**

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

endOfArray--;
```

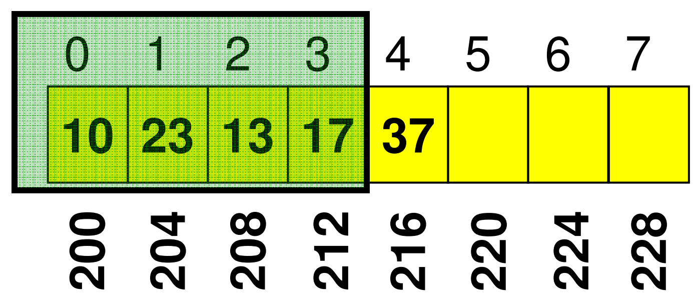
Assume deleteIndex = 4



maxSize = 8; endOfArray = 4

Before entry to the loop

Accessible portion of the array



maxSize = 8; endOfArray = 3

After exiting the loop

Time complexity of Delete

Worst Case: **If the delete is to be done at index 0**

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

endOfArray--;
```

Assume deleteIndex = 0

0	1	2	3	4	5	6	7
10	23	13	17	37			
200	204	208	212	216	220	224	228

maxSize = 8; endOfArray = 4

Before entry to the loop

index = 0

0	1	2	3	4	5	6	7
23	23	13	17	37			

index = 1

0	1	2	3	4	5	6	7
23	13	13	17	37			

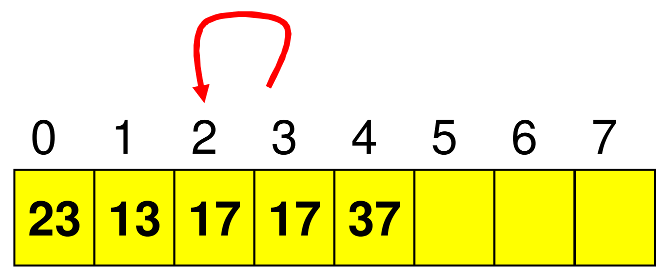
index = 2

0	1	2	3	4	5	6	7
23	13	17	17	37			

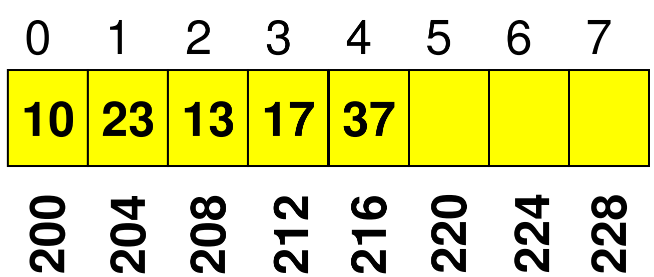
Worst Case: If the delete is to be done at index 0

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

endOfArray--;
```

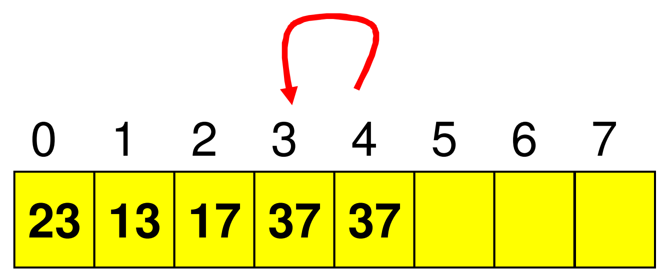
index = 2



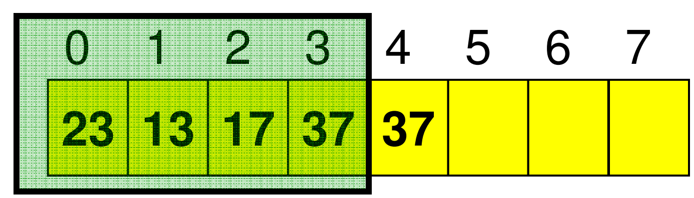
Assume deleteIndex = 0



index = 3



Accessible portion of the array



maxSize = 8; endOfArray = 3

Before entry to the loop

The worst case number of copy operations correspond to endOfArray, which is n-1 where n is maxSize.
Hence, the time complexity of the Delete Operation is O(n-1) = 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

Pros and Cons of Implementing Dynamic List using Array

- Pros: Θ(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); inserting at the beginning of the list is the most time consuming.
  - 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. Note: Array is a contiguous block of memory
  - Also, note that when we double the space for an array-based List, half of it could remain unused

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**

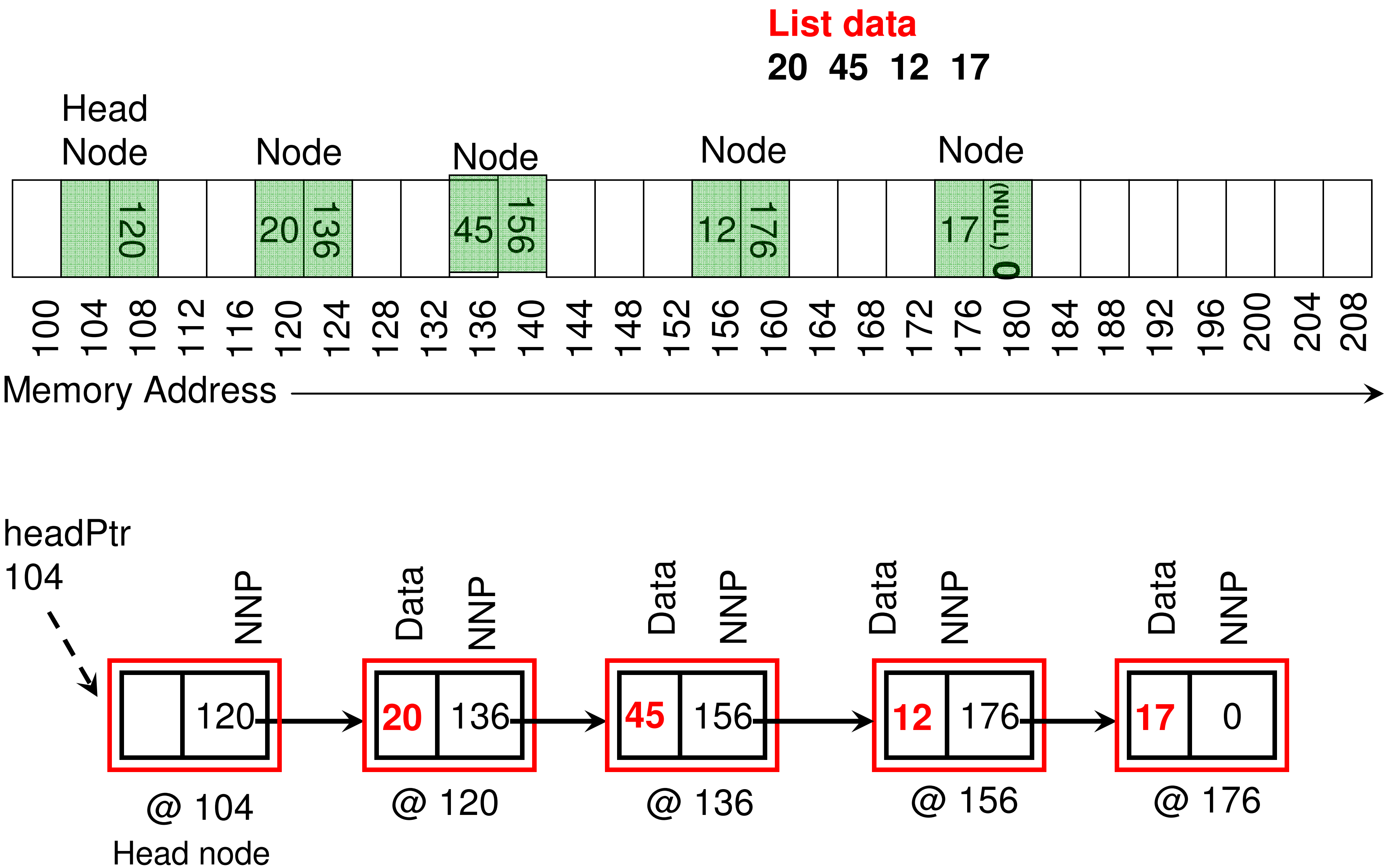
  nextNodePtr
- Doubly Linked List: Each node contains the address of the node with the subsequent value as well as the address of the node with the preceding value. There is also a head node pointing to the first node in the list and a tail node pointing to the last node in the list.

  prevNodePtr

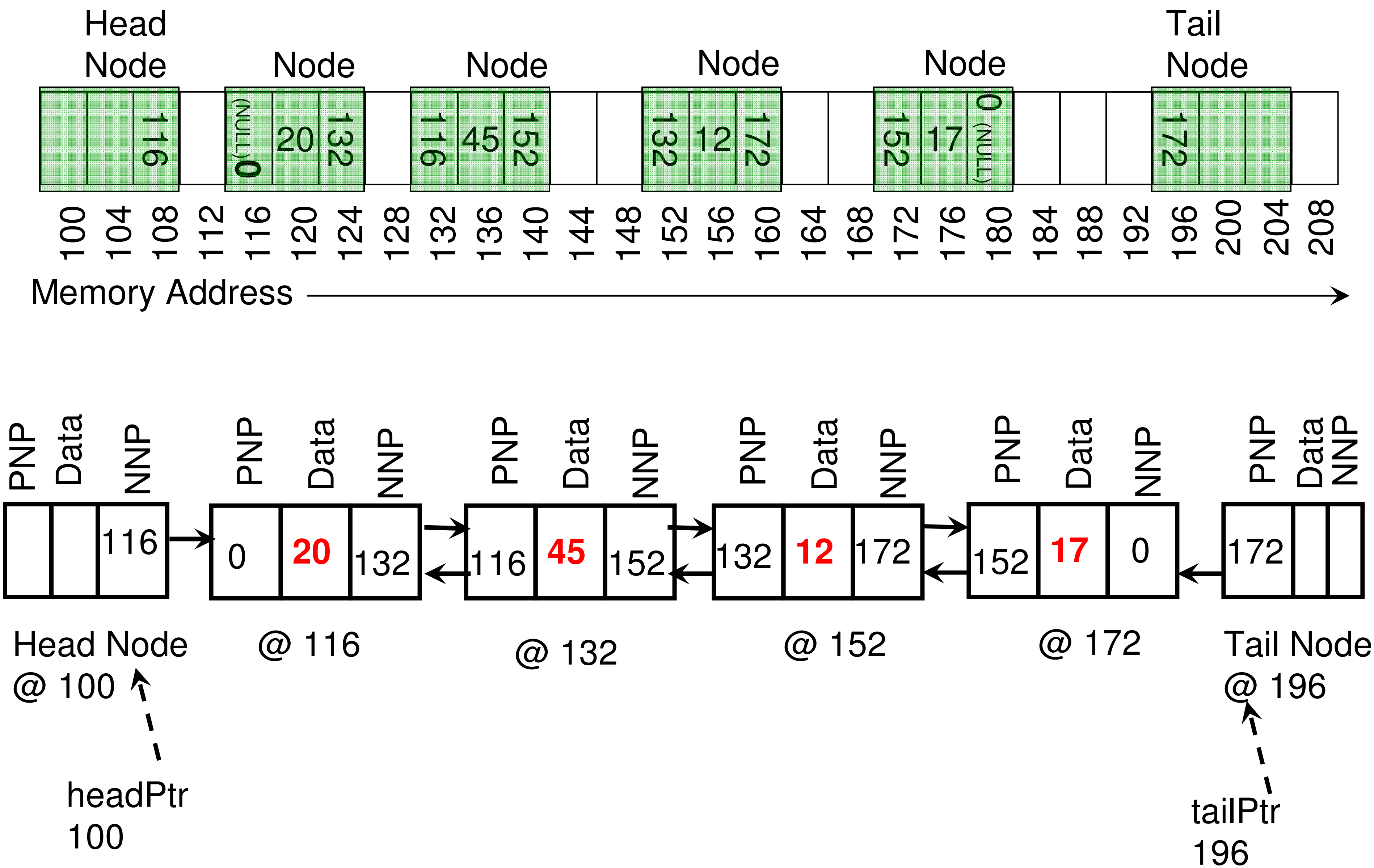
  Data **With doubly linked list – we can traverse in both directions**

  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



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;
   }
```

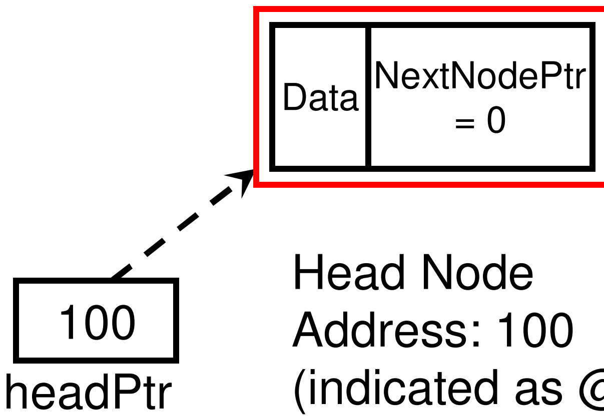
```
public:
   void setNextNodePtr(Node* nodePtr){
              nextNodePtr = nodePtr;
   }

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

Singly Linked List: Class List

Class List (C++)

```
private:
   Node *headPtr;

public:      /* Note that the data for the
   List( ){  Head node is not set */
         headPtr = new Node();
         headPtr->setNextNodePtr(0);
   }
```

Initialization of List Object

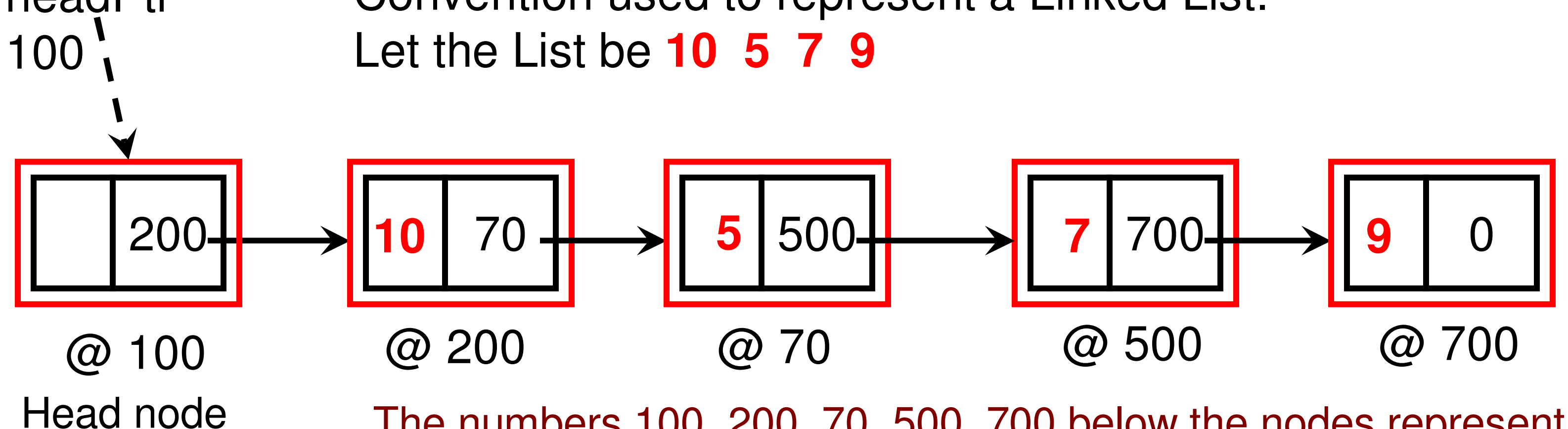


Head Node
Address: 100
(indicated as @ 100)

// 0 – indicates NULL
// i.e., the ptr is not
// pointing to
// any address

headPtr
100

Convention used to represent a Linked List.
Let the List be **10 5 7 9**



The numbers 100, 200, 70, 500, 700 below the nodes represent the address at which these nodes are stored, indicated with an @ symbol

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);
}
```

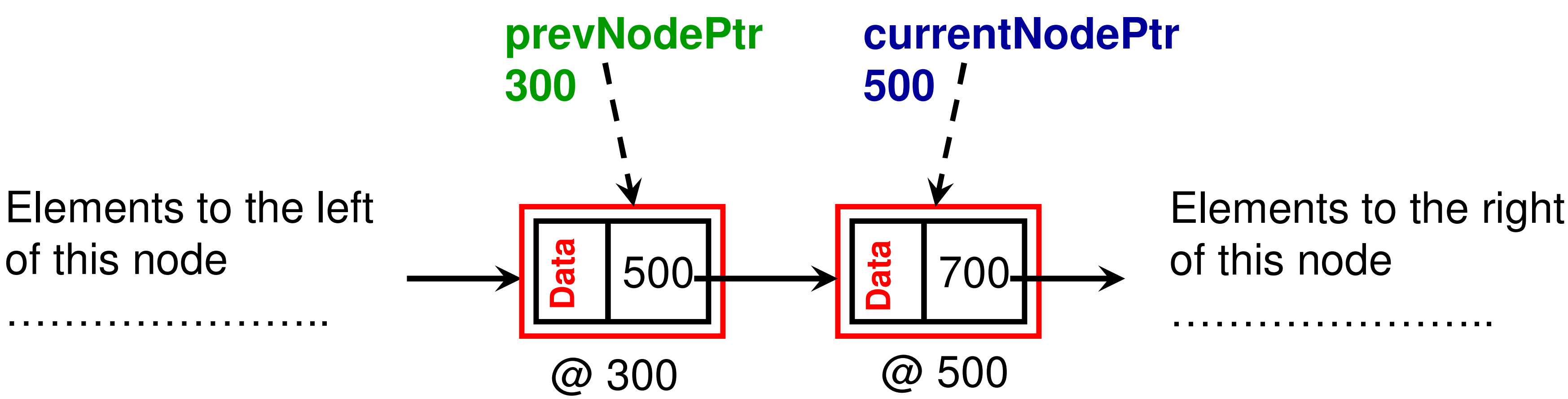
Move the currentNode ptr from first node in the list to end of the list. When we come out of the 'while' loop, the prevNode ptr is the last node in the list and currentNode ptr points to null (0).

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;
}
```

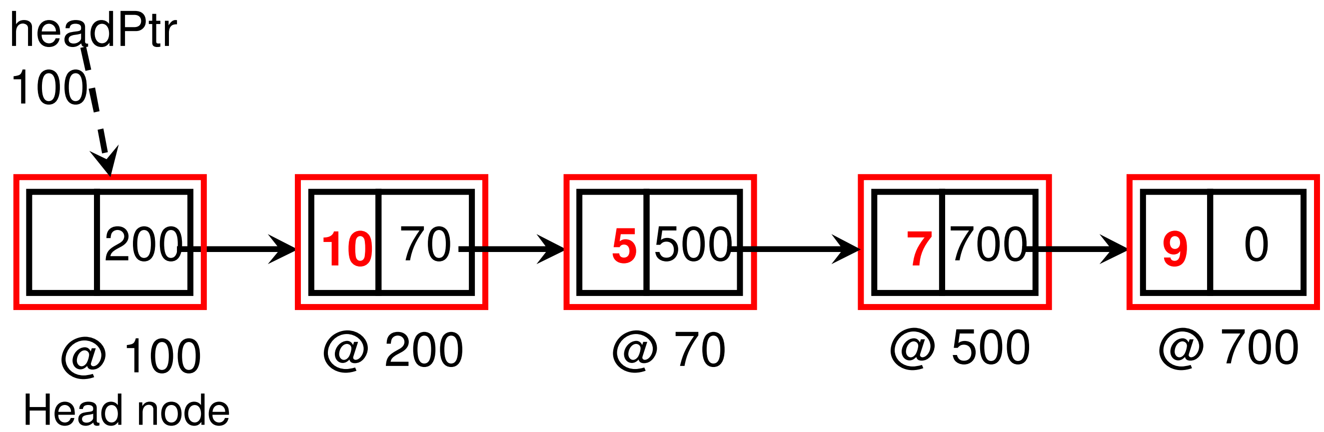
prevNodePtr and currentNodePtr

- As we traverse through the list, node by node, we will maintain two pointers: the prevNodePtr and currentNodePtr.
  - The currentNodePtr has the address for the node that is currently being visited/ processed.
  - The prevNodePtr has the address for the node that was just visited before the current node.
- We have reached the end of the list when currentNodePtr is 0 (i.e., does not point to any node).



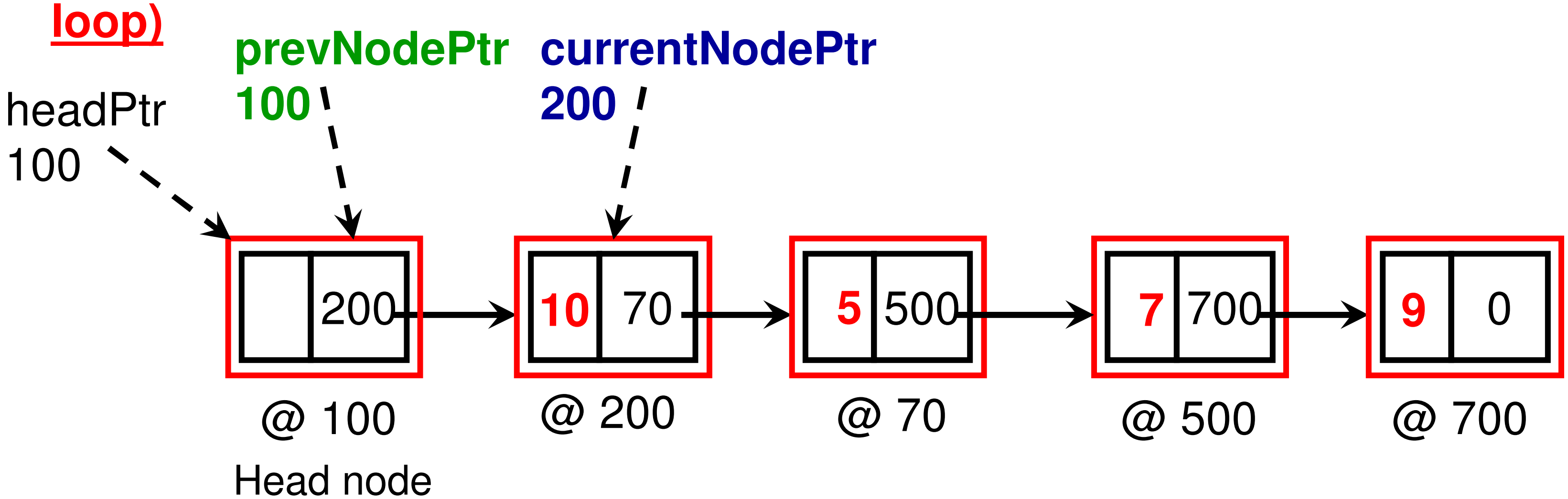
Example: Insertion at the End of the List (1)

Let the List be **10 5 7 9** and now we want to insert element '30' at the end.



Initialization of prevNodePtr and currentNodePtr (before the while loop)

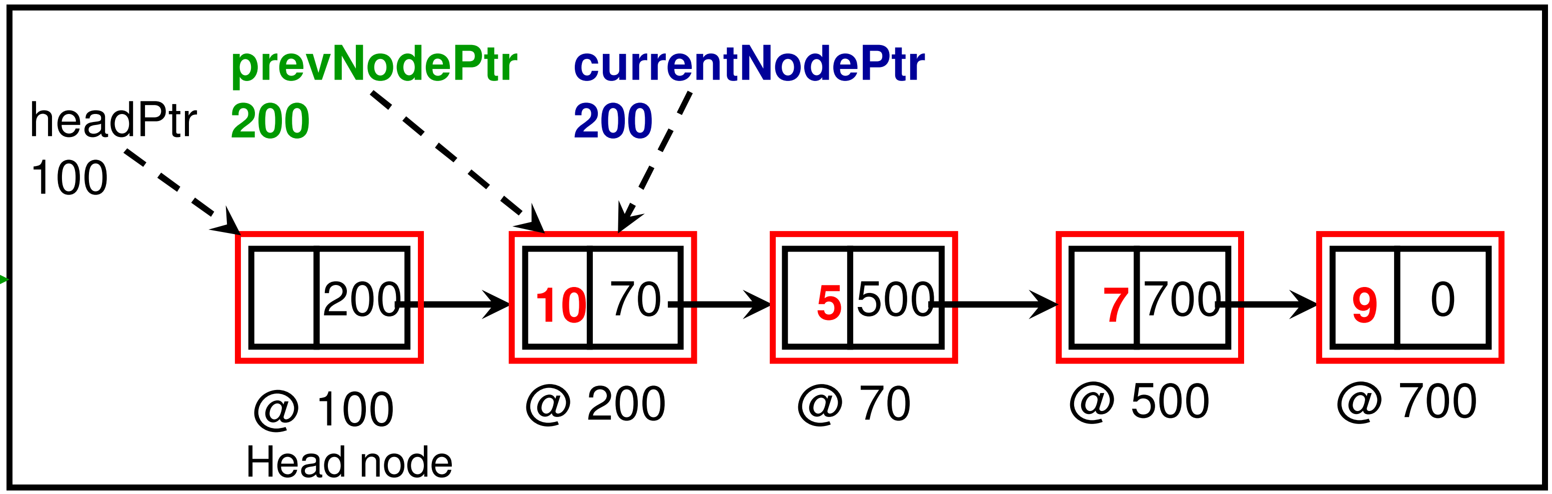
```
Node* currentNodePtr = headPtr->getNextNodePtr();
Node* prevNodePtr = headPtr;
```

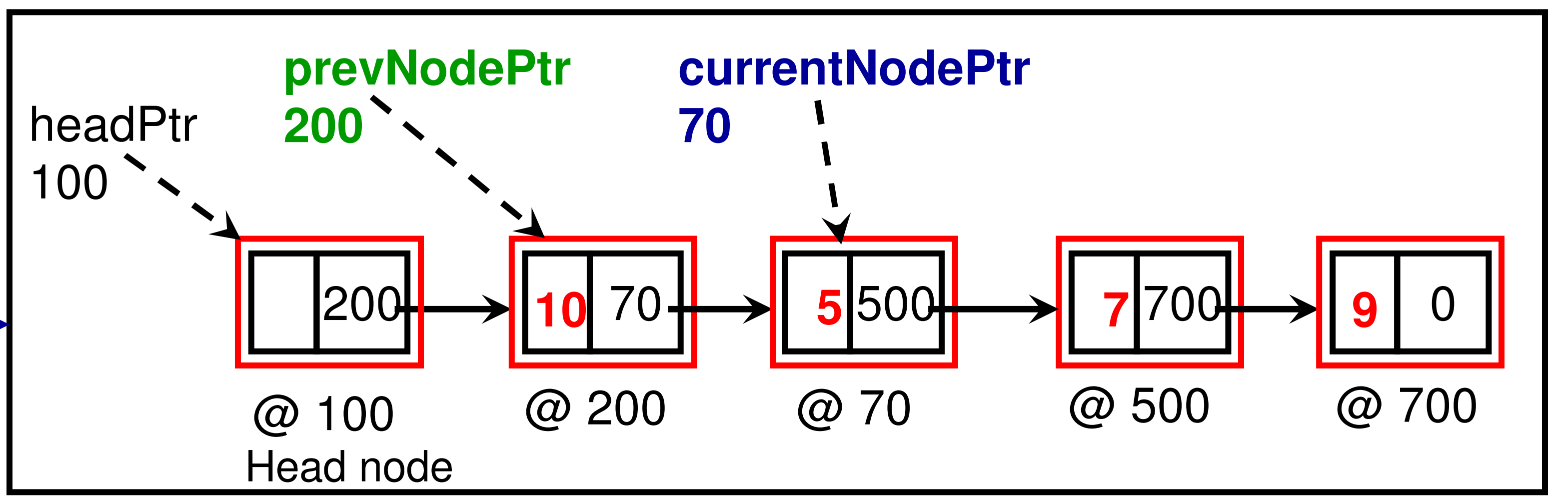


Inside the while loop

Example continued (2)

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

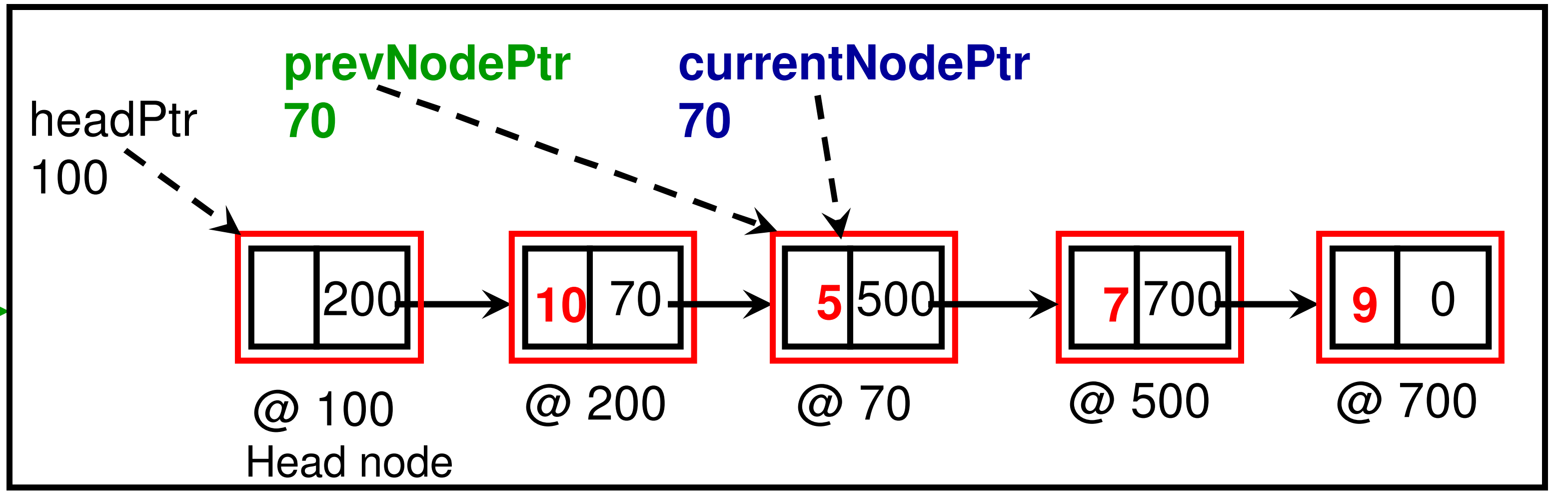


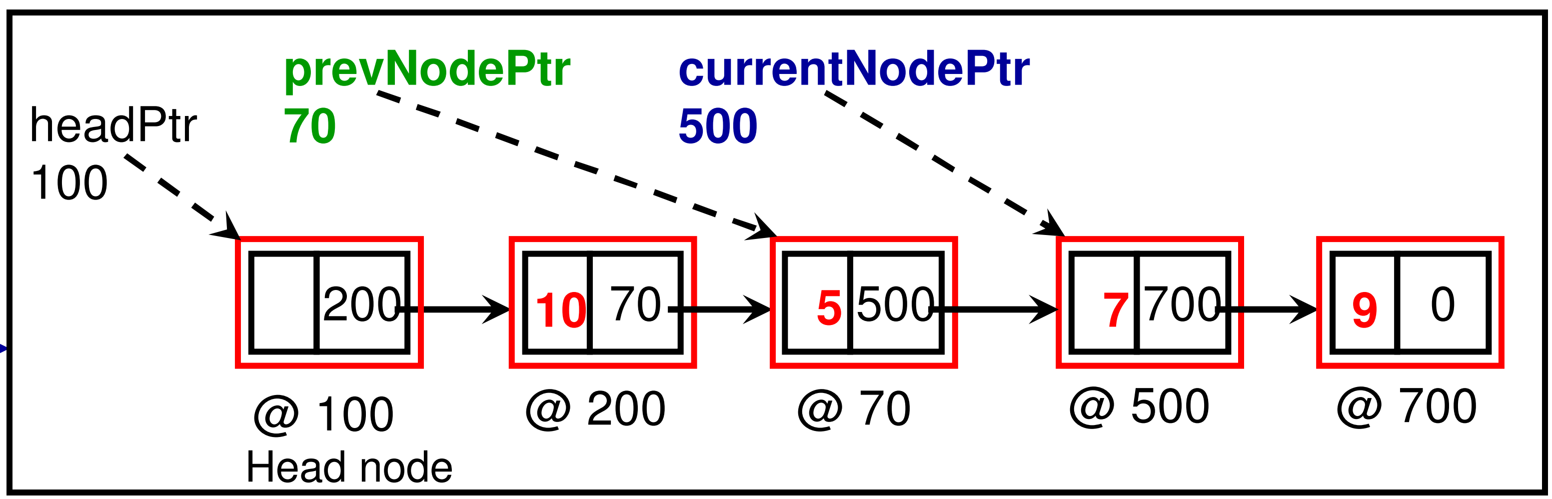


Inside the while loop

Example continued (3)

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

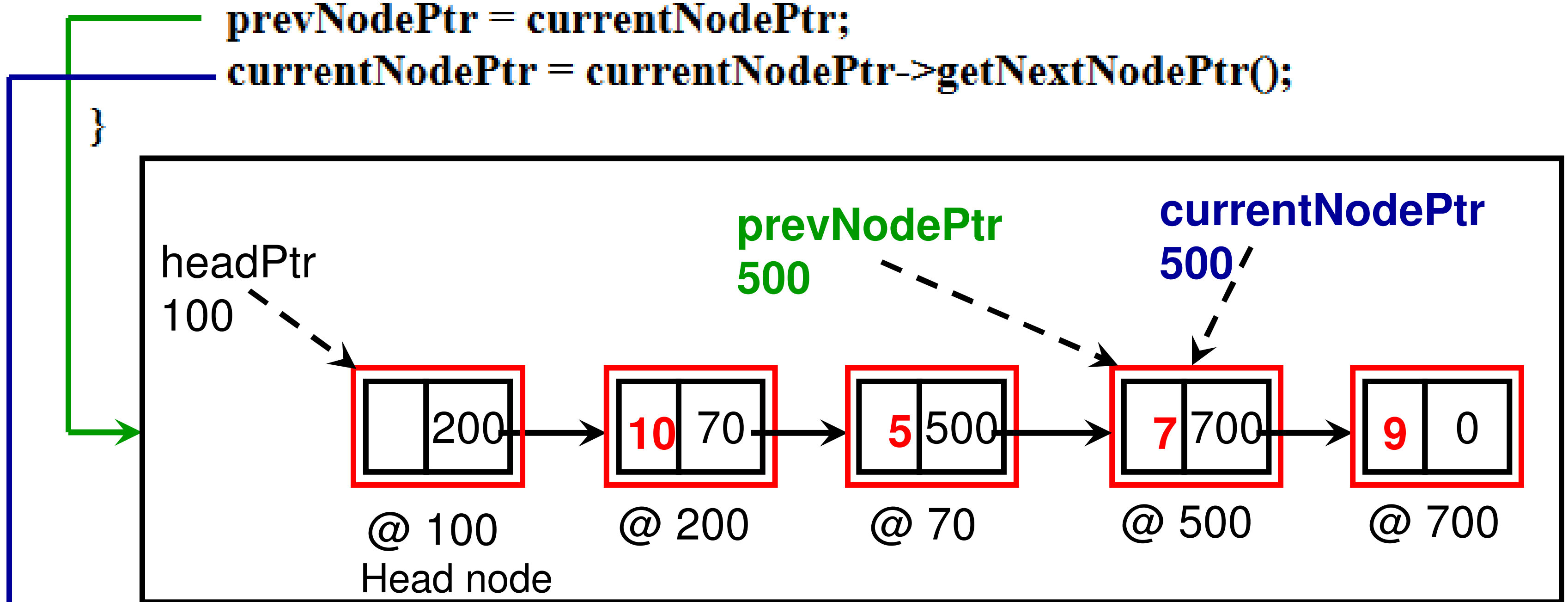


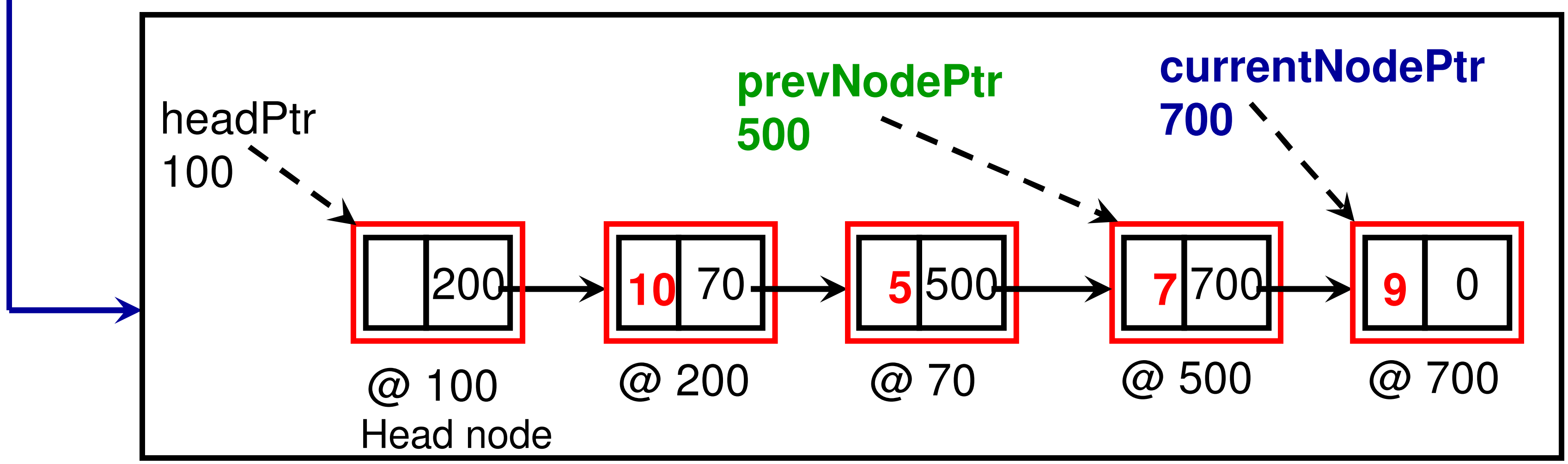


Inside the while loop

Example continued (4)

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

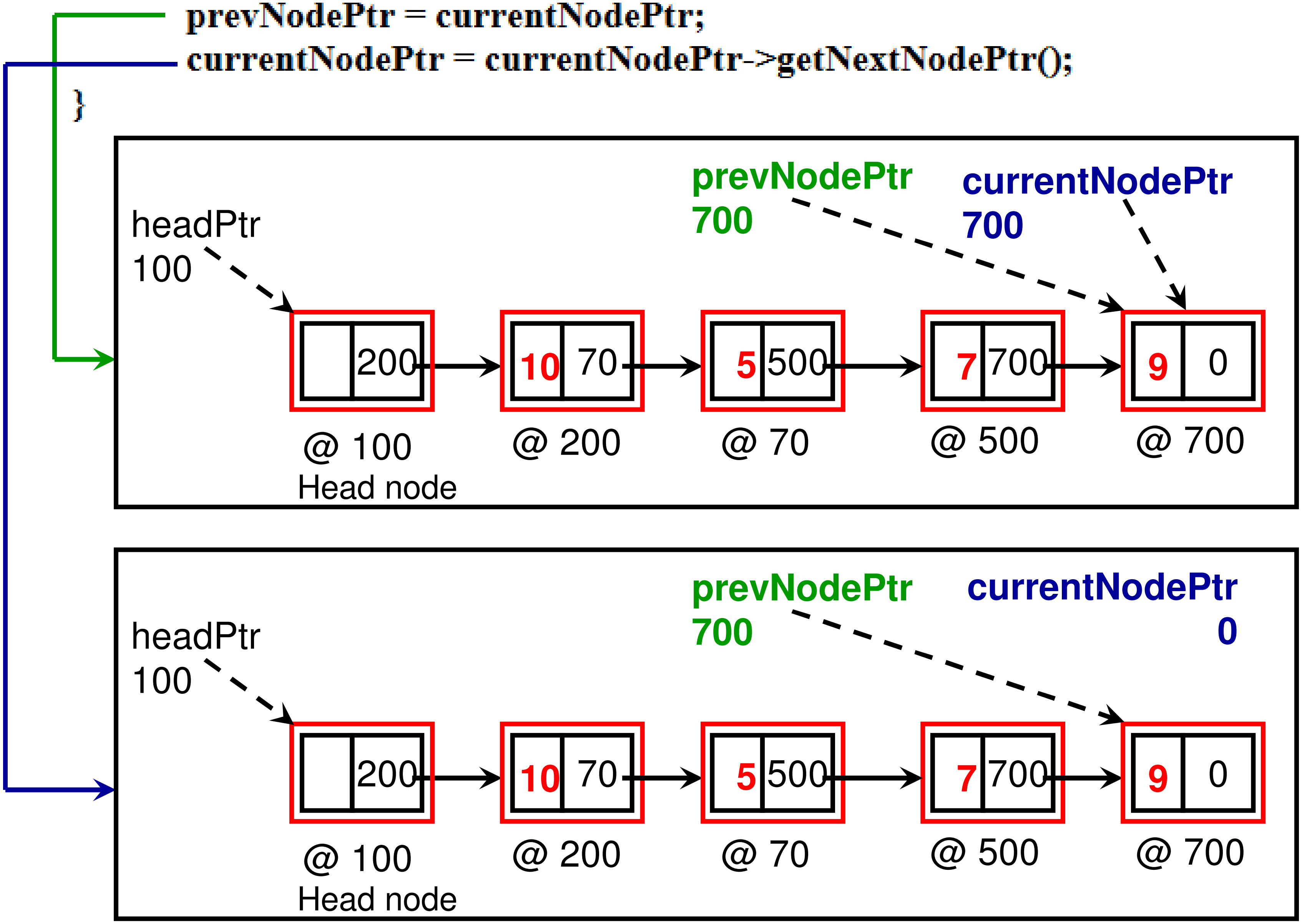




Inside the while loop

Example continued (5)

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

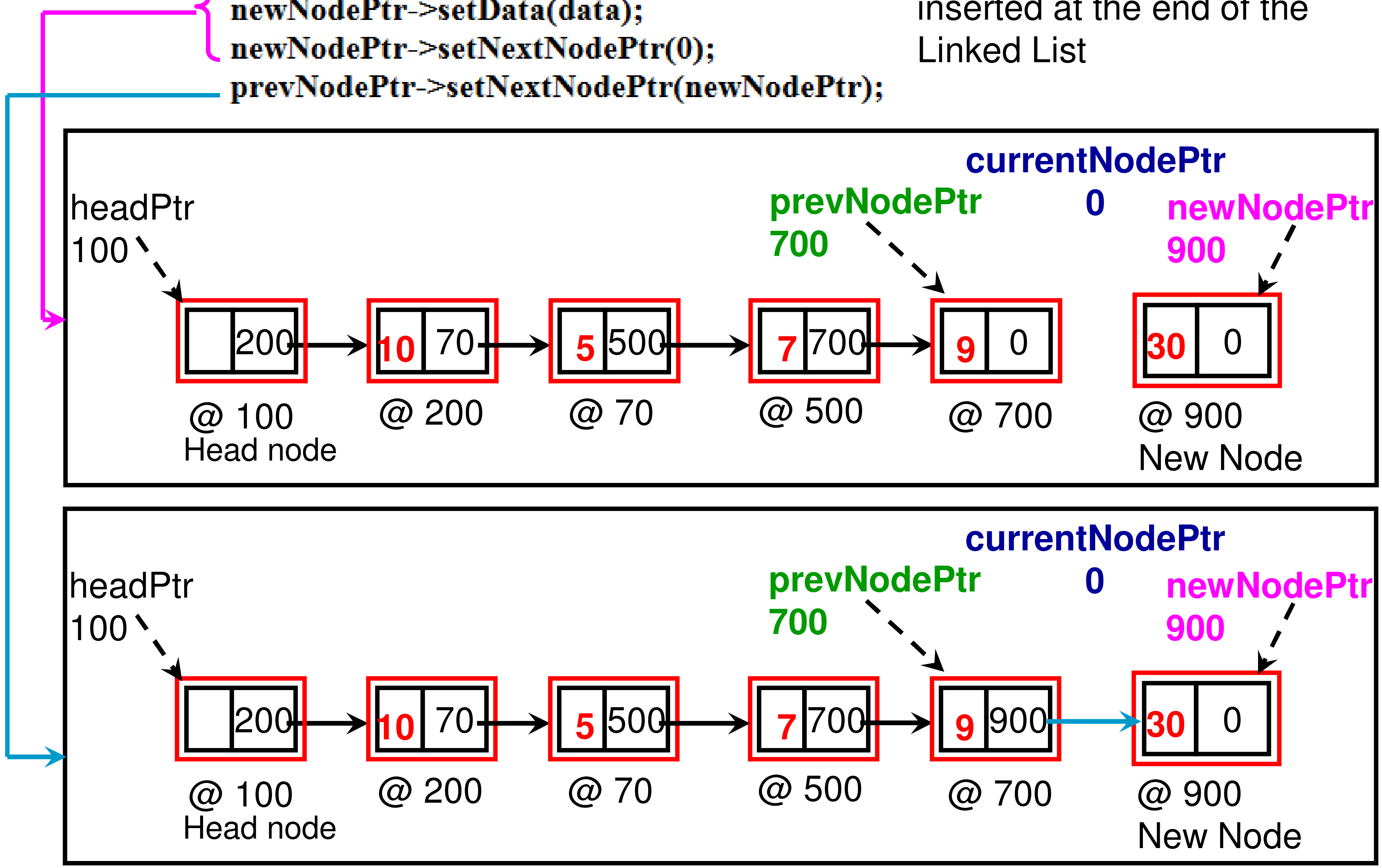


After the while loop

Example continued (6)

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

Let '30' be the data to be inserted at the end of the Linked List



Class List (C++)

index refers to the node pointed by currentNodePtr at any time

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.

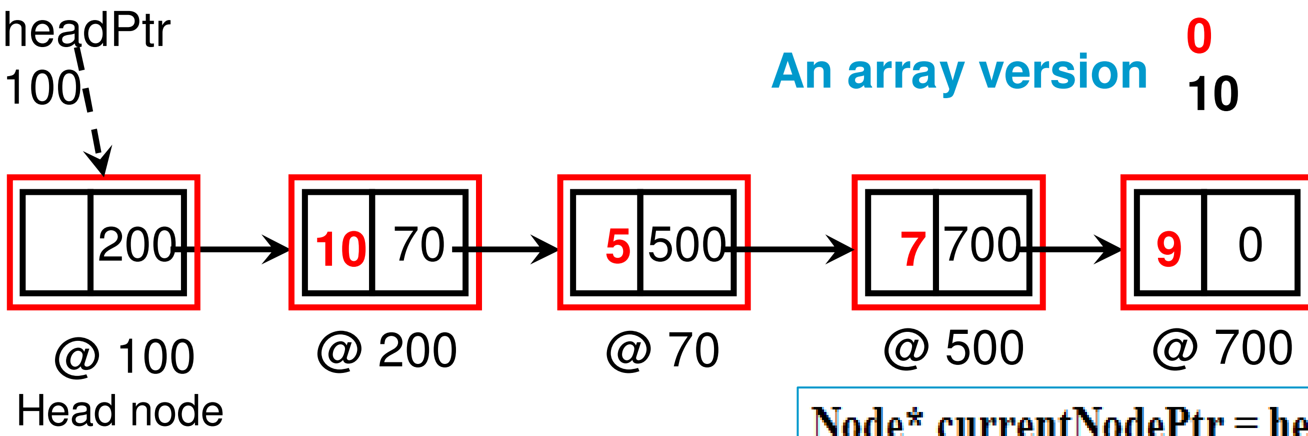
```
void insertAtIndex(int insertIndex, int data){

    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);
}
```

Example: Insertion at *insertIndex* = 2 (1)

Let the List be **10 5 7 9** and let us say we want to insert element '30' at *insertIndex* = 2

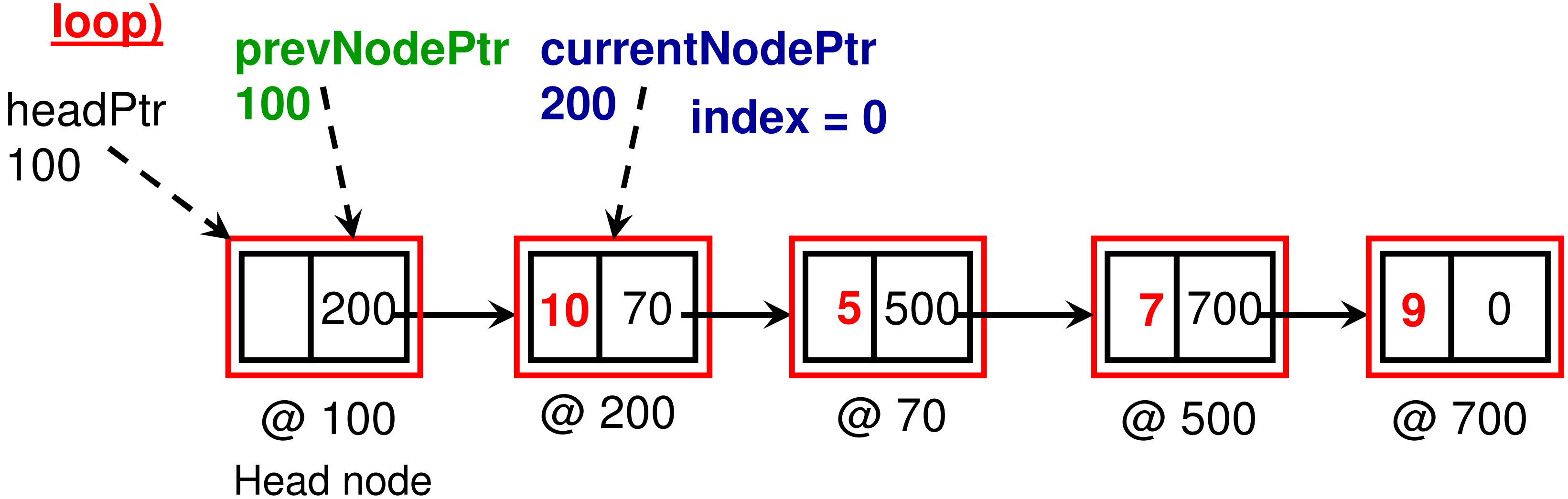
An array version

0	1	2	3
10	**5**	**7**	**9**




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

Initialization of prevNodePtr and currentNodePtr (before the while loop)

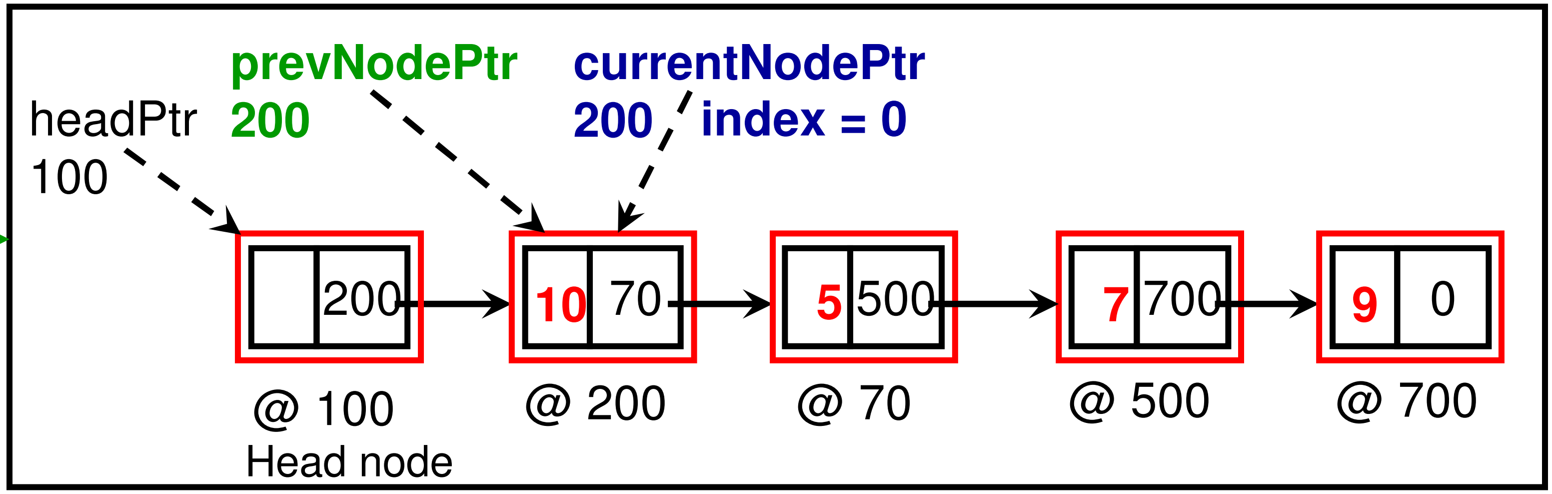


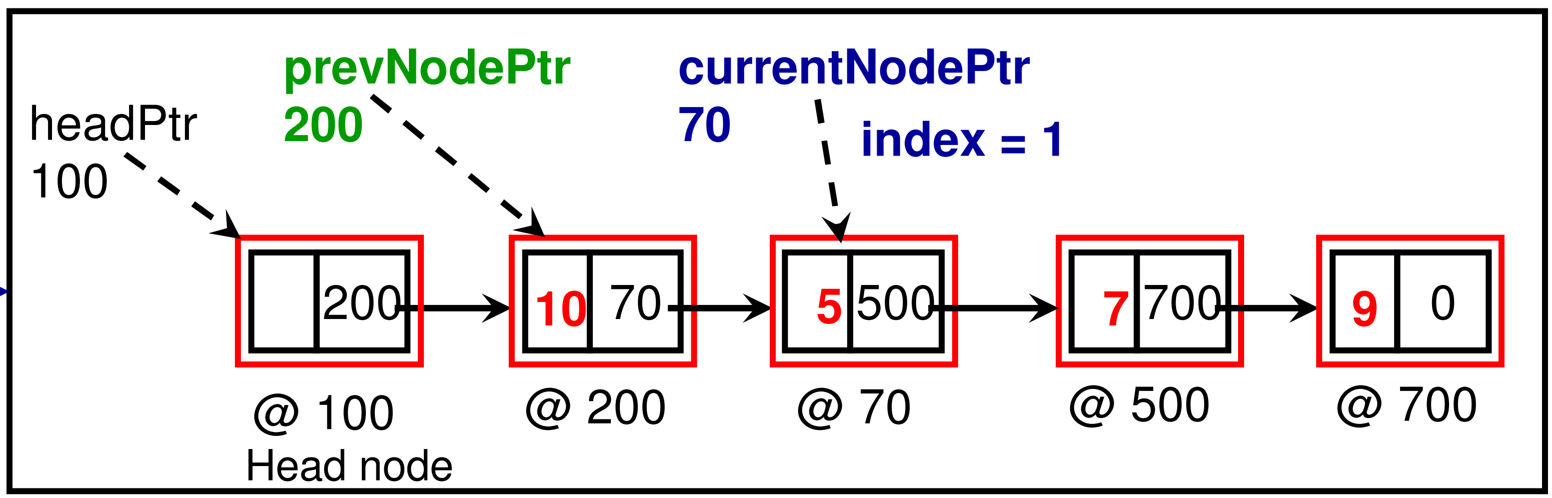
Inside the while loop

Example continued (2)

insertIndex = 2

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



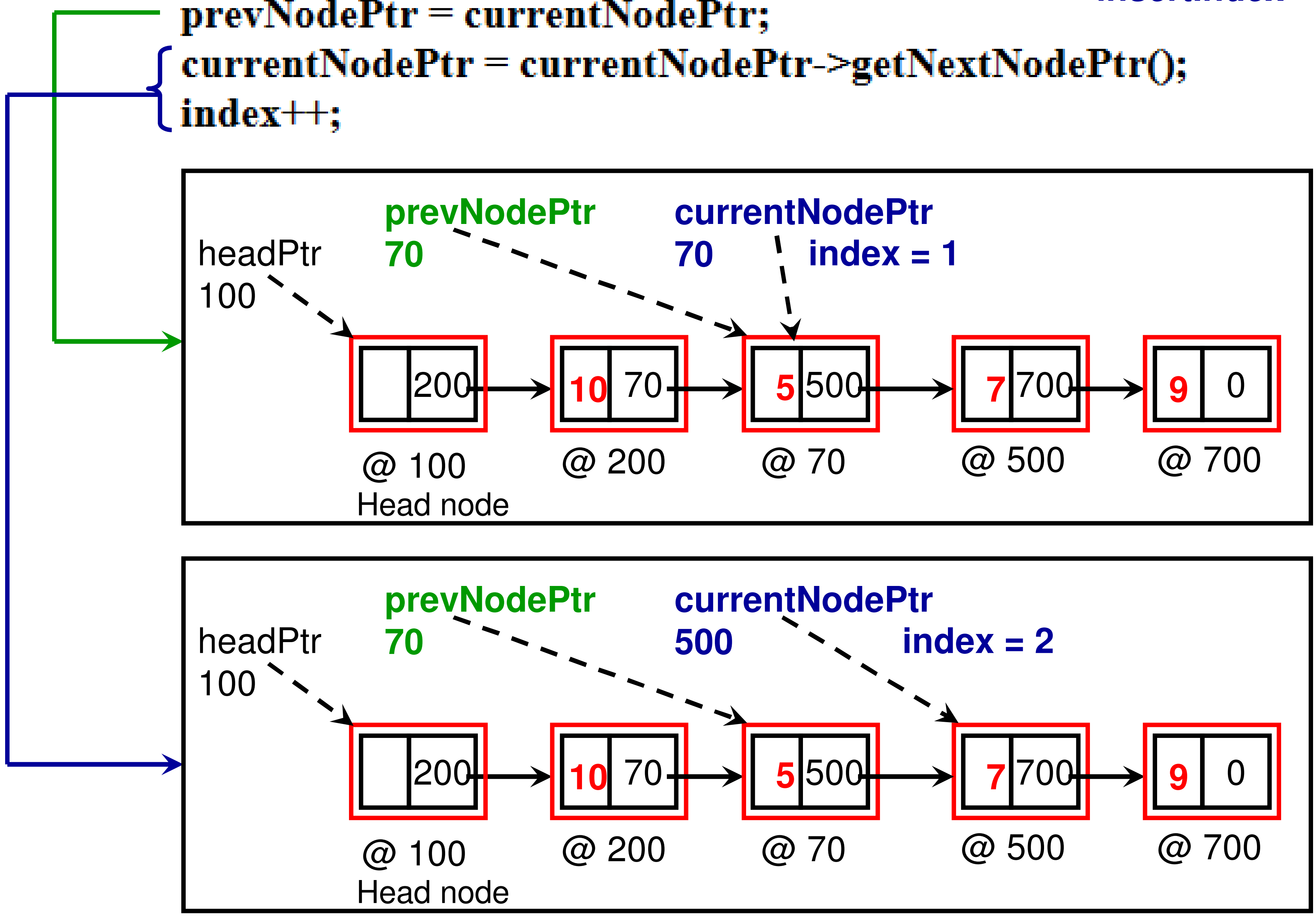


Inside the while loop

Example continued (3)

insertIndex = 2

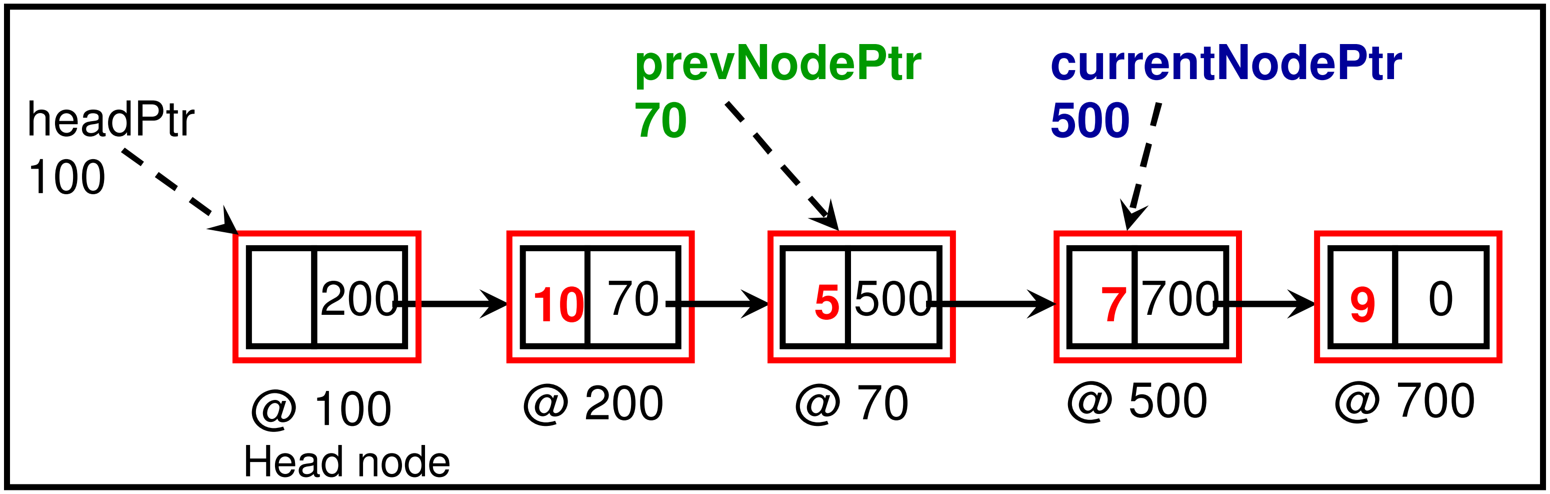
```
prevNodePtr = currentNodePtr;
currentNodePtr = currentNodePtr->getNextNodePtr();
index++;
```



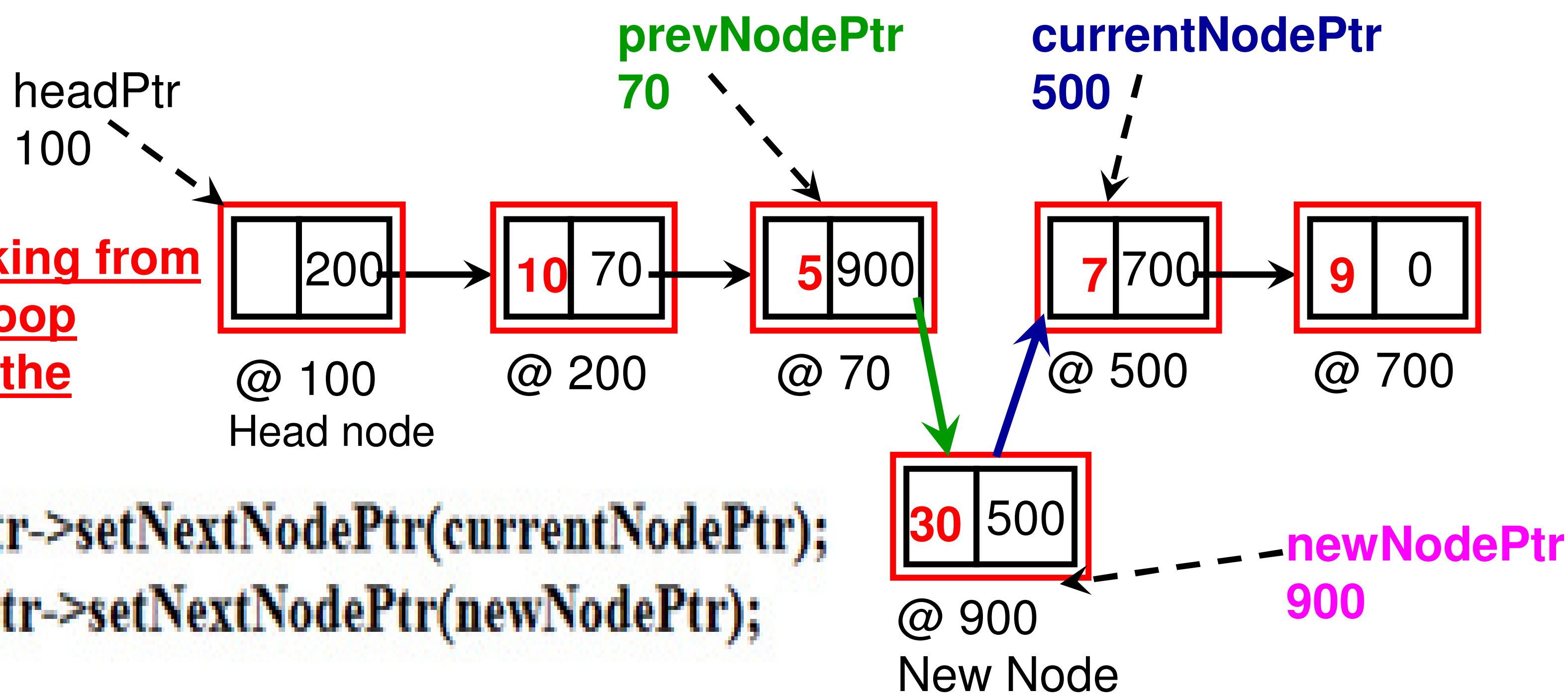
At the time of breaking from the while loop

Example continued (3)

index = 2 **_insertIndex_ = 2**



After breaking from the while loop Linking of the newNode




```
newNodePtr->setNextNodePtr(currentNodePtr);
prevNodePtr->setNextNodePtr(newNodePtr);
```

Class List (C++)

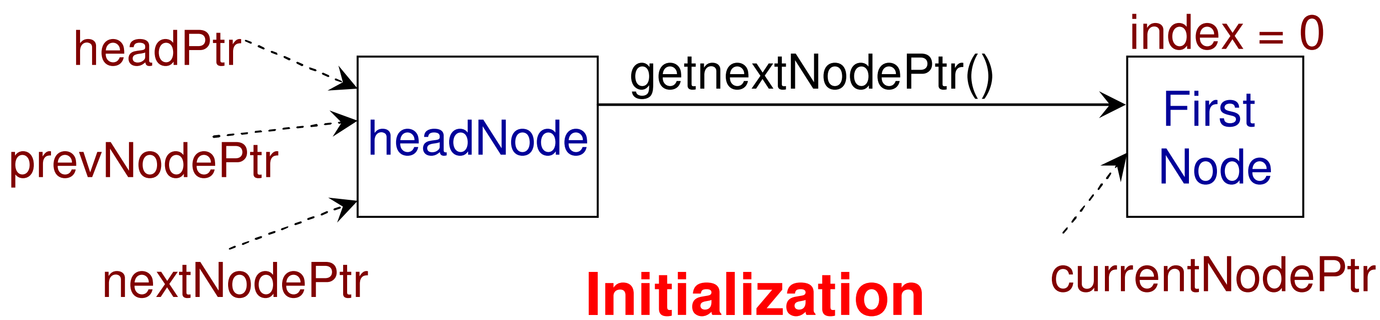
```
int read(int readIndex){
    Node* currentNodePtr = headPtr->getNextNodePtr();
    Node* prevNodePtr = headPtr;
    int index = 0;

    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

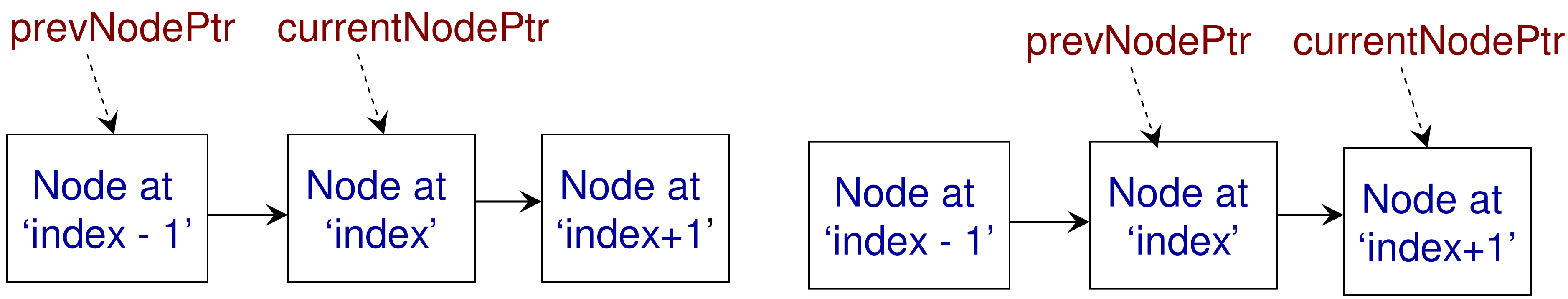
Class List (C++)

```
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++;
        }
}
```

Delete (deleteIndex) Function



Initialization



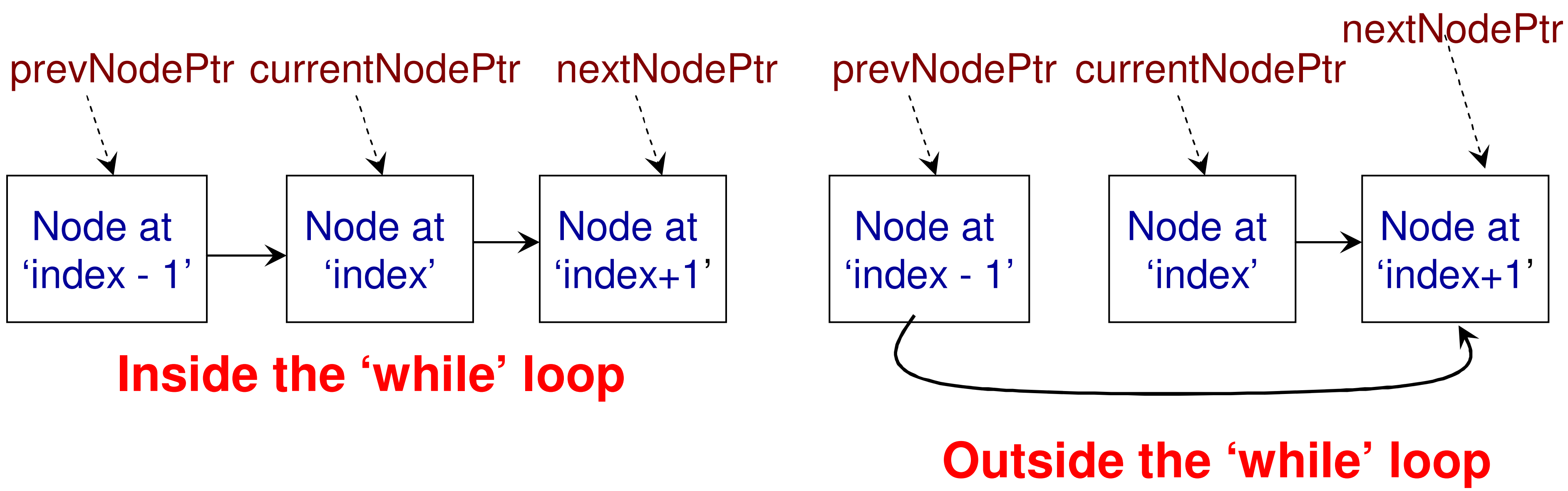
At the beginning of an iteration inside the 'while' loop

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

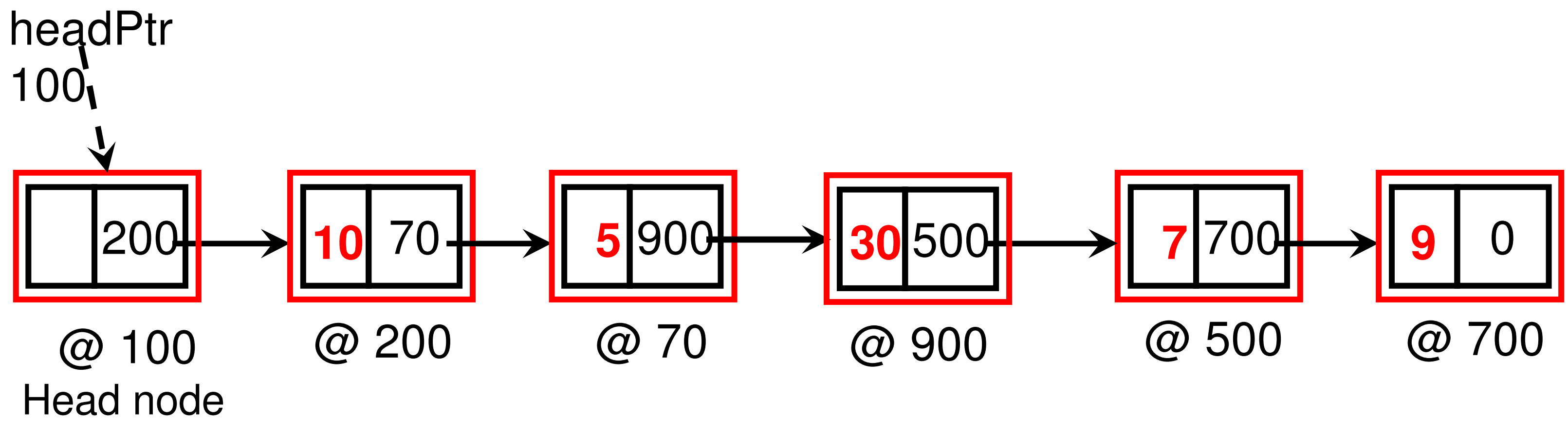
Class List (C++)

```
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);
}
```

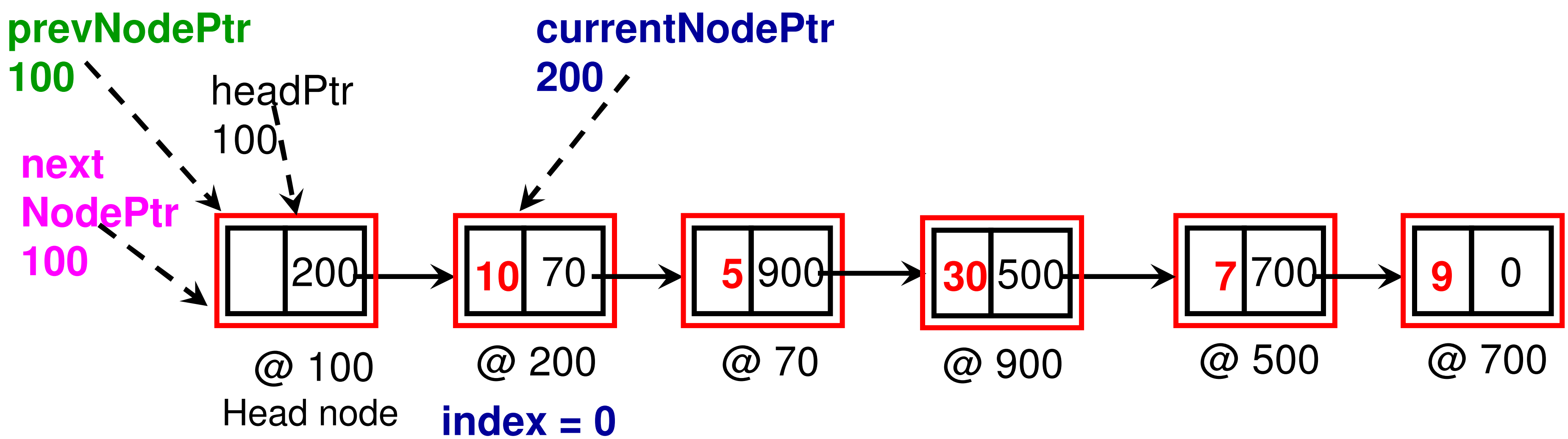
The next node for 'prevNode' ptr is now 'next node' and not 'current node'

Example: Deletion at *deleteIndex = 2*

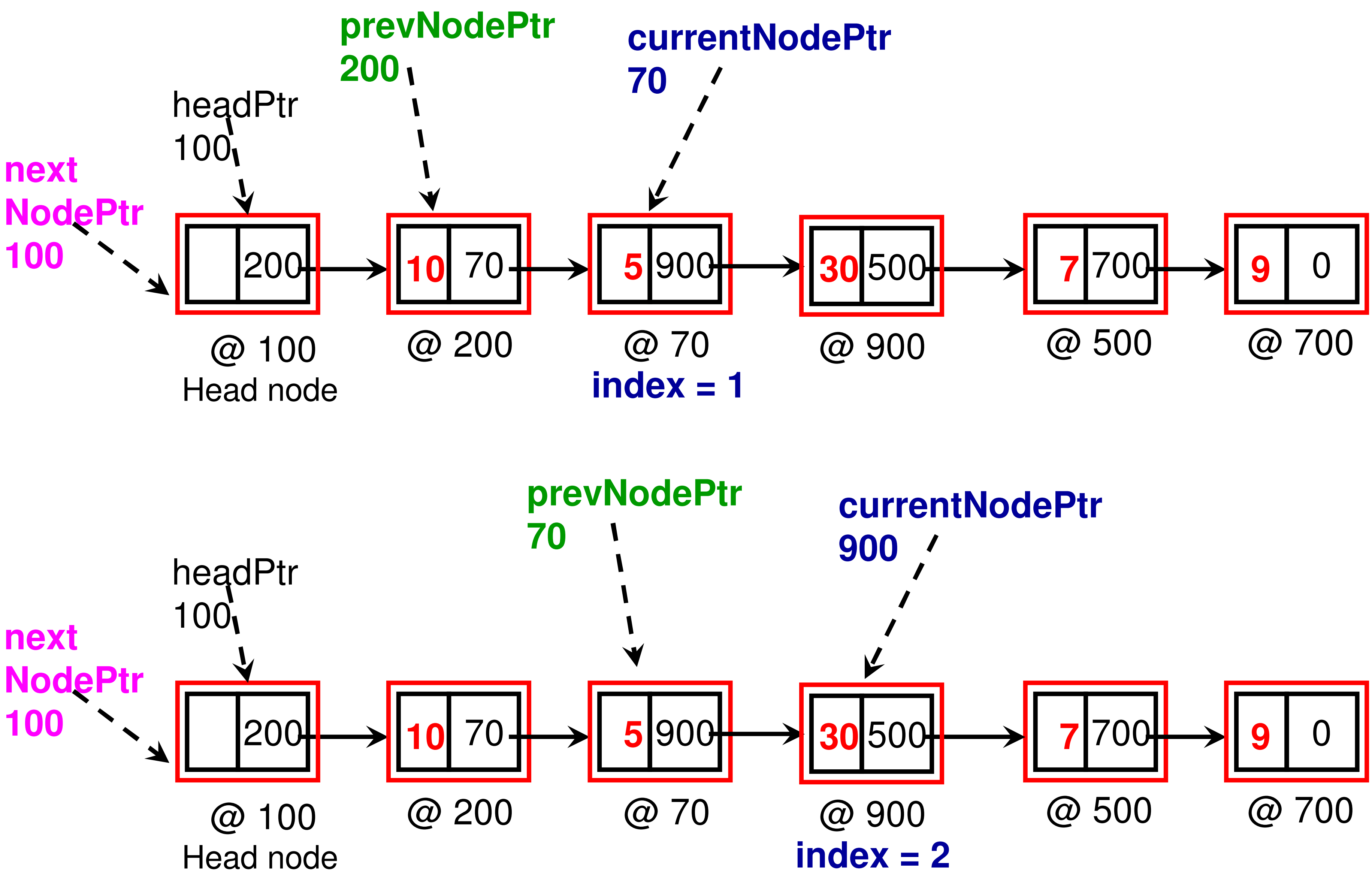
Let the List be **10 5 30 7 9** and now we want to delete '30' at *deleteIndex* = 2



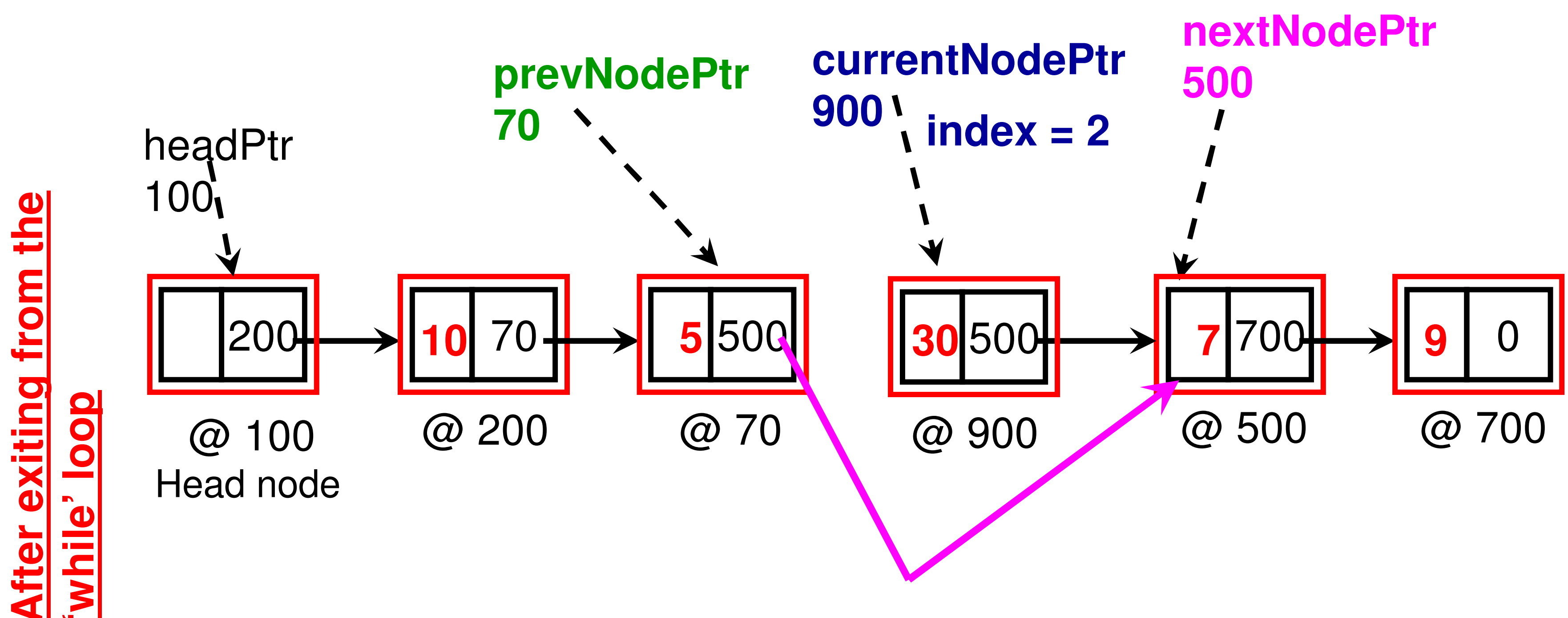
Initialization of the pointers



Example: Deletion at *deleteIndex* = 2



```
while (currentNodePtr != 0){

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

}

prevNodePtr->setNextNodePtr(nextNodePtr);
```



Delinking of Node at index = 2 (element '30' from the Linked List)

Iterative Print

Class List (C++)

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

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

        cout << endl;
  }
```

Linked List vs. Arrays: Memory Usage

	Data size	Next Node Ptr	Prev Node Ptr	Node Size	%ovh
Singly Linked List	4 (int)	4	N/A	8 bytes	100%
Singly Linked List	32	4	N/A	36 bytes	12.5%
Doubly Linked List	4 (int)	4	4	12 bytes	200%
Doubly Linked List	32	4	4	40 bytes	25%

Code 6: Run Time Complexity Analysis

```
#include <iostream>
#include <ctime>
#include <ratio>
#include <chrono>
using namespace std;

int main ()
{
  using namespace std::chrono;

  high_resolution_clock::time_point t1 = high_resolution_clock::now();

  cout << "printing out 1000 stars...\n";
  for (int i=0; i<1000; ++i) cout << "*";
  cout << endl;

  high_resolution_clock::time_point t2 = high_resolution_clock::now();

  duration<double, std::nano> time_span_nano = t2 - t1;
  duration<double, std::micro> time_span_micro = t2 - t1;
  duration<double, std::milli> time_span_milli = t2 - t1;

  cout << "It took me " << time_span_nano.count() << " nanoseconds." << endl;
  cout << "It took me " << time_span_micro.count() << " microseconds." << endl;
  cout << "It took me " << time_span_milli.count() << " milliseconds." << endl;
  cout << endl;

  return 0;
}
```

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	**Θ(1)**	**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 (remember: arrays come with the overhead of creating a new block of memory, if needed, and copying the elements to the new block)

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.

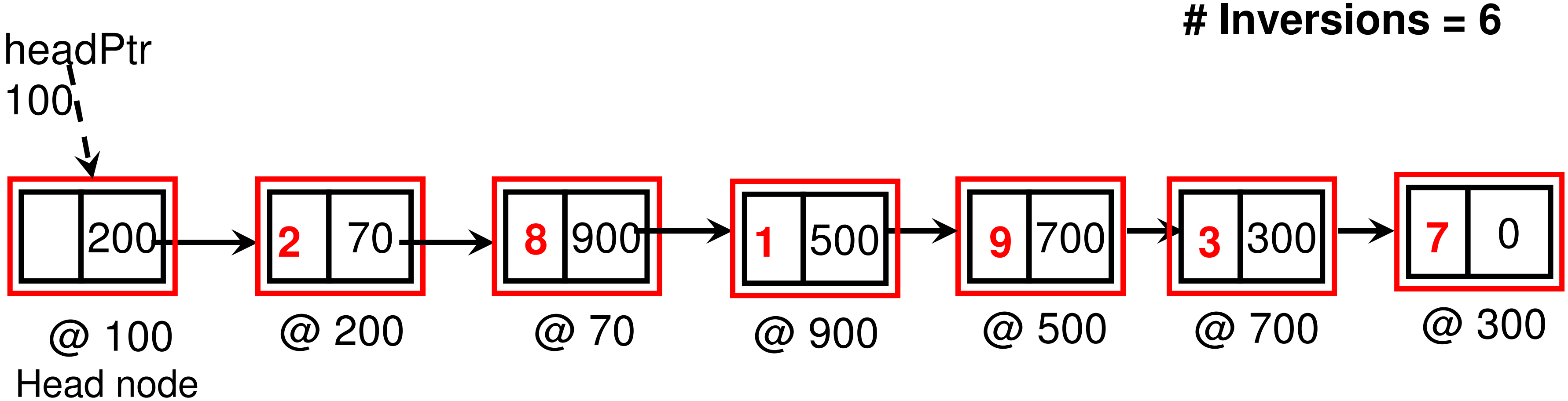
Number of Inversions in a Singly Linked List

- Given a Singly Linked List of data (say integers), an inversion is said to have occurred if an integer i is more closer to the head node compared to an integer j, but i > j

- **Example**

Inverted Pairs
(2, 1)
(8, 1)
(8, 3)
(8, 7)
(9, 3)
(9, 7)

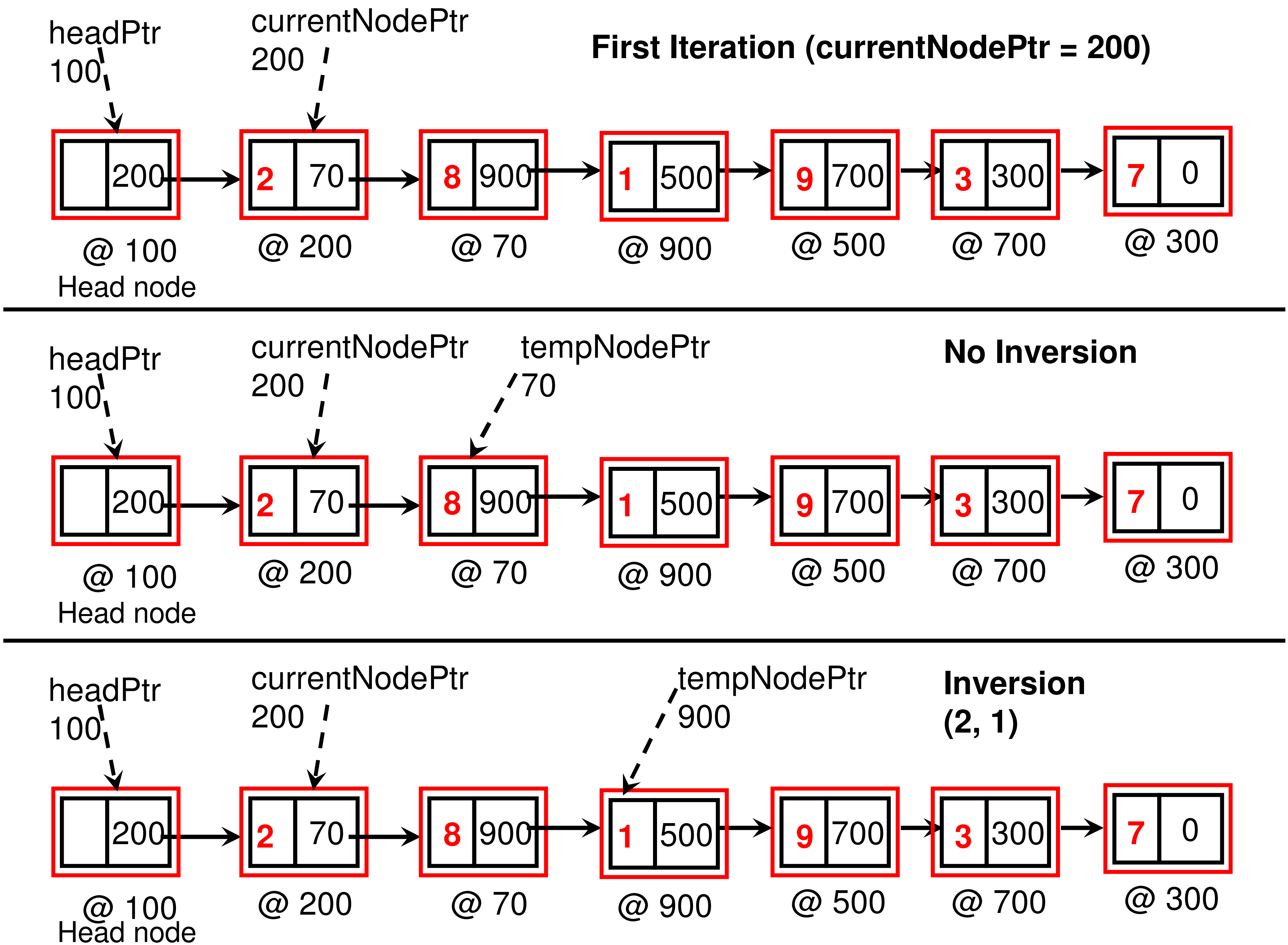
 # Inversions = 6



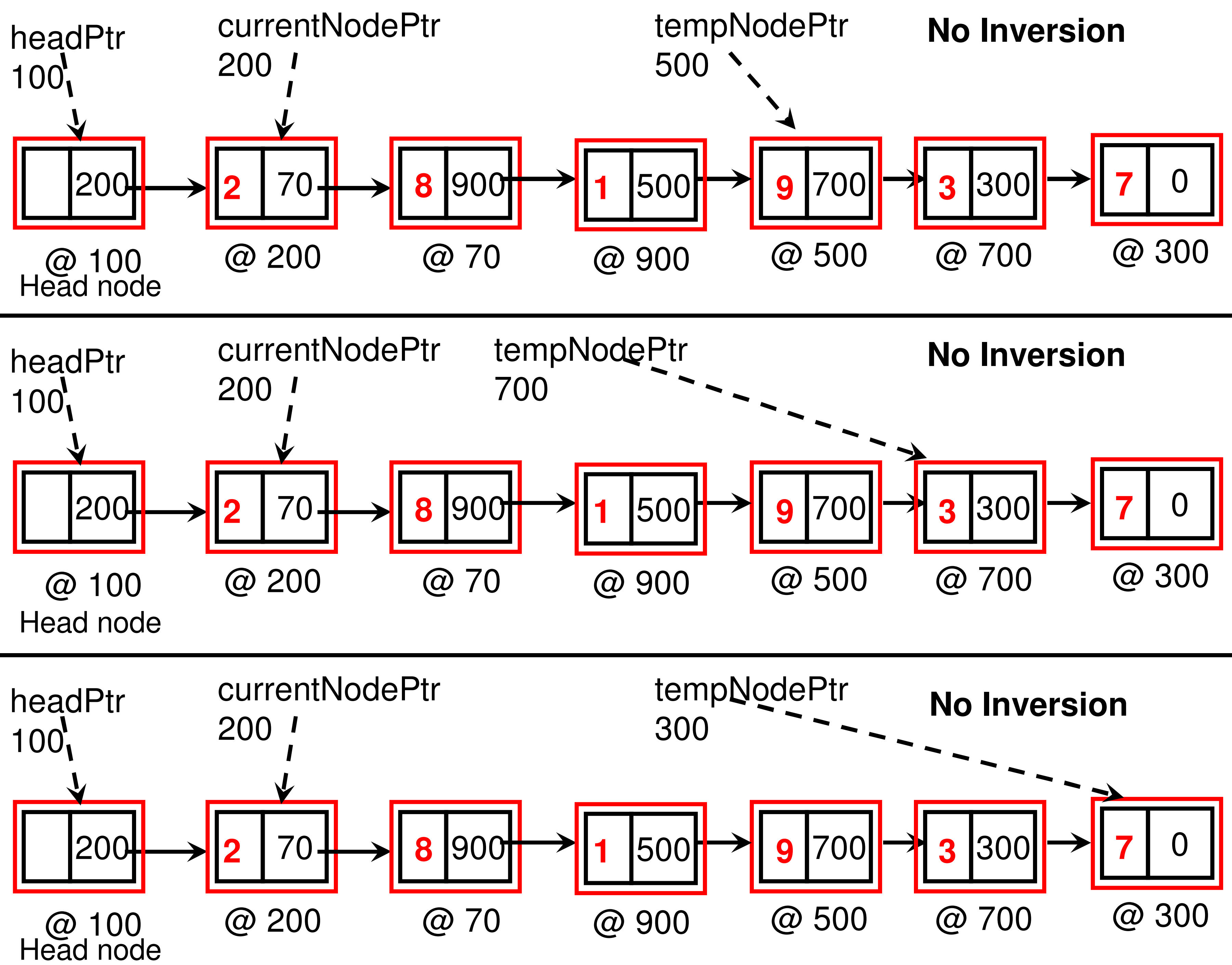
Code 8: Number of Inversions in a Singly Linked List

Note: This code is in the main( ) function, assuming listSize is a variable input by the user for the number of elements in the list

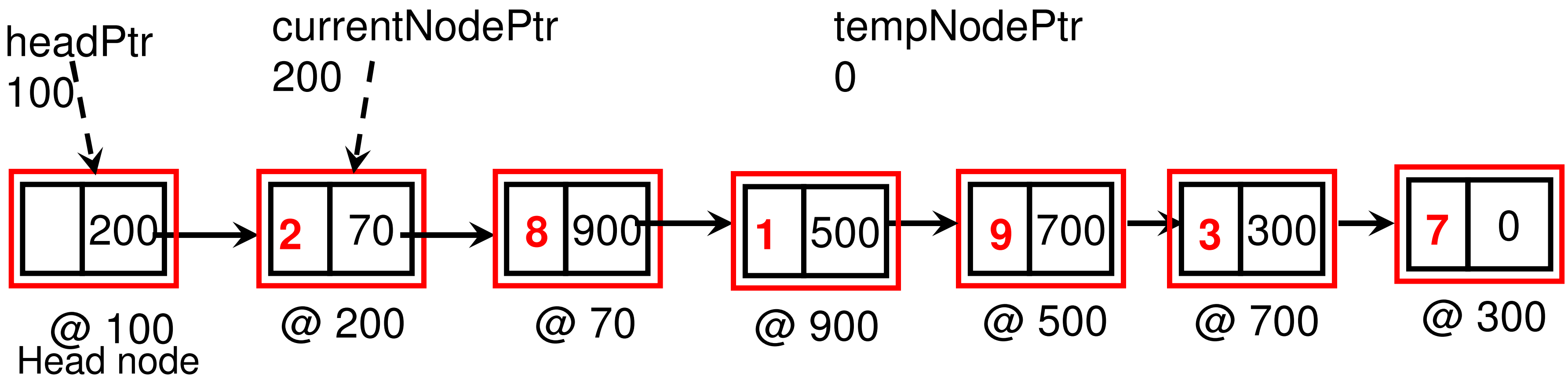
```
int numInversions = 0;
if (listSize > 1){
        Node* headPtr = integerList.getHeadPtr();
        Node* currentNodePtr = headPtr->getNextNodePtr();
        while (currentNodePtr != 0){
                Node* tempNodePtr = currentNodePtr->getNextNodePtr();
                while (tempNodePtr != 0){
                        if (currentNodePtr->getData() > tempNodePtr->getData()){
                                cout << "(" << currentNodePtr->getData() << ", " <<
                                        tempNodePtr->getData() << ")" << endl;
                                numInversions++;
                        }
                        tempNodePtr = tempNodePtr->getNextNodePtr();
                }
                currentNodePtr = currentNodePtr->getNextNodePtr();
        }
        cout << "The number of inversions is " << numInversions << endl;
}
else{
        cout << "As the list size is <= 1; there cannot be any inversions " << endl;
}
```

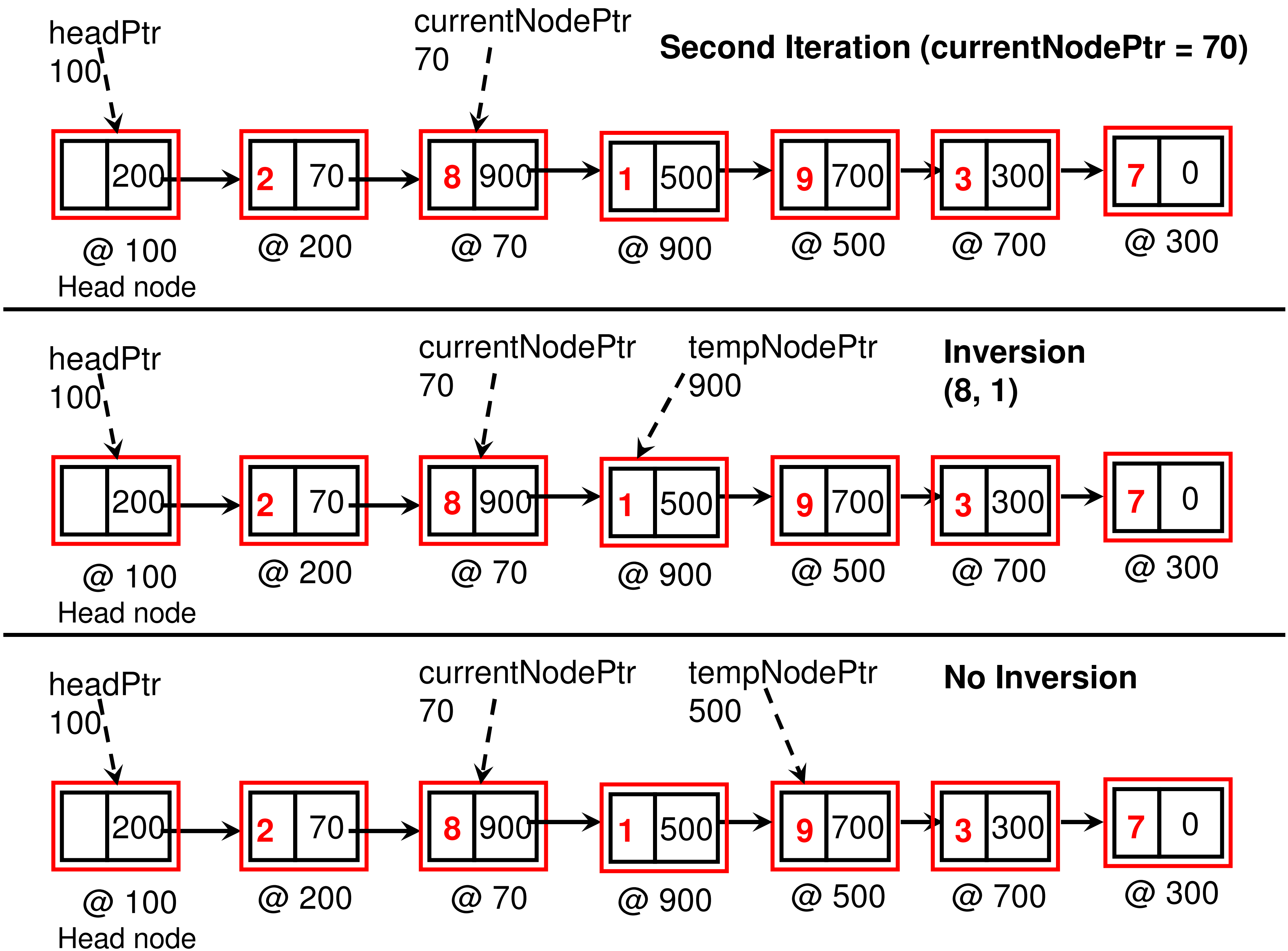


First Iteration (currentNodePtr = 200)
headPtr
100
currentNodePtr
200
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300
No Inversion
headPtr
100
currentNodePtr
200
tempNodePtr
70
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300
Inversion
(2, 1)
headPtr
100
currentNodePtr
200
tempNodePtr
900
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300

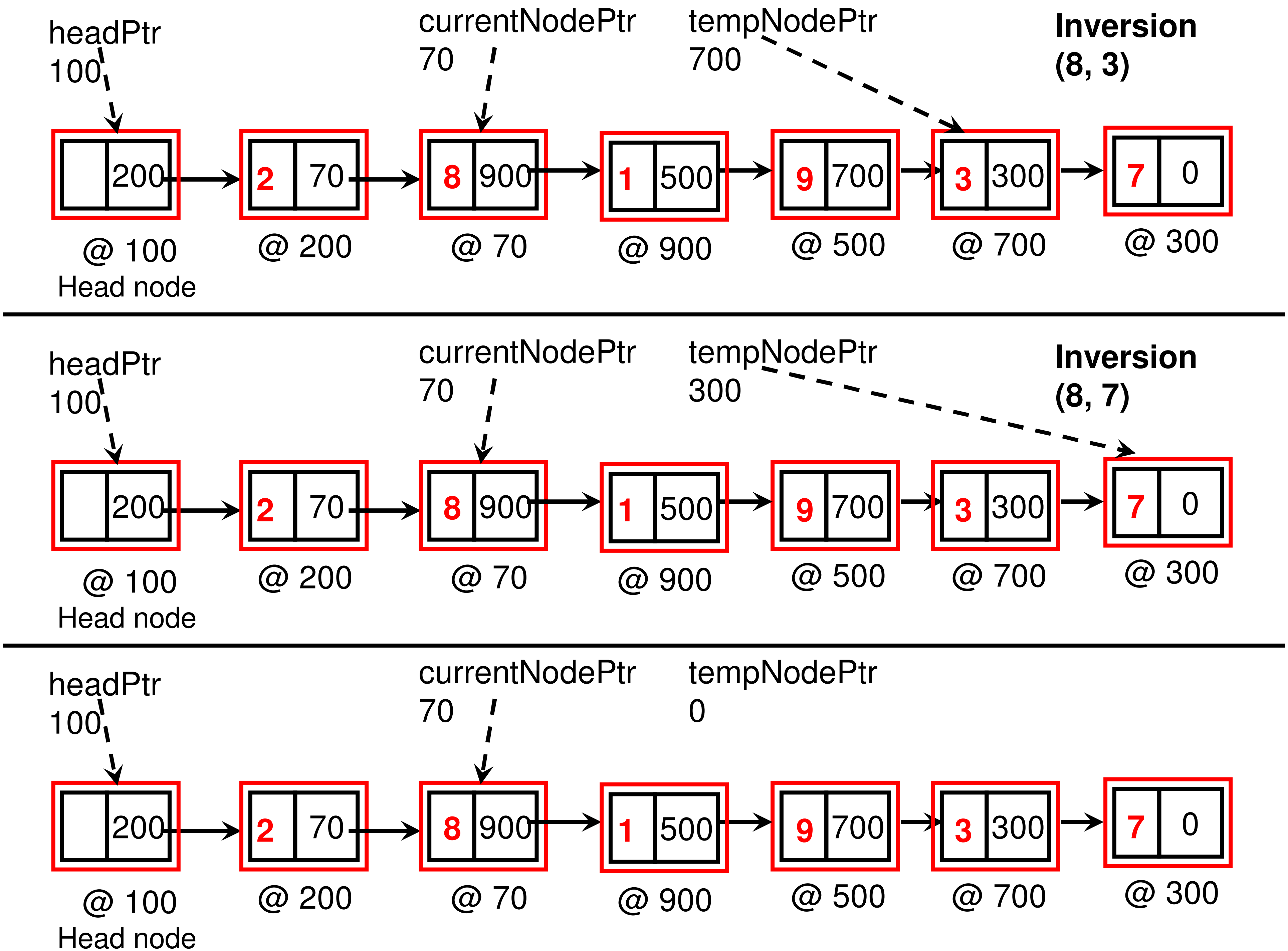


headPtr
100
currentNodePtr
200
tempNodePtr
500
No Inversion
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300
headPtr
100
currentNodePtr
200
tempNodePtr
700
No Inversion
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300
headPtr
100
currentNodePtr
200
tempNodePtr
300
No Inversion
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300


headPtr
100
currentNodePtr
200
tempNodePtr
0
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300



Second Iteration (currentNodePtr = 70)
headPtr
100
currentNodePtr
70
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300
Inversion
(8, 1)
headPtr
100
currentNodePtr
70
tempNodePtr
900
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300
No Inversion
headPtr
100
currentNodePtr
70
tempNodePtr
500
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300


headPtr
100
currentNodePtr
70
tempNodePtr
700
Inversion
(8, 3)
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300
headPtr
100
currentNodePtr
70
tempNodePtr
300
Inversion
(8, 7)
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300
headPtr
100
currentNodePtr
70
tempNodePtr
0
200
2
70
8
900
1
500
9
700
3
300
7
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
@ 300

Inserting at an Appropriate Location Decided at Run time

- Consider the problem of maintaining a list of integers such that it always has the negative integers followed by positive integers.
- Newly input integers are to be inserted on a *last input last insert basis*.
  - i.e., a positive integer is inserted at the end of all the positive integers in the list (which basically corresponds to the end of the list).
  - a negative integer is inserted at the end of all the negative integers that are currently in the list.

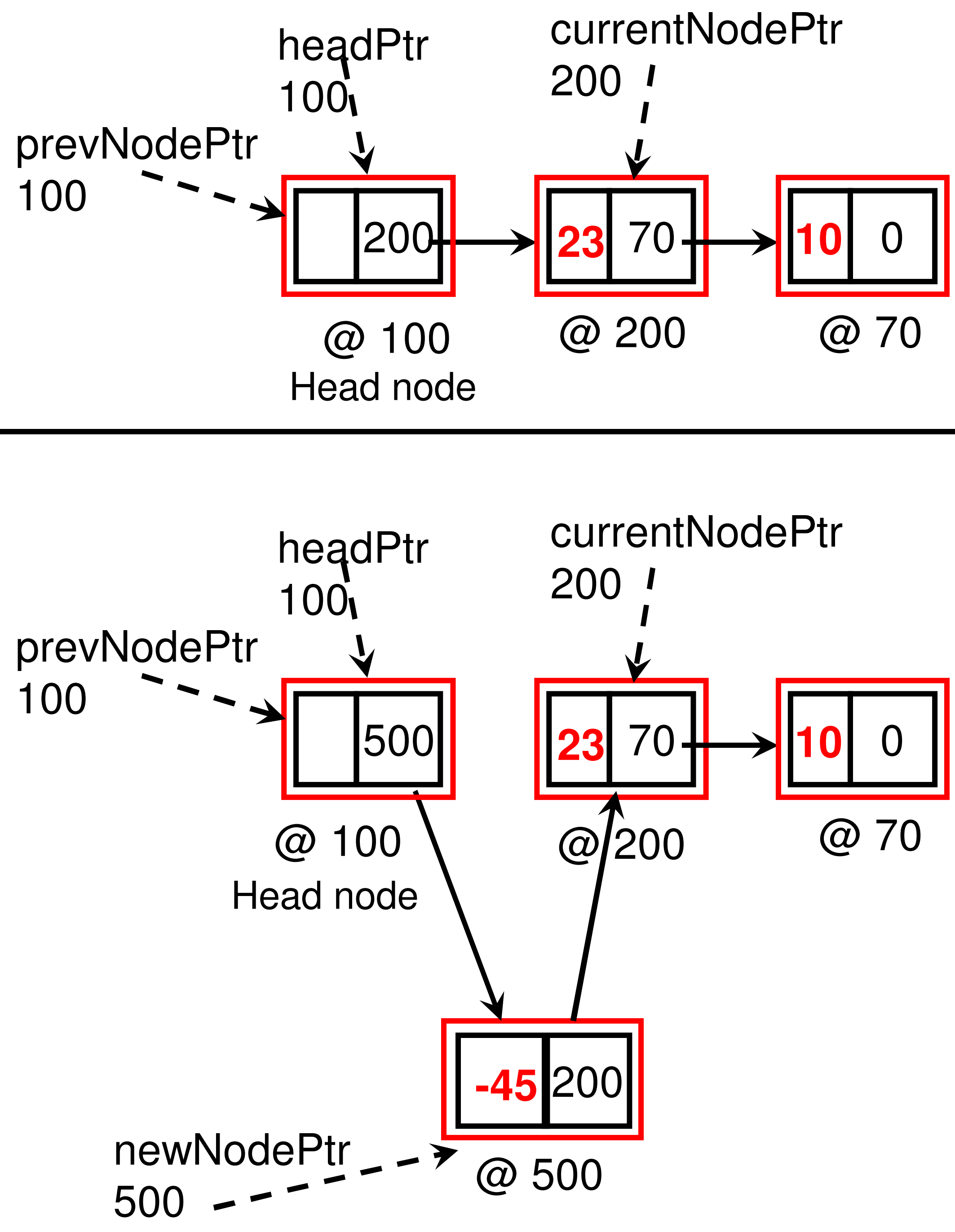
Inserting at an Appropriate Location Decided at Run time

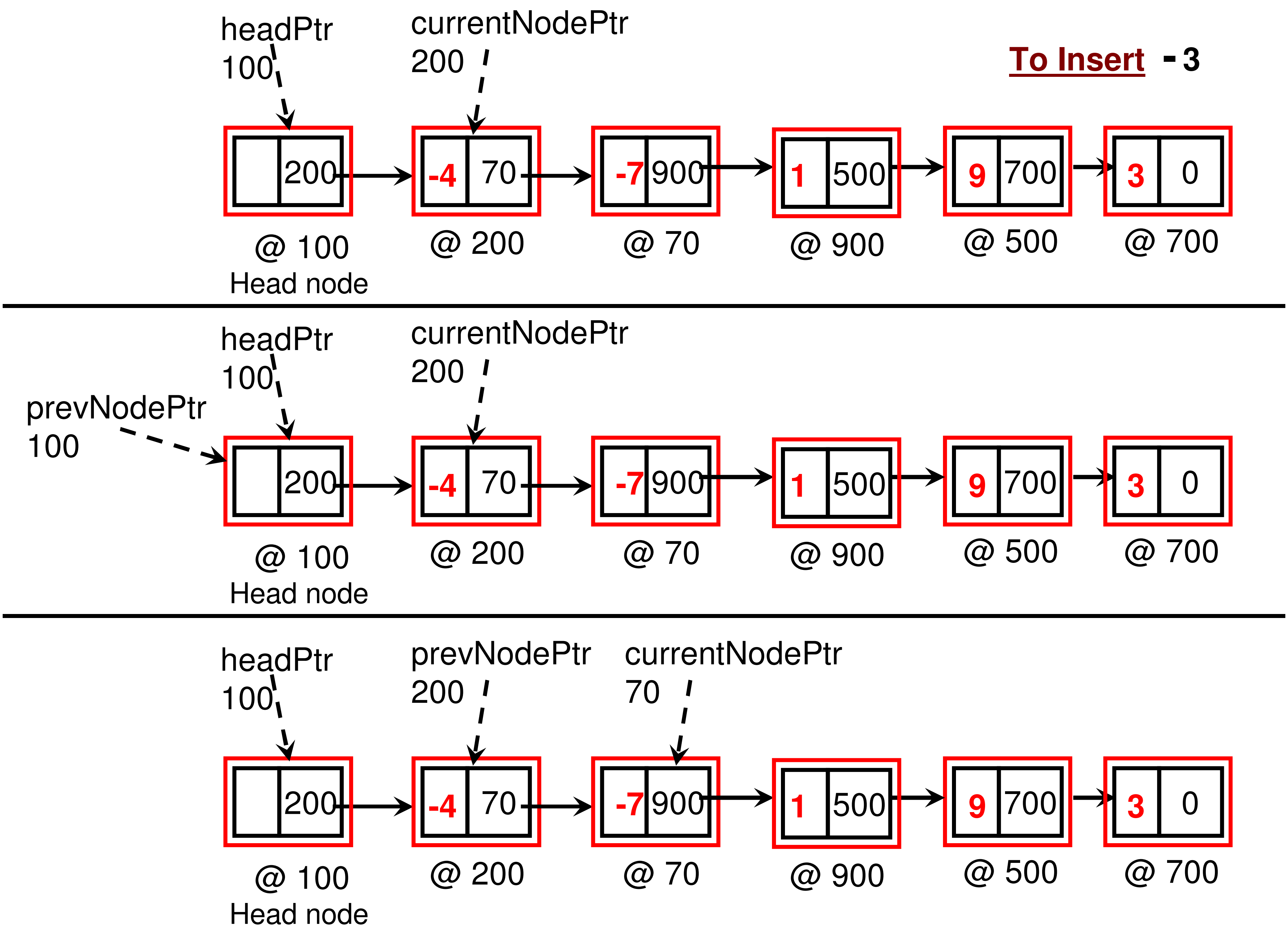
```
Enter the number of elements you want to insert: 10
Enter element # 0 : 23
Contents of the List: 23
Enter element # 1 : 10
Contents of the List: 23 10
Enter element # 2 : -45
Contents of the List: -45 23 10
Enter element # 3 : -78
Contents of the List: -45 -78 23 10
Enter element # 4 : 56
Contents of the List: -45 -78 23 10 56
Enter element # 5 : -11
Contents of the List: -45 -78 -11 23 10 56
Enter element # 6 : -12
Contents of the List: -45 -78 -11 -12 23 10 56
Enter element # 7 : -11
Contents of the List: -45 -78 -11 -12 -11 23 10 56
Enter element # 8 : 0
Contents of the List: -45 -78 -11 -12 -11 23 10 56 0
Enter element # 9 : 4
Contents of the List: -45 -78 -11 -12 -11 23 10 56 0 4
```

Code 9: Inserting at an Appropriate Location Decided at Run time

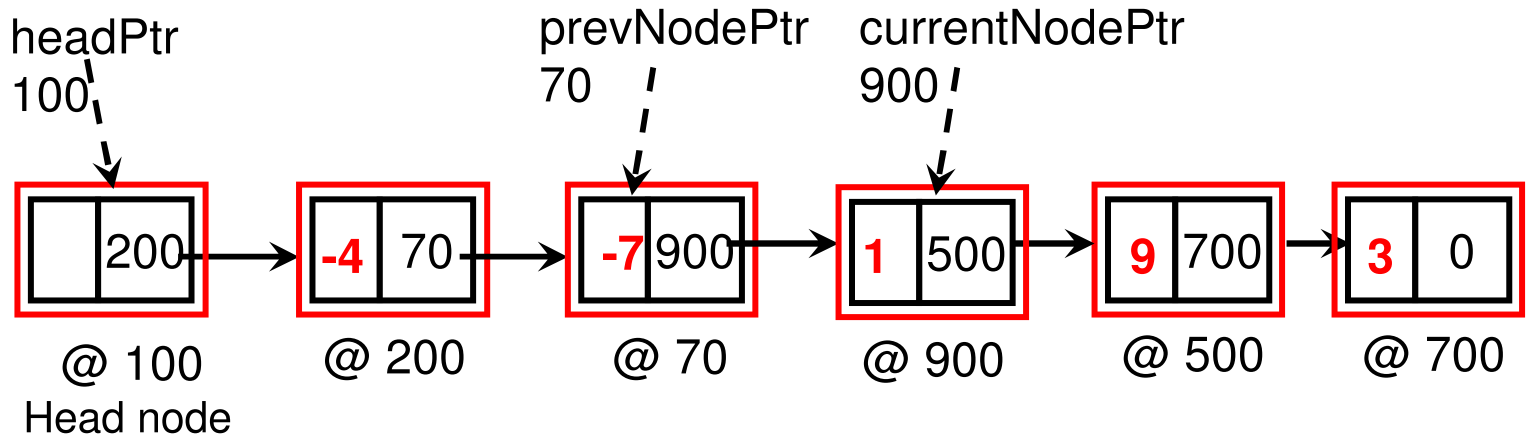
```
void insertElement(int data){

        if (data >= 0){
                insert(data);
                return;
        }

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

        while (currentNodePtr != 0){

                if (currentNodePtr->getData() >= 0){
                        break;
                }

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

        }

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

}
```

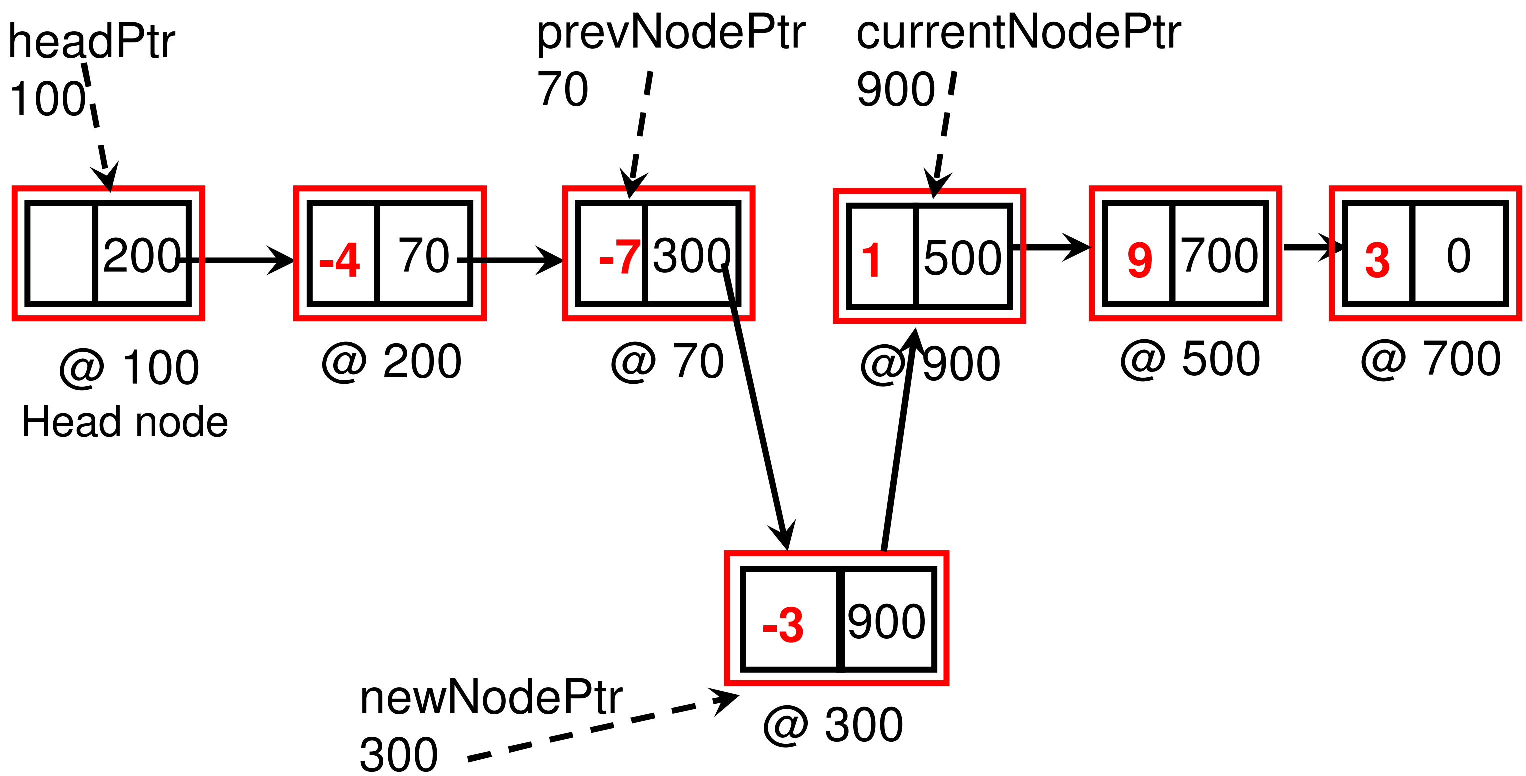
To Insert -45


headPtr
100
currentNodePtr
200
prevNodePtr
100
200
23
70
10
0
@ 100
Head node
@ 200
@ 70
headPtr
100
currentNodePtr
200
prevNodePtr
100
500
23
70
10
0
@ 100
Head node
@ 200
@ 70
-45
200
newNodePtr
500
@ 500



To Insert -3
headPtr
100
currentNodePtr
200
200
-4
70
-7
900
1
500
9
700
3
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
headPtr
100
currentNodePtr
200
prevNodePtr
100
200
-4
70
-7
900
1
500
9
700
3
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
headPtr
100
prevNodePtr
200
currentNodePtr
70
200
-4
70
-7
900
1
500
9
700
3
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700


headPtr
100
prevNodePtr
70
currentNodePtr
900
200
-4
70
-7
900
1
500
9
700
3
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700


headPtr
100
prevNodePtr
70
currentNodePtr
900
200
-4
70
-7
300
1
500
9
700
3
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
-3
900
newNodePtr
300
@ 300

Sorting Algorithm: Selection Sort

- Given an array A[0…n-1], we proceed for a total of n-1 iterations
- In iteration i ($0 \le i \le n-2$), we initially assume i to be the index (minIndex) where the minimum element is. We compare the value of the element at minIndex with those at indexes i+1 to n-1 and update minIndex accordingly (i.e., if any index has an element with further lower value). At the end of the ith iteration, we swap the element at minIndex with the element at index i.

```
Algorithm Selection Sort
// Input: An array A[0...n-1] of orderable elements
// Output: Array A[0...n-1] sorted in non-decreasing order

for (index i = 0 to n-2) do

    minIndex = i

    for (index j = i+1 to n-1) do

        if (A[j] < A[minIndex])
            minIndex = j
        end if

    end for

    swap A[i] and A[minIndex]

end for
```

 # Comparisons
$(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = \Theta(n^2)$

There is no best or worst case. In the ith Iteration, we have to find if there exists any element that is less than the element at index i.

	0	1	2	3	4	5	6	7	8	9
Given Array	5	6	5	4	3	10	9	1	7	8
Iteration 0	5	6	5	4	3	10	9	1	7	8
Iteration 0 (After)	1	6	5	4	3	10	9	5	7	8
Iteration 1	1	6	5	4	3	10	9	5	7	8
Iteration 1 (After)	1	3	5	4	6	10	9	5	7	8
Iteration 2	1	3	5	4	6	10	9	5	7	8
Iteration 2 (After)	1	3	4	5	6	10	9	5	7	8

Iteration 3

0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	10	9	5	7	8

Iteration 3 (After)

0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	10	9	5	7	8

Iteration 4

0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	10	9	5	7	8

Iteration 4 (After)

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	10	9	6	7	8

Iteration 5

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	10	9	6	7	8

Iteration 5 (After)

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	9	10	7	8

Iteration 6

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	9	10	7	8

Iteration 6 (After)

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	7	10	9	8

Iteration 7

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	7	10	9	8

Iteration 7 (After)

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	7	8	9	10

Iteration 8

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	7	8	9	10

Iteration 8 (After)

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	7	8	9	10

Final Sorted Array

0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	7	8	9	10

Code 7

Find the address of the node with the minimum data

```
Node* findMinimumDataNodeAddress(List list){

    Node* headPtr = list.getHeadPtr();
    Node* currentNodePtr = headPtr->getNextNodePtr();

    Node* minDataNodePtr;

    if (currentNodePtr != 0){
        minDataNodePtr = currentNodePtr;
        currentNodePtr = currentNodePtr->getNextNodePtr();
    }
    else{   // return '0' (null) if the list is empty
        return 0;
    }

    while (currentNodePtr != 0){

        if (minDataNodePtr->getData() > currentNodePtr->getData())
            minDataNodePtr = currentNodePtr;

        currentNodePtr = currentNodePtr->getNextNodePtr();

    }

    return minDataNodePtr;

}
```

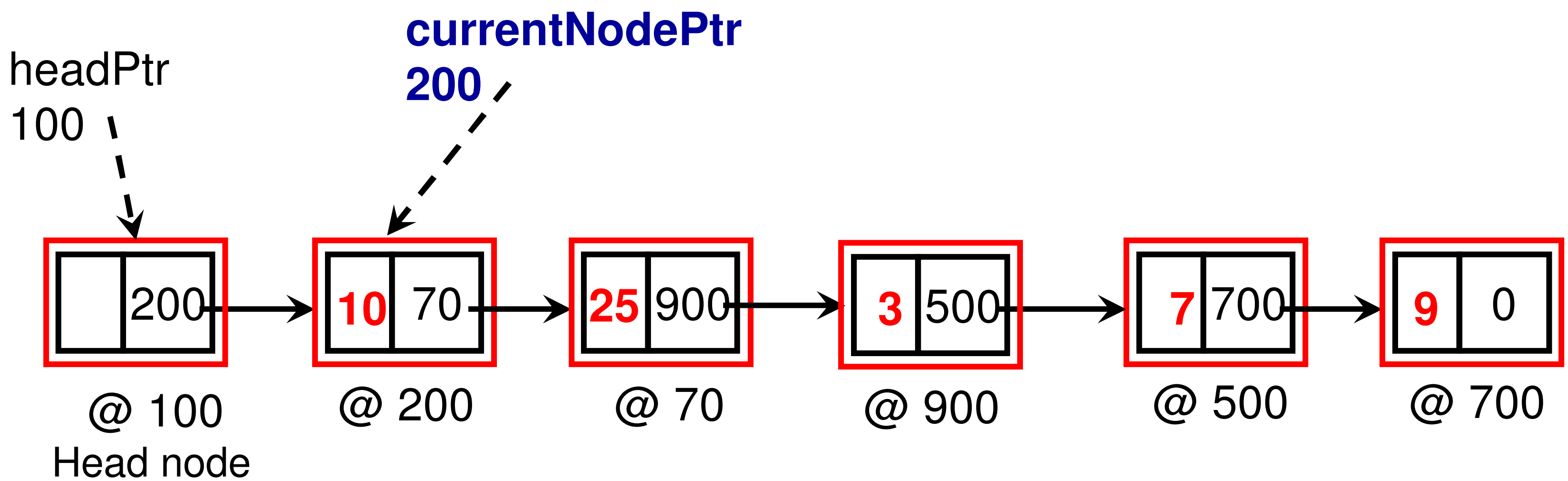
/* Assigns the address of the first data node as the initial value of minDataNodePtr */

/* Inside this loop, minDataNodePtr will be set to the address of the node (if any exists) whose data is less than the data of the node whose address is stored in minDataNodePtr */

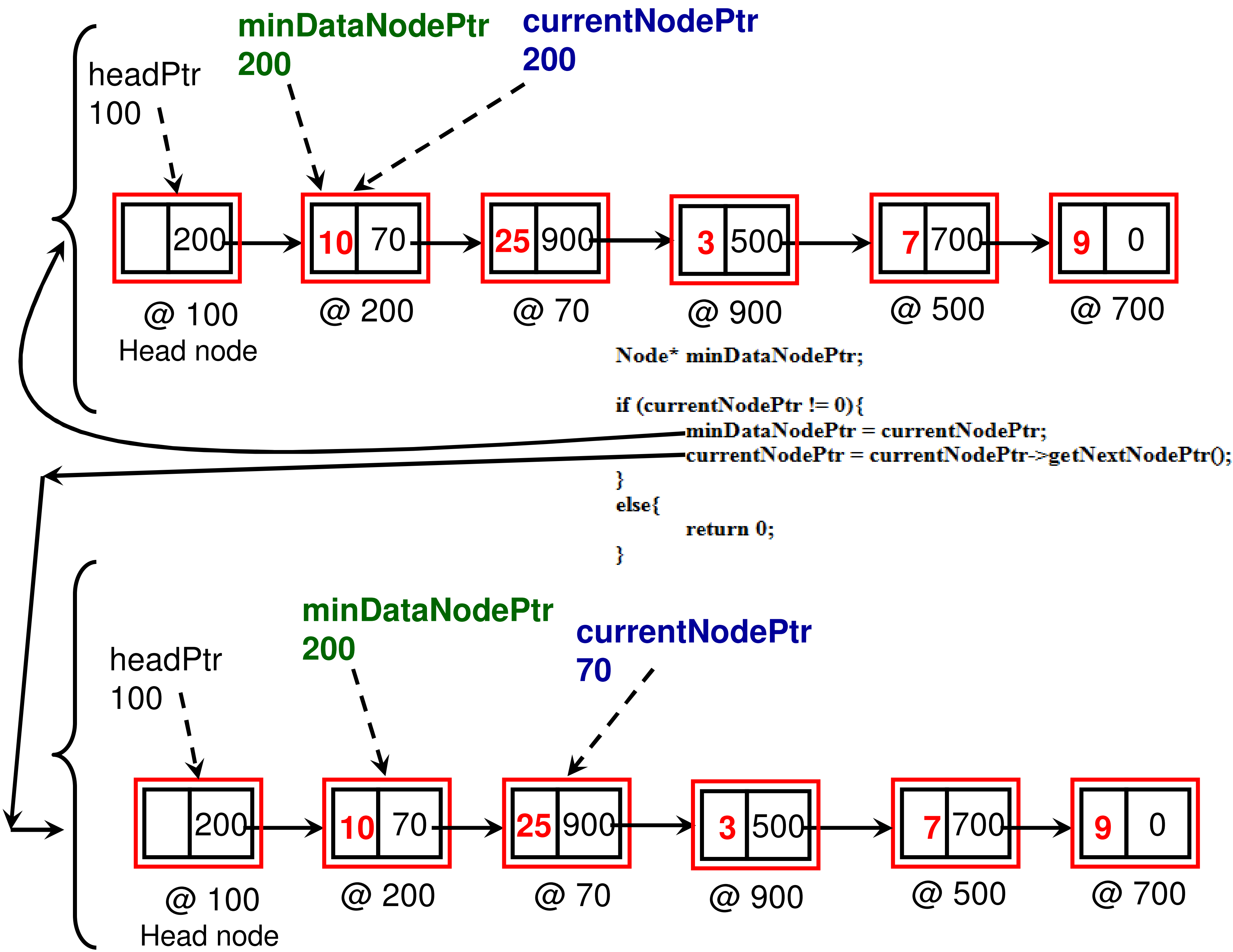
```
Node* MinDataNodePtr = findMinimumDataNodeAddress(integerList);
if (MinDataNodePtr != 0)
    cout << "Minimum data is: " << MinDataNodePtr->getData() << endl;
else
    cout << "The list is empty!!" << endl;
```

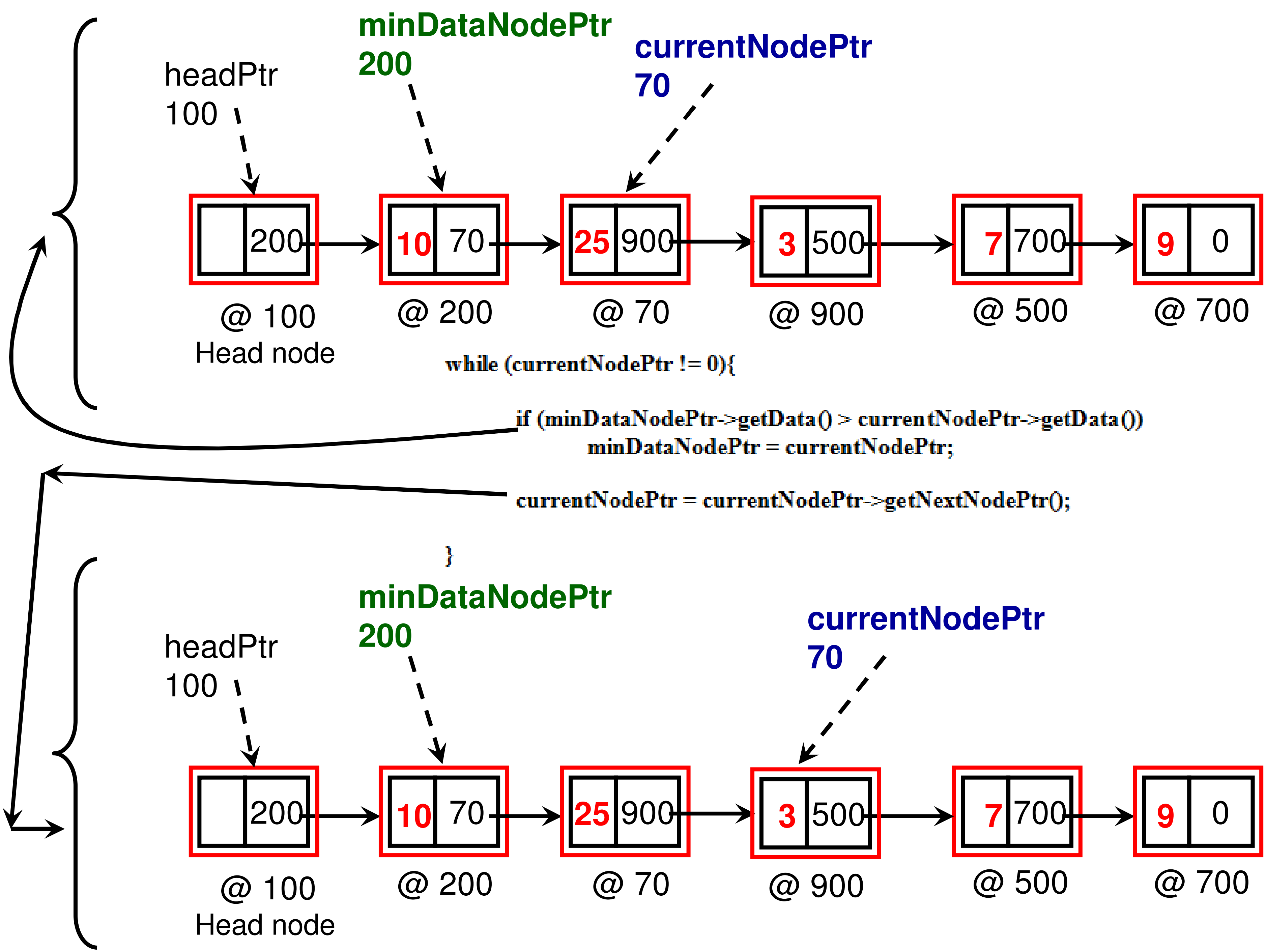
Find the address of the node with the minimum data

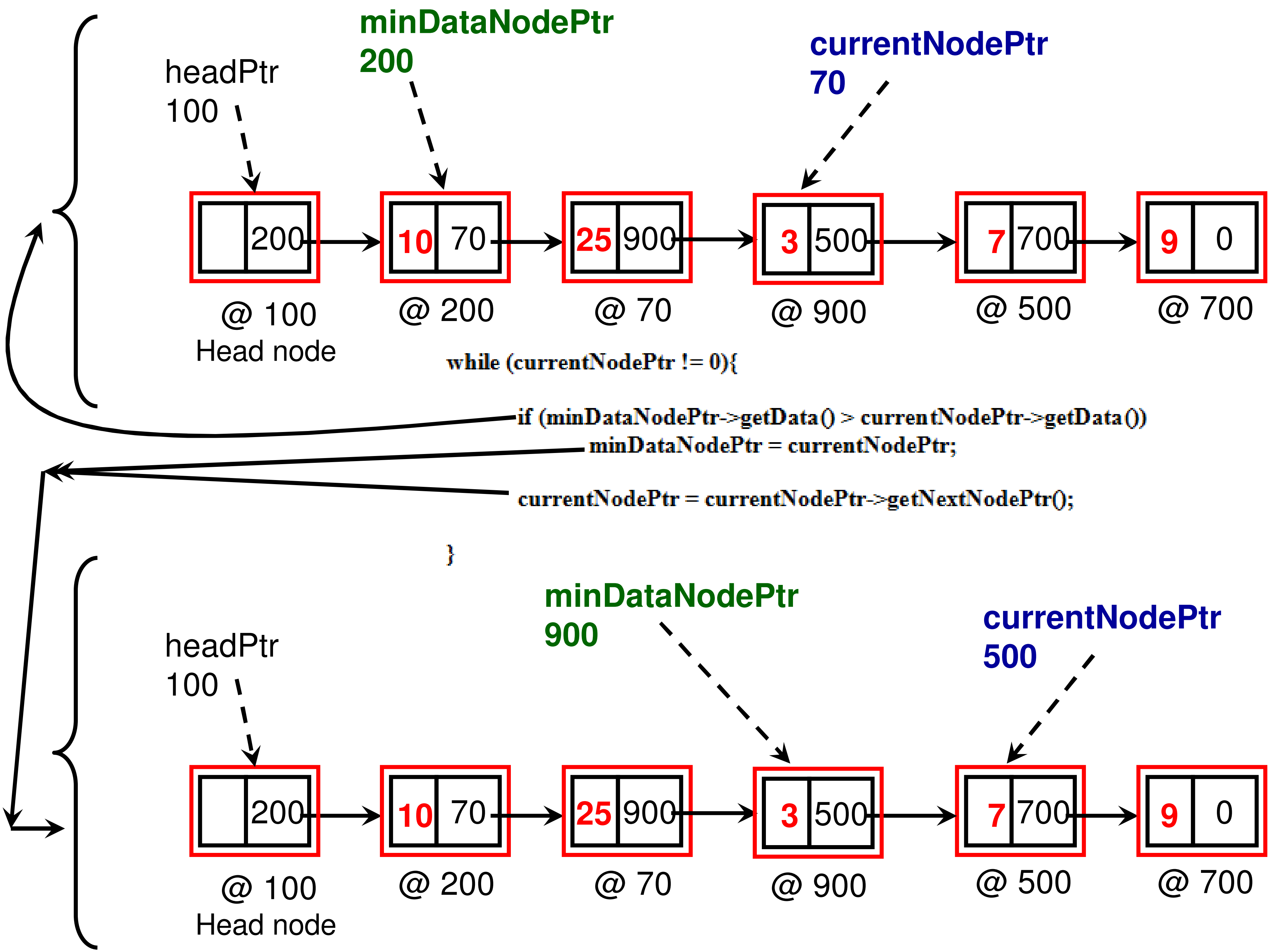
```
Node* headPtr = list.getHeadPtr();
Node* currentNodePtr = headPtr->getNextNodePtr();
```



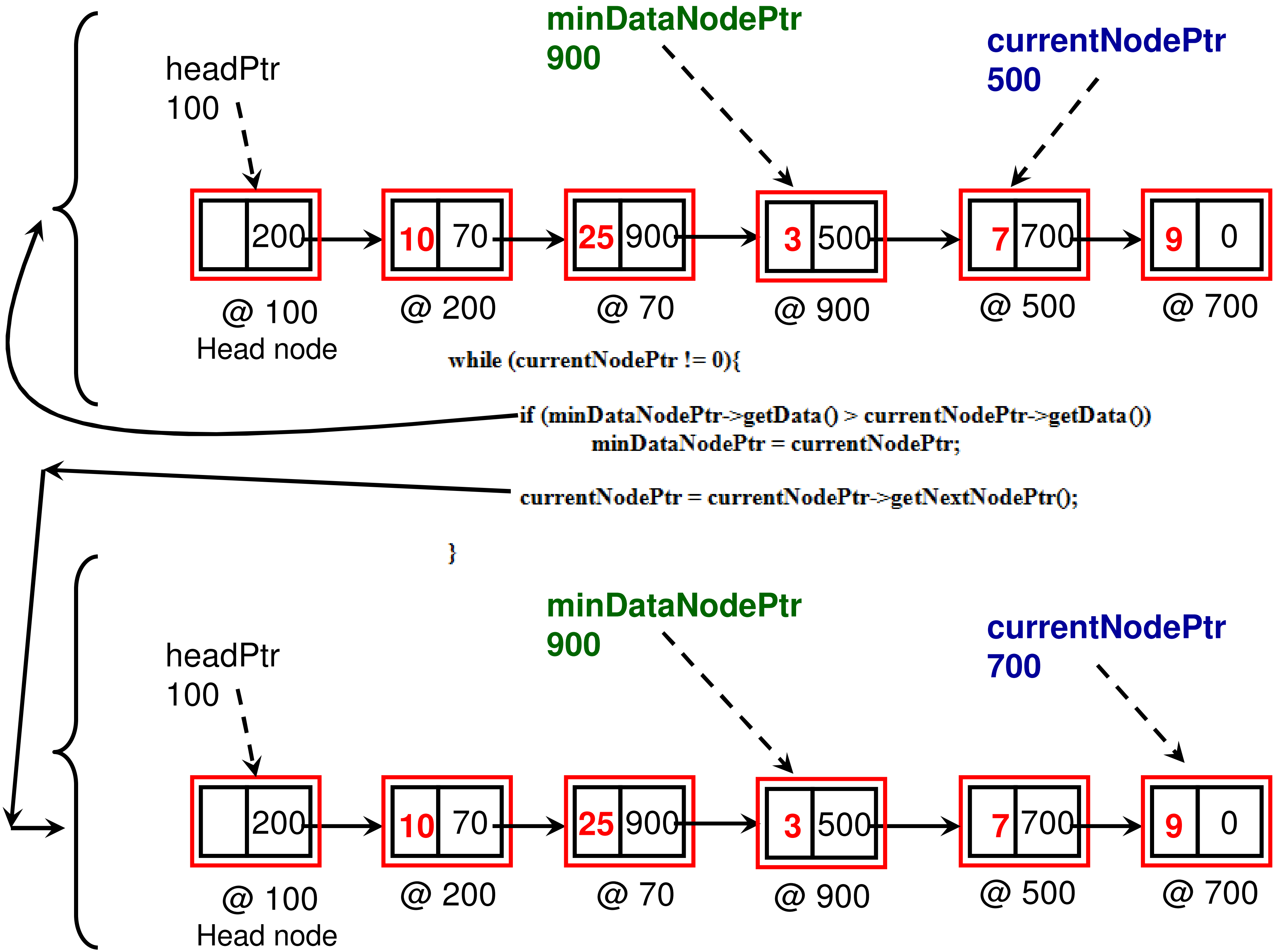

```
Node* minDataNodePtr;

if (currentNodePtr != 0){
        minDataNodePtr = currentNodePtr;
        currentNodePtr = currentNodePtr->getNextNodePtr();
}
else{
        return 0;
}
```

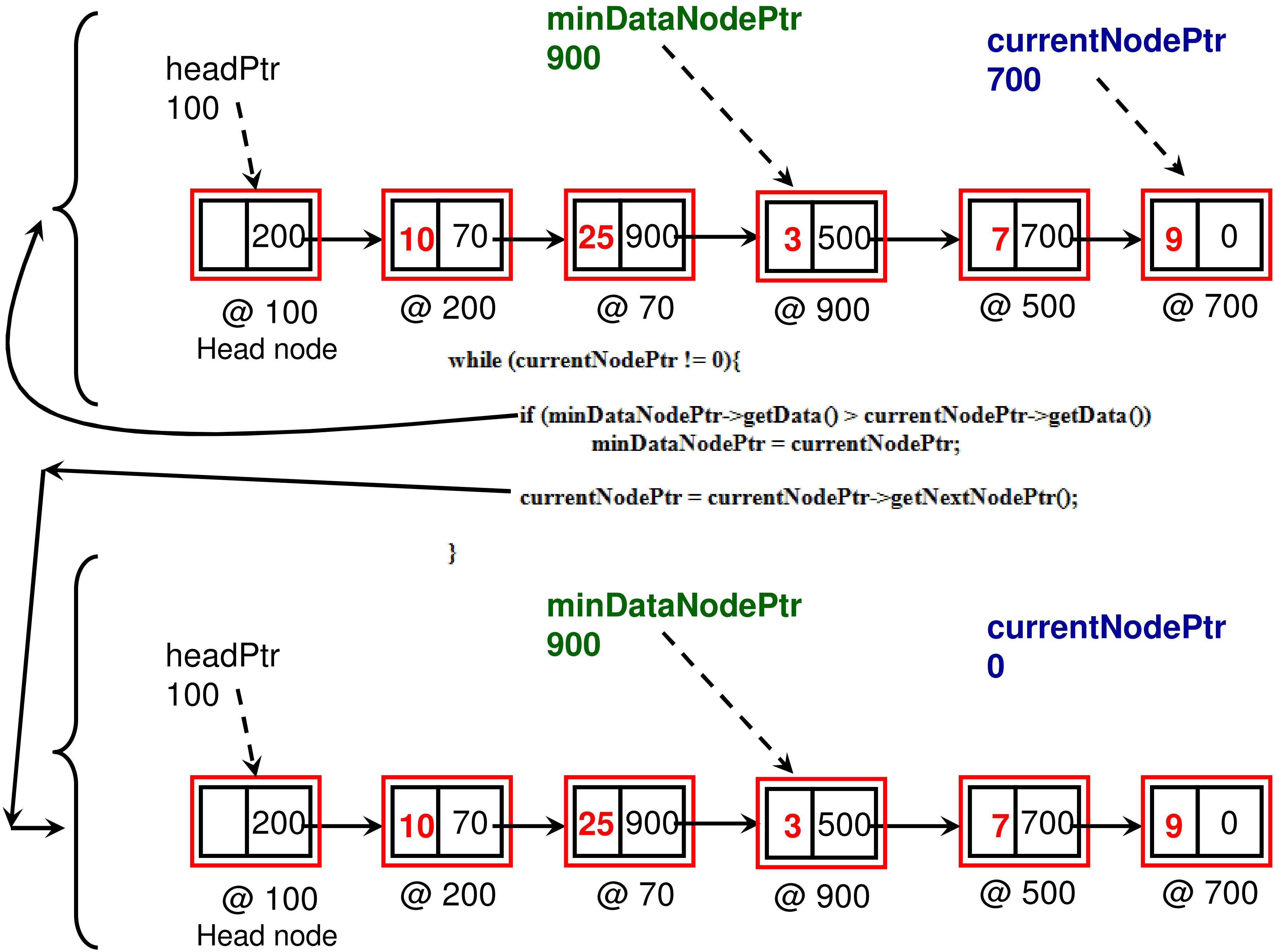

minDataNodePtr
200
currentNodePtr
200
headPtr
100
200
10
70
25
900
3
500
7
700
9
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
Node* minDataNodePtr;
if (currentNodePtr != 0){
minDataNodePtr = currentNodePtr;
currentNodePtr = currentNodePtr->getNextNodePtr();
}
else{
return 0;
}
minDataNodePtr
200
currentNodePtr
70
headPtr
100
200
10
70
25
900
3
500
7
700
9
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700


headPtr
100
minDataNodePtr
200
currentNodePtr
70
200
10
70
25
900
3
500
7
700
9
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
while (currentNodePtr != 0){
if (minDataNodePtr->getData() > currentNodePtr->getData())
minDataNodePtr = currentNodePtr;
currentNodePtr = currentNodePtr->getNextNodePtr();
}
headPtr
100
minDataNodePtr
200
currentNodePtr
70
200
10
70
25
900
3
500
7
700
9
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700

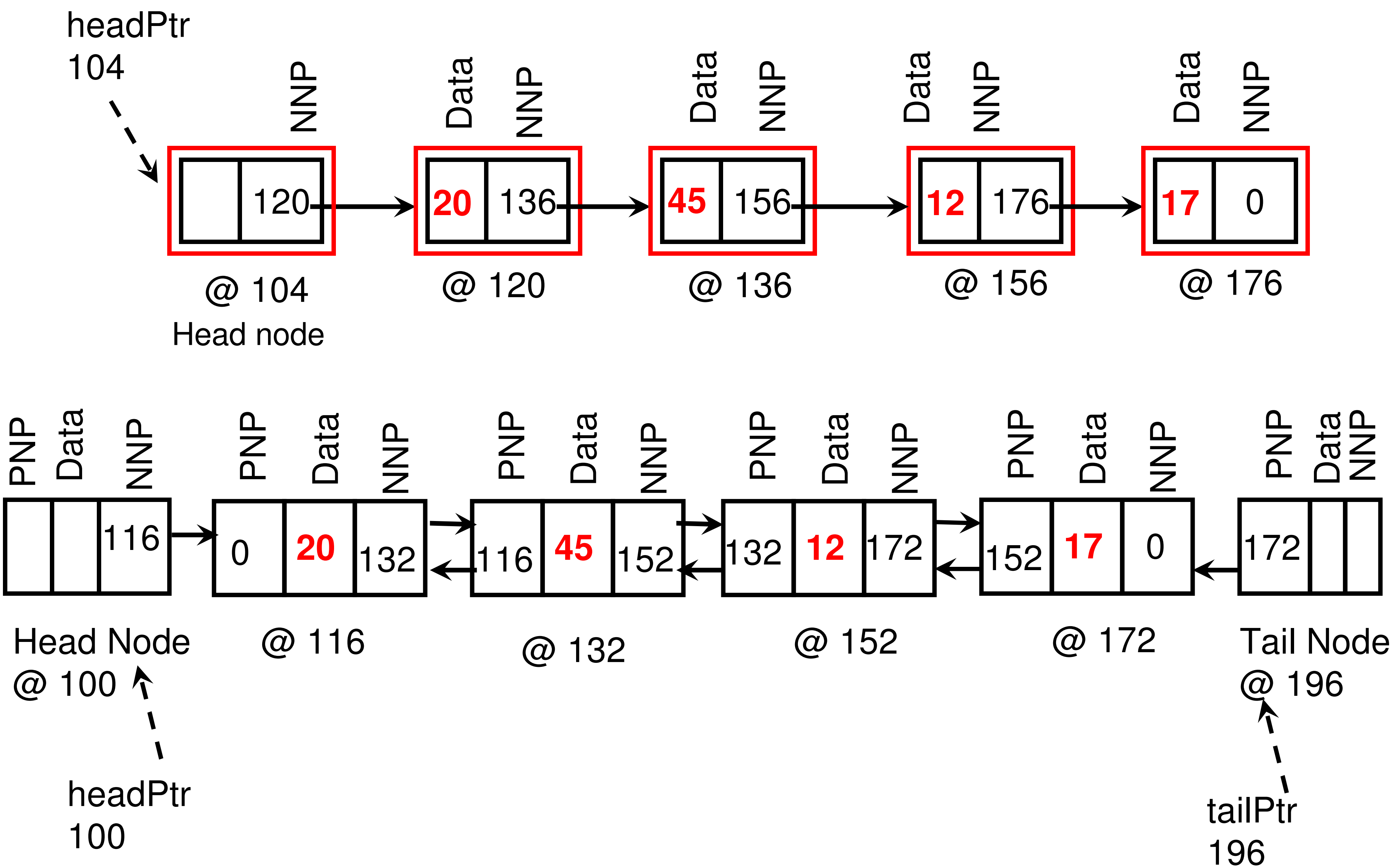


minDataNodePtr
200
currentNodePtr
70
headPtr
100
200
10
70
25
900
3
500
7
700
9
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
while (currentNodePtr != 0){
if (minDataNodePtr->getData() > currentNodePtr->getData())
minDataNodePtr = currentNodePtr;
currentNodePtr = currentNodePtr->getNextNodePtr();
}
minDataNodePtr
900
currentNodePtr
500
headPtr
100
200
10
70
25
900
3
500
7
700
9
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700


minDataNodePtr
900
currentNodePtr
500
headPtr
100
200
10
70
25
900
3
500
7
700
9
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
while (currentNodePtr != 0){
if (minDataNodePtr->getData() > currentNodePtr->getData())
minDataNodePtr = currentNodePtr;
currentNodePtr = currentNodePtr->getNextNodePtr();
}
minDataNodePtr
900
currentNodePtr
700
headPtr
100
200
10
70
25
900
3
500
7
700
9
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700


minDataNodePtr
900
currentNodePtr
700
headPtr
100
200
10
70
25
900
3
500
7
700
9
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700
while (currentNodePtr != 0){
if (minDataNodePtr->getData() > currentNodePtr->getData())
minDataNodePtr = currentNodePtr;
currentNodePtr = currentNodePtr->getNextNodePtr();
}
minDataNodePtr
900
currentNodePtr
0
headPtr
100
200
10
70
25
900
3
500
7
700
9
0
@ 100
Head node
@ 200
@ 70
@ 900
@ 500
@ 700

Singly vs. Doubly Linked List



Singly vs. Doubly Linked List

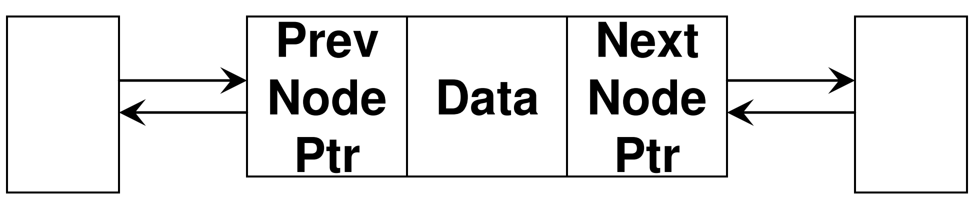
- A doubly linked list has two additional nodes: a head node and tail node
- A doubly linked list could be traversed in either direction (from head node or from tail node).
  - nextNodePtr values at the nodes are used to access in the forward direction (from head node)
  - prevNodePtr values at the nodes are used to access in the reverse direction (from the tail node)
- In the forward direction: The headPtr points to the head node whose next node is the first data node in the list and the node previous to the tail node is the last data node whose nextNodePtr is set to 0 (null).
- In the reverse direction: The tailPtr points to the tail node whose previous node is the first data node and the node next to the head node is the last data node whose prevNodePtr is set to 0 (null).

Class Node
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;
        }
```

Initialization of a Doubly Linked List