Linked List: Basic Operations

- The basic operations of a linked list are as follows:
  - Search the list for a particular item
  - Insert an item in the list
  - Delete an item from the list

- These operations require the list to be traversed. That is, given a pointer to the first node of the list, we must step through the nodes of the list.

Linked List Class

```
class LinkedList {
   private:
      Node* head;
   public:
      LinkedList(); /* Constructor */
      ~LinkedList(); /* Destructor */

      void traverse();
      Node* search(int val);
      void insertNode(int leftValue, int value);
      void deleteNode(int value);
};
```

Linked List: Traverse a List

- We always want **head** to point to the first node in the list. Why? Because if we use head to traverse the list, we would lose the nodes of the list. The problem occurs because the links are in only one direction. The pointer **head** contains the address of the first node, which contains the address of the second node, and so on. If we move **head** to the second node, the first node is lose (unless we save a pointer to this node). If we keep advancing **head** to the next node, we will lose all of the nodes of the list (unless we save a pointer to each node before advancing **head**, which is impractical due to computer time and memory).

Linked List: Traverse a List

```
current = head;

while (current != nullptr)
{
   cout << current->data << “ ”;
   current = current->next;
}
// or use the following method
for (current = head; current != 0; current = current->next)
{
   cout << current->data << “->”;
}
```

Linked List Class

```
class LinkedList {
   private:
      Node* head;
   public:
      LinkedList(); /* Constructor */
      ~LinkedList(); /* Destructor */

      void traverse();
      Node* search(int val);
      void insertNode(int leftValue, int value);
      void deleteNode(int value);
};
```

Linked List: Search a List

```
Node* LinkedList::search(int val) {
    Node* current = head;
    while (current != 0) {
        if (current->data == val)
            return current;
        current = current->next;
    }
    return 0;
}
```

Linked List: Search a List

```
Node* LinkedList::search(int val) {
    Node* current = head;
    while (current != nullptr &&
           current->data != val)
        current = current->next;
    }
    return current;
}
```

Linked List Class

```
class LinkedList {
   private:
      Node* head;
   public:
      LinkedList(); /* Constructor */
      ~LinkedList(); /* Destructor */

      void traverse();
      Node* search(int val);
      void insertNode(int leftValue, int value);
      void deleteNode(int value);
};
```

Linked List: Inserting a Node

- 3 ways to insert a node
  1. At the front of the linked list
  2. After a given node
  3. At the end of the linked list

Linked List: Inserting a Node

– Note: There are various implementations!

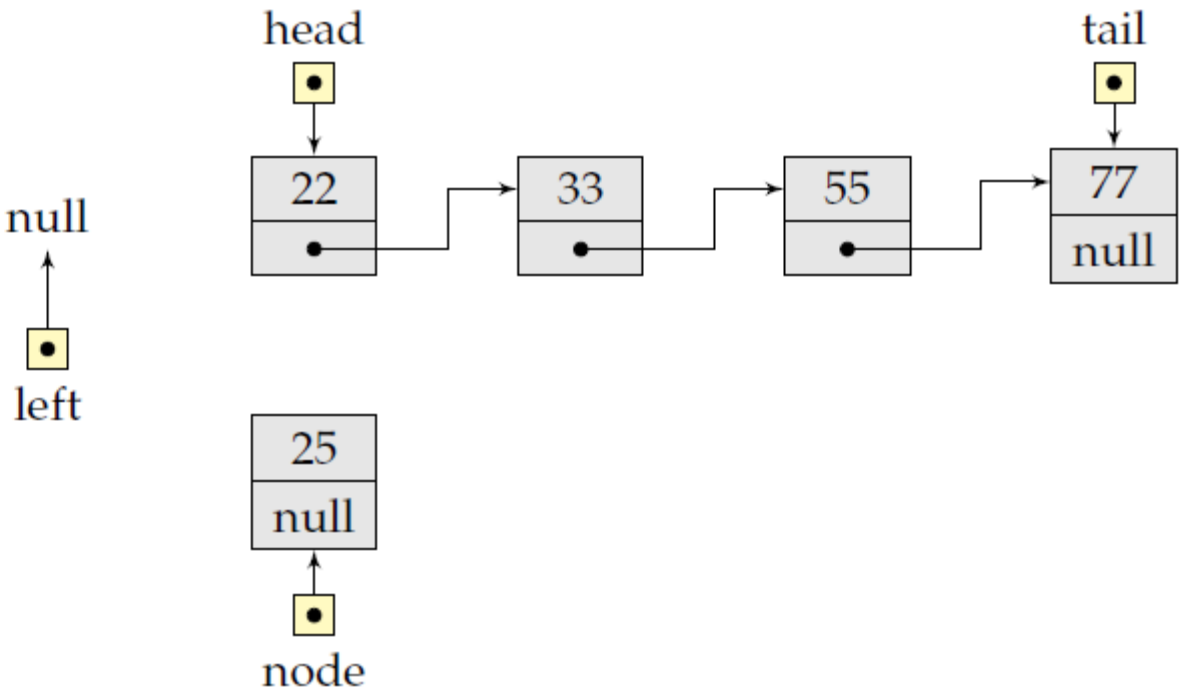
– Insert the given “value” in the list immediately to the right of the first node having value equal to “leftValue”.

– If there is no node in the list with value equal to “leftValue” then insert the node at the head of the list.

```
int insertNode(int leftValue, int value)
{
    Node *node = new Node(value);
    Node *left = search(leftValue);
    …
}
```

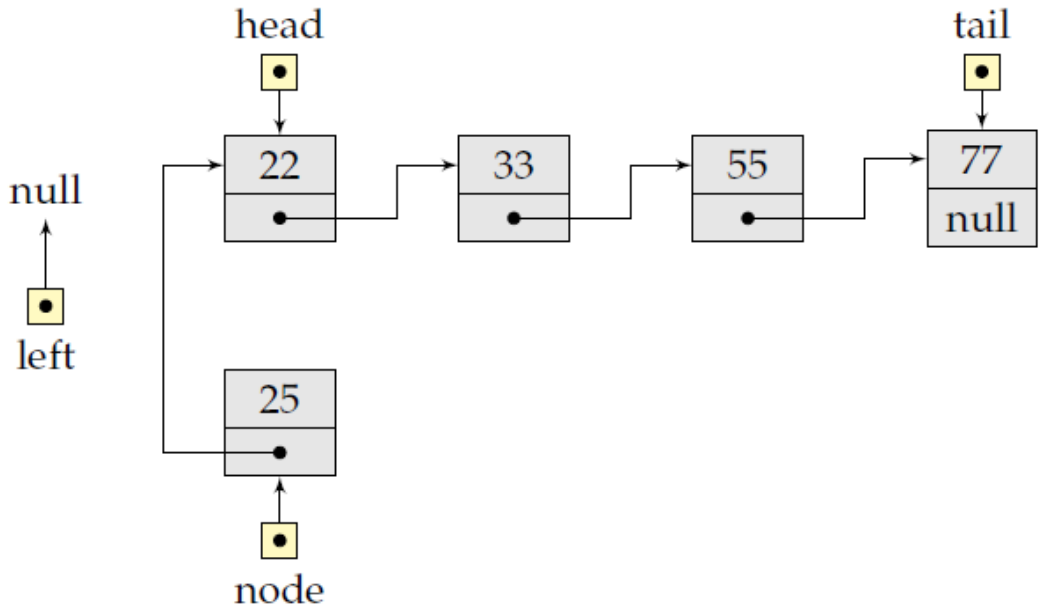
Linked List: Inserting a Node

Case 1: `(left == 0)`



Linked List: Inserting a Node

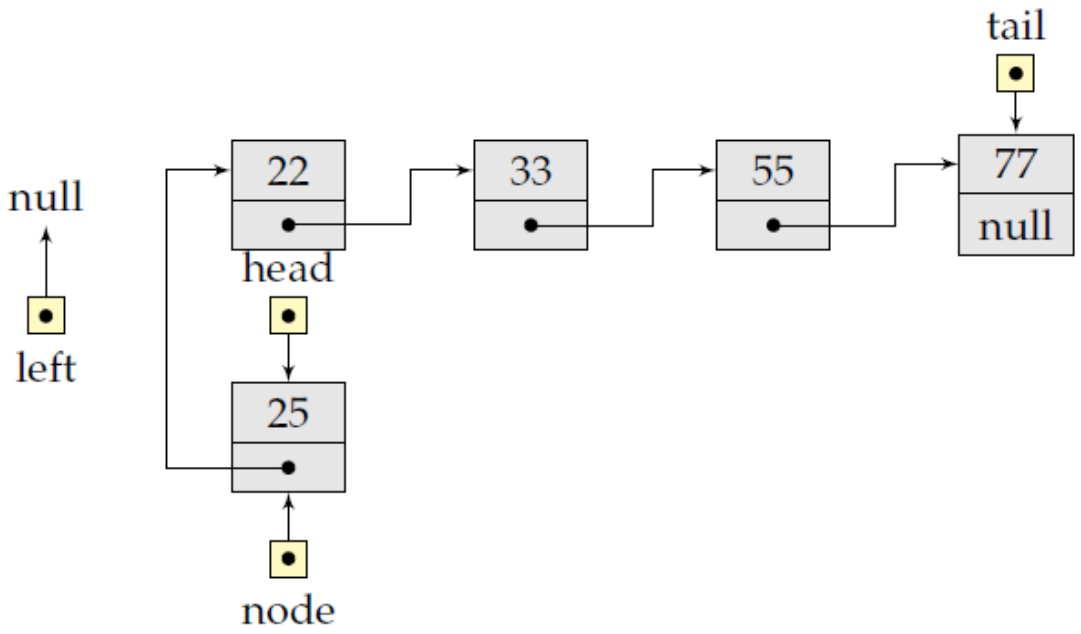
Case 1: `(left == 0)`



1. `node->next = head;`
2. `head = node;`

Linked List: Inserting a Node

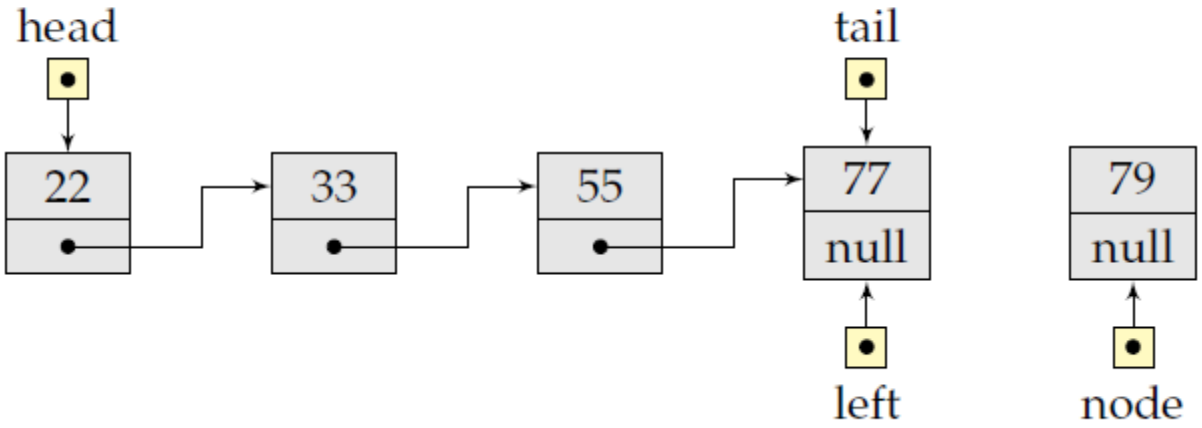
Case 1: `(left == 0)`



1. `node->next = head;`
2. `head = node;`

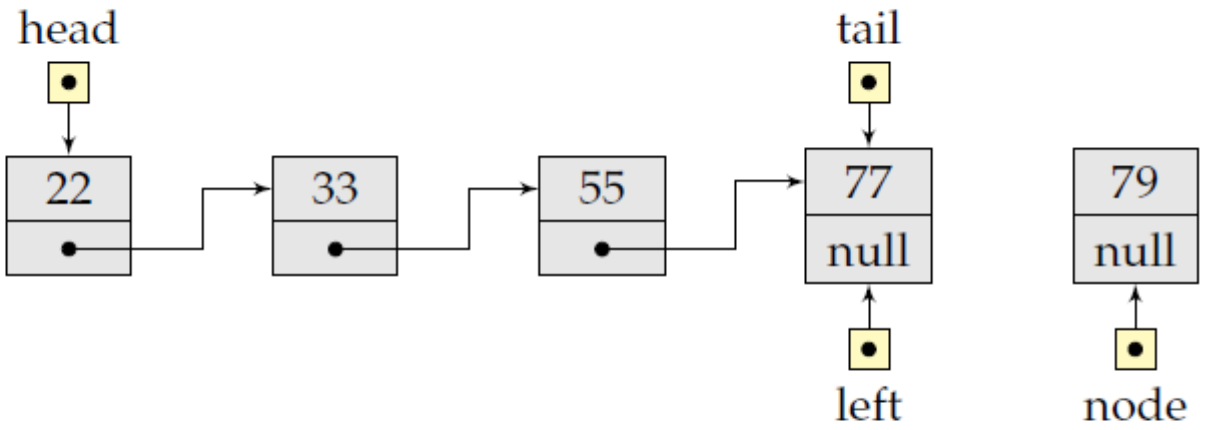
Linked List: Inserting a Node

Case 2: `(left->next == 0)`



Linked List: Inserting a Node

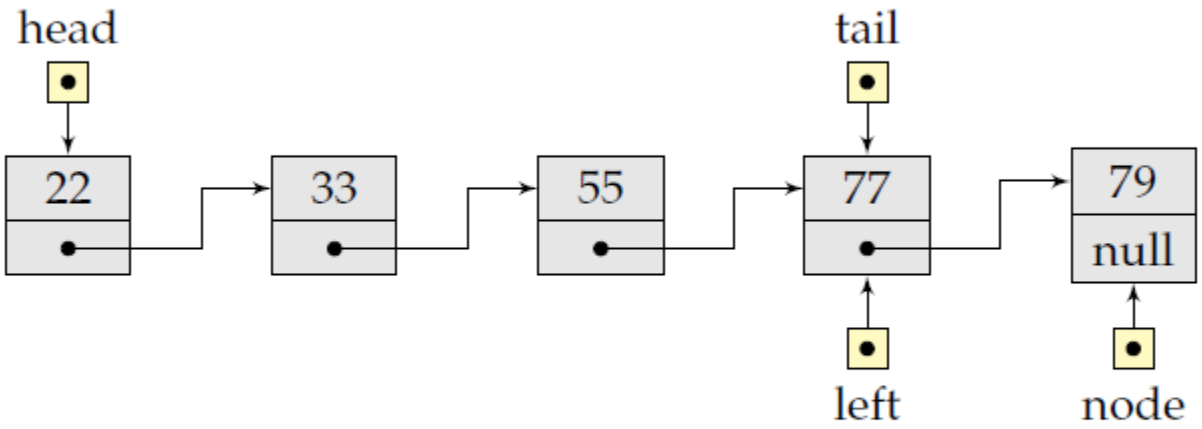
Case 2: `(left->next == 0)`



1. `tail->next = node;`
2. `tail = node;`

Linked List: Inserting a Node

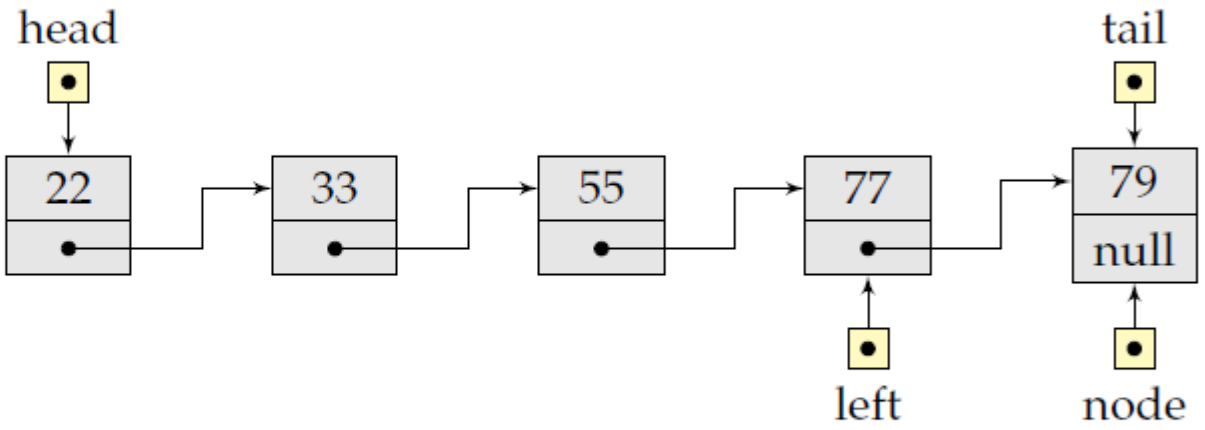
Case 2: `(left->next == 0)`



1. `tail->next = node;`
2. `tail = node;`

Linked List: Inserting a Node

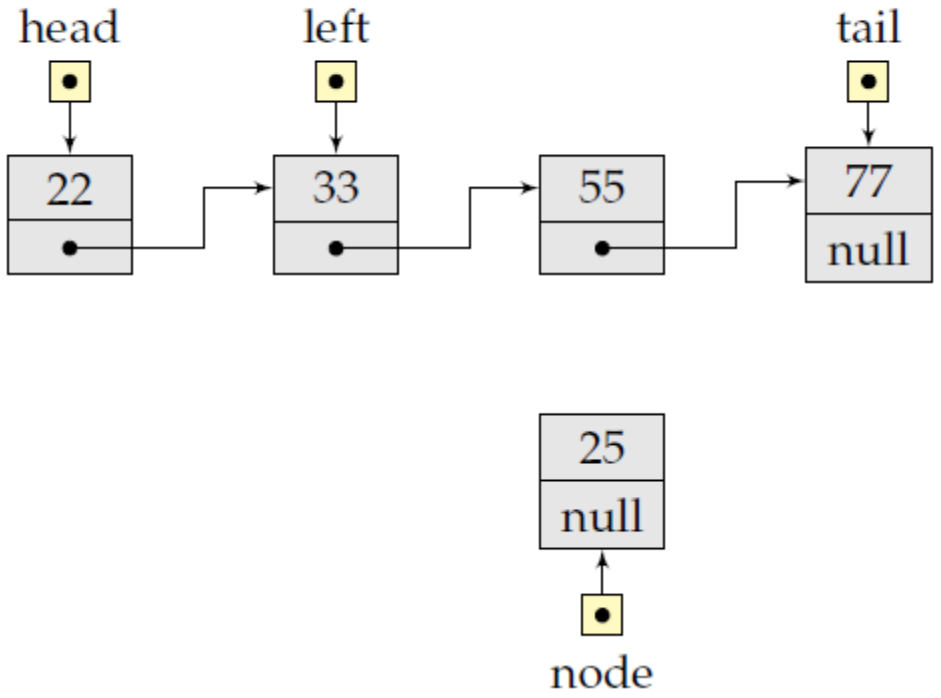
Case 2: `(left->next == 0)`



1. `tail->next = node;`
2. `tail = node;`

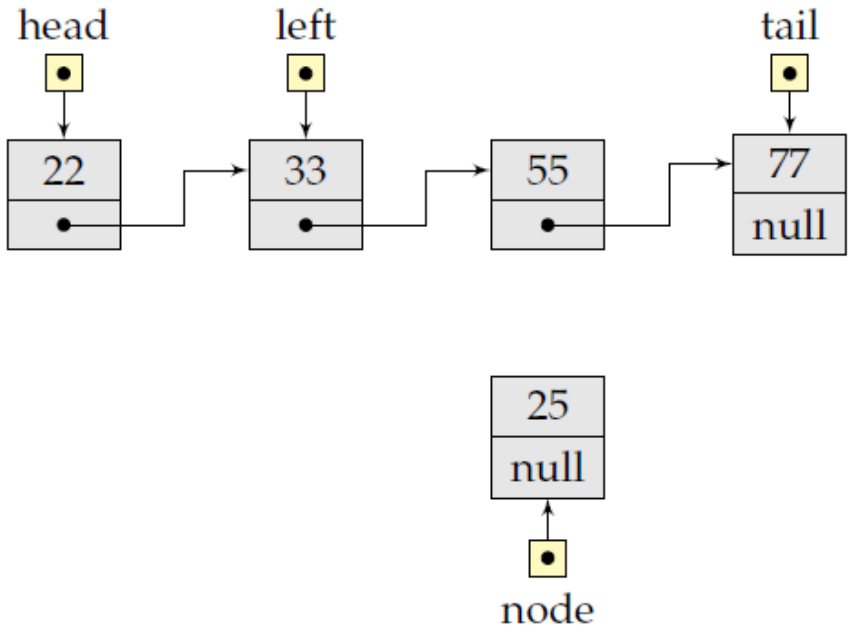
Linked List: Inserting a Node

Case 3: Otherwise:



Linked List: Inserting a Node

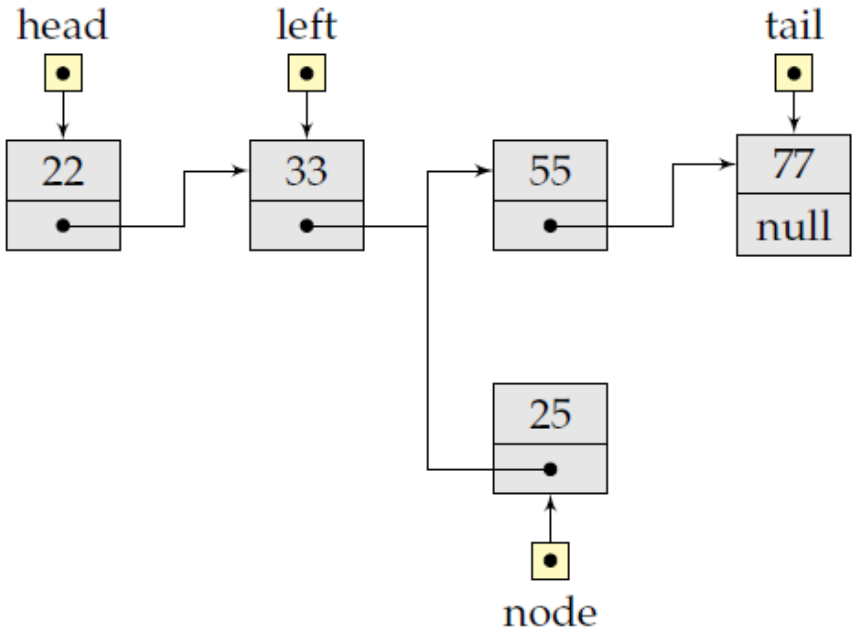
Case 3: Otherwise:



1. `node->next = left->next;`
2. `left->next = node;`

Linked List: Inserting a Node

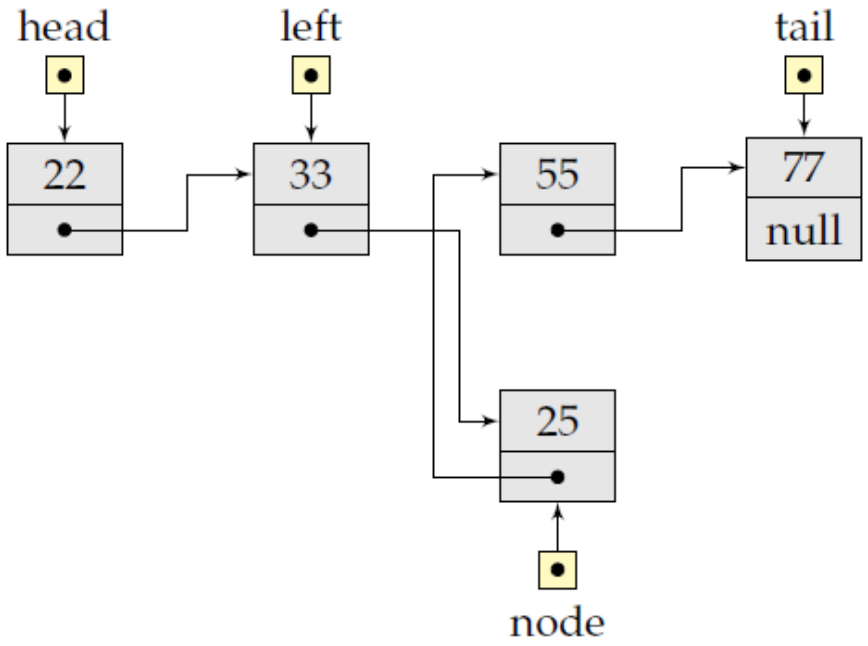
Case 3: Otherwise:



1. `node->next = left->next;`
2. `left->next = node;`

Linked List: Inserting a Node

Case 3: Otherwise:



1. `node->next = left->next;`
2. `left->next = node;`

Linked List: Inserting a Node

```
struct Node {
   int data; /* Data field */
   Node *next; /* Next pointer */

   Node() {
      data = -1;
      next = 0;
   }

   Node(int data_) {
      data = data_;
      next = 0;
   }

   Node(int data_, Node* next_) {
      data = data_;
      next = next_;
   }
};
```

Linked List: Inserting a Node

```
void LinkedList::insertNode(int leftValue, int value) {
   Node* left = search(leftValue);
   Node* node = new Node(value);

   if (left == 0) { /* inserting a new head node */
      node->next = head;
      head = node;
      if (tail == 0)
         tail = head;
   }
   else if (left->next == 0) { /* inserting a new tail node */
      left->next = node;
      tail = node;
      if (head == 0)
         head = node;
   }
   else { /* inserting a node in the middle */
      node->next = left->next;
      left->next = node;
   }
}
```