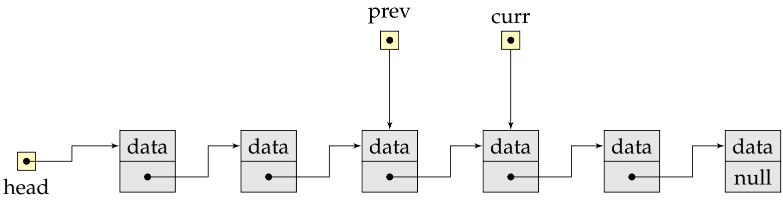
CSCI 2270: Data Structures

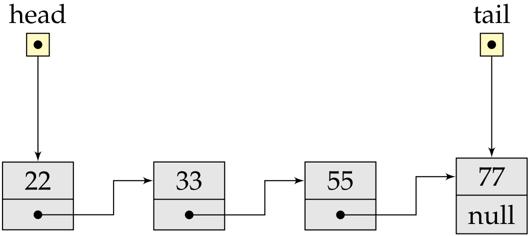
Lecture 08: Linked Lists (Contd.)

Ashutosh Trivedi

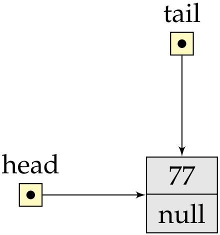


Department of Computer Science
University of Colorado Boulder

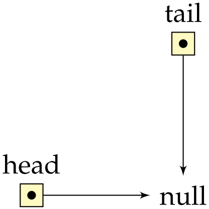
Linked List



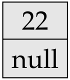
Linked List: with only one element



Linked List: empty list

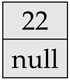


1. Defining a node of the list.



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

1. Defining a node of the list.



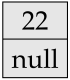
22

null

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

- self-referential pointers

Defining a node of the list.



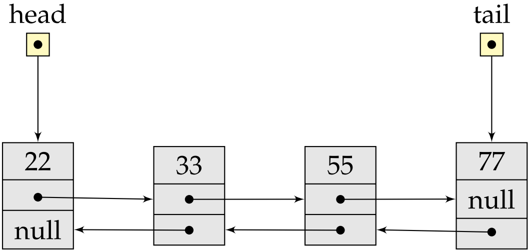
22

null

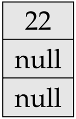
```
struct Node {
  int data;        /* Data field */
  Node *next;       /* Next pointer */

  Node() {          /* Default Constructor */
    data = -1;
    next = 0;
  }
  Node(int data_){  /* Fills data field  */
    data = data_;
    next = 0;
  }
  Node(int data_, Node* next_){ /* Fills both fields */
    data = data_;
    next = next_;
  }
};
```

- constructor and destructor

Doubly Linked List

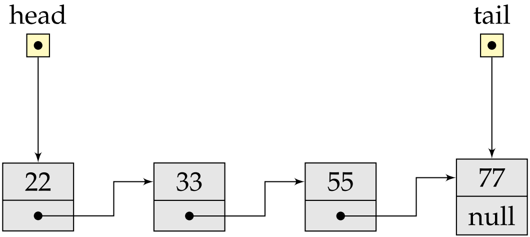


Defining a Doubly Linked List Node.



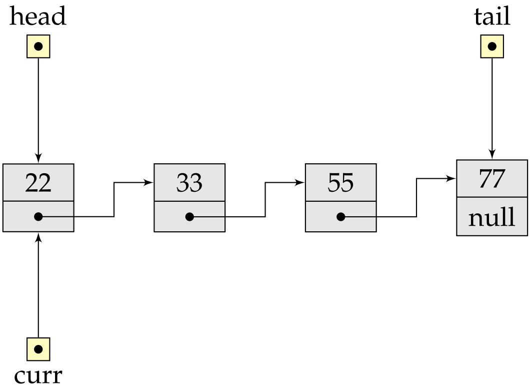
```
struct Node {
  int data;        /* Data field */
  Node *next;       /* Next pointer */
  Node *prev;       /* Previous pointer */

  Node() {          /* Default Constructor */
    data = -1;
    next = 0;
    prev = 0;
  }
  Node(int data_){  /* Fills data field  */
    data = data_;
    next = 0;
    prev = 0;
  }
  Node(int data_, Node* next_){ /* Fills data and next fields */
    data = data_;
    next = next_;
  }
  Node(int data_, Node* next_, Node* prev_){ /* Fills all fields */
    data = data_;
    next = next_;
    prev = prev_;
  }
};
```

Linked List (Abstract Data Type)



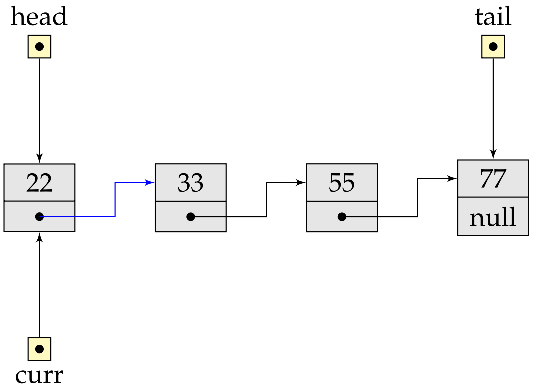
```
class LinkedList {
private:
  Node* head;
  Node* tail;

public:
  LinkedList();       /* Constructor */
  ~LinkedList();      /* Destructor */

  void traverse();         /* Traverse and print the list */
  Node* search(int val);    /* Search the list to find a value */
  void insertNode(int leftValue, int value); /* Insert a node in the list */
  void deleteNode(int value); /* delete the value from the list*/
};
```

Linked List: Traverse list



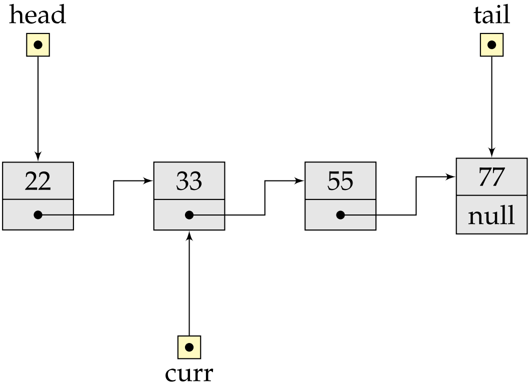
```
void LinkedList::traverse() {
  Node* curr = head;

  std::cout<<"head->";
  while (curr != 0) {
    std::cout << curr->data << "->";
    curr = curr->next;
  }
  std::cout<<"tail" << std::endl;
}
```

Linked List: Traverse list



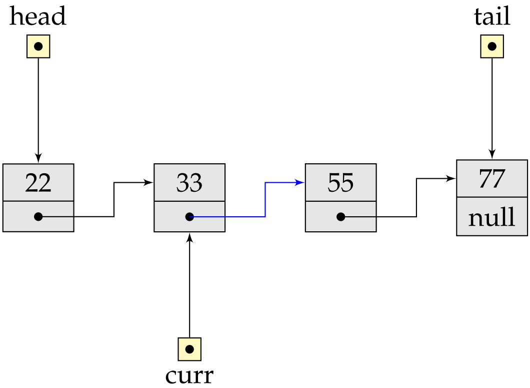
```
void LinkedList::traverse() {
  Node* curr = head;

  std::cout<<"head->";
  while (curr != 0) {
    std::cout << curr->data << "->";
    curr = curr->next;
  }
  std::cout<<"tail" << std::endl;
}
```

Linked List: Traverse list



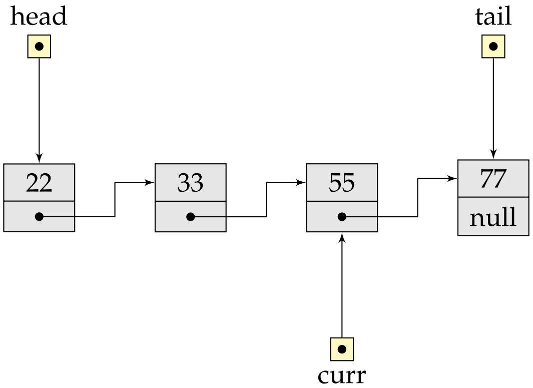
```
void LinkedList::traverse() {
  Node* curr = head;

  std::cout<<"head->";
  while (curr != 0) {
    std::cout << curr->data << "->";
    curr = curr->next;
  }
  std::cout<<"tail" << std::endl;
}
```

Linked List: Traverse list



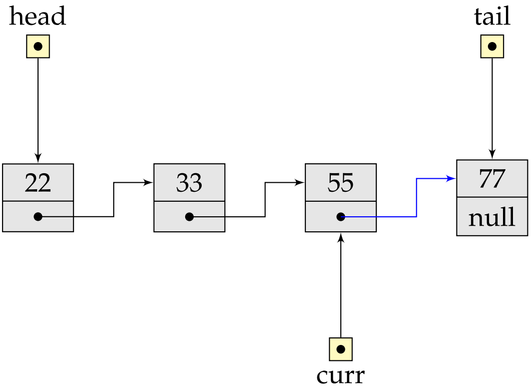
```
void LinkedList::traverse() {
  Node* curr = head;

  std::cout<<"head->";
  while (curr != 0) {
    std::cout << curr->data << "->";
    curr = curr->next;
  }
  std::cout<<"tail" << std::endl;
}
```

Linked List: Traverse list



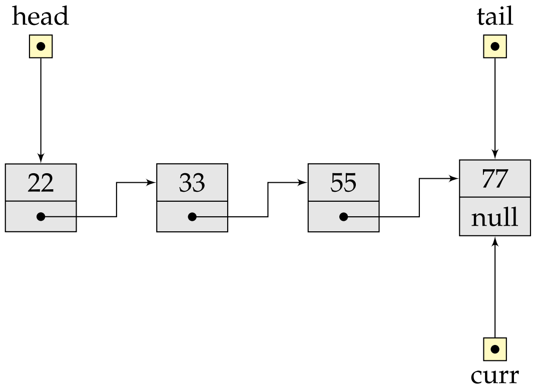
```
void LinkedList::traverse() {
  Node* curr = head;

  std::cout<<"head->";
  while (curr != 0) {
    std::cout << curr->data << "->";
    curr = curr->next;
  }
  std::cout<<"tail" << std::endl;
}
```

Linked List: Traverse list



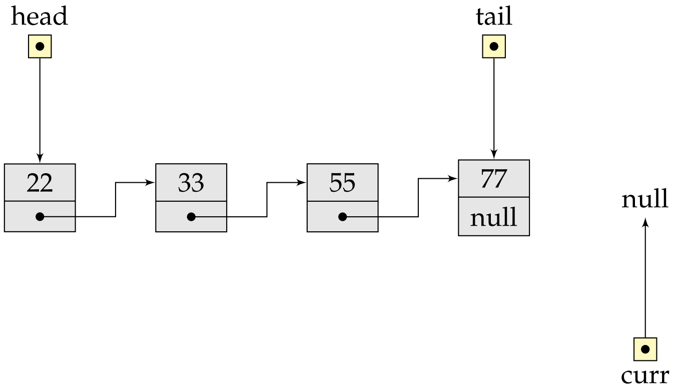
```
void LinkedList::traverse() {
  Node* curr = head;

  std::cout<<"head->";
  while (curr != 0) {
    std::cout << curr->data << "->";
    curr = curr->next;
  }
  std::cout<<"tail" << std::endl;
}
```

Linked List: Traverse list



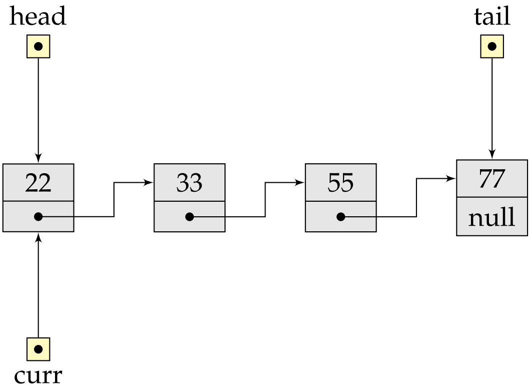
```
void LinkedList::traverse() {
  Node* curr = head;

  std::cout<<"head->";
  while (curr != 0) {
    std::cout << curr->data << "->";
    curr = curr->next;
  }
  std::cout<<"tail" << std::endl;
}
```

Linked List: Traverse list



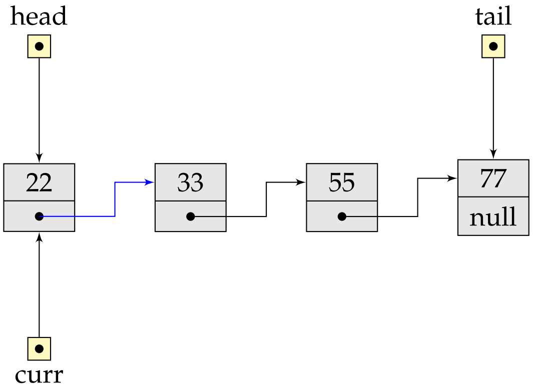
```
void LinkedList::traverse() {
  Node* curr = head;

  std::cout<<"head->";
  while (curr != 0) {
    std::cout << curr->data << "->";
    curr = curr->next;
  }
  std::cout<<"tail" << std::endl;
}
```

Linked List: Search list: search(55)



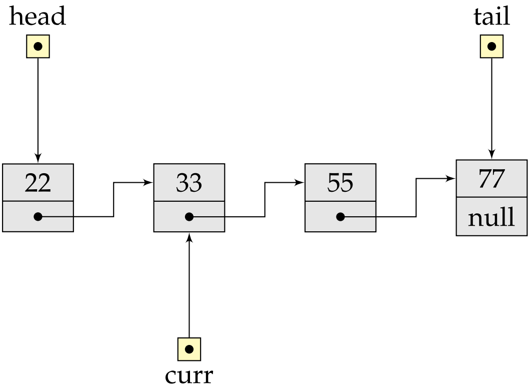
```
Node* LinkedList::search(int val) {
  Node* curr = head;

  while (curr != 0) {
    if (curr->data == val) return curr;
    curr = curr->next;
  }

  return 0;
}
```

Linked List: Search list: search(55)



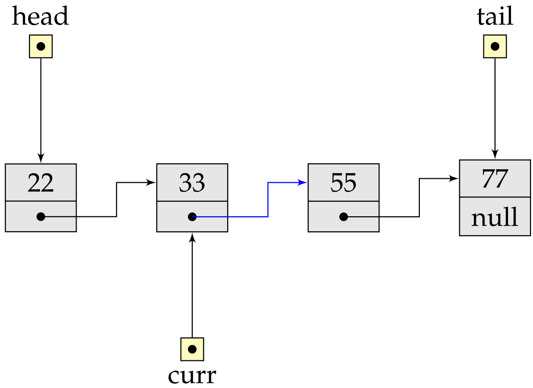
```
Node* LinkedList::search(int val) {
  Node* curr = head;

  while (curr != 0) {
    if (curr->data == val) return curr;
    curr = curr->next;
  }

  return 0;
}
```

Linked List: Search list: search(55)



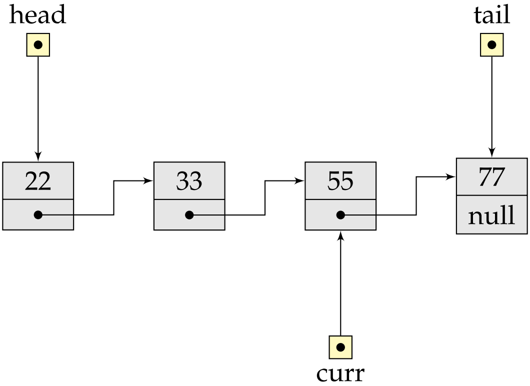
```
Node* LinkedList::search(int val) {
  Node* curr = head;

  while (curr != 0) {
    if (curr->data == val) return curr;
    curr = curr->next;
  }

  return 0;
}
```

Linked List: Search list: search(55)



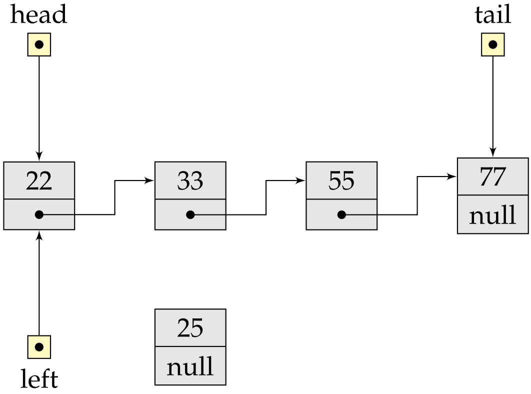
```
Node* LinkedList::search(int val) {
  Node* curr = head;

  while (curr != 0) {
    if (curr->data == val) return curr;
    curr = curr->next;
  }

  return 0;
}
```

Linked List: Search list: search(55)

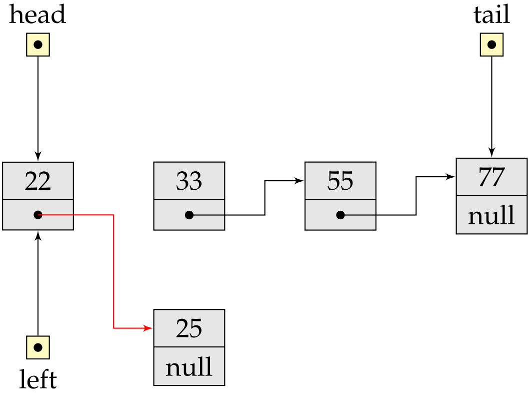


```
Node* LinkedList::search(int val) {
  Node* curr = head;

  while (curr != 0) {
    if (curr->data == val) return curr;
    curr = curr->next;
  }

  return 0;
}
```

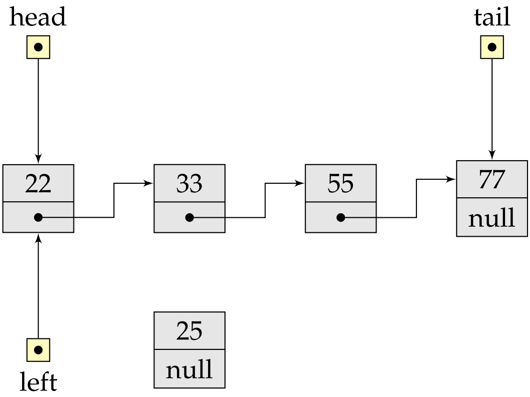
Linked List: InsertNode (case 1: at the head)



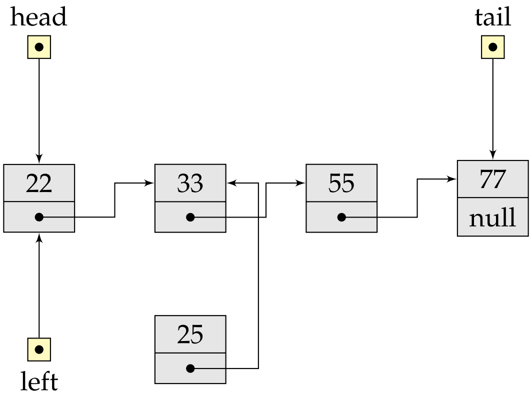
Linked List: InsertNode (case 1: at the head)



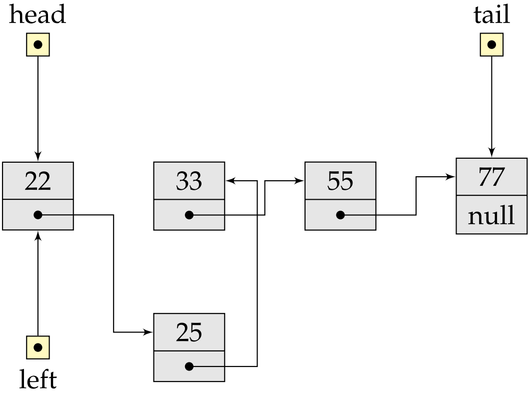
Linked List: InsertNode (case 1: at the head)



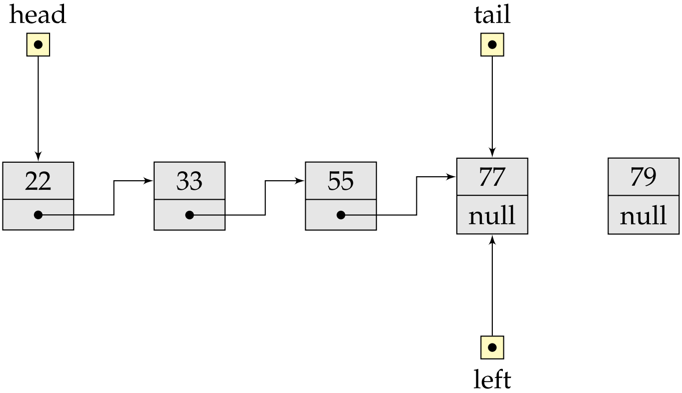
Linked List: InsertNode (case 1: at the head)



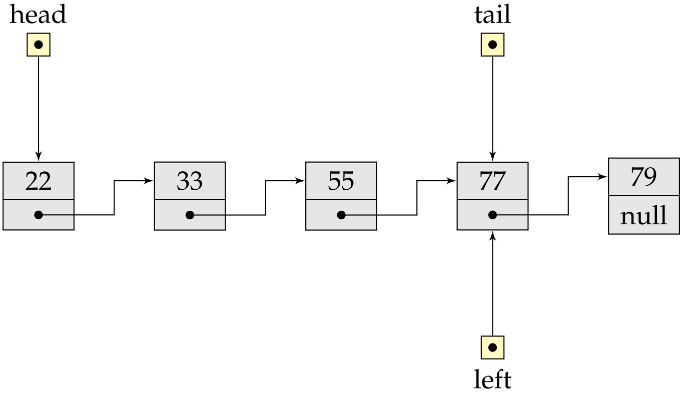
Linked List: InsertNode (case 1: at the head)



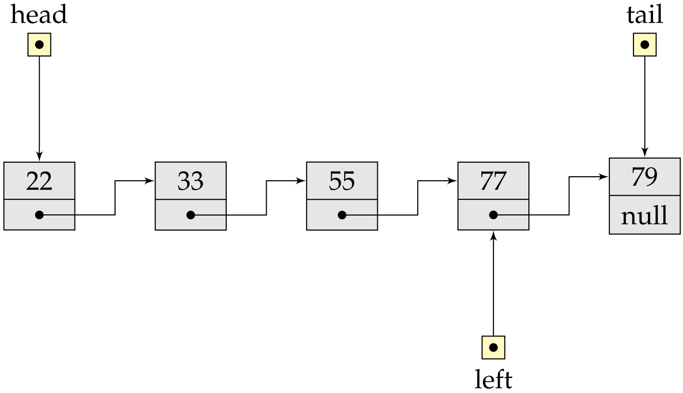
Linked List: InsertNode (case 2: at the tail)



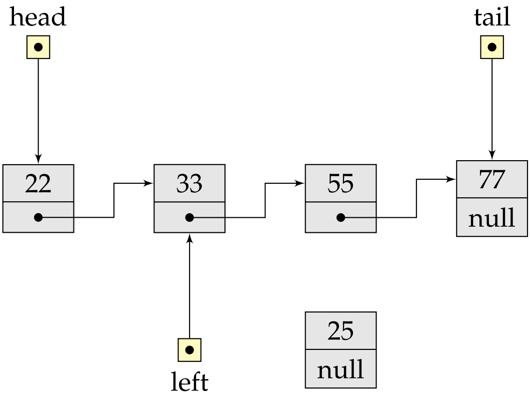
Linked List: InsertNode (case 2: at the tail)



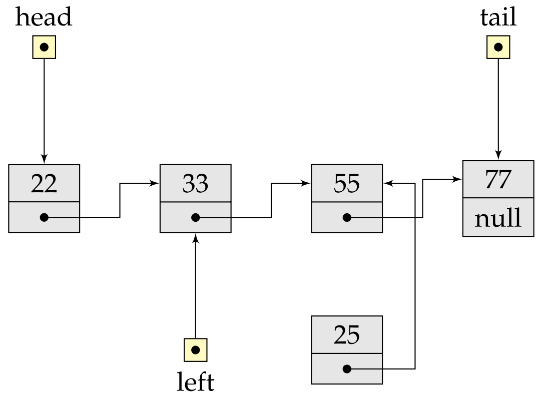
Linked List: InsertNode (case 2: at the tail)



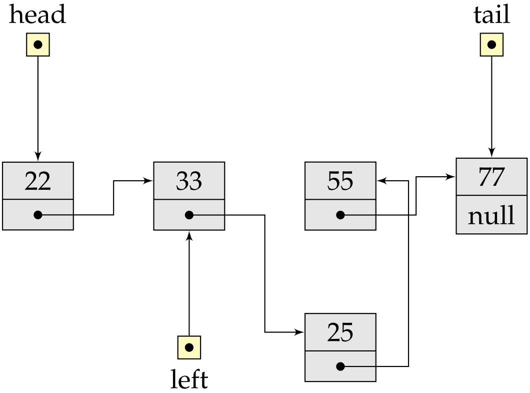
Linked List: InsertNode (case 3: in the middle)



Linked List: InsertNode (case 3: in the middle)



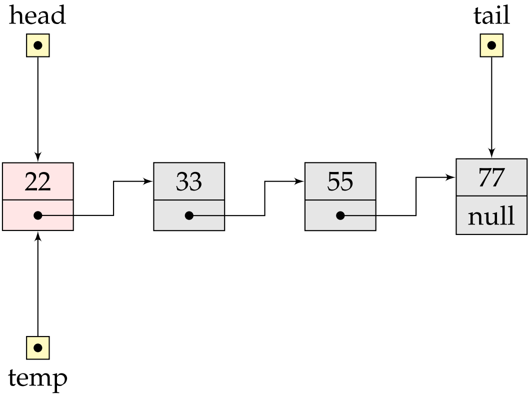
Linked List: InsertNode (case 3: in the middle)



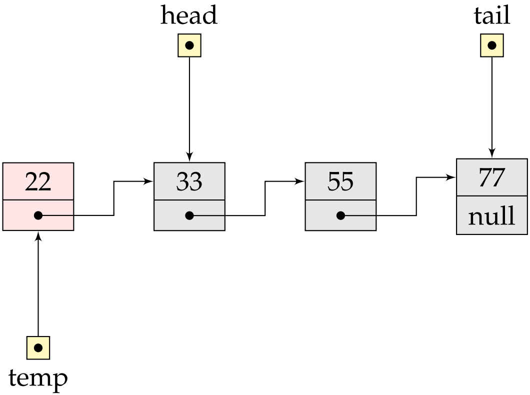
Linked List: InsertNode

```
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;
  }
}
```

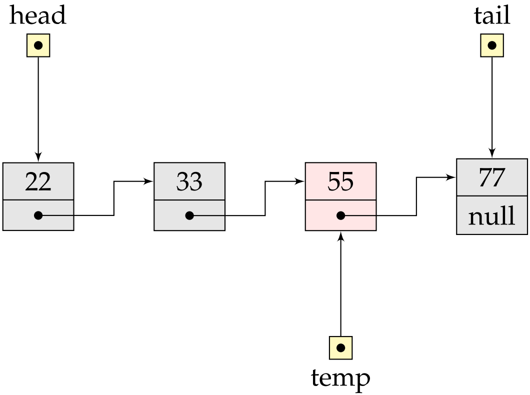
Linked List: deleteNode (head node)



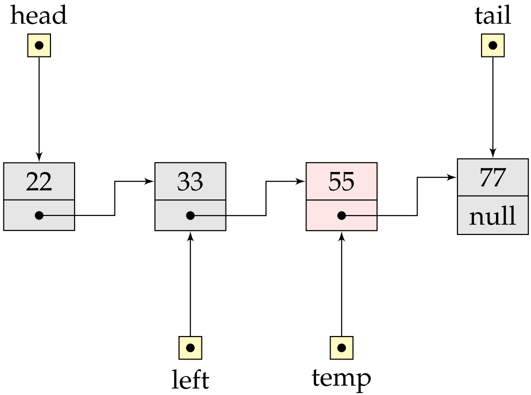
Linked List: deleteNode (head node)



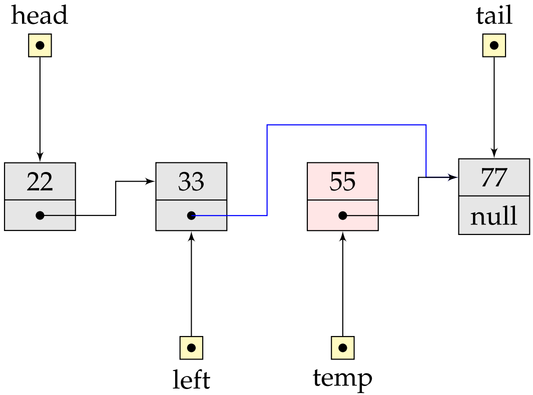
Linked List: deleteNode (middle node)



Linked List: deleteNode (middle node)



Linked List: deleteNode (middle node)



Linked List: deleteNode

```
void LinkedList::deleteNode(int value) {

  if (head->data == value) {
    Node* temp = head;
    head = head->next;
    delete temp;
  }
  else { /*either tail node or middle node */
    Node* left = head;
    Node* temp = left->next;
    bool isFound = false;
    while (temp != 0 && isFound != true) {
      if (temp->data == value) {
  if (temp->next == 0) { /* tail node */
    left->next = 0;
    tail = left;
  }
  else {
    left->next = temp->next;
  }
  delete temp;
  isFound = true;
      }
      else {
  left = temp;
  temp = temp-> next;
      }
    }
  }
}
```

Common Pitfall

- Memory leaks!
- Portion of lists are lost!

Doubly Linked Lists

- Except delete all other operations are similar.
- You need to keep track of both previous and next pointers.
- Delete operation is significantly simpler!