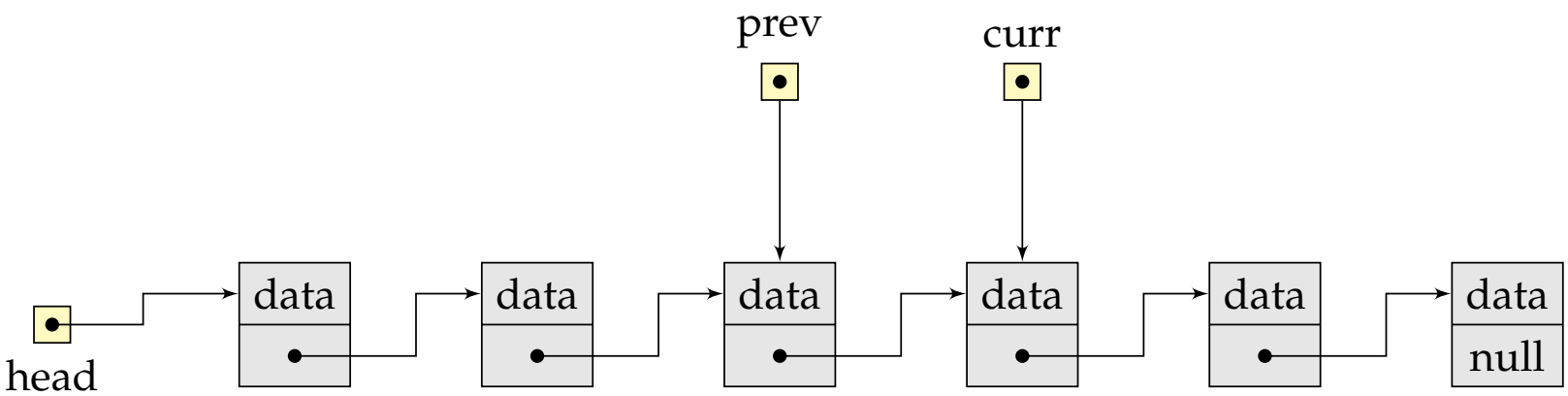
CSCI 2270: Data Structures

Lecture 10: Linked Lists (Insertion and Deletion)

Ashutosh Trivedi



Department of Computer Science
University of Colorado Boulder

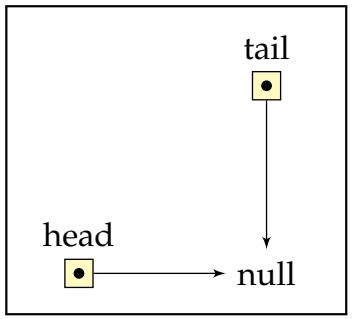
Linked Lists: ADT

Linked List: Inserting a node

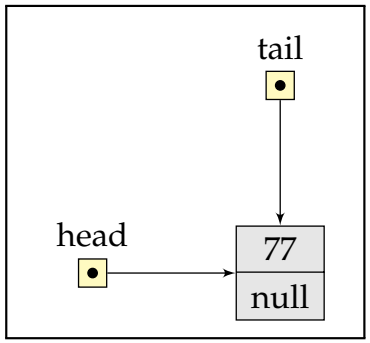
Linked List: Deleting a node

Doubly-Linked List

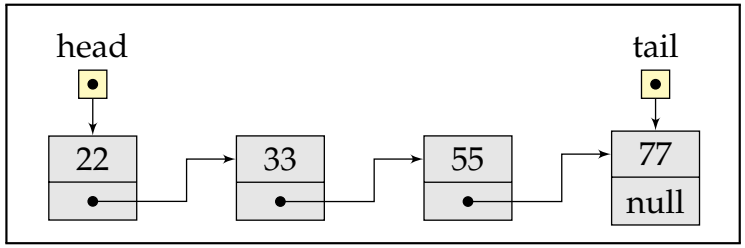
Linked List



a) An empty list



b) A list with a single node



c) A list with $n = 4$ nodes.

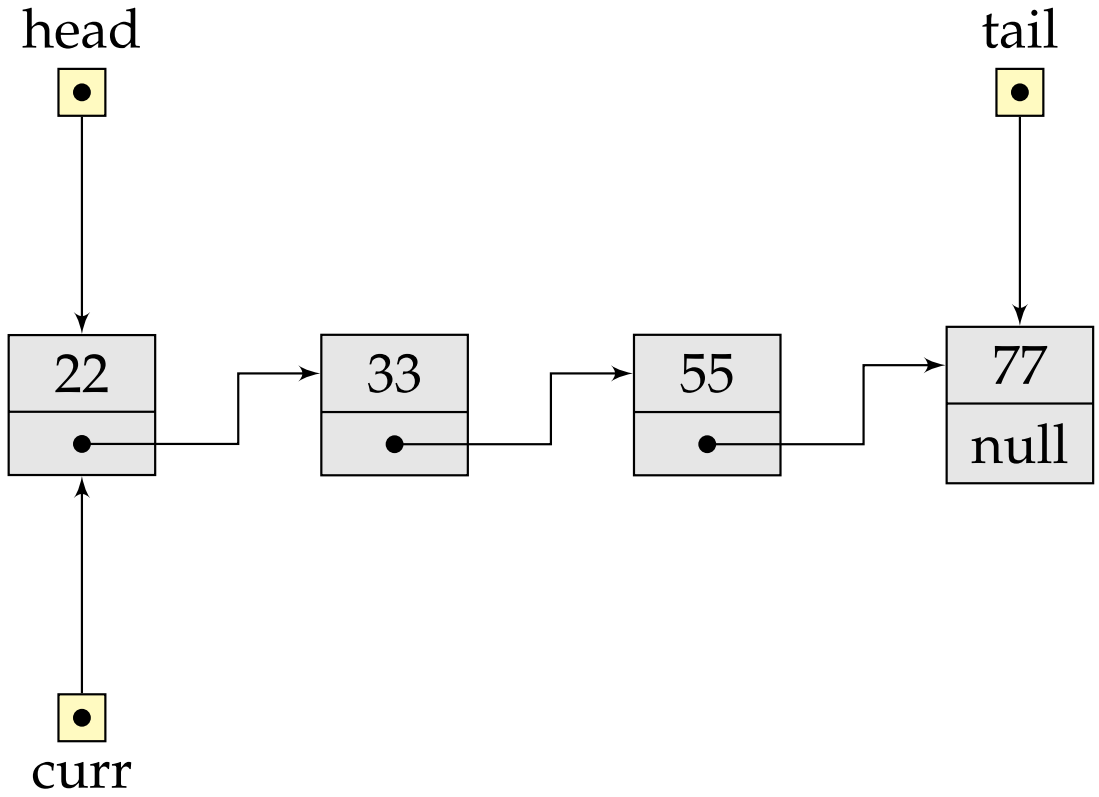
```
class LinkedList {
private:
  Node* head;
  Node* tail;

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*/
};
```

Abstract Data Type: List with a linked-list implementation.

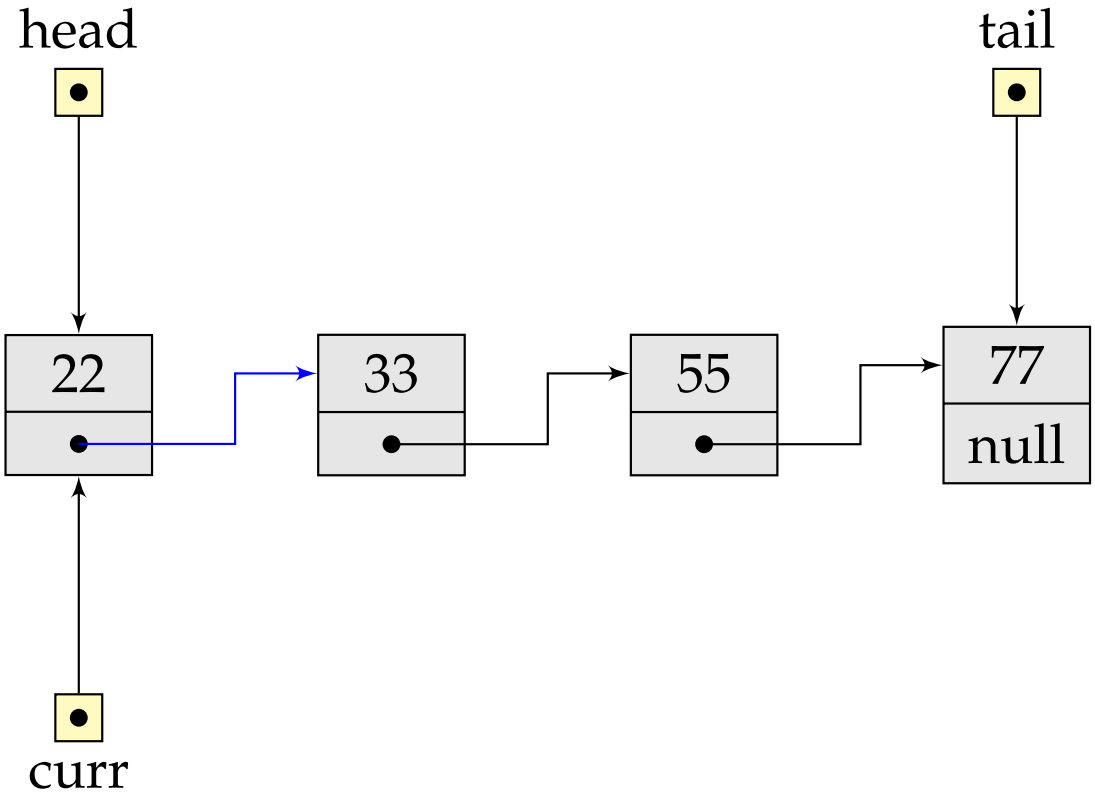
Linked List: Traverse list




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

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

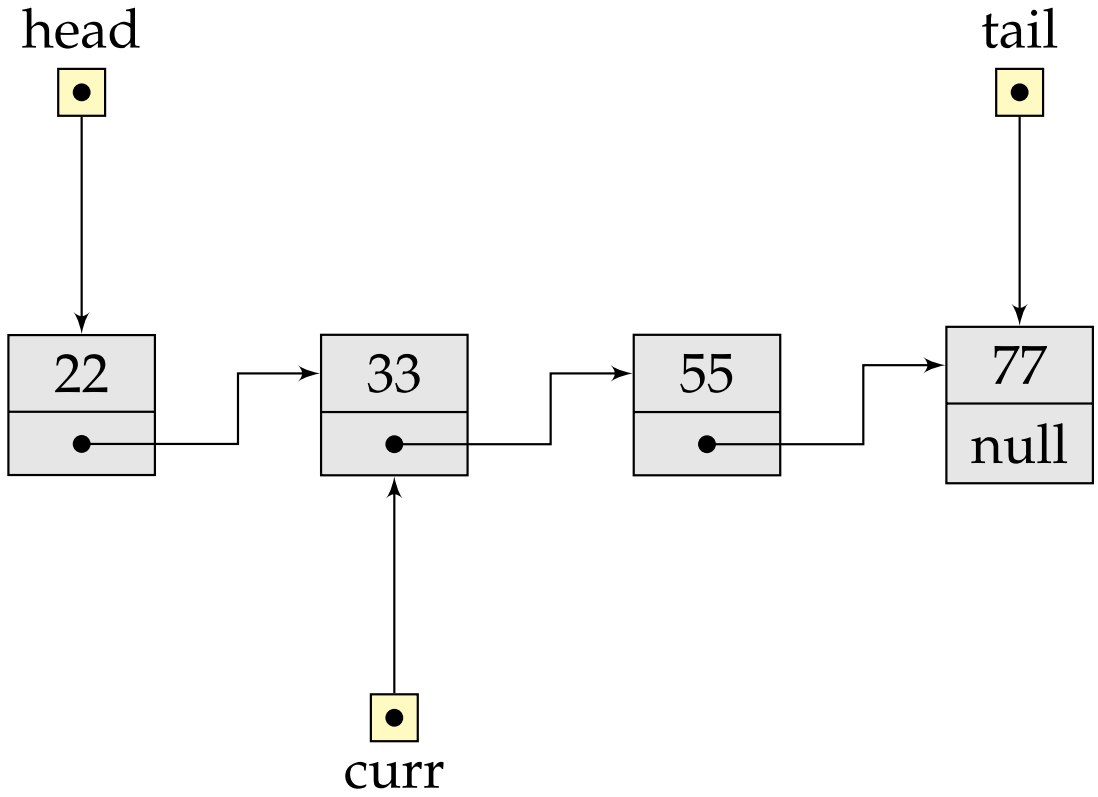
Linked List: Traverse list




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

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

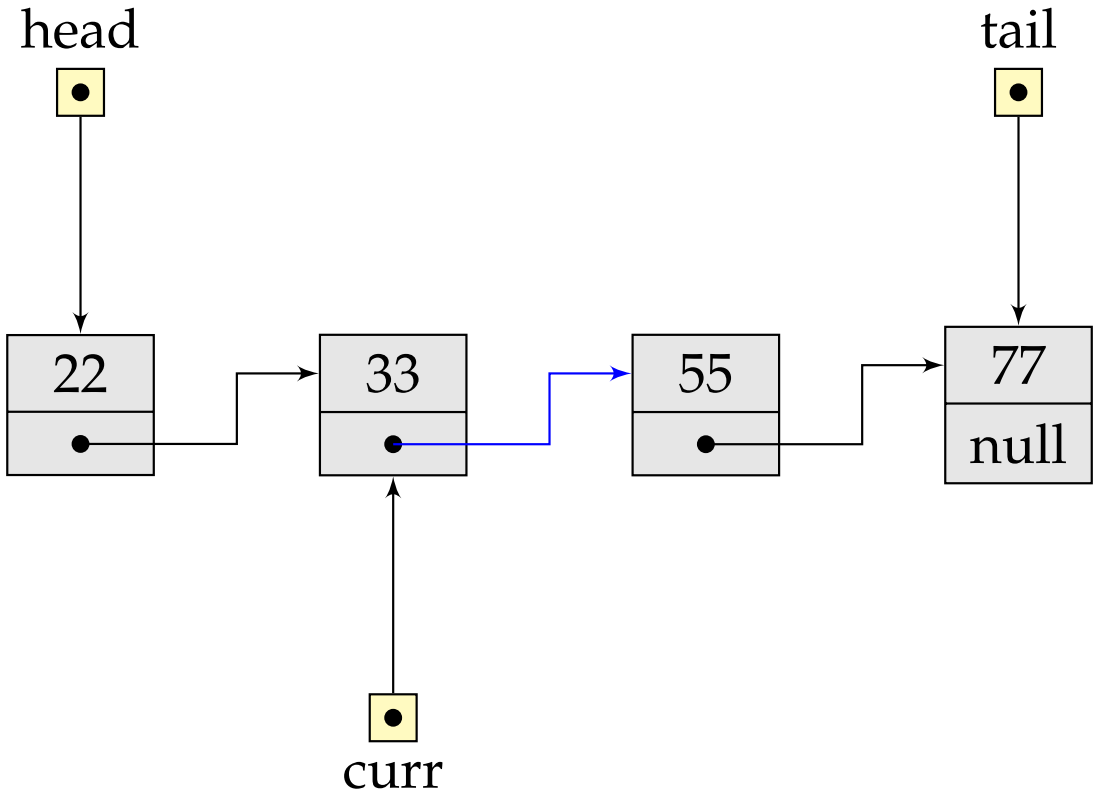
Linked List: Traverse list




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

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

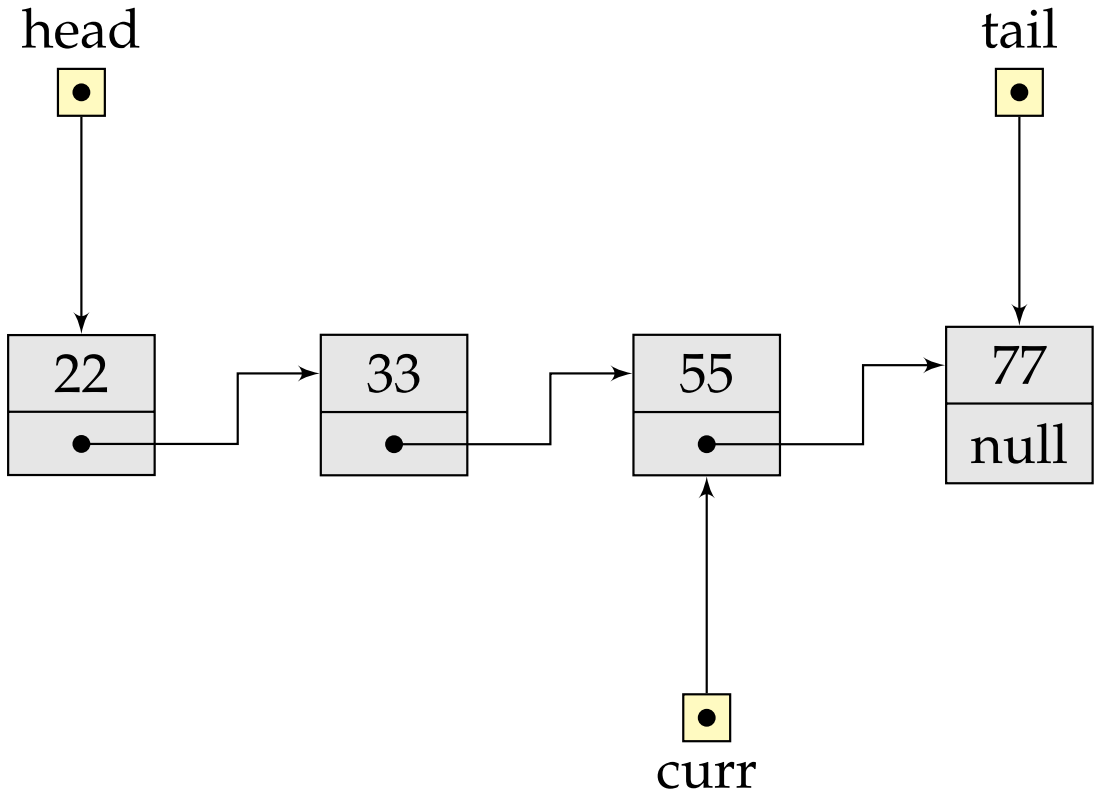
Linked List: Traverse list




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

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

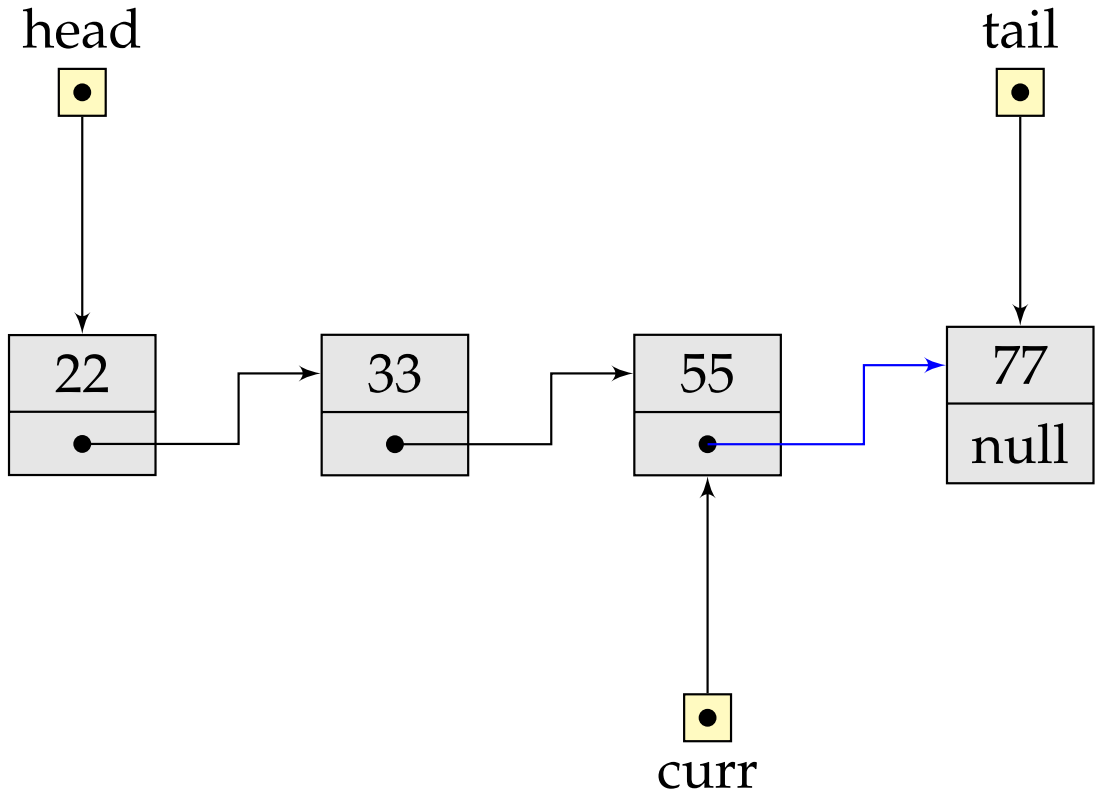
Linked List: Traverse list




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

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

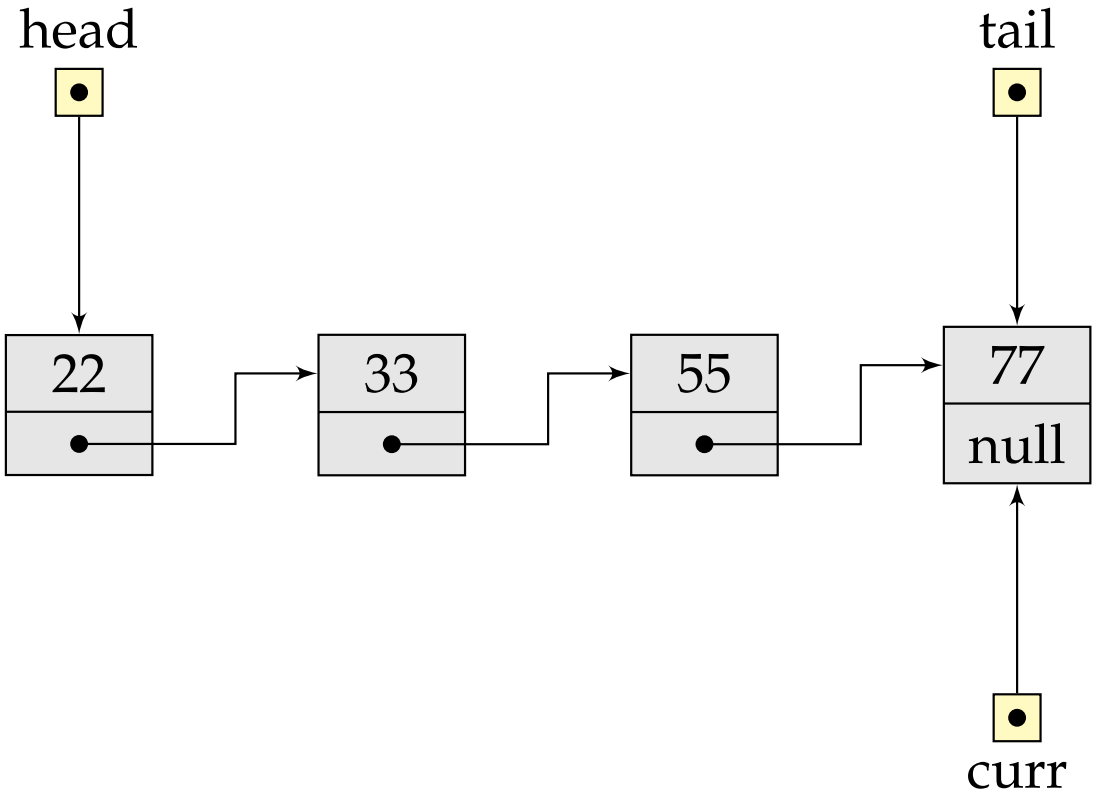
Linked List: Traverse list




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

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

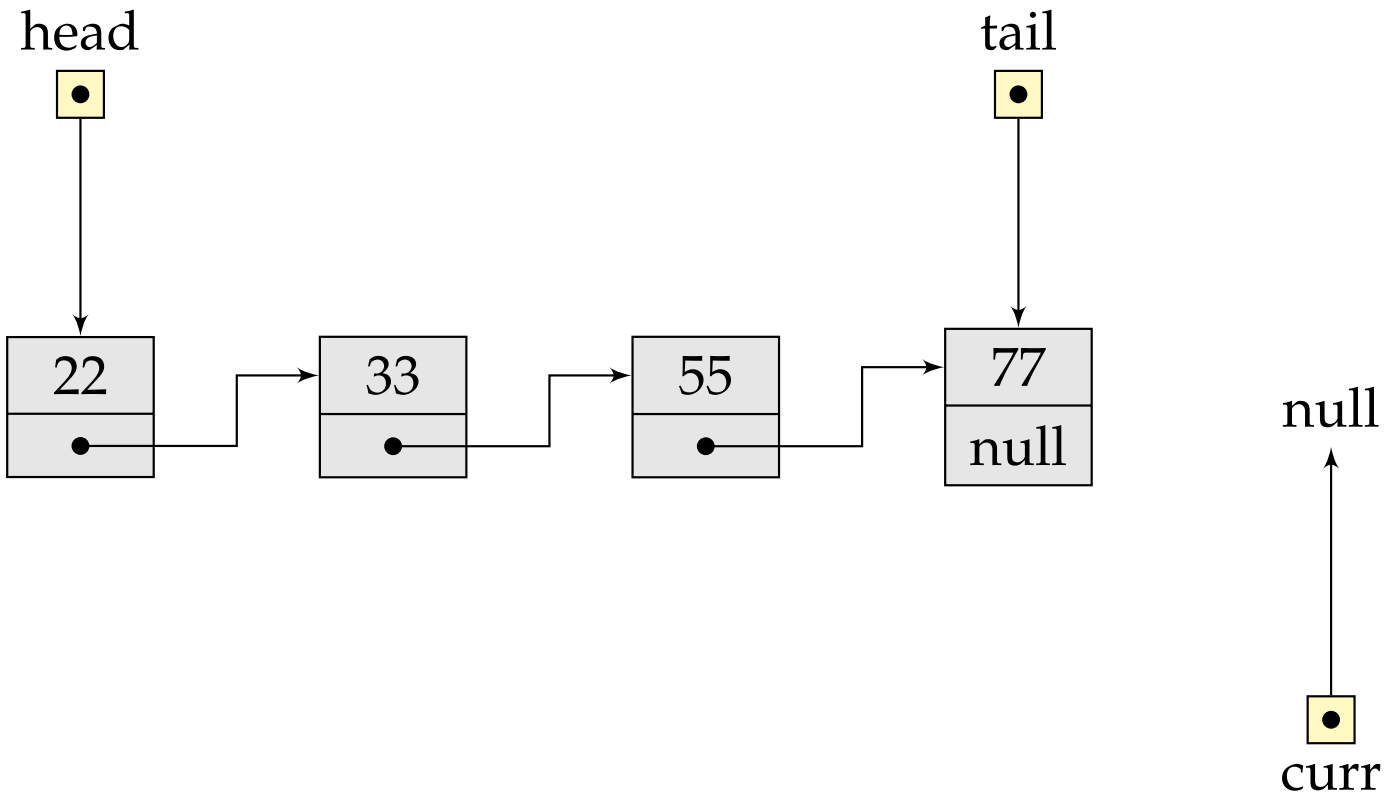
Linked List: Traverse list




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

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

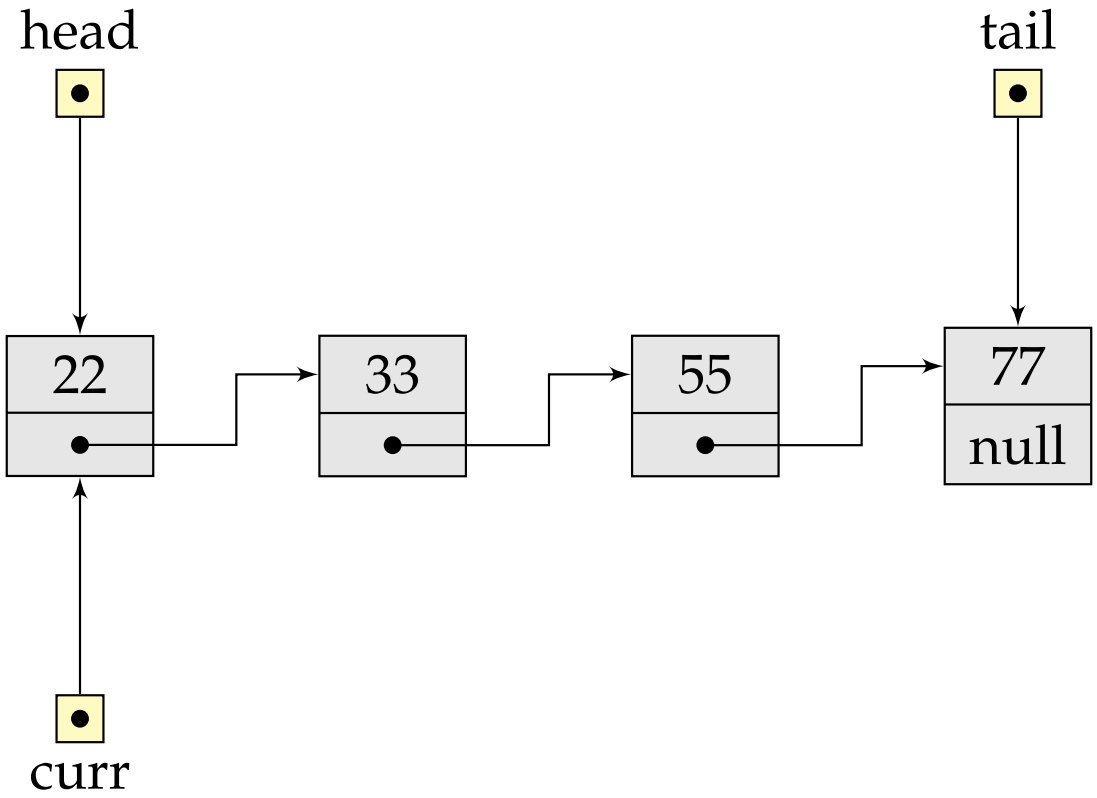
Linked List: Traverse list




```
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: Traverse list



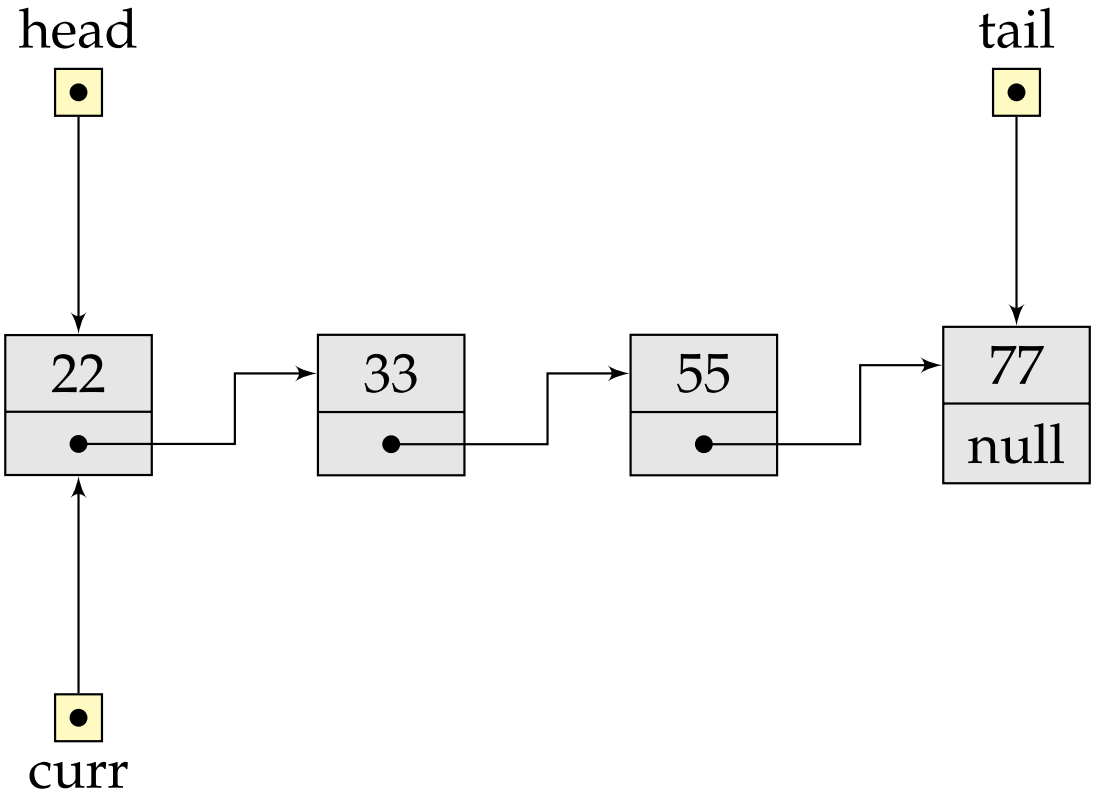

```
void LinkedList::traverse() {
  Node* curr;

  std::cout<<"head->";

  for (curr = head; curr != 0; curr = curr->next) {
    std::cout << curr->data << "->";
  }

  std::cout<<"tail" << std::endl;
}
```

Linked List: Search list: search(55)

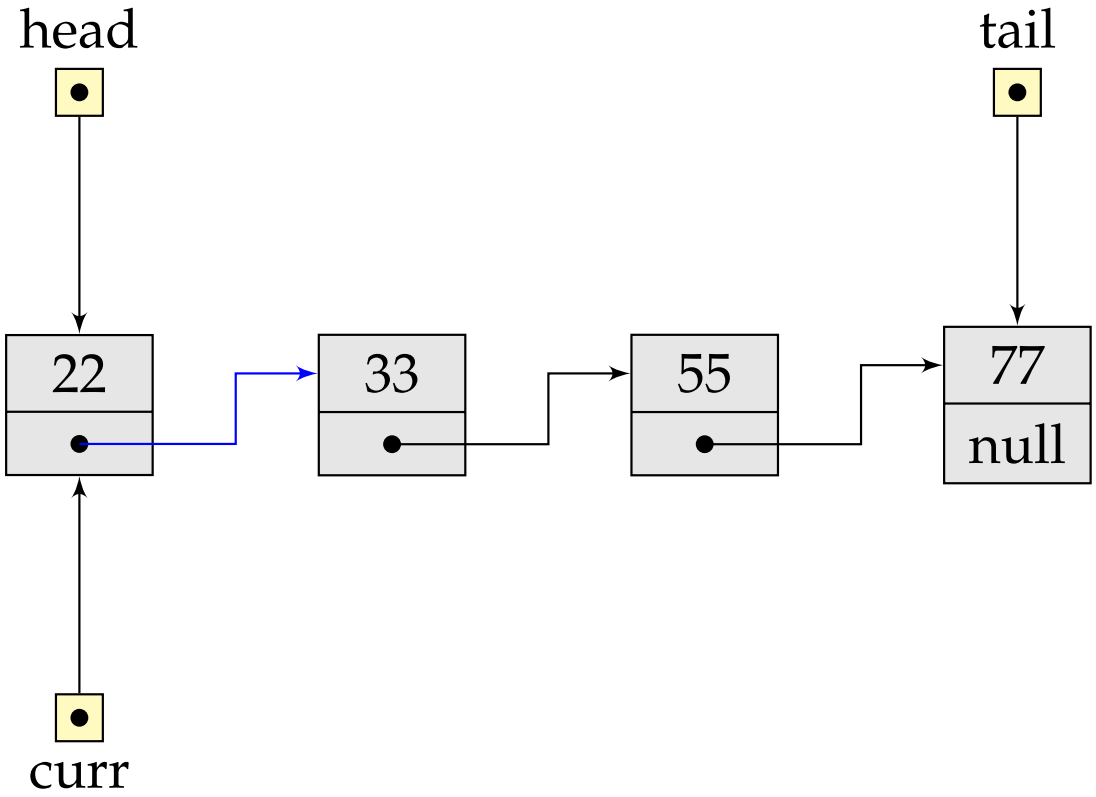



```
Node* LinkedList::search(int val) {
  Node* curr = head;

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

  return 0;
}
```

Linked List: Search list: search(55)

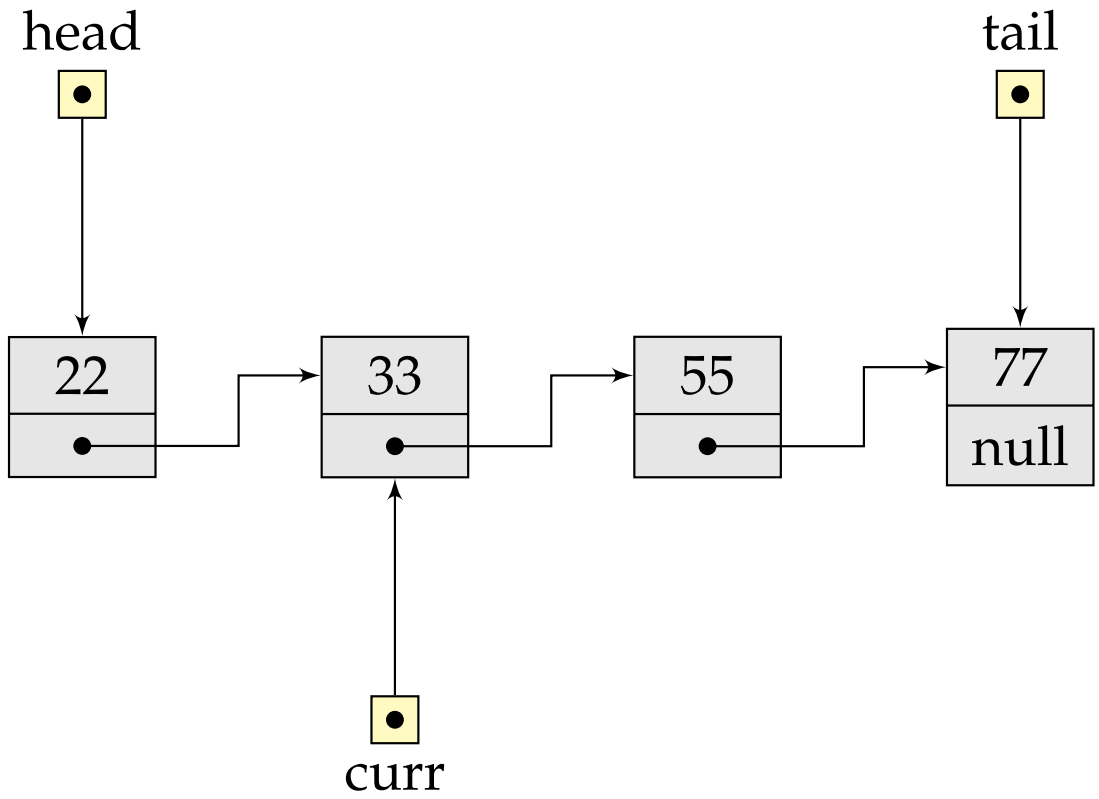



```
Node* LinkedList::search(int val) {
  Node* curr = head;

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

  return 0;
}
```

Linked List: Search list: search(55)

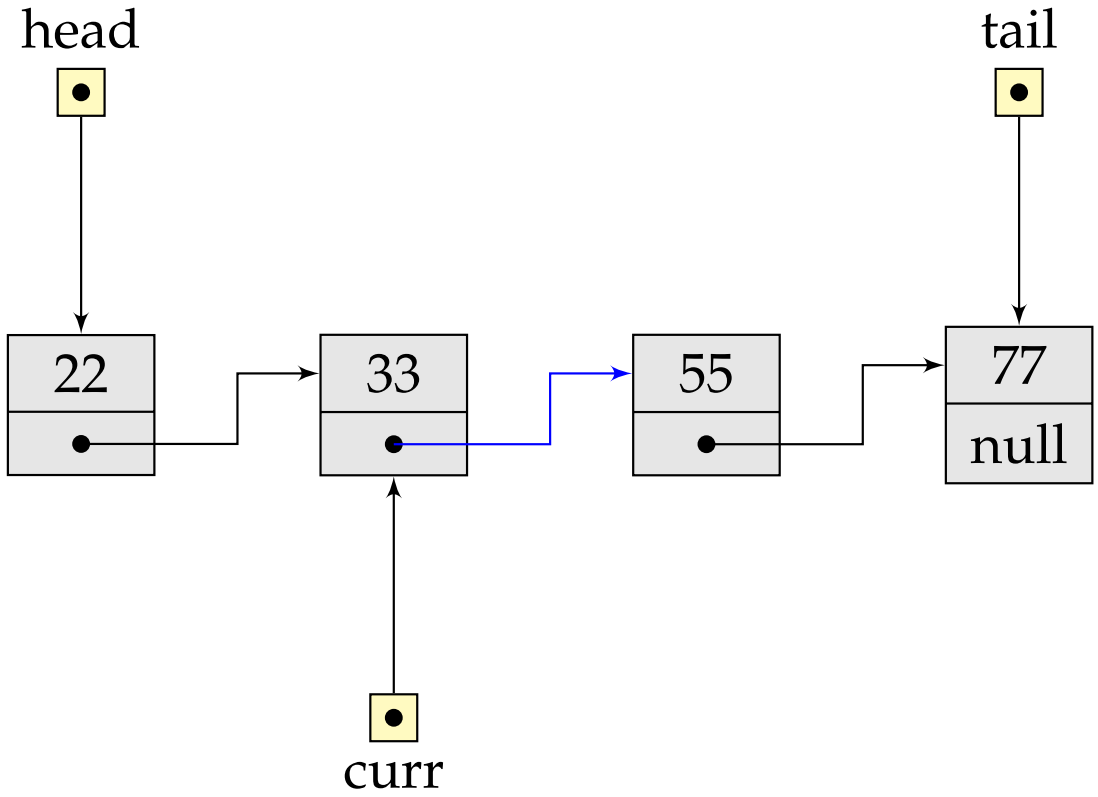



```
Node* LinkedList::search(int val) {
  Node* curr = head;

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

  return 0;
}
```

Linked List: Search list: search(55)

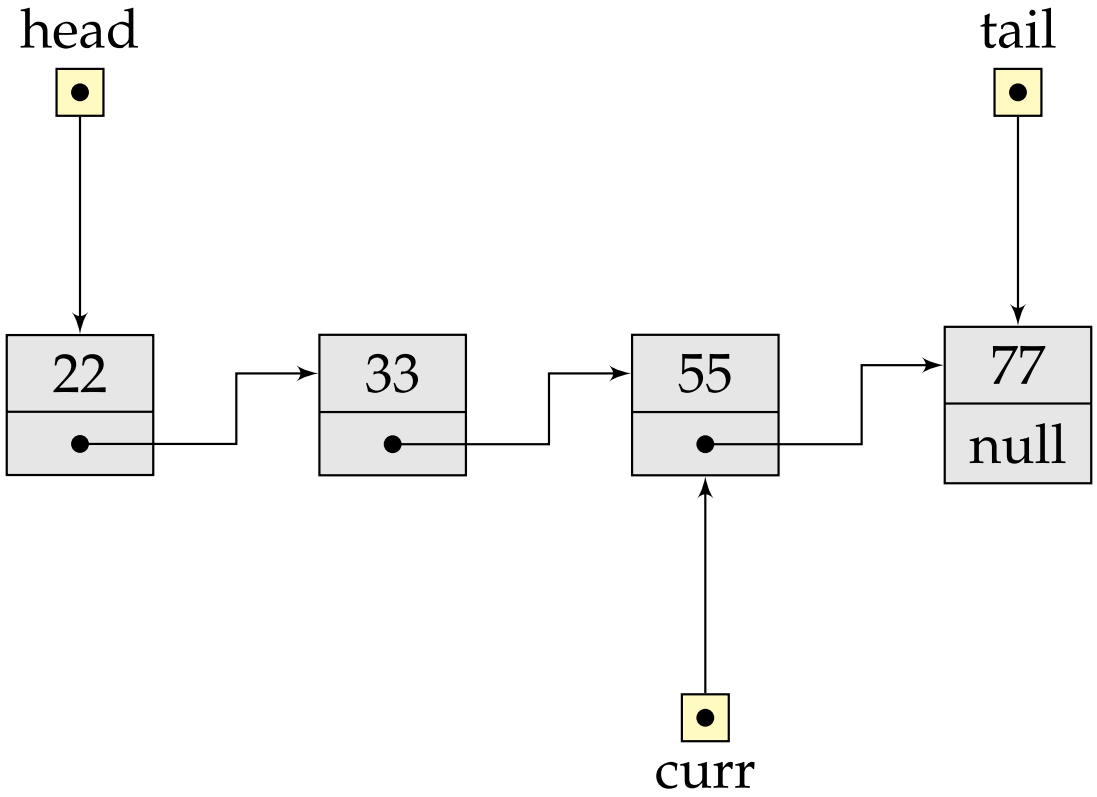



```
Node* LinkedList::search(int val) {
  Node* curr = head;

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

  return 0;
}
```

Linked List: Search list: search(55)

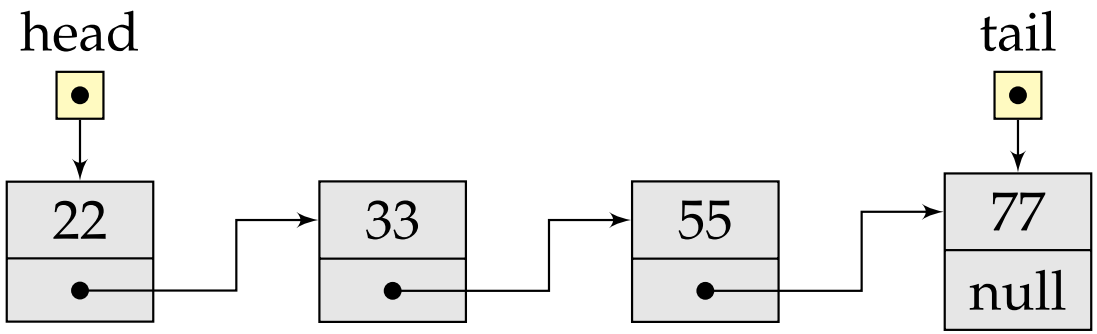



```
Node* LinkedList::search(int val) {
  Node* curr = head;

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

  return 0;
}
```

Linked List (Abstract Data Type)

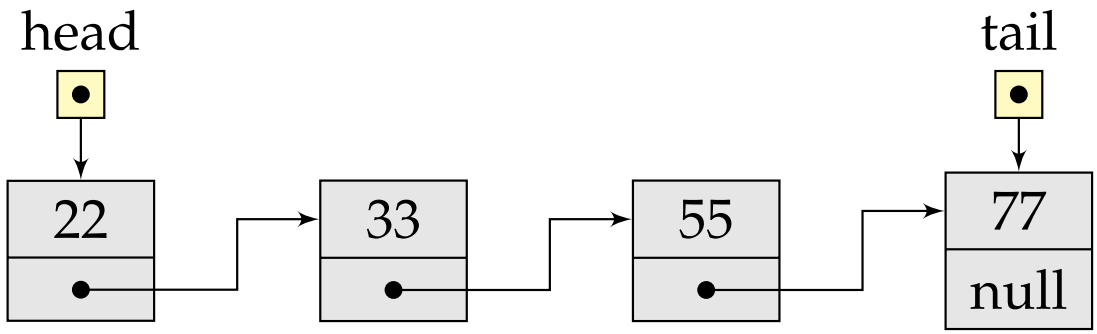



```
class LinkedList {
private:
  Node* head;
  Node* tail;

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*/
};
```

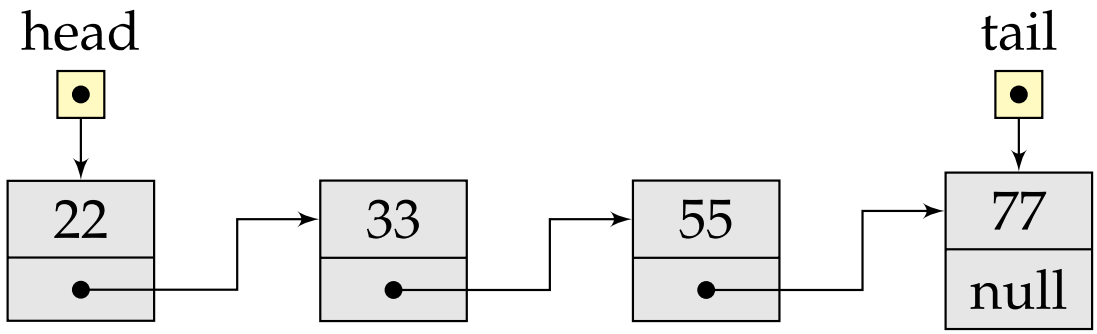
Insert a node in a linked list



Insert Node:

- `int insertNode(int leftValue, int value)`

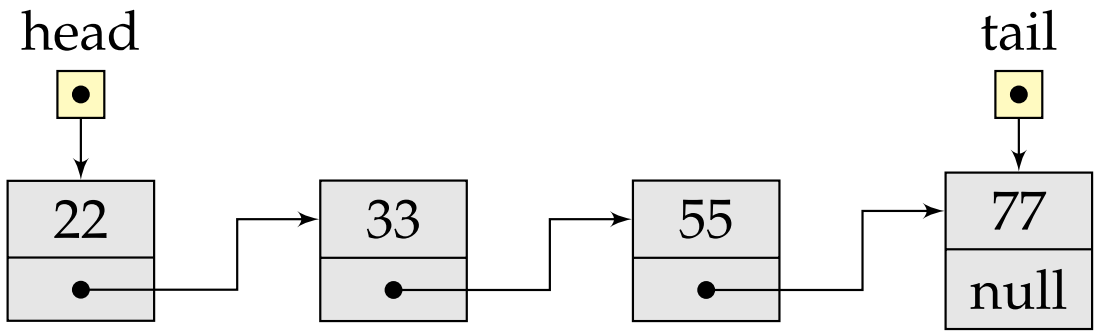
Insert a node in a linked list



Insert Node:

- `int insertNode(int leftValue, int value)`
- Insert the given "value" in the list immediately right to the first node having value equal to "leftValue".

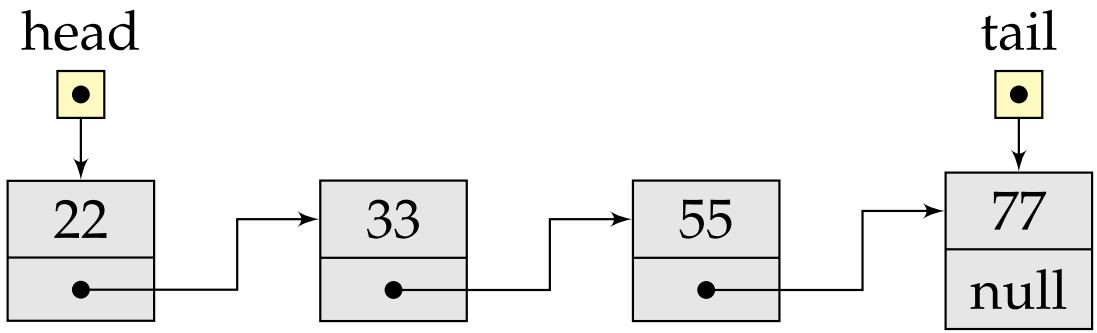
Insert a node in a linked list



Insert Node:

- `int insertNode(int leftValue, int value)`
- Insert the given "value" in the list immediately right to 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.

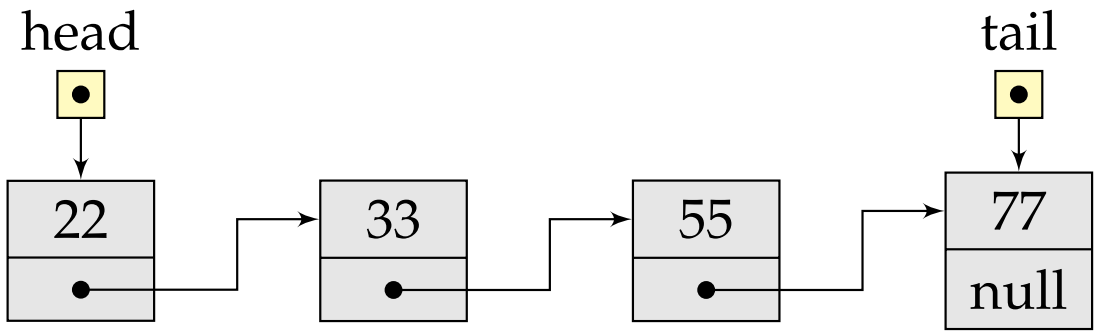
Insert a node in a linked list



Insert Node:

- `int insertNode(int leftValue, int value)`
- Insert the given "value" in the list immediately right to 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.
- `Node *node= new Node(value);`

Insert a node in a linked list

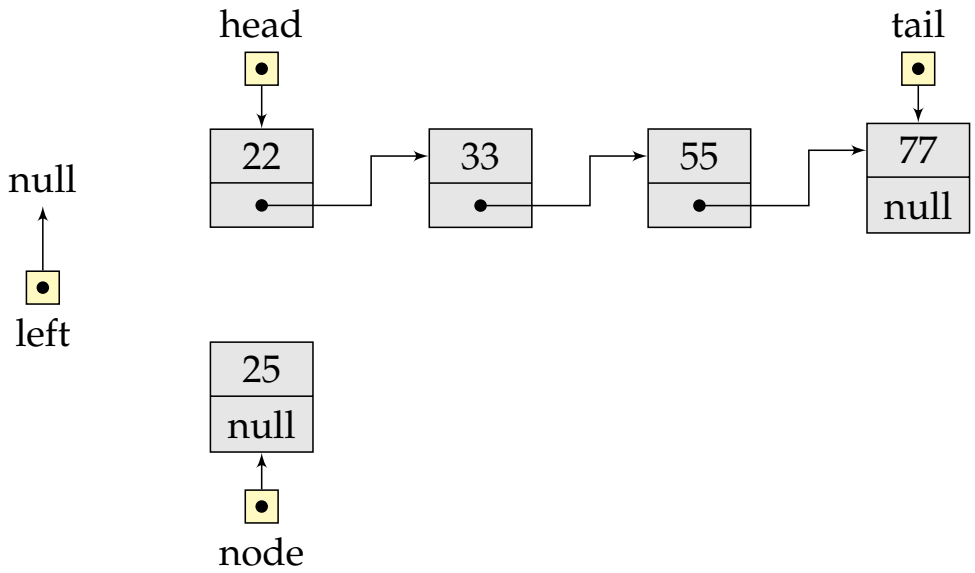


Insert Node:

- `int insertNode(int leftValue, int value)`
- Insert the given "value" in the list immediately right to 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.
- `Node *node= new Node(value);`
- `Node *left=search(leftValue);`

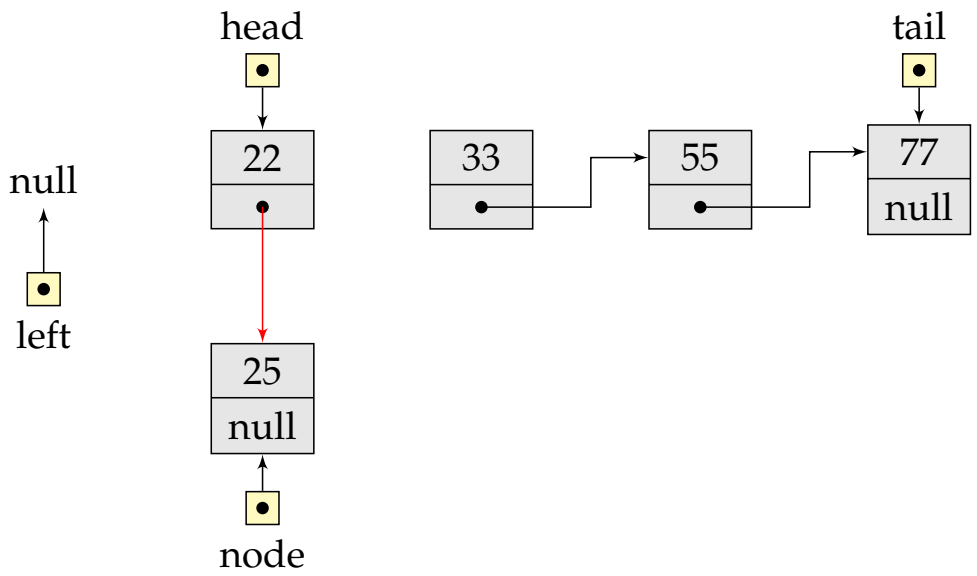
Insert a node in a linked list

Case 1: `(left == 0)`



Insert a node in a linked list

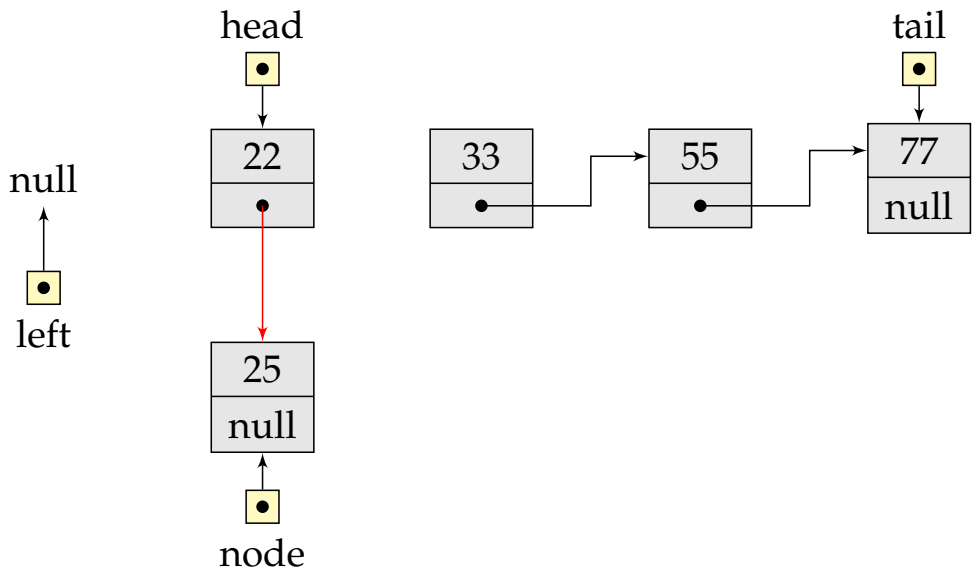
Case 1: `(left == 0)`



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

Insert a node in a linked list

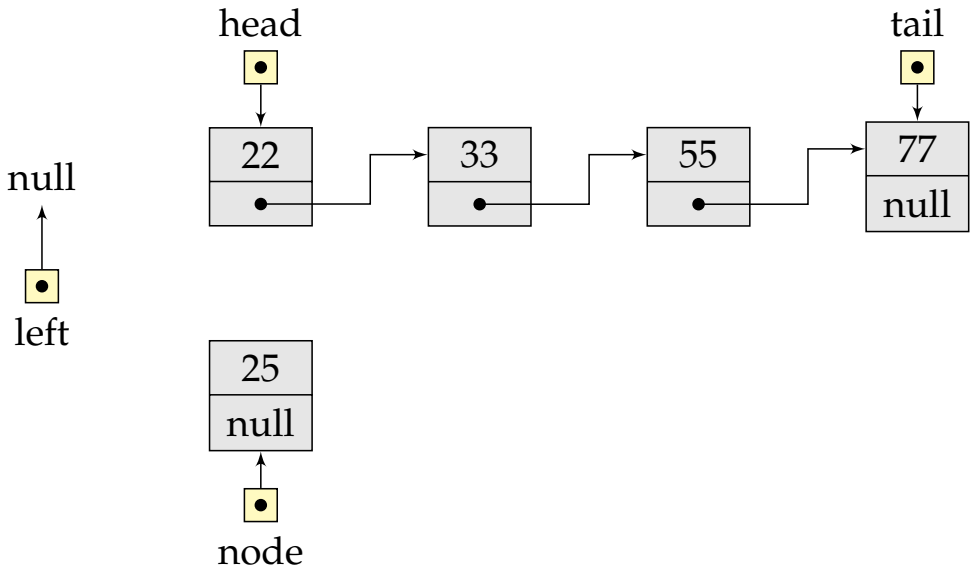
Case 1: `(left == 0)`



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

Insert a node in a linked list

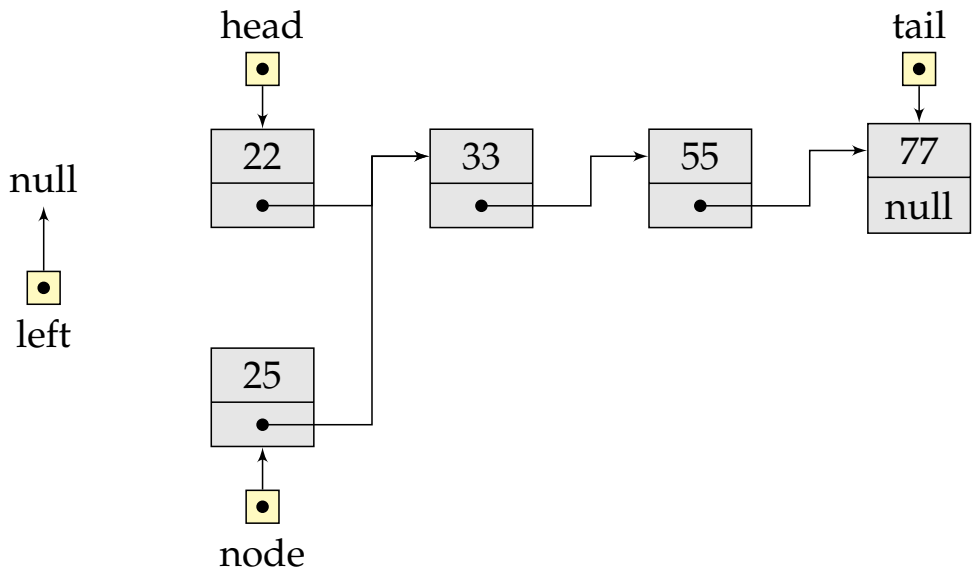
Case 1: `(left == 0)`



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

Insert a node in a linked list

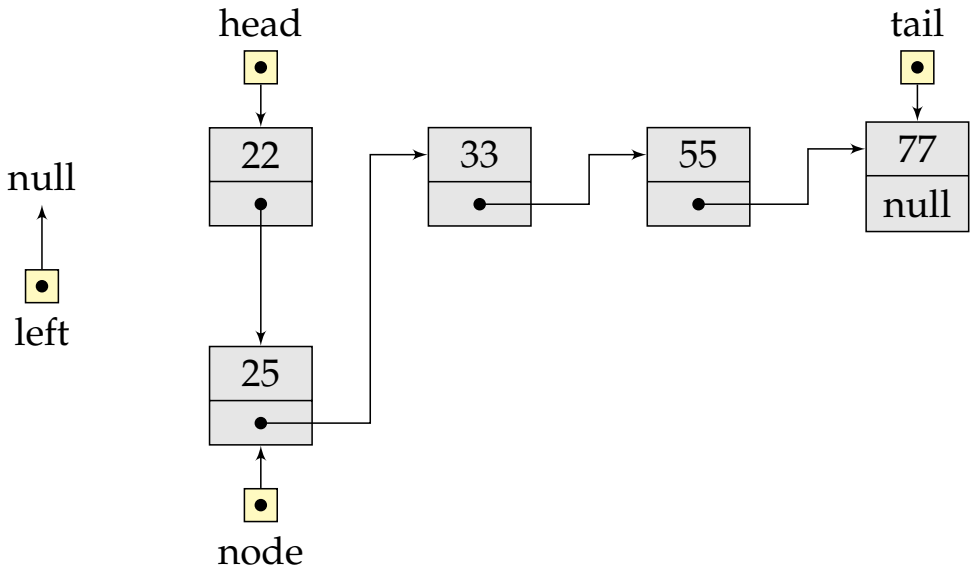
Case 1: `(left == 0)`



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

Insert a node in a linked list

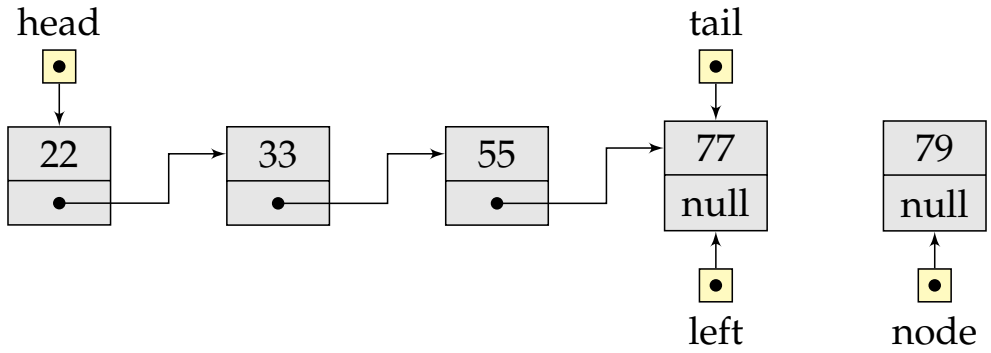
Case 1: `(left == 0)`



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

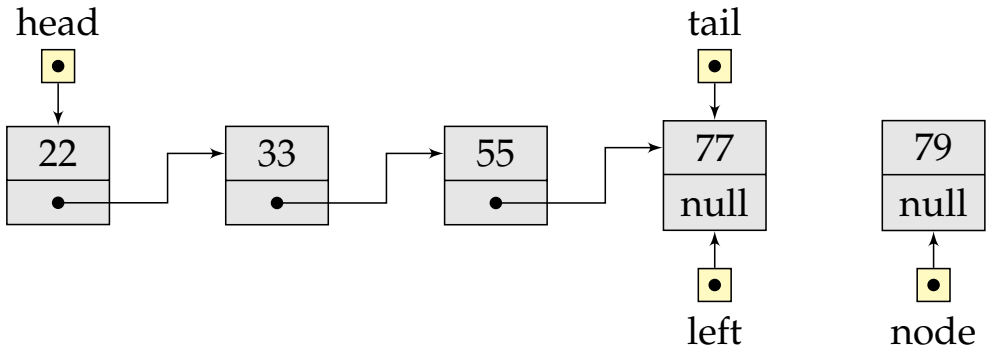
Linked List: InsertNode (case 2: at the tail)

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



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

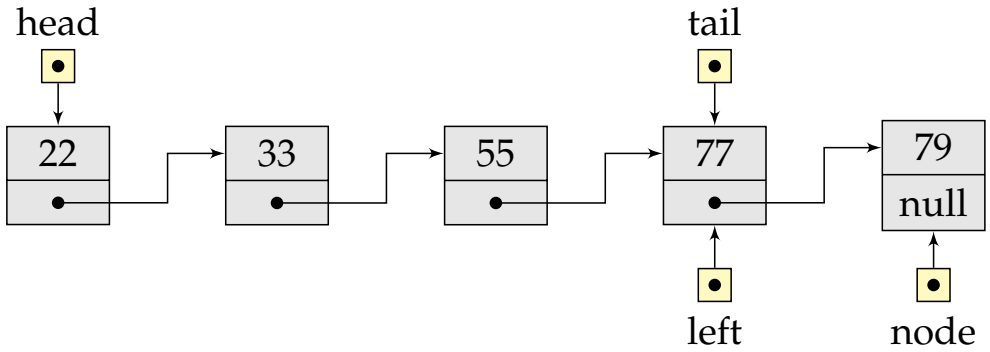
Case 2: `(left->next == 0)`



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

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

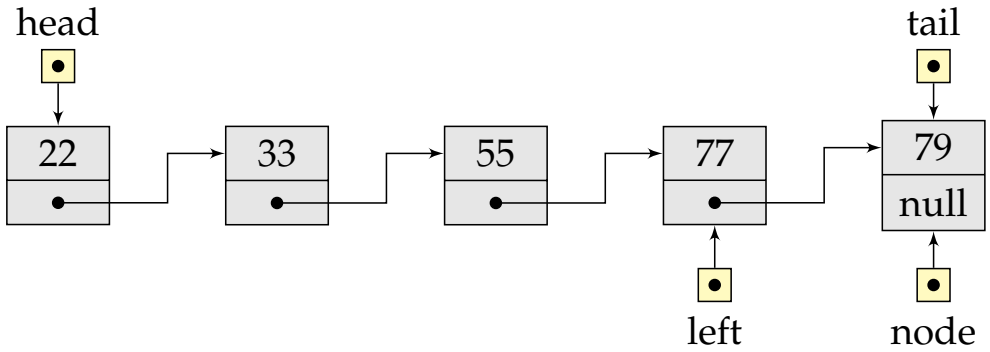
Case 2: `(left->next == 0)`



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

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

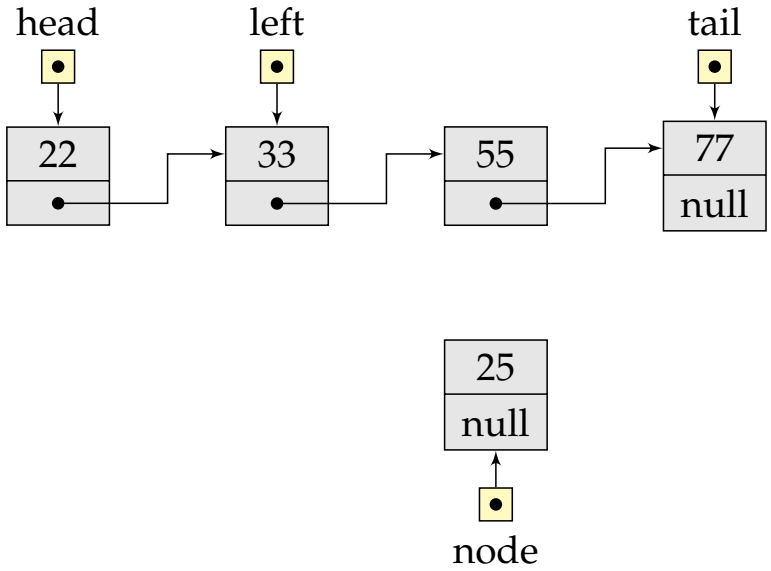
Case 2: `(left->next == 0)`



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

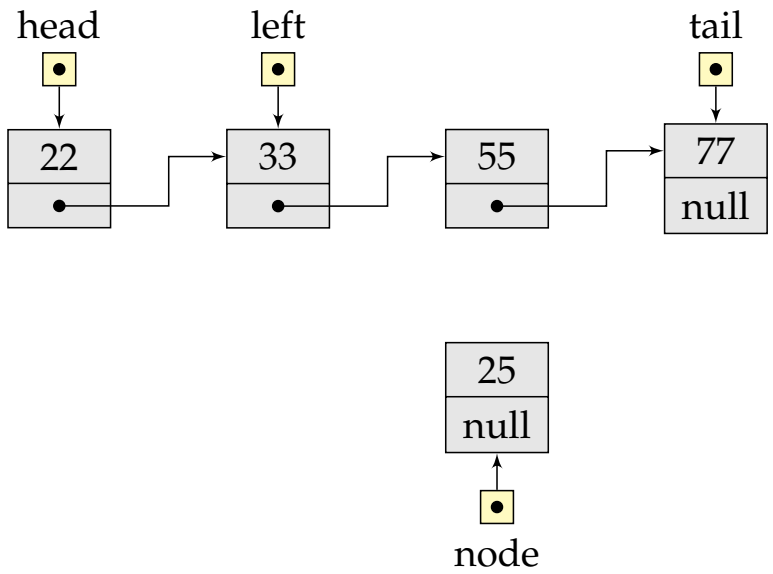
Linked List: InsertNode (case 3: in the middle)

Case 3: Otherwise:



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

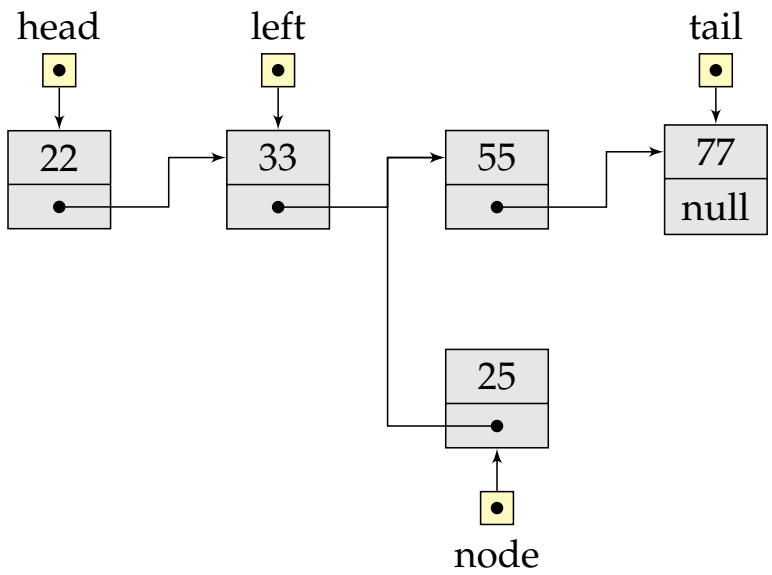
Case 3: Otherwise:



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

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

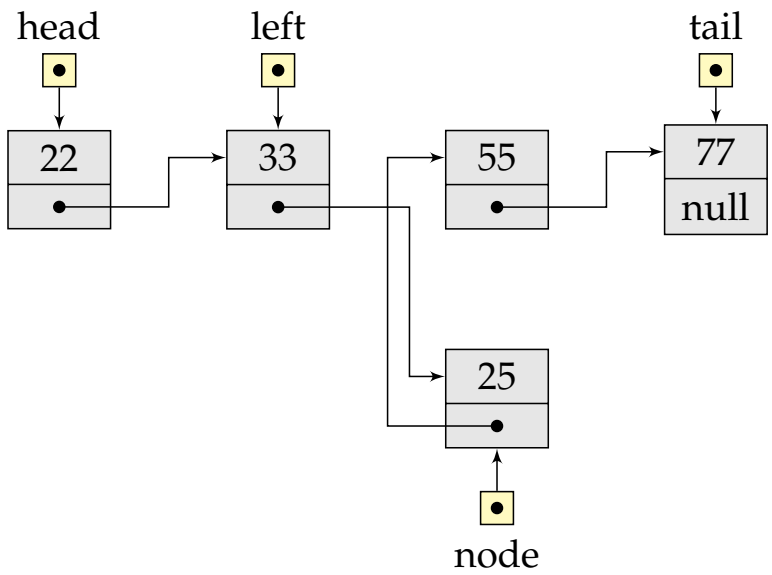
Case 3: Otherwise:



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

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

Case 3: Otherwise:



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

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;
  }
  else {                              /* inserting a node in the middle */
    node->next = left->next;
    left->next = node;
  }
}
```

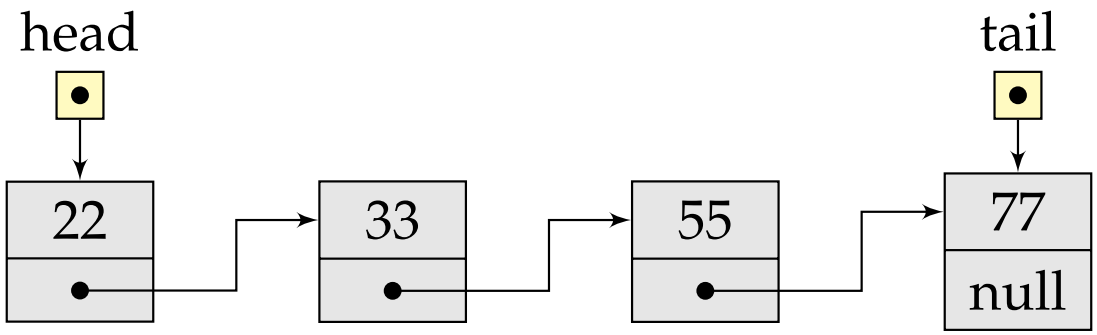
Linked Lists: ADT

Linked List: Inserting a node

Linked List: Deleting a node

Doubly-Linked List

Linked List (Abstract Data Type)

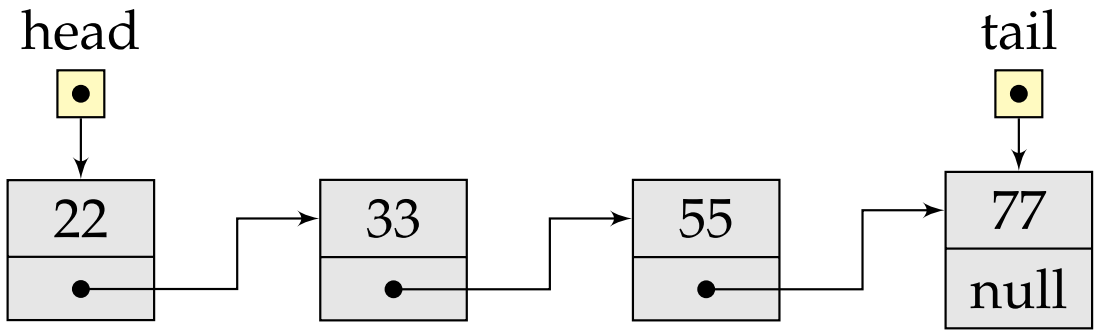



```
class LinkedList {
private:
  Node* head;
  Node* tail;

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*/
};
```

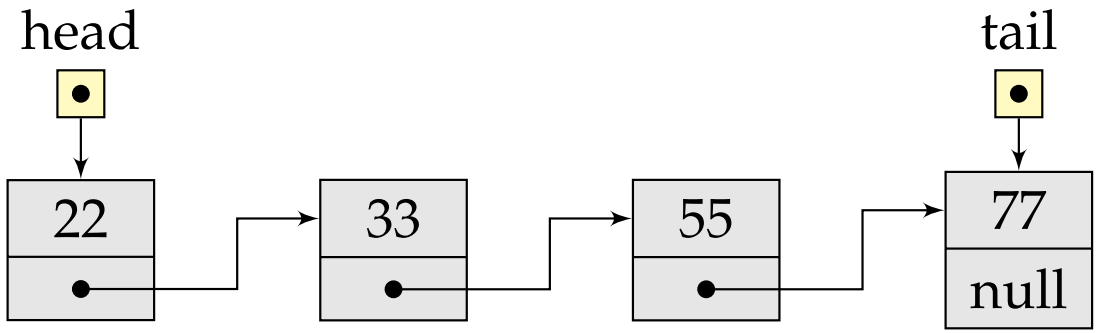
Delete a node in a linked list



Delete Node:

- `int deleteNode(int value)`

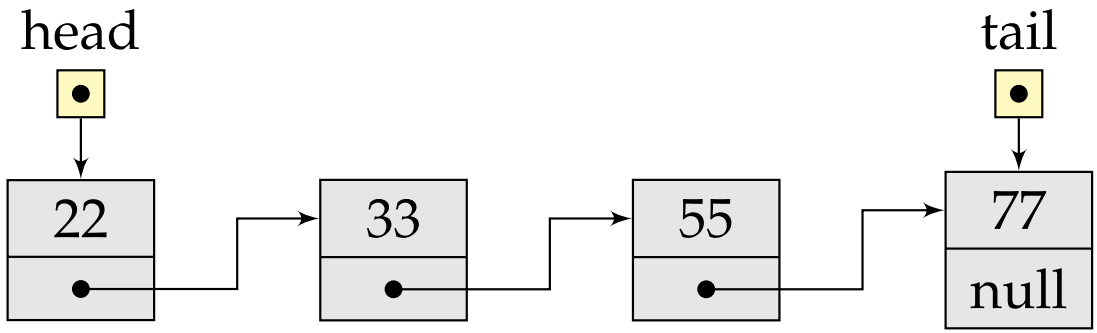
Delete a node in a linked list



Delete Node:

- `int deleteNode(int value)`
- If there is a node having the value equal to "value", delete first such occurrence.

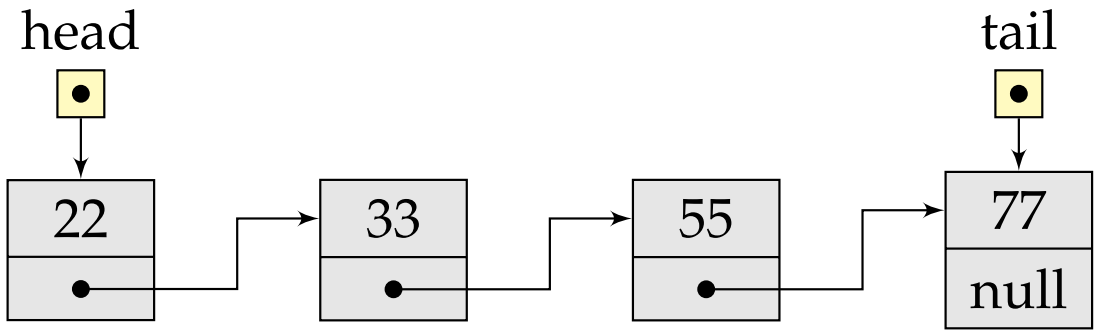
Delete a node in a linked list



Delete Node:

- `int deleteNode(int value)`
- If there is a node having the value equal to "value", delete first such occurrence.
- If there is no node in the list with value equal to "value", keep the lsit untouched.

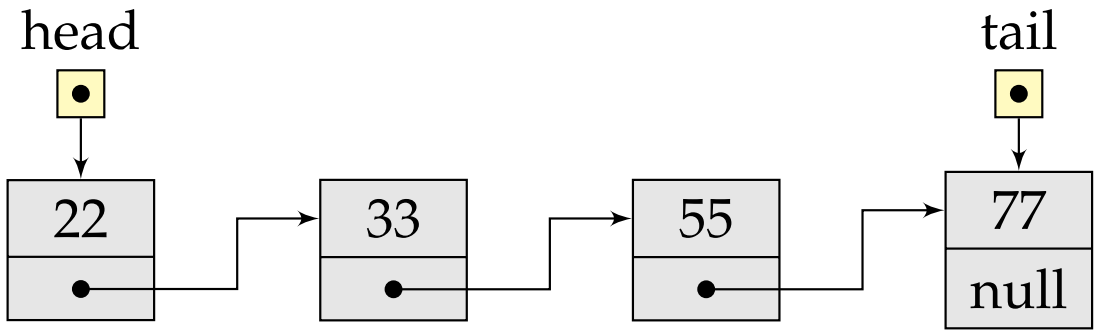
Delete a node in a linked list



Delete Node:

- `int deleteNode(int value)`
- If there is a node having the value equal to "value", delete first such occurrence.
- If there is no node in the list with value equal to "value", keep the lsit untouched.
- **`Node *temp=search(value);`**

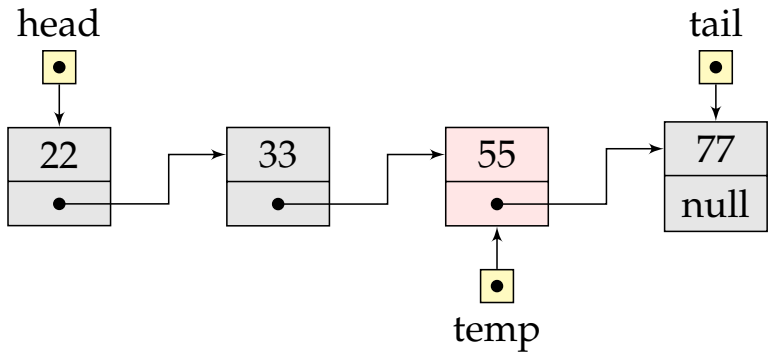
Delete a node in a linked list



Delete Node:

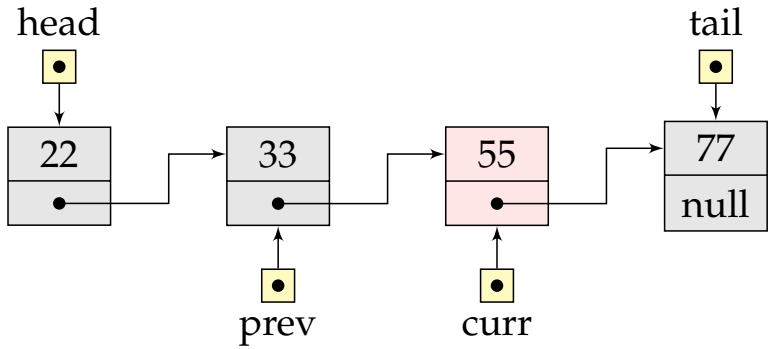
- `int deleteNode(int value)`
- If there is a node having the value equal to "value", delete first such occurrence.
- If there is no node in the list with value equal to "value", keep the lsit untouched.
- **`Node *temp=search(value);`**
- **`if (temp == 0) return; else ``remove Node''`**

Linked List: deleteNode (head node)



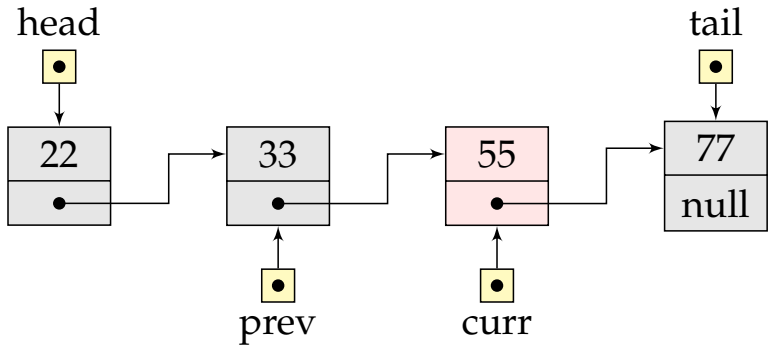
- `Node *temp=search(value);`
- `if (temp == 0) return; else ``remove Node''`

Linked List: deleteNode (head node)



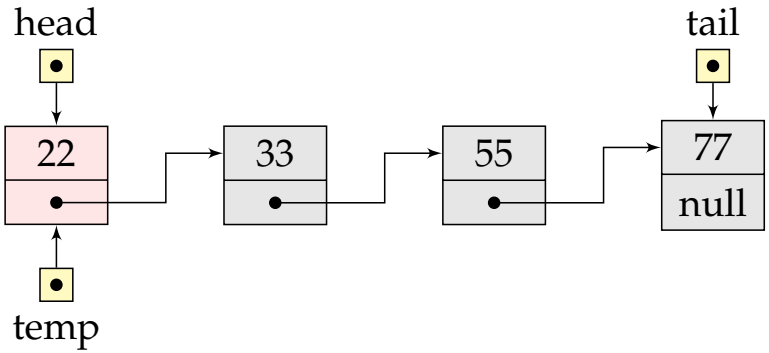

```
prev = curr = head;
while (curr != 0) {
  if (curr == value) carefullyDelete();
  else {prev=curr; curr = curr->next;}
}
```

Linked List: deleteNode (head node)



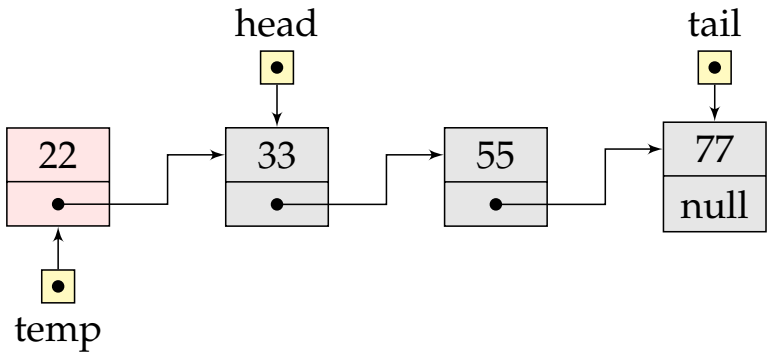
- `prev = curr = head;`
- `while (curr != 0) {`
- `  if (curr == value) carefullyDelete();`
- `  else {prev=curr; curr = curr->next;}`
- `}`
- Three cases:
  1. `curr = head`
  2. `curr = tail`
  3. `otherwise.`

Linked List: deleteNode (head node)



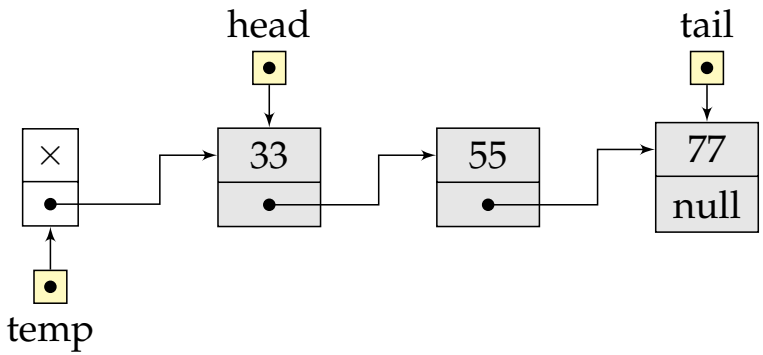

```
1. if (head->data == value) {
2.     Node *temp = head;
3.     head = head->next;
4.     delete temp;
5. }
```

Linked List: deleteNode (head node)



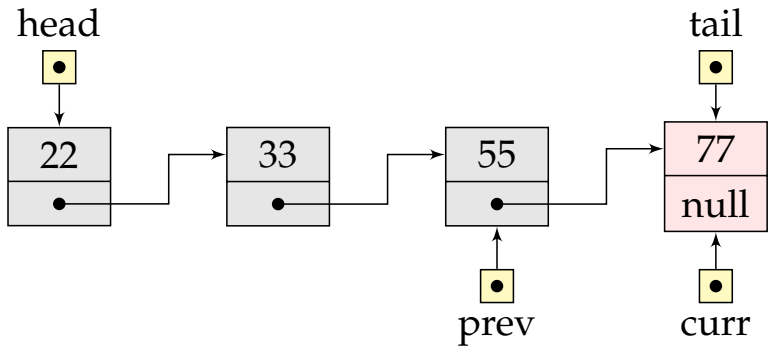

```
1. if (head->data == value) {
2.     Node *temp = head;
3.     head = head->next;
4.     delete temp;
5. }
```

Linked List: deleteNode (head node)



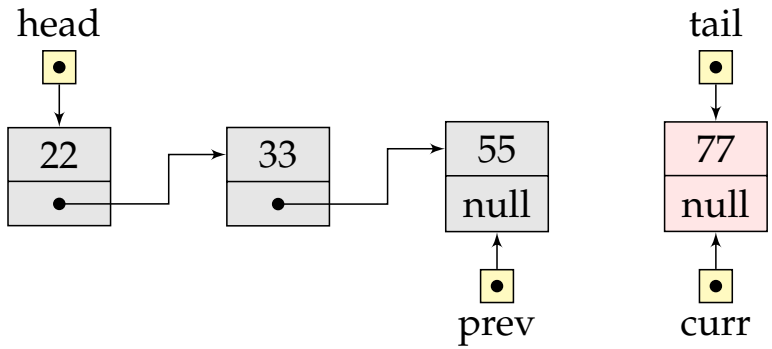

```
1. if (head->data == value) {
2.     Node *temp = head;
3.     head = head->next;
4.     delete temp;
5. }
```

Linked List: deleteNode (last node)



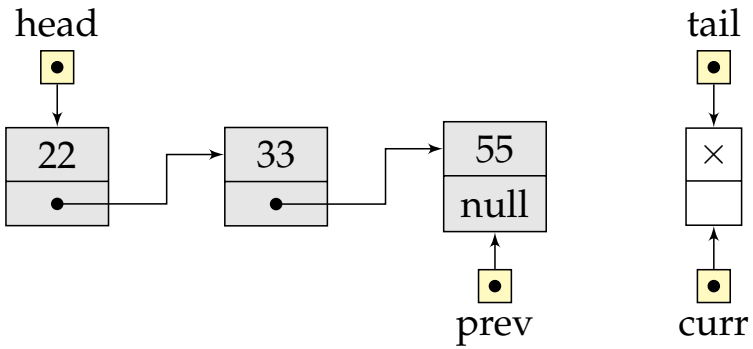

```
1. if ((curr->data == value) && (curr->next = 0)) {
2.     prev->next = 0;
3.     delete curr;
4.     tail = prev;
5. }
```

Linked List: deleteNode (last node)



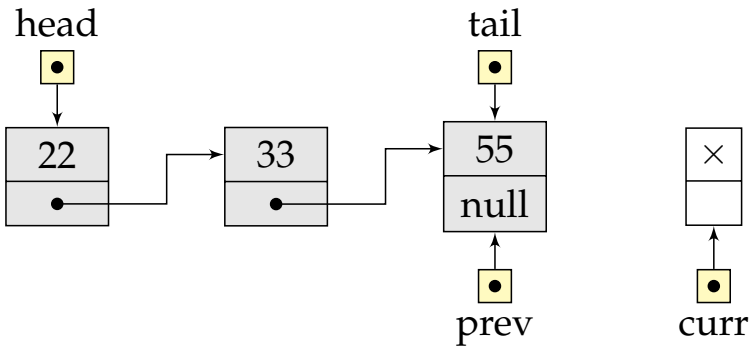

```
1. if ((curr->data == value) && (curr->next = 0)) {
2.     prev->next = 0;
3.     delete curr;
4.     tail = prev;
5. }
```

Linked List: deleteNode (last node)



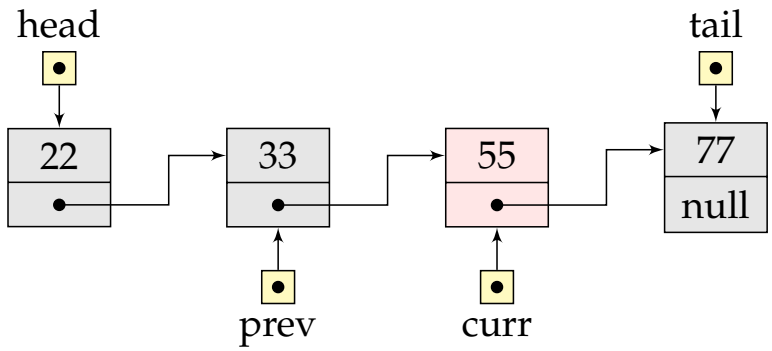

```
1. if ((curr->data == value) && (curr->next = 0)) {
2.     prev->next = 0;
3.     delete curr;
4.     tail = prev;
5. }
```

Linked List: deleteNode (last node)



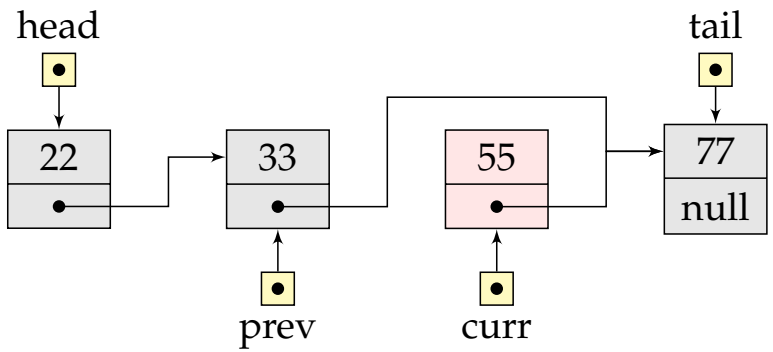

```
1. if ((curr->data == value) && (curr->next = 0)) {
2.     prev->next = 0;
3.     delete curr;
4.     tail = prev;
5. }
```

Linked List: deleteNode (middle node)



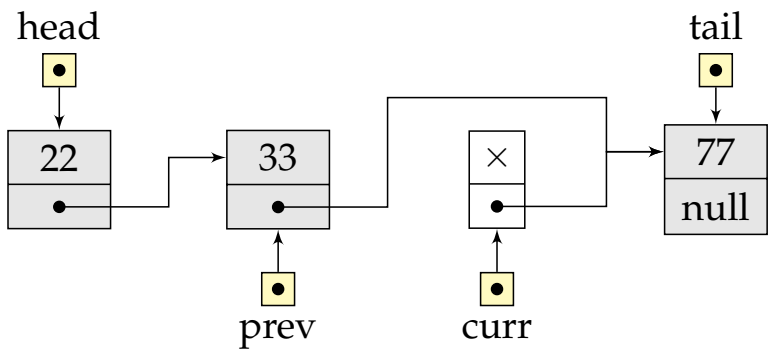

```
1. if (curr->data == value) {
2.     prev->next = curr->next;
3.     delete curr;
4. }
```

Linked List: deleteNode (middle node)




```
1. if (curr->data == value) {
2.     prev->next = curr->next;
3.     delete curr;
4. }
```

Linked List: deleteNode (middle node)




```
1. if (curr->data == value) {
2.     prev->next = curr->next;
3.     delete curr;
4. }
```

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

Common Pitfall

- Memory leaks!
- Portion of lists are lost!

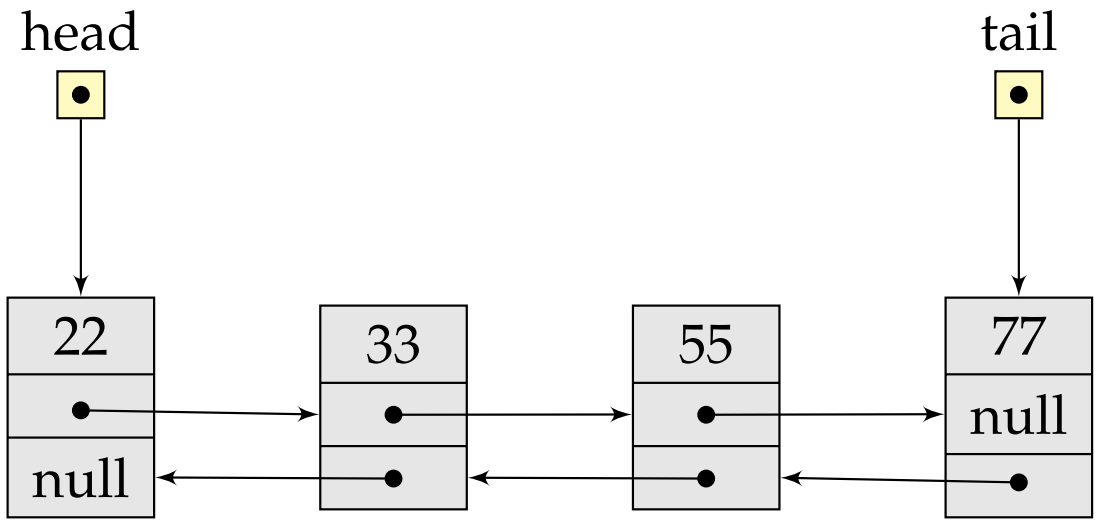
Linked Lists: ADT

Linked List: Inserting a node

Linked List: Deleting a node

Doubly-Linked List

Doubly Linked Lists



- Apart from "deleteNode" all other operations are similar.
- You need to keep track of both previous and next pointers.
- Delete operation is significantly simpler for the tail node!