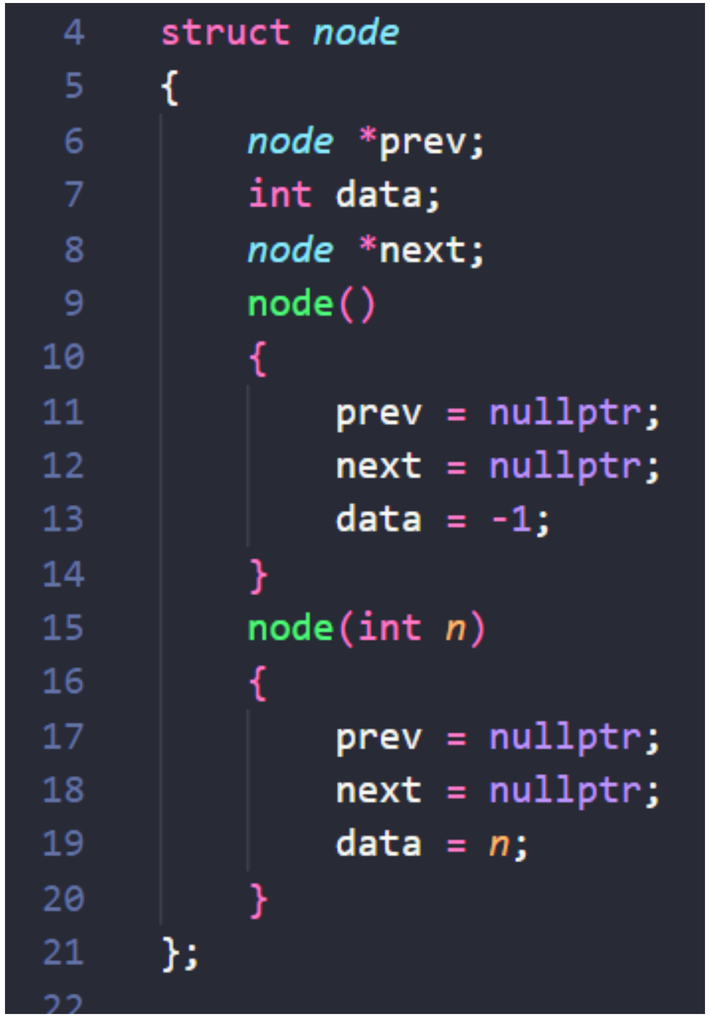
Doubly Linked Lists

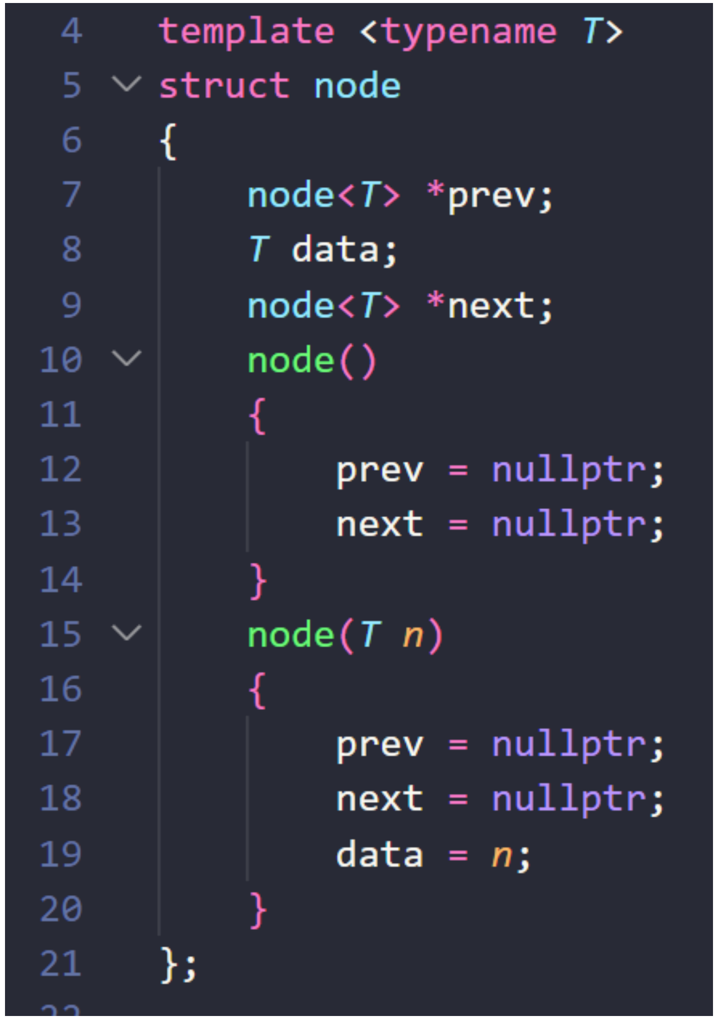
Insertion

Class Structure

- Class is nearly identical to Singly-Linked List
- Changes to node struct
    - Add node* prev pointer
- Templates help to create an Abstract Data Type (ADT)
    - Every method/class/struct must have template declaration
    - Replace every node with node<T>



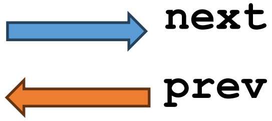
```
struct node
{
    node *prev;
    int data;
    node *next;
    node()
    {
        prev = nullptr;
        next = nullptr;
        data = -1;
    }
    node(int n)
    {
        prev = nullptr;
        next = nullptr;
        data = n;
    }
};
```

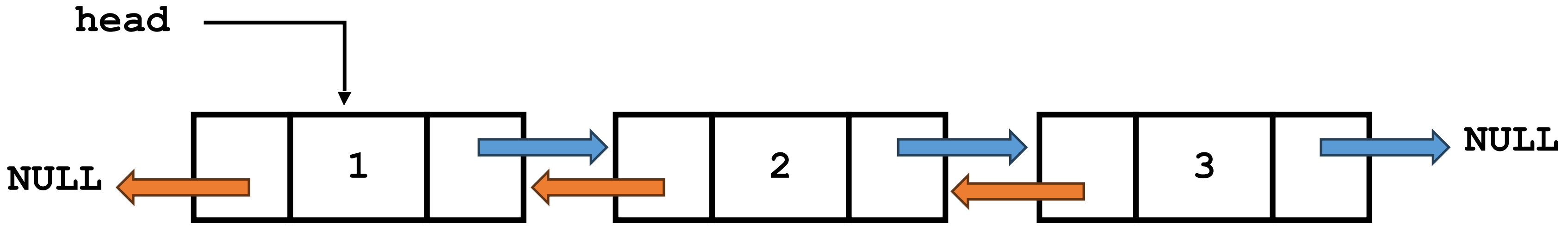


```
template <typename T>
struct node
{
    node<T> *prev;
    T data;
    node<T> *next;
    node()
    {
        prev = nullptr;
        next = nullptr;
    }
    node(T n)
    {
        prev = nullptr;
        next = nullptr;
        data = n;
    }
};
```

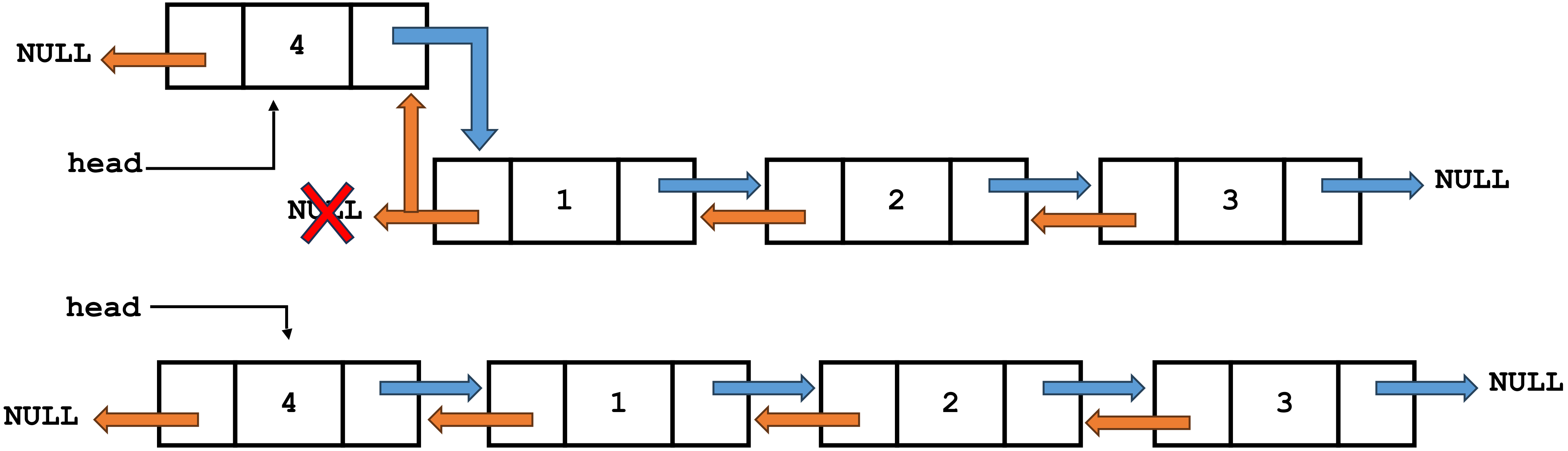
Insert at head

Pointers

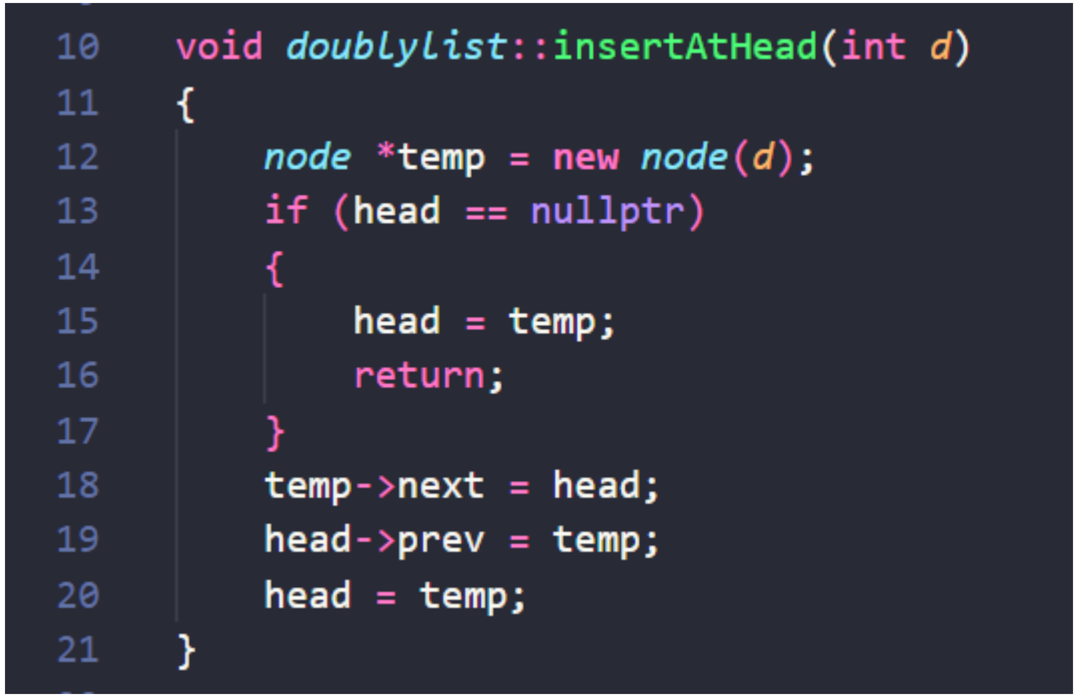




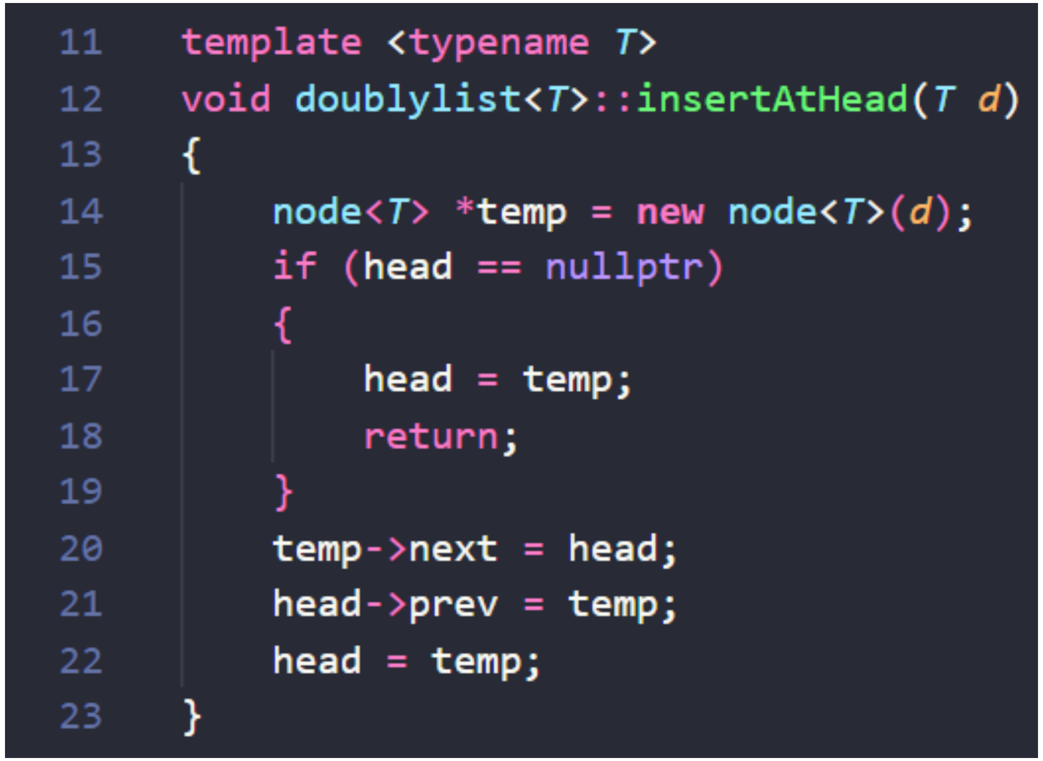
Insert node 4



Insert at head

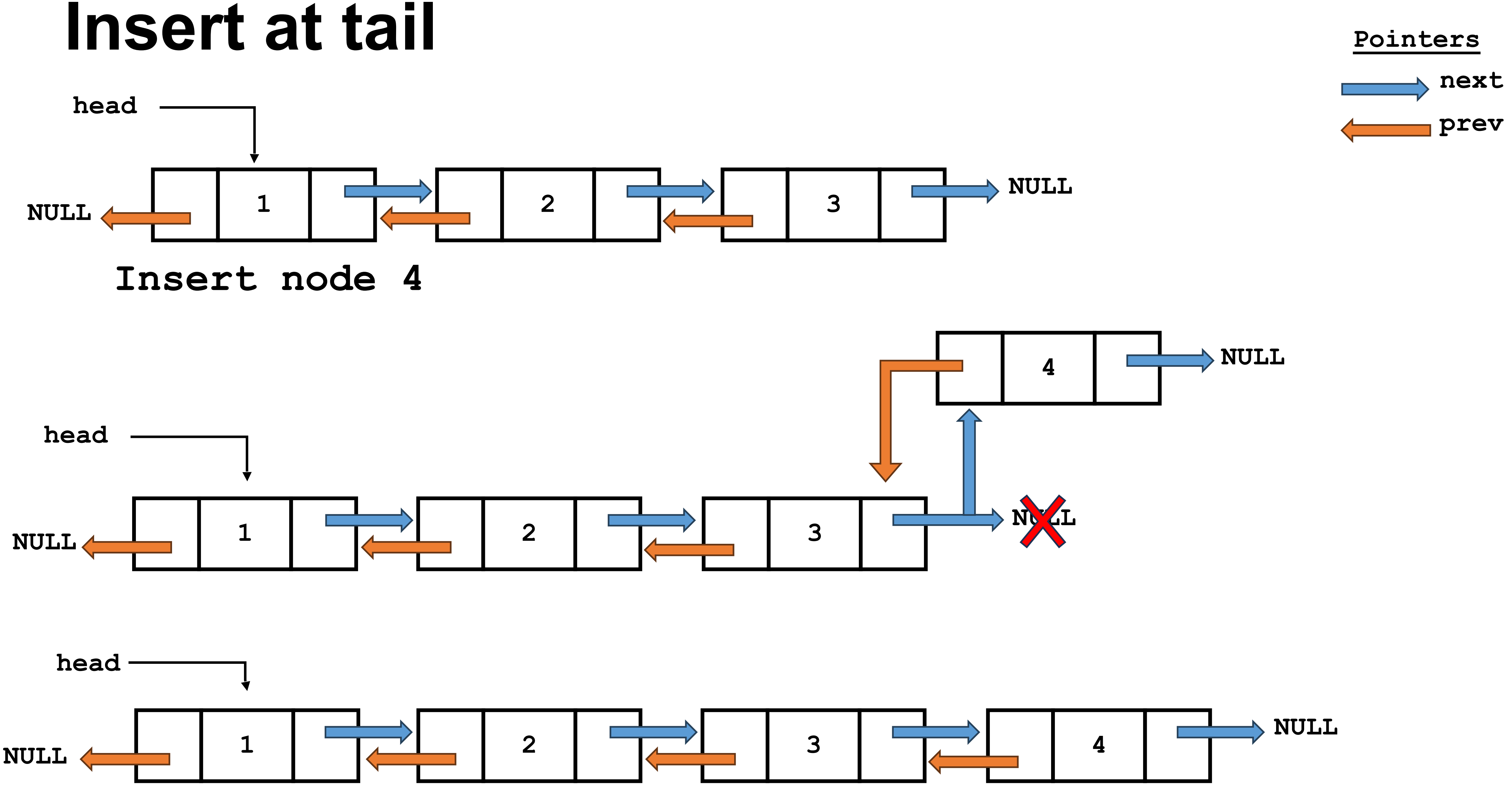


```cpp
void doublylist::insertAtHead(int d)
{
    node *temp = new node(d);
    if (head == nullptr)
    {
        head = temp;
        return;
    }
    temp->next = head;
    head->prev = temp;
    head = temp;
}
```



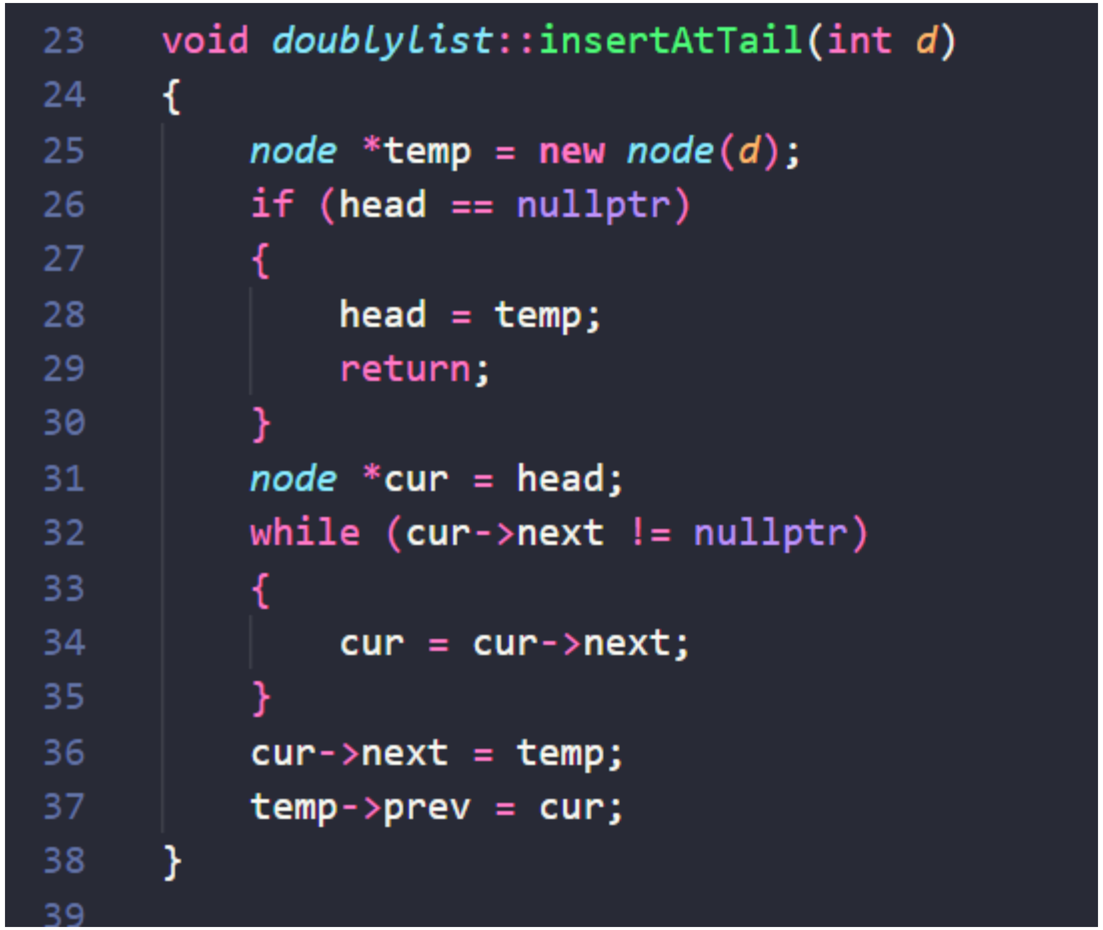
```cpp
template <typename T>
void doublylist<T>::insertAtHead(T d)
{
    node<T> *temp = new node<T>(d);
    if (head == nullptr)
    {
        head = temp;
        return;
    }
    temp->next = head;
    head->prev = temp;
    head = temp;
}
```

Insert at tail

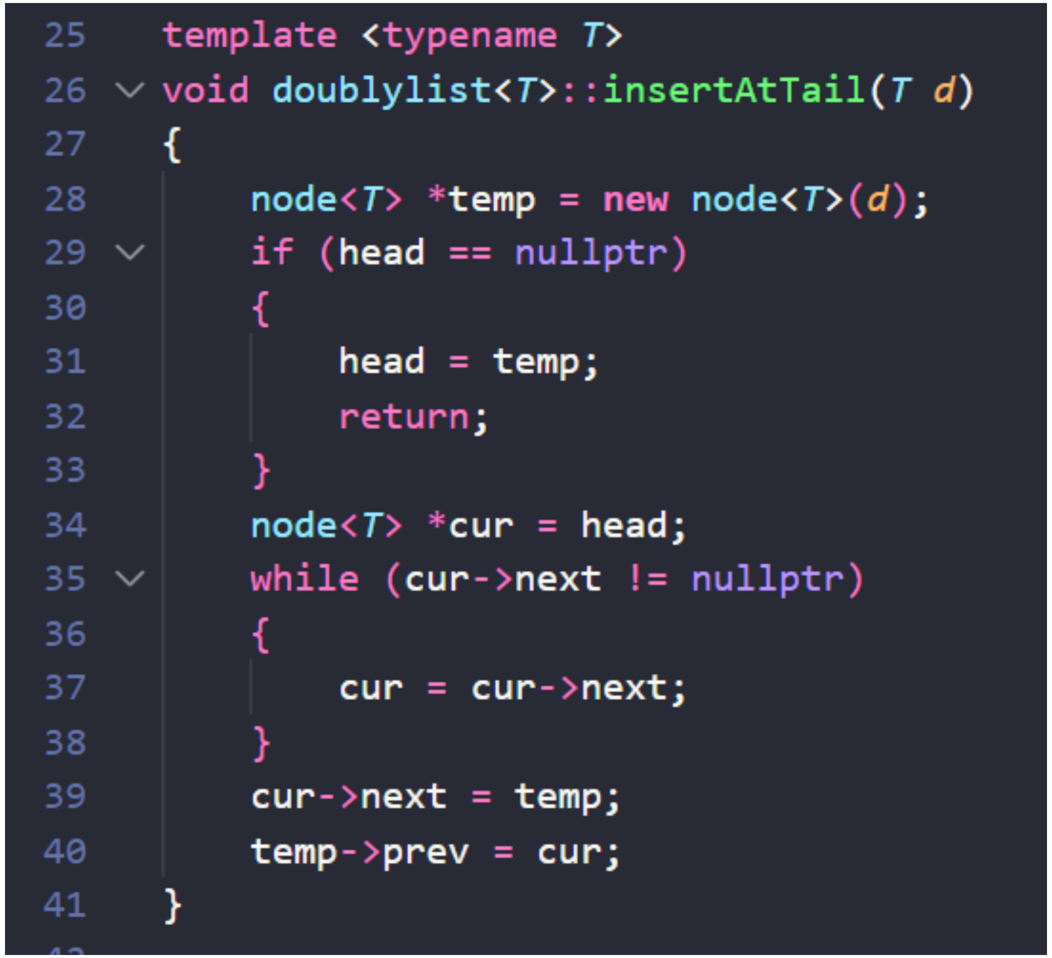


Insert node 4

Insert at tail



```cpp
void doublylist::insertAtTail(int d)
{
    node *temp = new node(d);
    if (head == nullptr)
    {
        head = temp;
        return;
    }
    node *cur = head;
    while (cur->next != nullptr)
    {
        cur = cur->next;
    }
    cur->next = temp;
    temp->prev = cur;
}
```

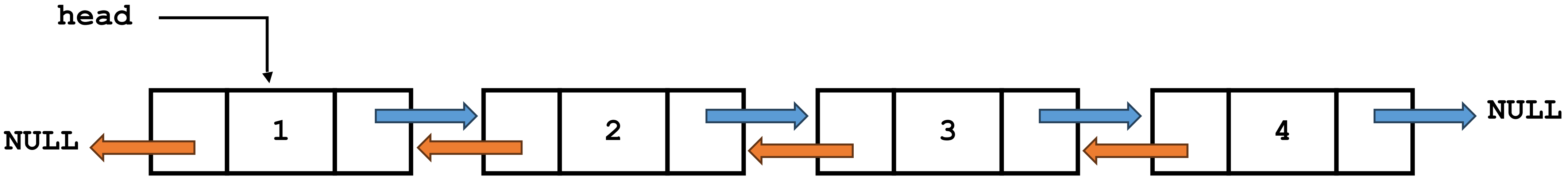


```cpp
template <typename T>
void doublylist<T>::insertAtTail(T d)
{
    node<T> *temp = new node<T>(d);
    if (head == nullptr)
    {
        head = temp;
        return;
    }
    node<T> *cur = head;
    while (cur->next != nullptr)
    {
        cur = cur->next;
    }
    cur->next = temp;
    temp->prev = cur;
}
```

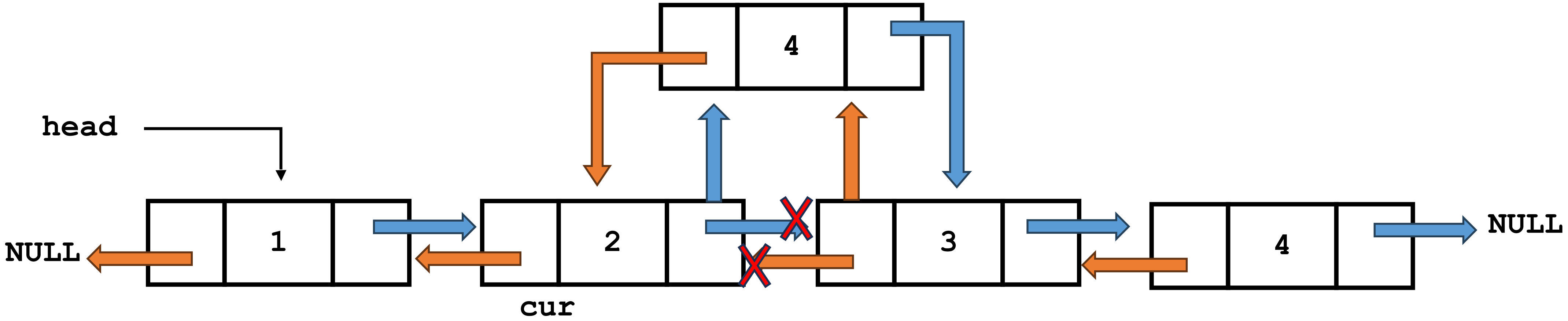
Insert at index

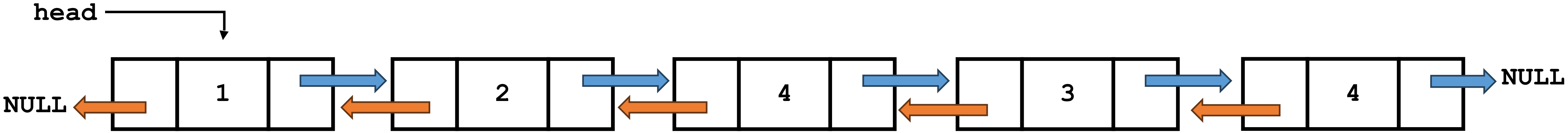
```
1. If index = 0 (or less)
     Insert at head
2. If index = size (or more)
     Insert at tail
3. Else
     Insert at index
```

Insert at index

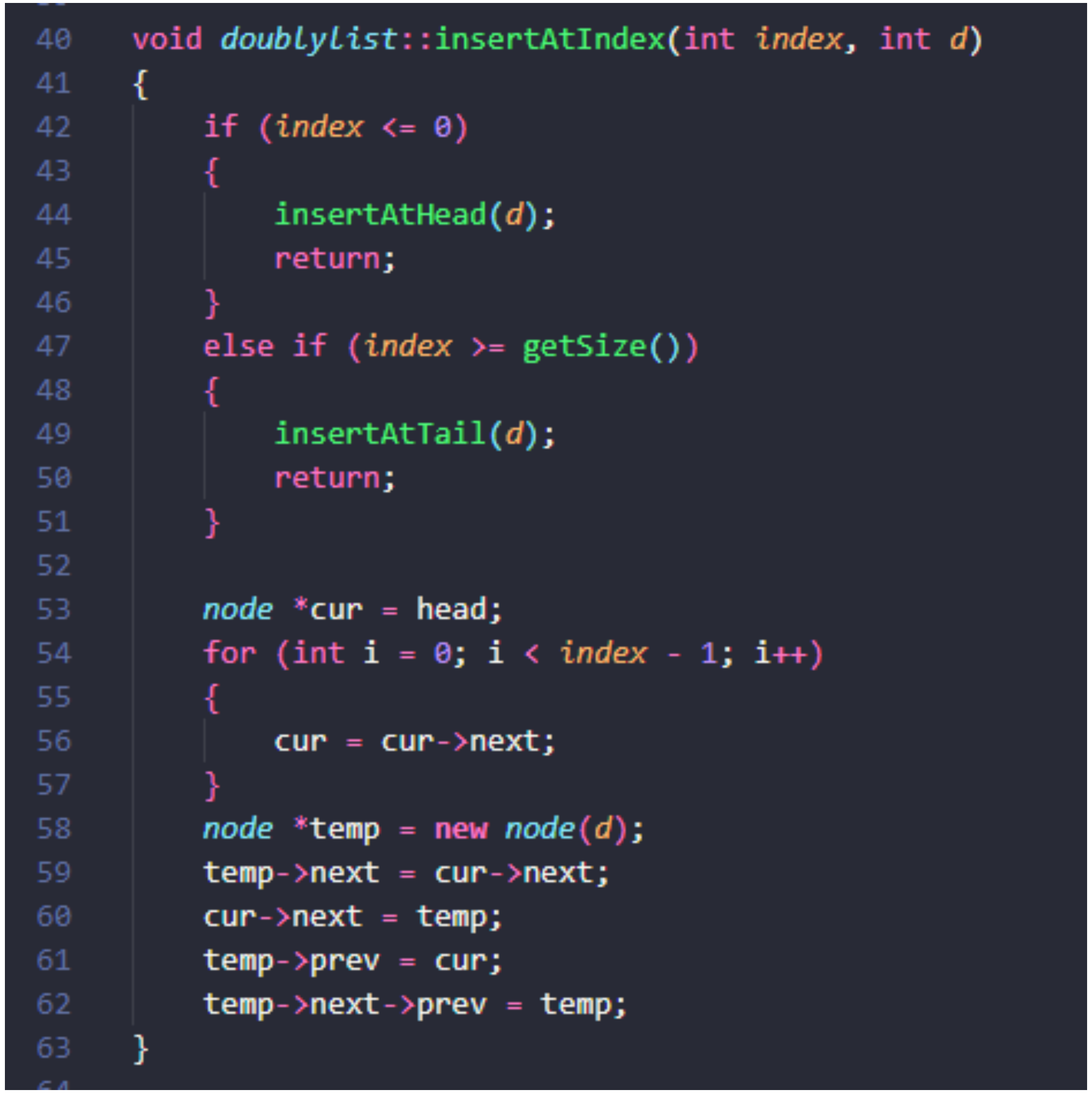


Insert node 4 at index 2

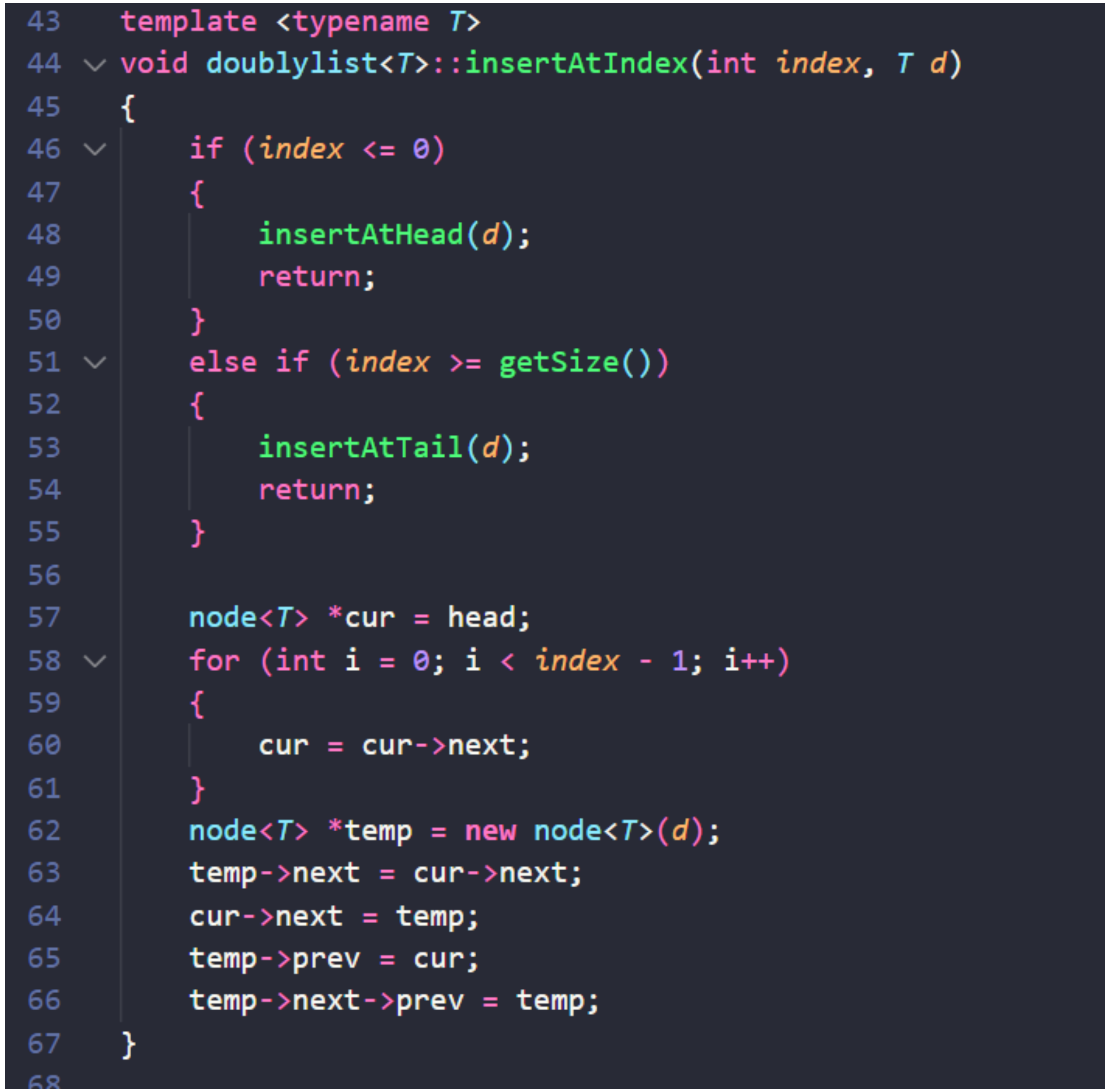




Insert at index

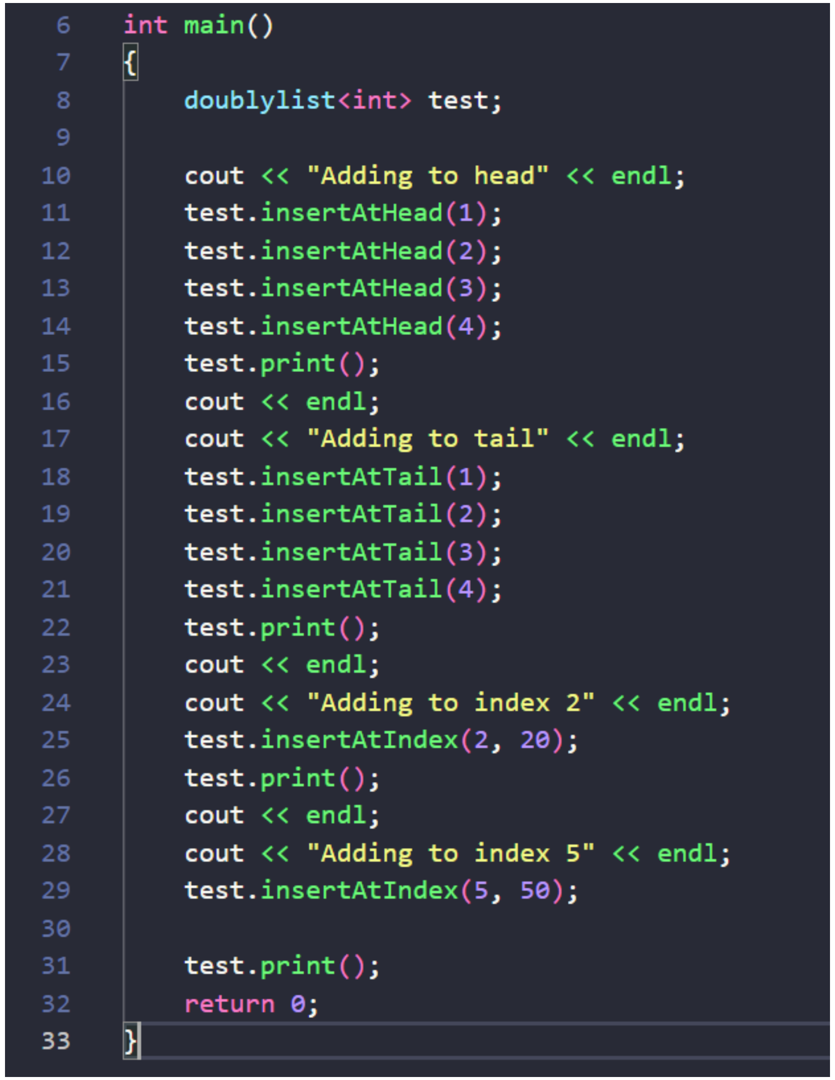


```cpp
void doublylist::insertAtIndex(int index, int d)
{
    if (index <= 0)
    {
        insertAtHead(d);
        return;
    }
    else if (index >= getSize())
    {
        insertAtTail(d);
        return;
    }

    node *cur = head;
    for (int i = 0; i < index - 1; i++)
    {
        cur = cur->next;
    }
    node *temp = new node(d);
    temp->next = cur->next;
    cur->next = temp;
    temp->prev = cur;
    temp->next->prev = temp;
}
```



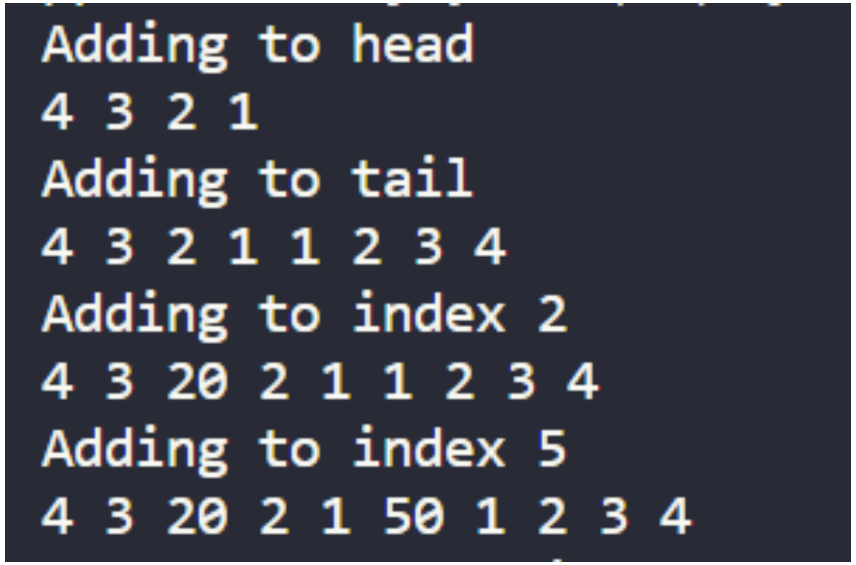
```cpp
template <typename T>
void doublylist<T>::insertAtIndex(int index, T d)
{
    if (index <= 0)
    {
        insertAtHead(d);
        return;
    }
    else if (index >= getSize())
    {
        insertAtTail(d);
        return;
    }

    node<T> *cur = head;
    for (int i = 0; i < index - 1; i++)
    {
        cur = cur->next;
    }
    node<T> *temp = new node<T>(d);
    temp->next = cur->next;
    cur->next = temp;
    temp->prev = cur;
    temp->next->prev = temp;
}
```

Demo



```cpp
int main()
{
    doublylist<int> test;

    cout << "Adding to head" << endl;
    test.insertAtHead(1);
    test.insertAtHead(2);
    test.insertAtHead(3);
    test.insertAtHead(4);
    test.print();
    cout << endl;
    cout << "Adding to tail" << endl;
    test.insertAtTail(1);
    test.insertAtTail(2);
    test.insertAtTail(3);
    test.insertAtTail(4);
    test.print();
    cout << endl;
    cout << "Adding to index 2" << endl;
    test.insertAtIndex(2, 20);
    test.print();
    cout << endl;
    cout << "Adding to index 5" << endl;
    test.insertAtIndex(5, 50);

    test.print();
    return 0;
}
```

Output:

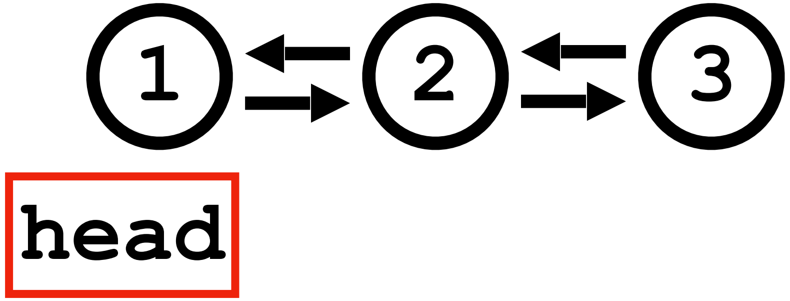


```
Adding to head
4 3 2 1
Adding to tail
4 3 2 1 1 2 3 4
Adding to index 2
4 3 20 2 1 1 2 3 4
Adding to index 5
4 3 20 2 1 50 1 2 3 4
```

Doubly Linked List

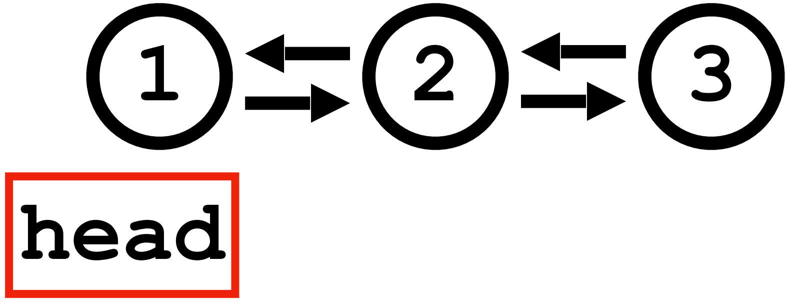
Deletion

Remove Head



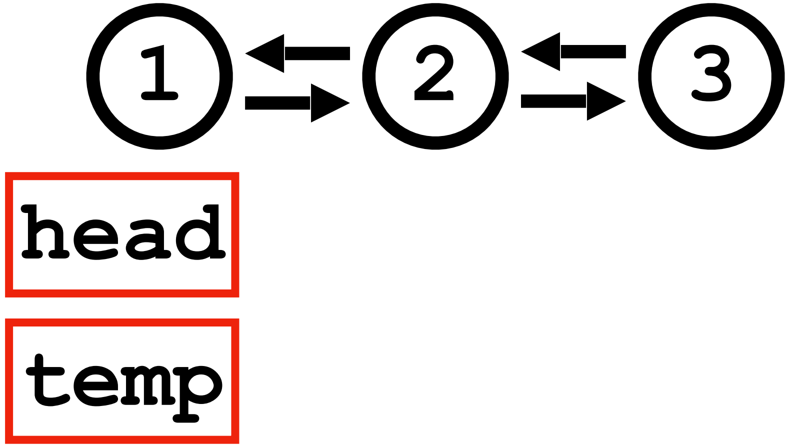
Step 1

Check if list is empty. If it is, return.



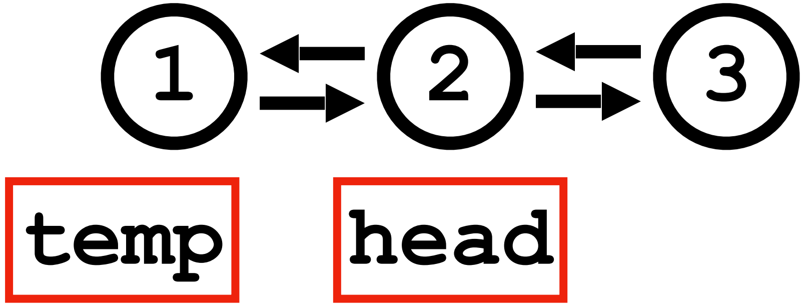
Step 2

Create a temporary pointer and set it to head.



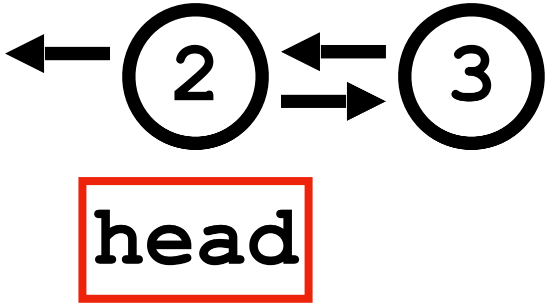
Step 3

Set head to head’s next pointer.



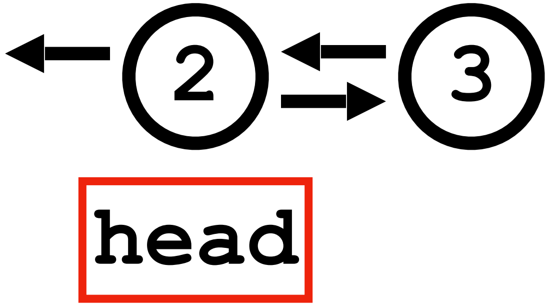
Step 4

Delete temp (which is pointing to the old head).



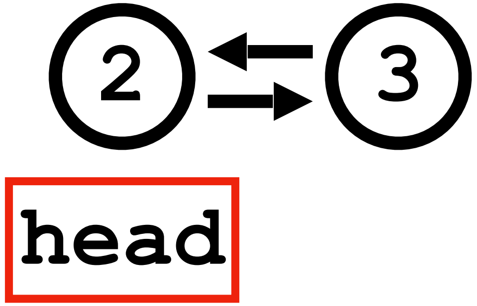
Step 5

Check if the new head is nullptr.



Step 6

Set head's previous pointer to nullptr.



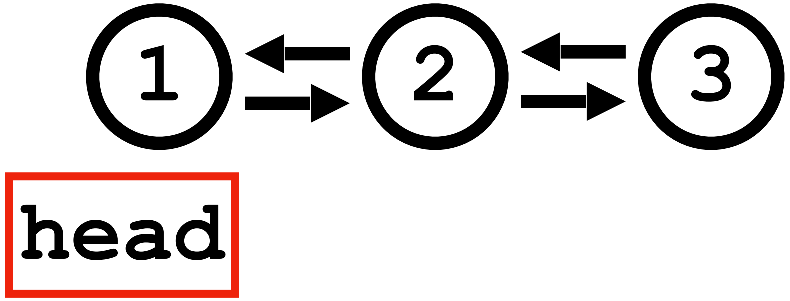
Remove Head - Code

```
void doublylist::removeHead(){
  if(head == nullptr)
    return;
  node *temp = head;
  head = head->next;
  delete temp;
  if(head != nullptr)
    head->prev = nullptr;
  return head;
}
```

Remove Head - Code with template

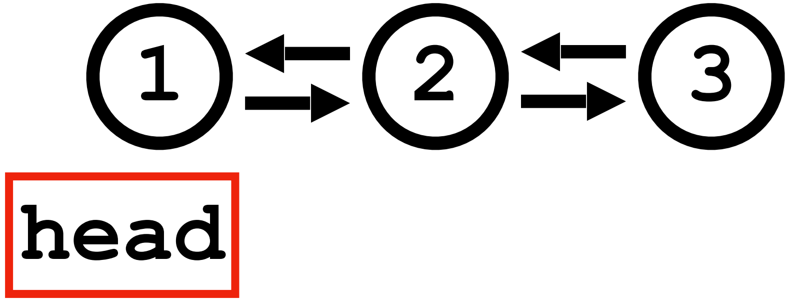
```
template <typename T>
void doublylist::removeHead(){
  if(head == nullptr)
    return;
  node<T> *temp = head;
  head = head->next;
  delete temp;
  if(head != nullptr)
    head->prev = nullptr;
  return head;
}
```

Remove Tail



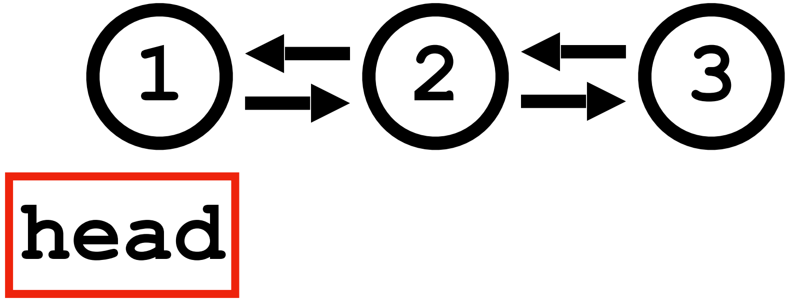
Step 1

Check if list is empty. If it is, return.



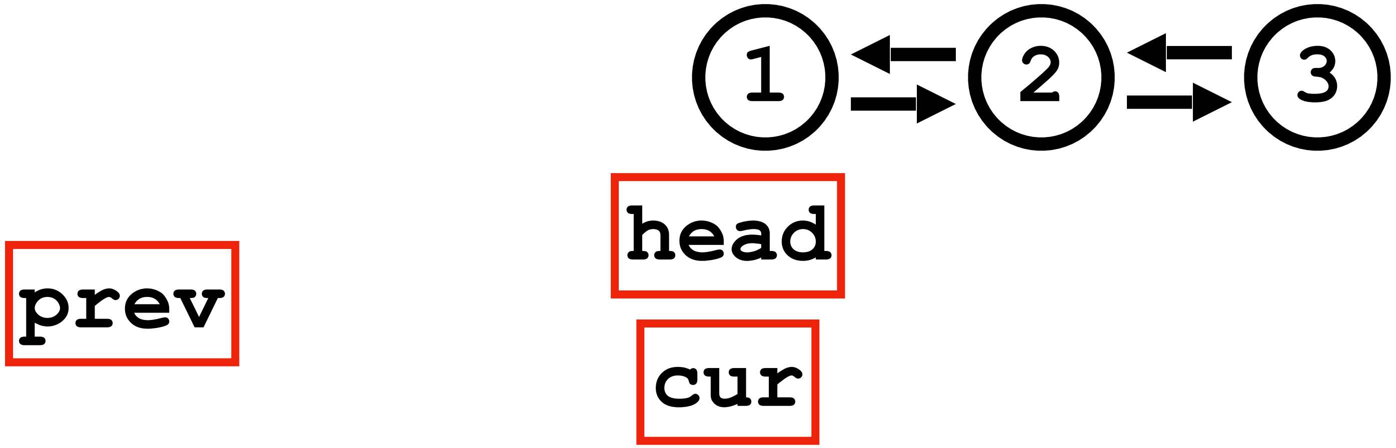
Step 2

Check if there is only one element in linked list.



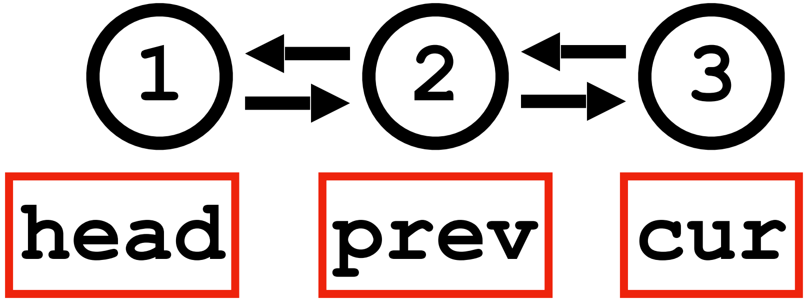
Step 3

Create two node pointers, `prev` and `cur`. Set to `cur` head and `prev` to nullptr.



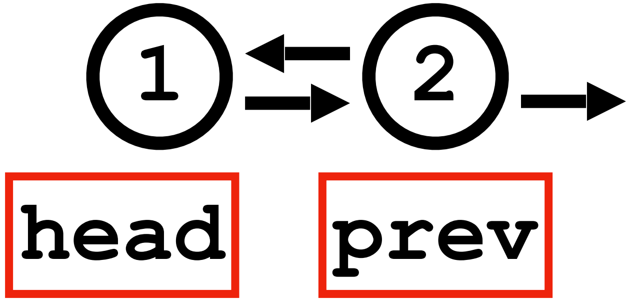
Step 4

Iterate `cur` until `cur`->next == nullptr. Iterate `prev` along with it.



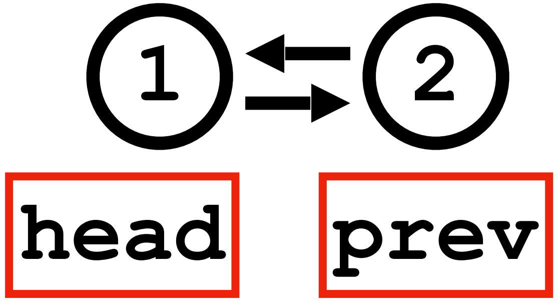
Step 5

Delete cur.



Step 6

Set prev’s next pointer equal to nullptr.



Remove Tail - Code

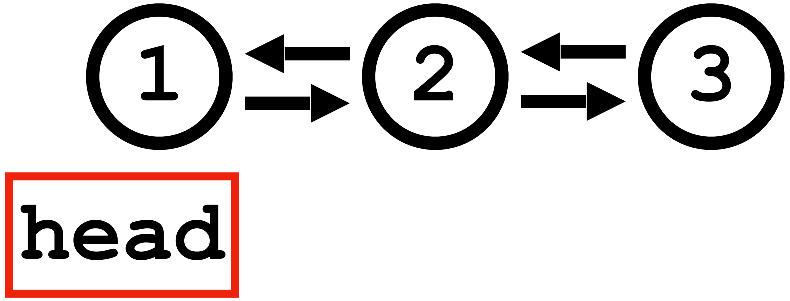
```
void doublylist::removeTail(){
  if(head == nullptr)
    return;
  else if(head->next == nullptr){
    removeHead();
    return;
  }
  node *prev = nullptr;
  node *cur = head;
  while(cur->next != nullptr){
    prev = cur;
    cur = cur->next;
  }
  prev->next = nullptr;
  delete cur;
  return head;
}
```

Remove Tail - Code with template

```
template <typename T>
void doublylist::removeTail(){
  if(head == nullptr)
    return;
  else if(head->next == nullptr){
    removeHead();
    return;
  }
  node<T> *prev = nullptr;
  node<T> *cur = head;
  while(cur->next != nullptr){
    prev = cur;
    cur = cur->next;
  }
  prev->next = nullptr;
  delete cur;
  return head;
}
```

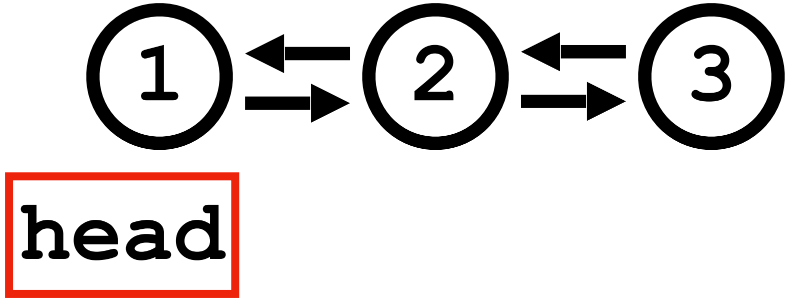
Remove At Position

Position is 1.



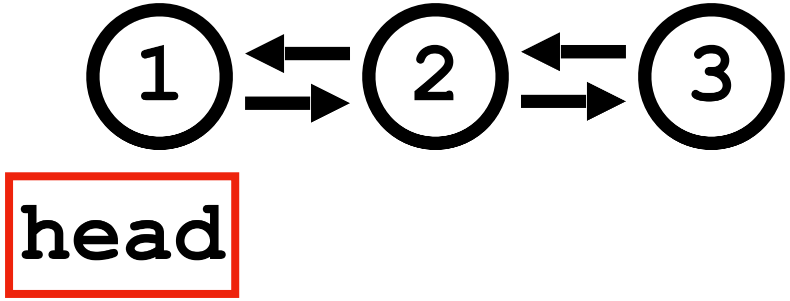
Step 1

Check if list is empty. If it is, return.



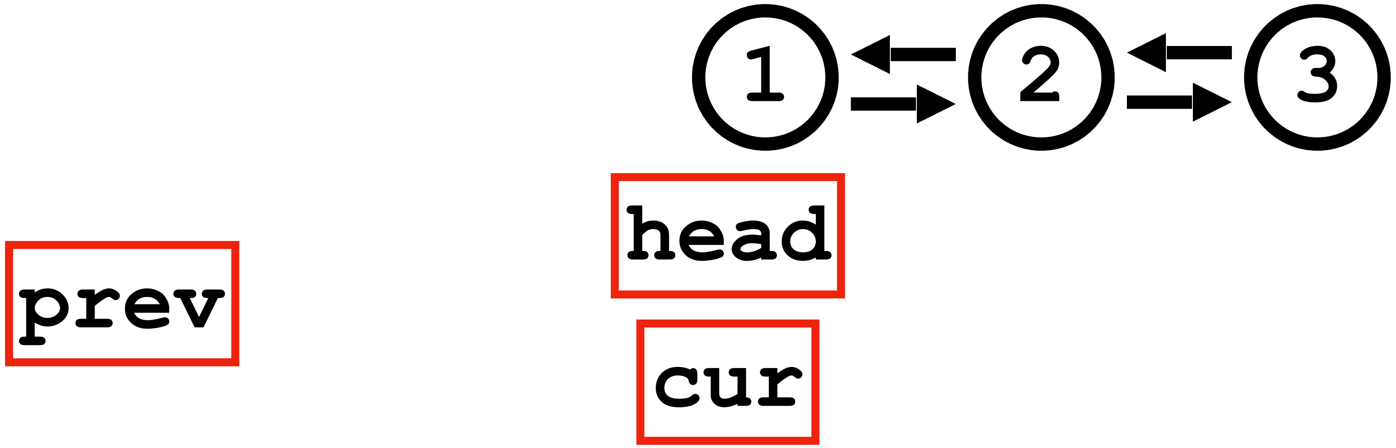
Step 2

Check if position is 0.



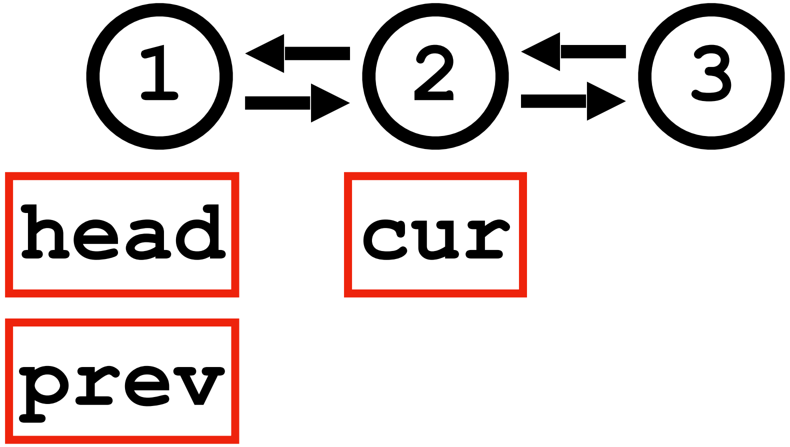
Step 3

Create two node pointers, `prev` and `cur`. Set to `cur` head and `prev` to nullptr.



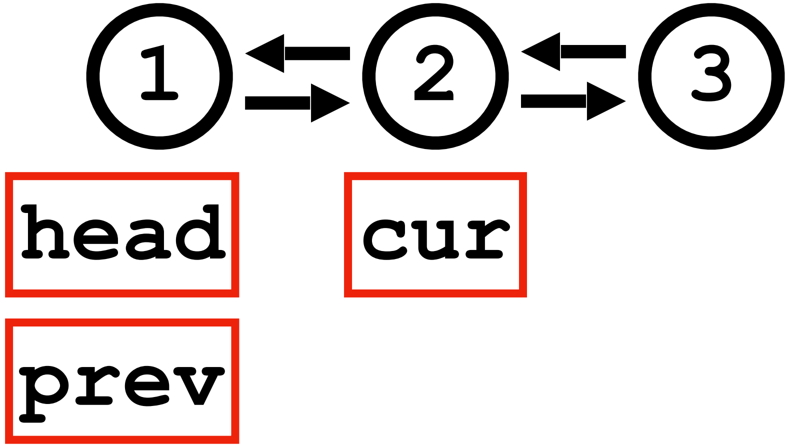
Step 4

Create a loop and iterate until you reach the position. If cur equals nullptr during the loop, position is invlaid so we return.



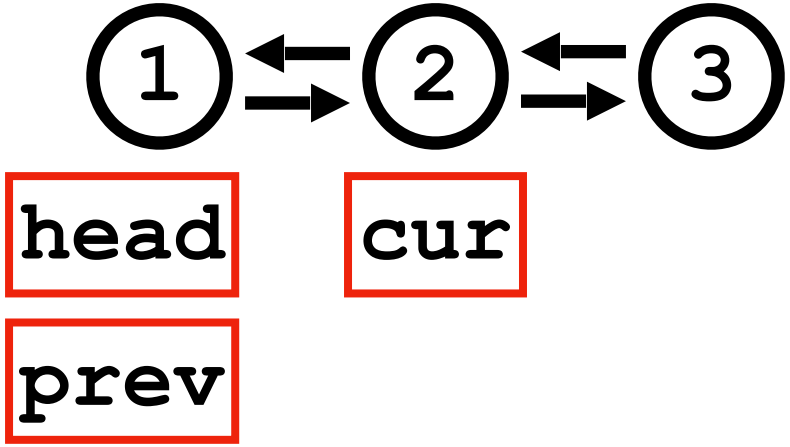
Step 5

Check if cur is nullptr. If it is return.



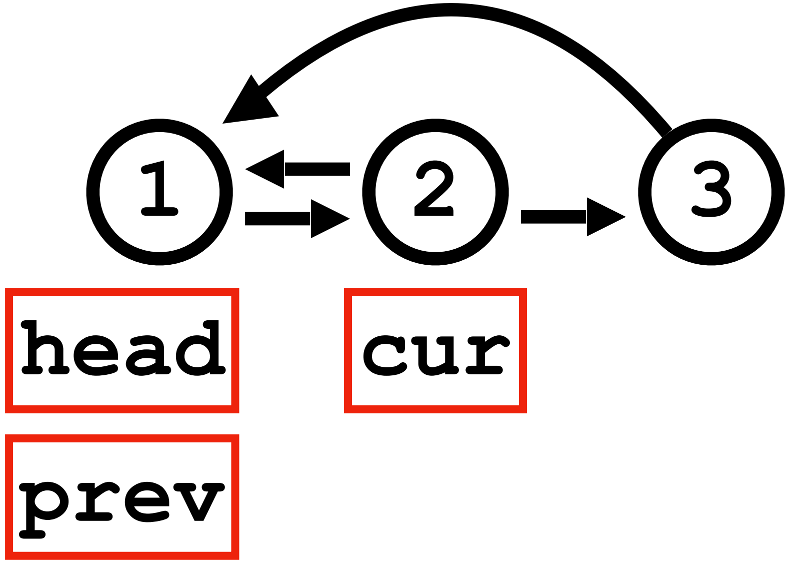
Step 6

Check if cur is tail. This way we don't get a segmentation fault when we access cur->next->prev.



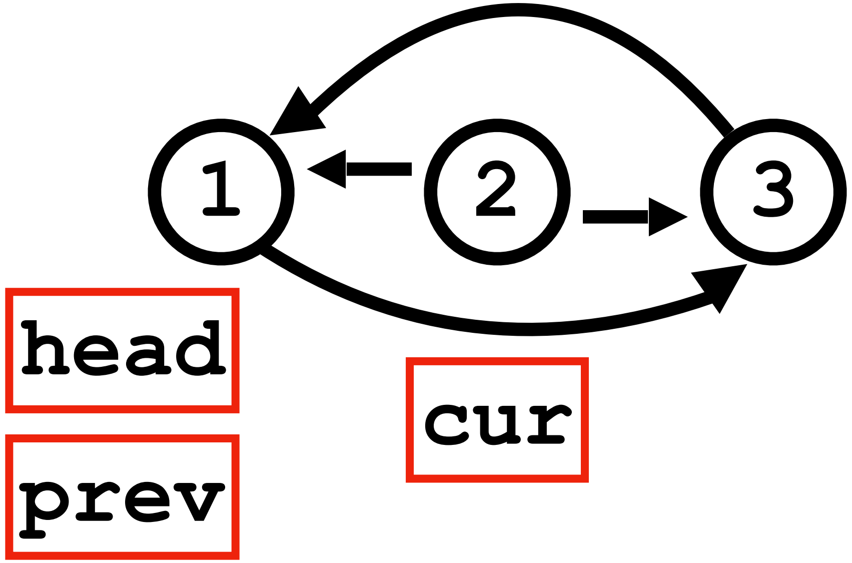
Step 7

Set set cur->next->prev equal to prev.



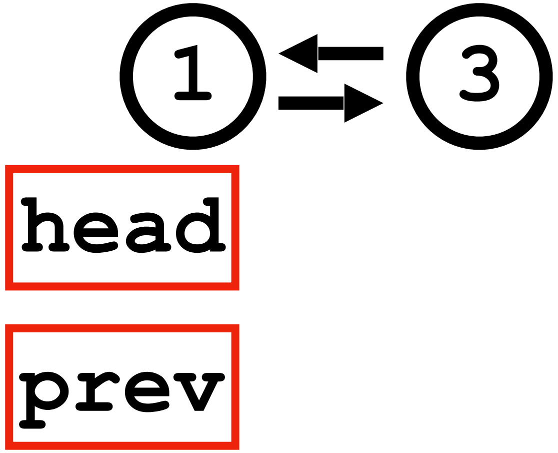
Step 8

Set prev->next equal to cur->next.



Step 9

Delete cur.



Remove At Position - Code

```
void doublylist::removeAtPosition(int pos){
  if(head == nullptr)
    return;
  else if(pos == 0){
    removeAtHead();
    return;
  }
  node *prev = nullptr;
  node *cur = head;
  for(int i = 0; i < pos; i++){
    if(cur == nullptr)
      return;
    prev = cur;
    cur = cur->next;
  }
  if(cur == nullptr)
    return;
  else if(cur->next != nullptr)
    cur->next->prev = prev;
  prev->next = cur->next;
  delete cur;
}
```

Remove At Position - Code with template

```
template <typename T>
void doublylist::removeAtPosition(int pos){
  if(head == nullptr)
    return;
  else if(pos == 0){
    removeAtHead();
    return;
  }
  node<T> *prev = nullptr;
  node<T> *cur = head;
  for(int i = 0; i < pos; i++){
    if(cur == nullptr)
      return;
    prev = cur;
    cur = cur->next;
  }
  if(cur == nullptr)
    return;
  else if(cur->next != nullptr)
    cur->next->prev = prev;
  prev->next = cur->next;
  delete cur;
}
```