Doubly Linked Lists

Insertion

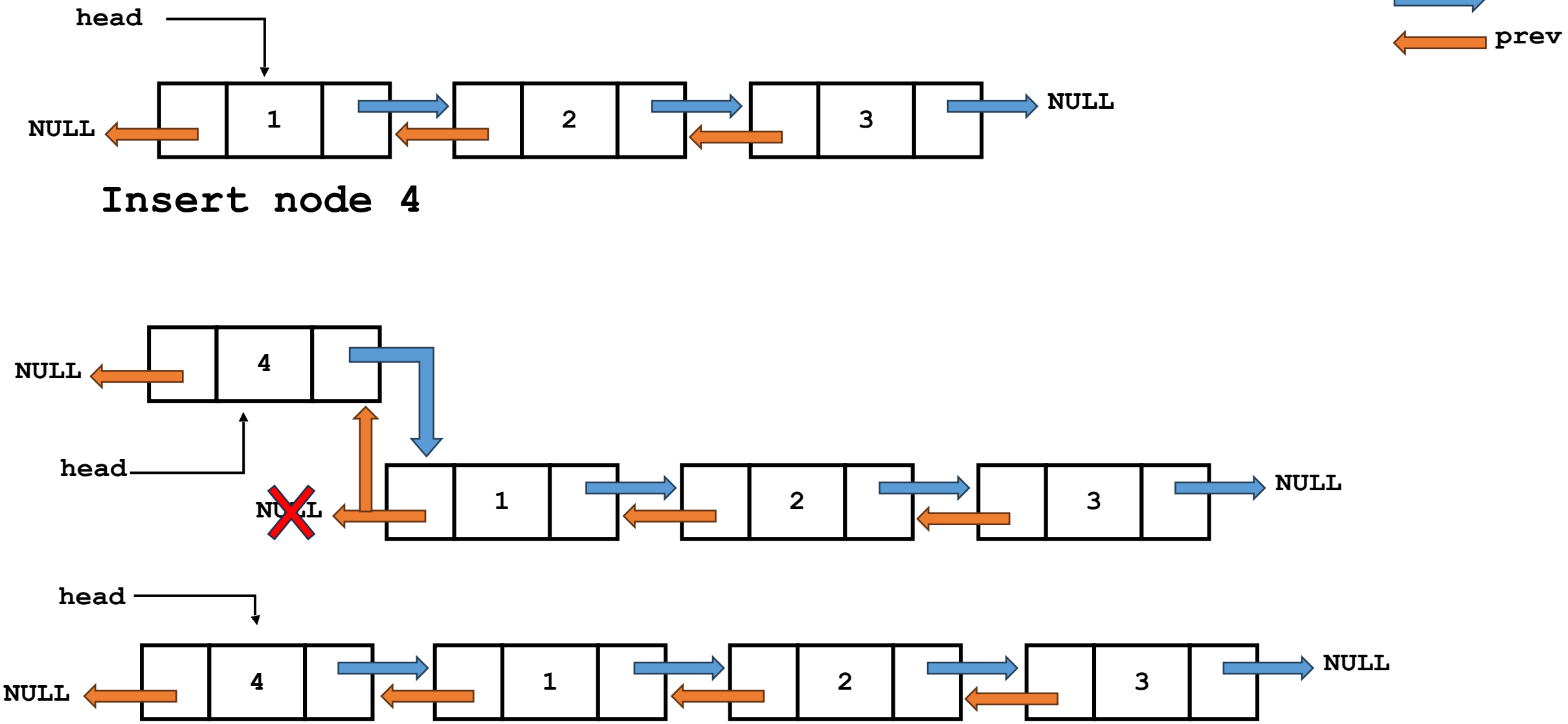
Class Structure

- 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>

```
4   struct node
5   {
6       node *prev;
7       int data;
8       node *next;
9       node()
10      {
11          prev = nullptr;
12          next = nullptr;
13          data = -1;
14      }
15      node(int n)
16      {
17          prev = nullptr;
18          next = nullptr;
19          data = n;
20      }
21  };
```

```
4   template <typename T>
5   struct node
6   {
7       node<T> *prev;
8       T data;
9       node<T> *next;
10      node()
11      {
12          prev = nullptr;
13          next = nullptr;
14      }
15      node(T n)
16      {
17          prev = nullptr;
18          next = nullptr;
19          data = n;
20      }
21  };
```

Insert at head

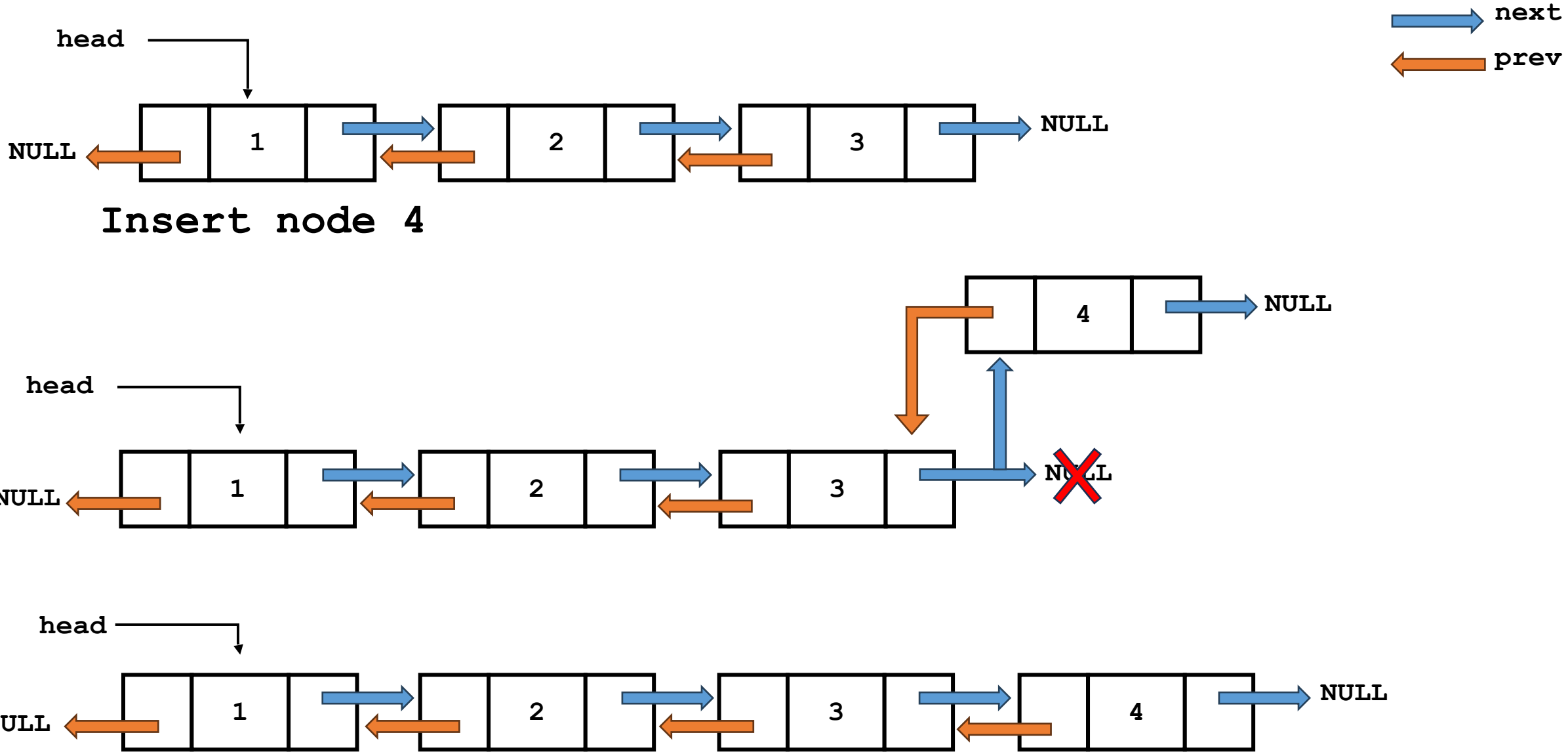


Insert at head

```
10  void doublylist::insertAtHead(int d)
11  {
12      node *temp = new node(d);
13      if (head == nullptr)
14      {
15          head = temp;
16          return;
17      }
18      temp->next = head;
19      head->prev = temp;
20      head = temp;
21  }
```

```
11  template <typename T>
12  void doublylist<T>::insertAtHead(T d)
13  {
14      node<T> *temp = new node<T>(d);
15      if (head == nullptr)
16      {
17          head = temp;
18          return;
19      }
20      temp->next = head;
21      head->prev = temp;
22      head = temp;
23  }
```

Insert at tail



Insert at tail

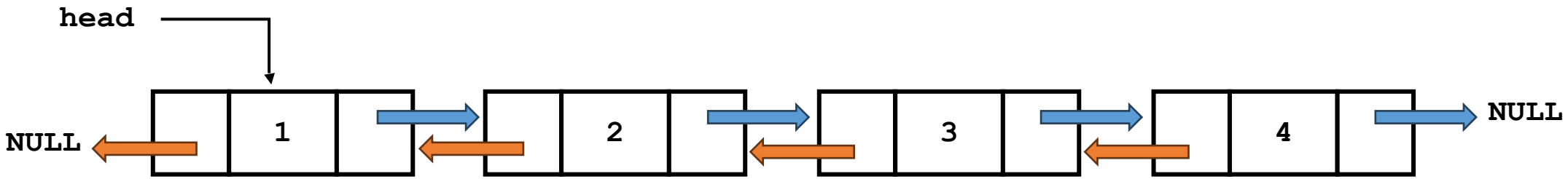
```
23  void doublylist::insertAtTail(int d)
24  {
25      node *temp = new node(d);
26      if (head == nullptr)
27      {
28          head = temp;
29          return;
30      }
31      node *cur = head;
32      while (cur->next != nullptr)
33      {
34          cur = cur->next;
35      }
36      cur->next = temp;
37      temp->prev = cur;
38  }
39
```

```
25  template <typename T>
26  void doublylist<T>::insertAtTail(T d)
27  {
28      node<T> *temp = new node<T>(d);
29      if (head == nullptr)
30      {
31          head = temp;
32          return;
33      }
34      node<T> *cur = head;
35      while (cur->next != nullptr)
36      {
37          cur = cur->next;
38      }
39      cur->next = temp;
40      temp->prev = cur;
41  }
```

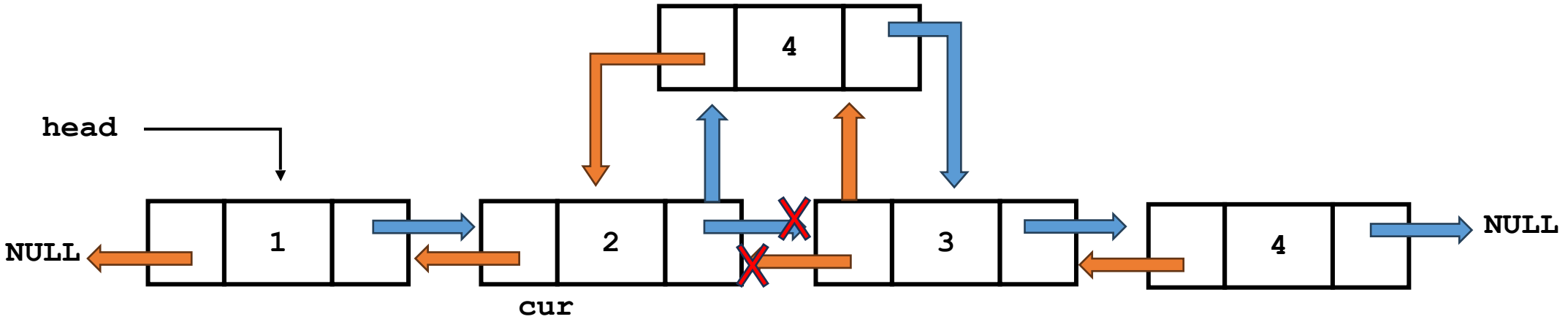
Insert at index

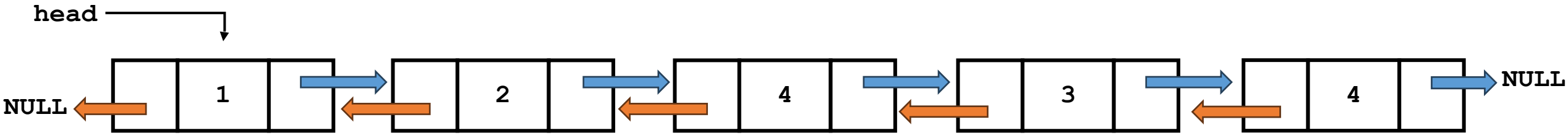
1. If index = 0 (or less)
   Insert at head
2. If index = size (or more)
   Insert at tail
3. Else
   Insert at index

Insert at index



Insert node 4 at index 2





Insert at index

```
40  void doublylist::insertAtIndex(int index, int d)
41  {
42      if (index <= 0)
43      {
44          insertAtHead(d);
45          return;
46      }
47      else if (index >= getSize())
48      {
49          insertAtTail(d);
50          return;
51      }
52
53      node *cur = head;
54      for (int i = 0; i < index - 1; i++)
55      {
56          cur = cur->next;
57      }
58      node *temp = new node(d);
59      temp->next = cur->next;
60      cur->next = temp;
61      temp->prev = cur;
62      temp->next->prev = temp;
63  }
```

```
43  template <typename T>
44  void doublylist<T>::insertAtIndex(int index, T d)
45  {
46      if (index <= 0)
47      {
48          insertAtHead(d);
49          return;
50      }
51      else if (index >= getSize())
52      {
53          insertAtTail(d);
54          return;
55      }
56
57      node<T> *cur = head;
58      for (int i = 0; i < index - 1; i++)
59      {
60          cur = cur->next;
61      }
62      node<T> *temp = new node<T>(d);
63      temp->next = cur->next;
64      cur->next = temp;
65      temp->prev = cur;
66      temp->next->prev = temp;
67  }
```

Demo

```
6   int main()
7   {
8       doublylist<int> test;
9
10      cout << "Adding to head" << endl;
11      test.insertAtHead(1);
12      test.insertAtHead(2);
13      test.insertAtHead(3);
14      test.insertAtHead(4);
15      test.print();
16      cout << endl;
17      cout << "Adding to tail" << endl;
18      test.insertAtTail(1);
19      test.insertAtTail(2);
20      test.insertAtTail(3);
21      test.insertAtTail(4);
22      test.print();
23      cout << endl;
24      cout << "Adding to index 2" << endl;
25      test.insertAtIndex(2, 20);
26      test.print();
27      cout << endl;
28      cout << "Adding to index 5" << endl;
29      test.insertAtIndex(5, 50);
30
31      test.print();
32      return 0;
33  }
```

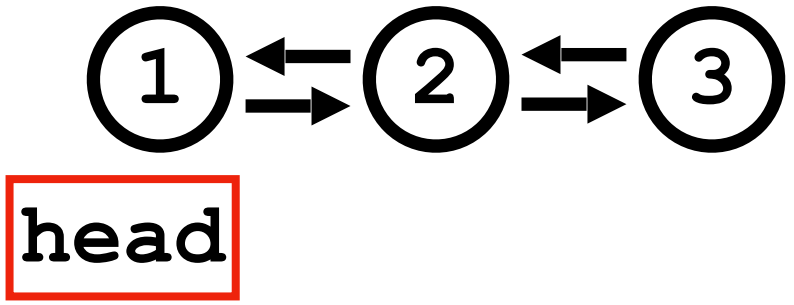
Output:

```
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

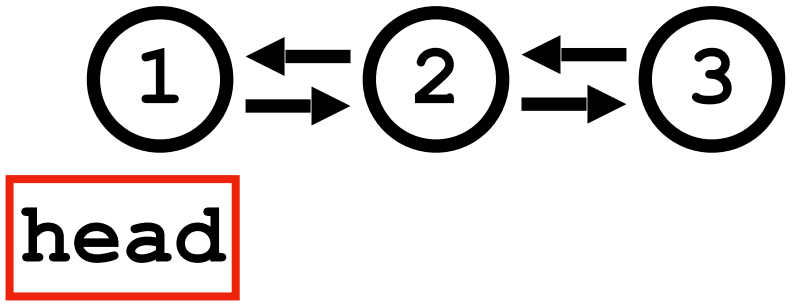
Deletion

Remove Head



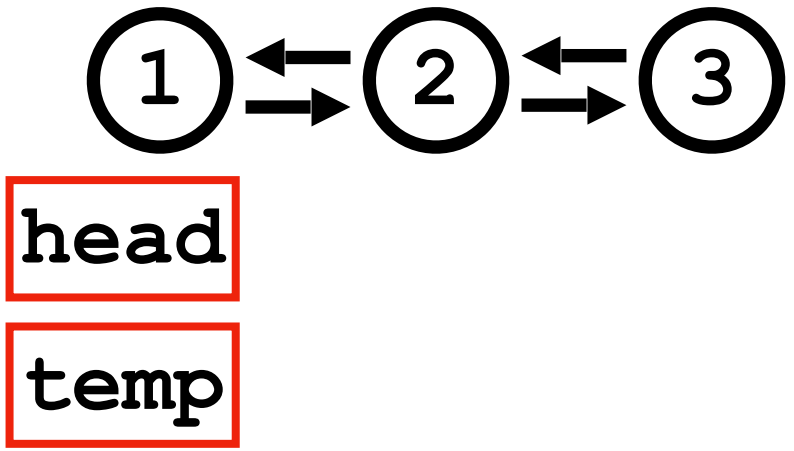
Step 1

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



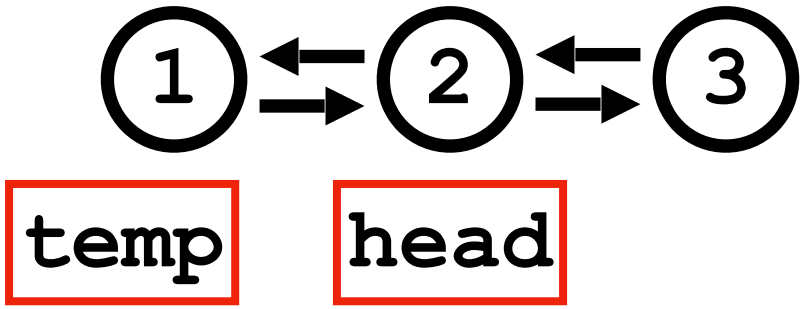
Step 2

Create a temporary pointer and set it to head.



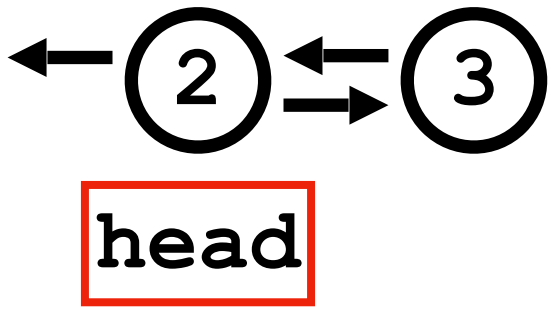
Step 3

Set head to head's next pointer.



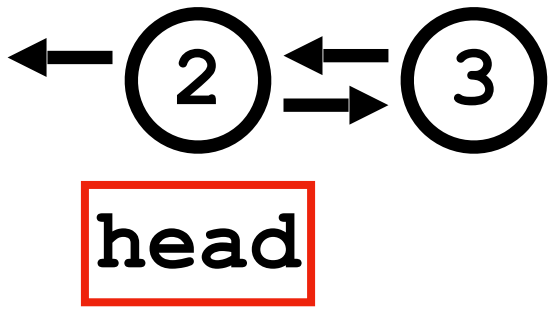
Step 4

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



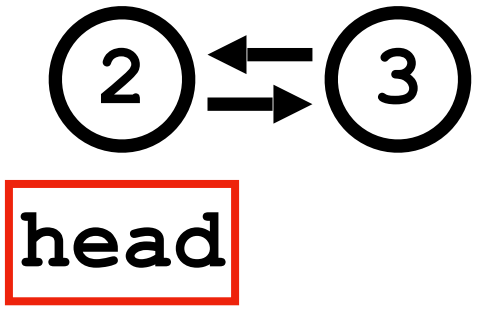
Step 5

Check if the new head is nullptr.



Step 6

Set head's previous pointer to nullptr.



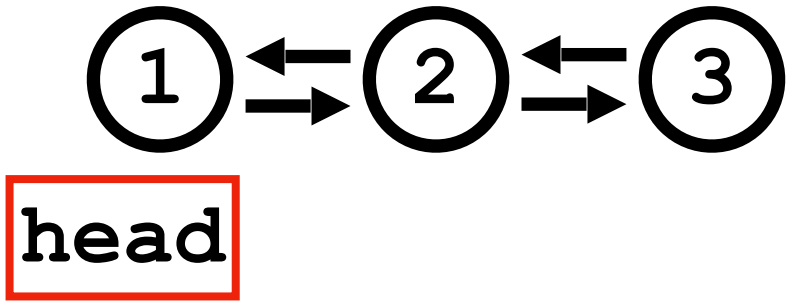
Remove Head - Code

```
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

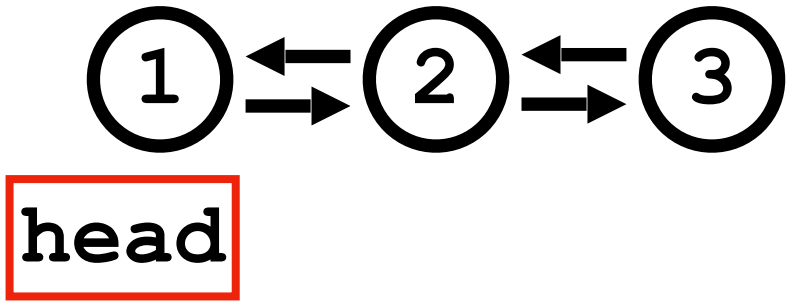
```
template <typename T>
void doublylist::removeHead(){
  if(head == nullptr)
    return;
  node<T> *temp = head;
  head = head->next;
  delete temp;
  if(head != nullptr)
    head->prev = nullptr;
  return head;
}
```

Remove Tail



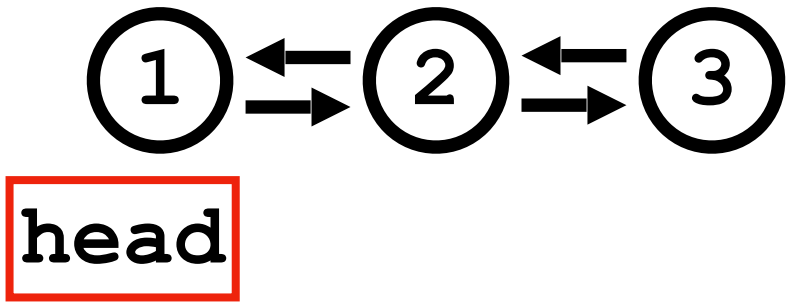
Step 1

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



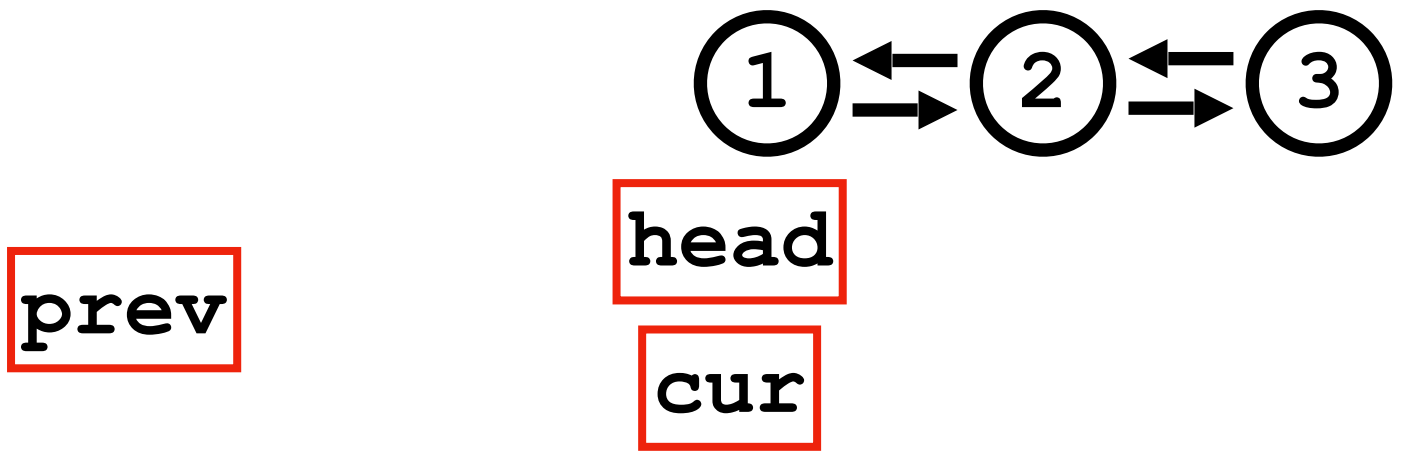
Step 2

Check if there is only one element in linked list.



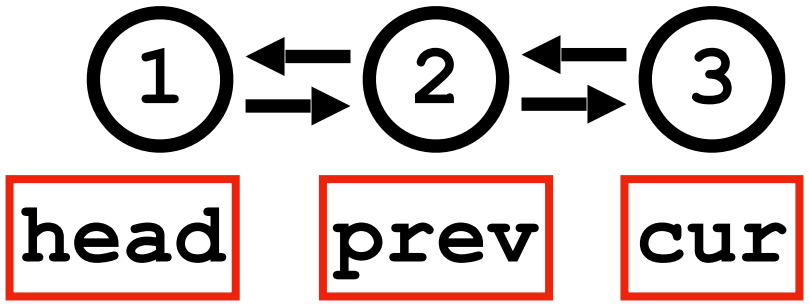
Step 3

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



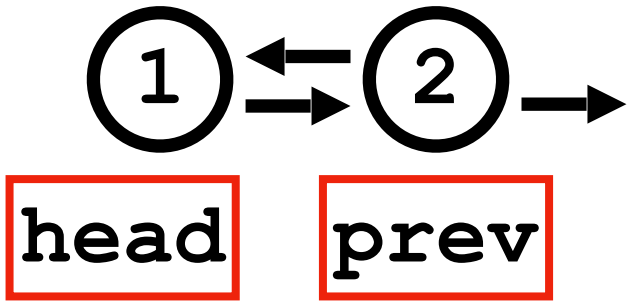
Step 4

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



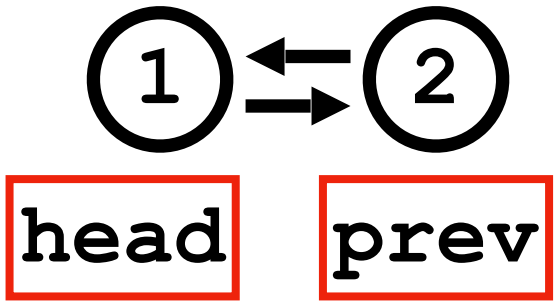
Step 5

Delete cur.



Step 6

Set prev's next pointer equal to nullptr.



Remove Tail - Code

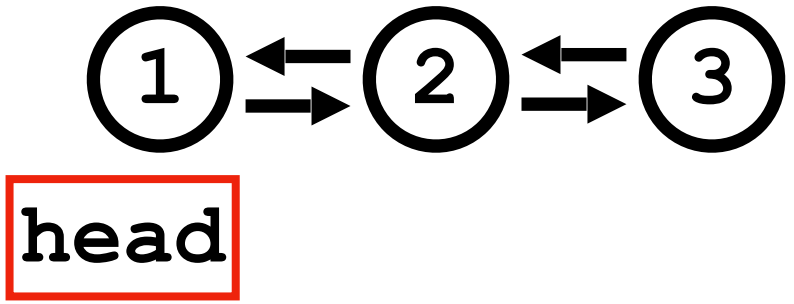
```
void doublylist::removeTail(){
  if(head == nullptr)
    return;
  else if(head->next == nullptr){
    removeHead();
    return;
  }
  node *prev = nullptr;
  node *cur = head;
  while(cur->next != nullptr){
    prev = cur;
    cur = cur->next;
  }
  prev->next = nullptr;
  delete cur;
  return head;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

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;
}
```

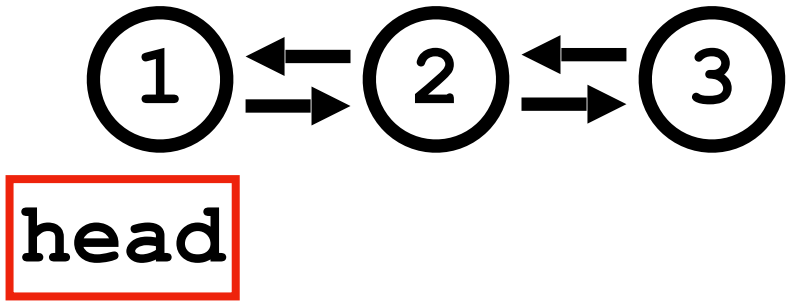
Remove At Position

Position is 1.



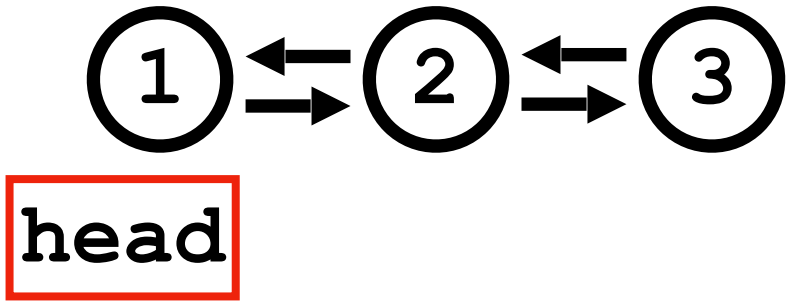
Step 1

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



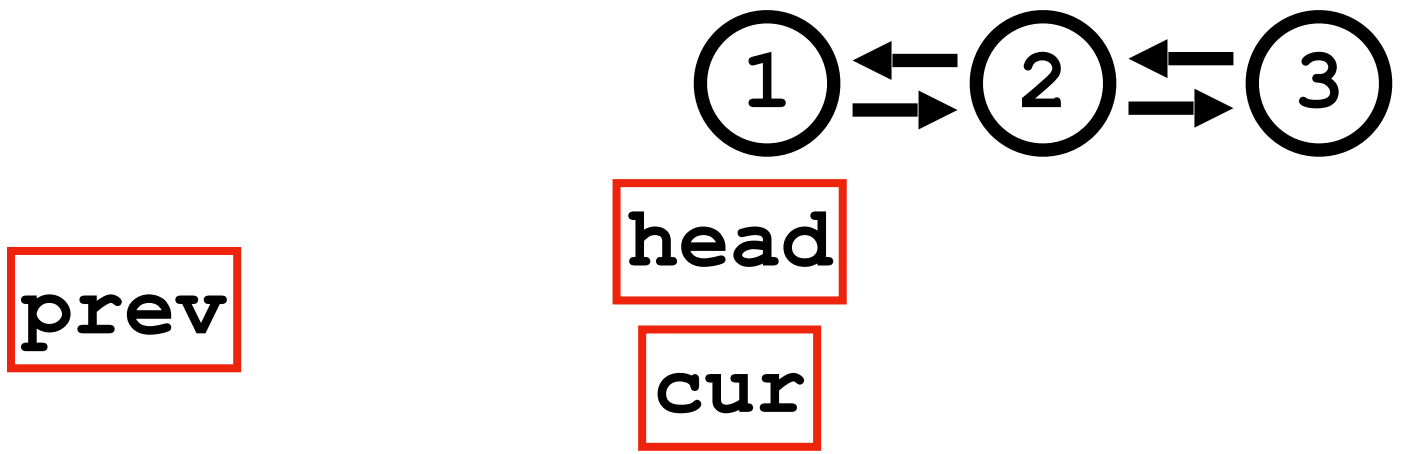
Step 2

Check if position is 0.



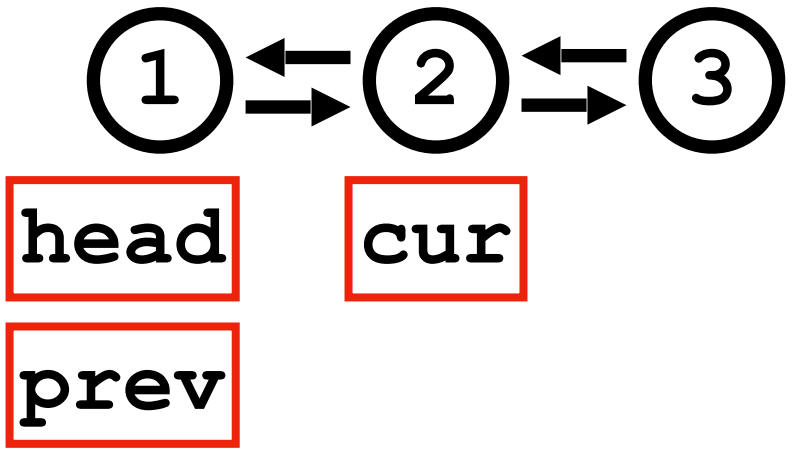
Step 3

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



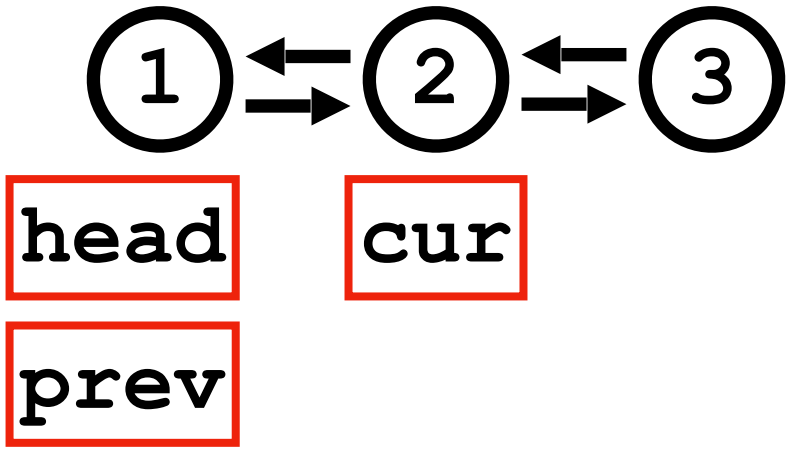
Step 4

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



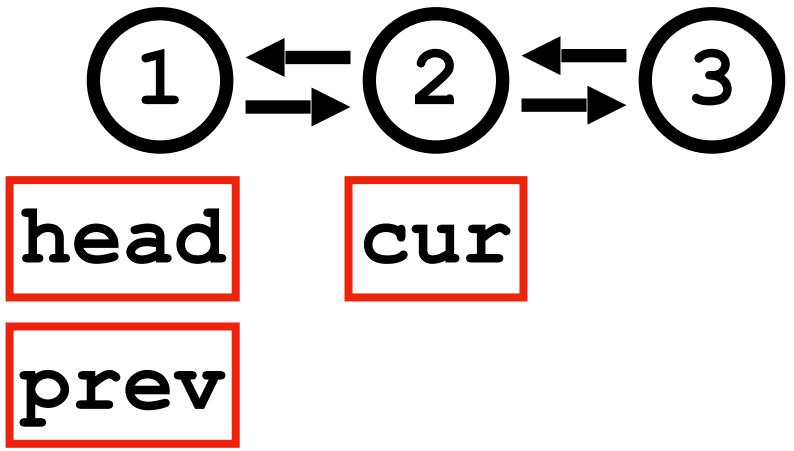
Step 5

Check if cur is nullptr. If it is return.



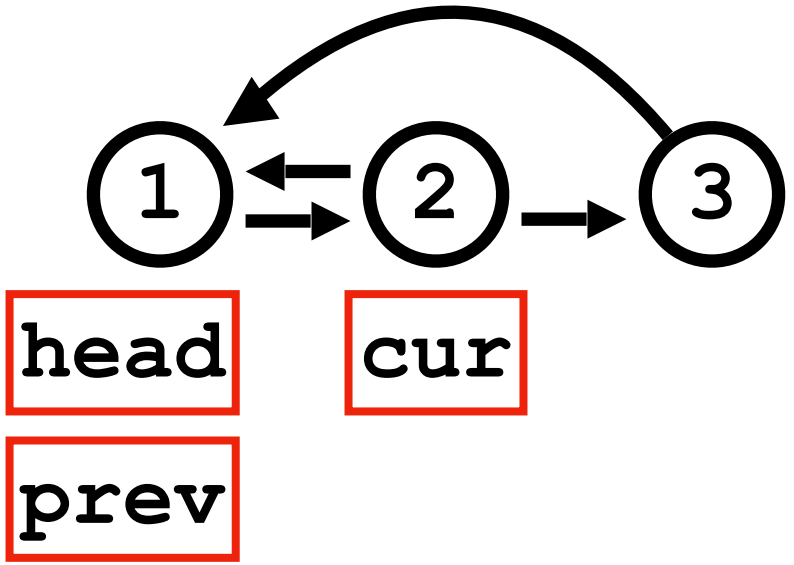
Step 6

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



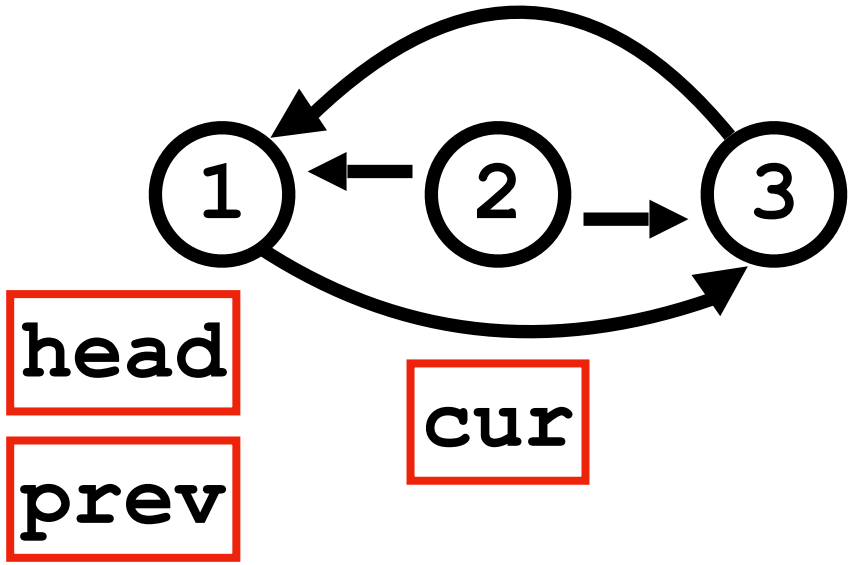
Step 7

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



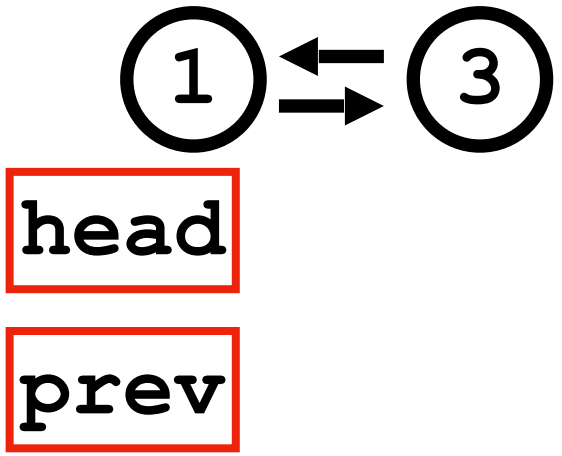
Step 8

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;
}
```