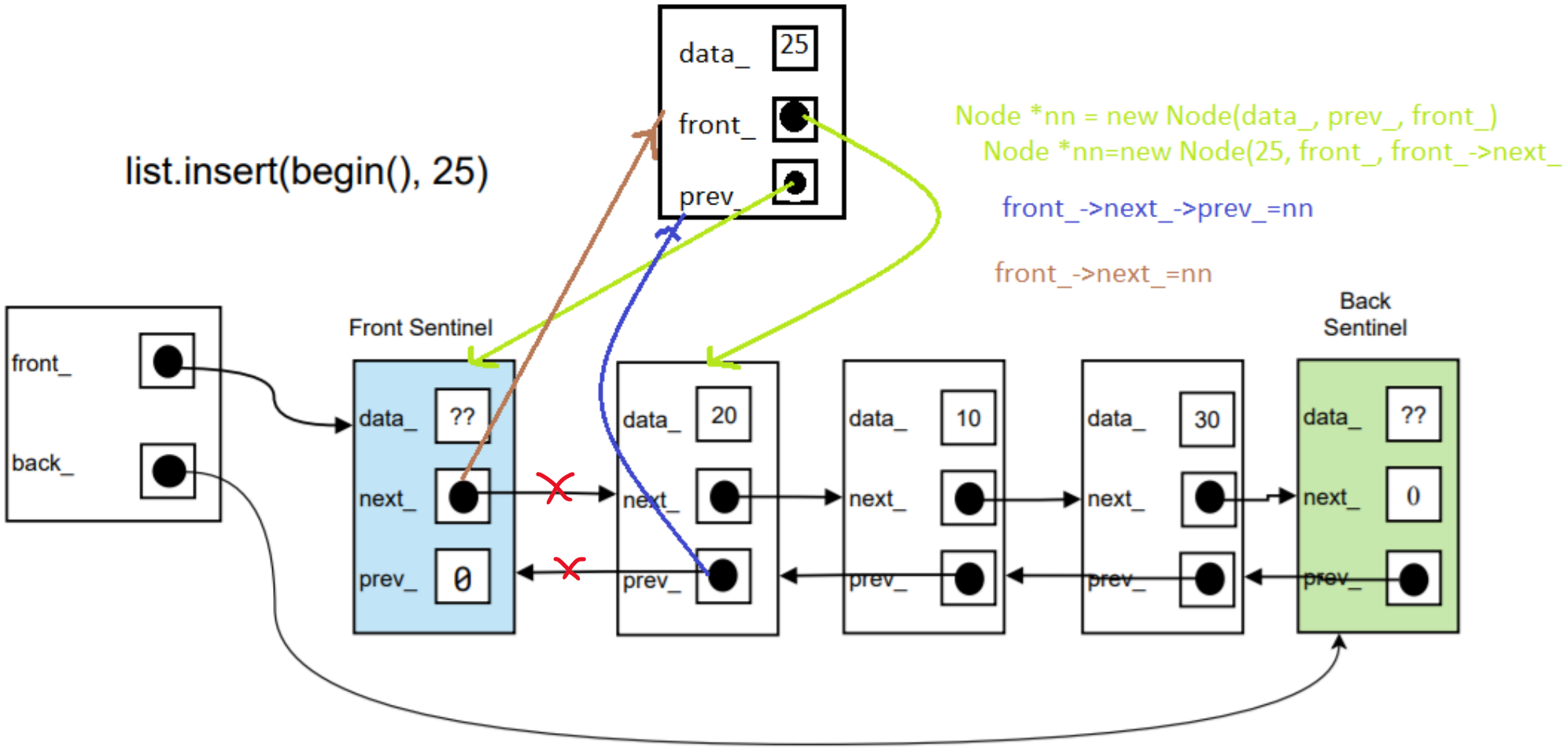
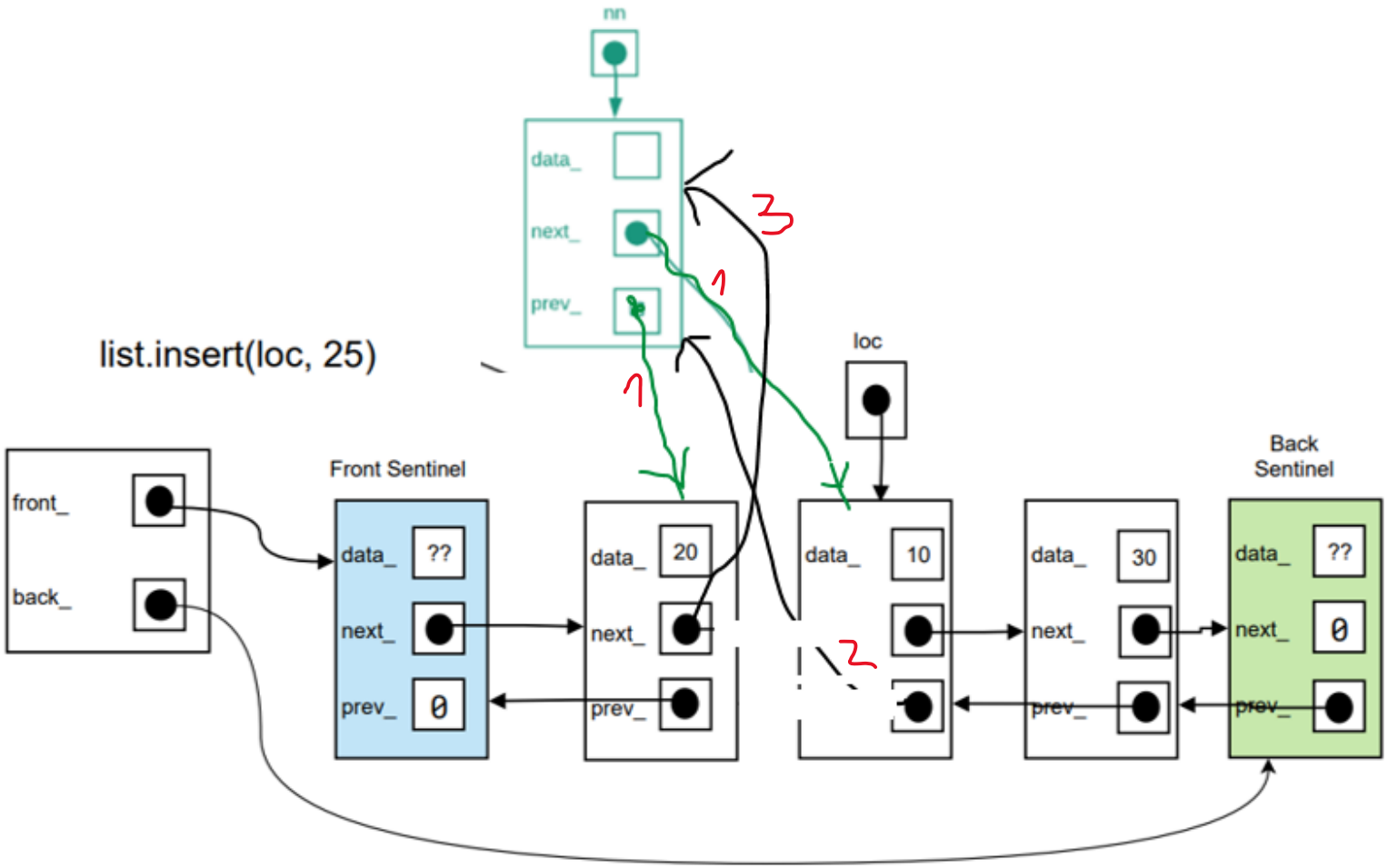
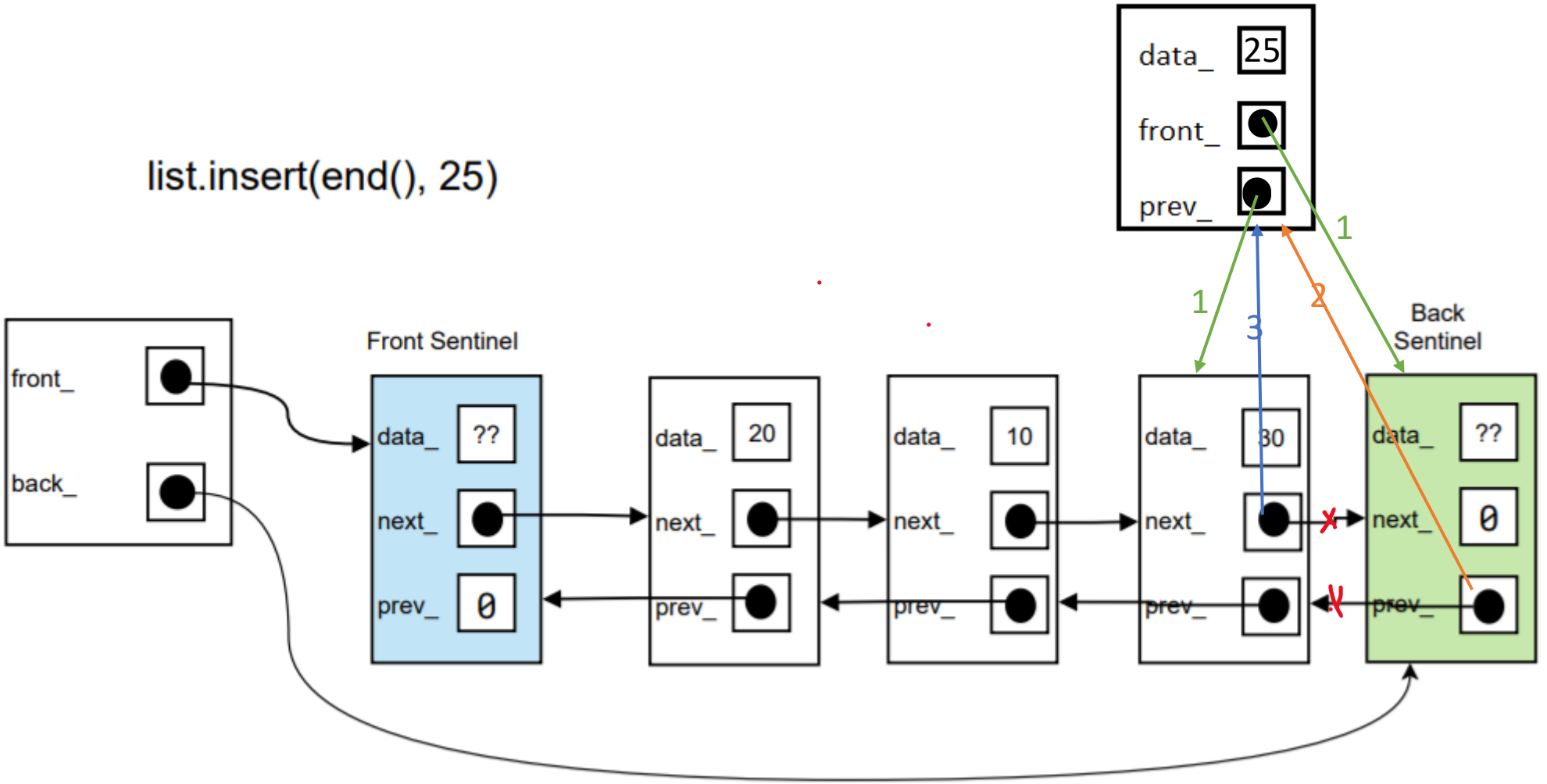

list.insert(begin(), 25)
data_ 25
front_
prev_
Node *nn = new Node(data_, prev_, front_)
Node *nn=new Node(25, front_, front_->next_
front_->next_->prev_=nn
front_->next_=nn
front_
back_
Front Sentinel
data_ ??
next_
prev_ 0
data_ 20
next_
prev_
data_ 10
next_
prev_
data_ 30
next_
prev_
Back Sentinel
data_ ??
next_ 0
prev_

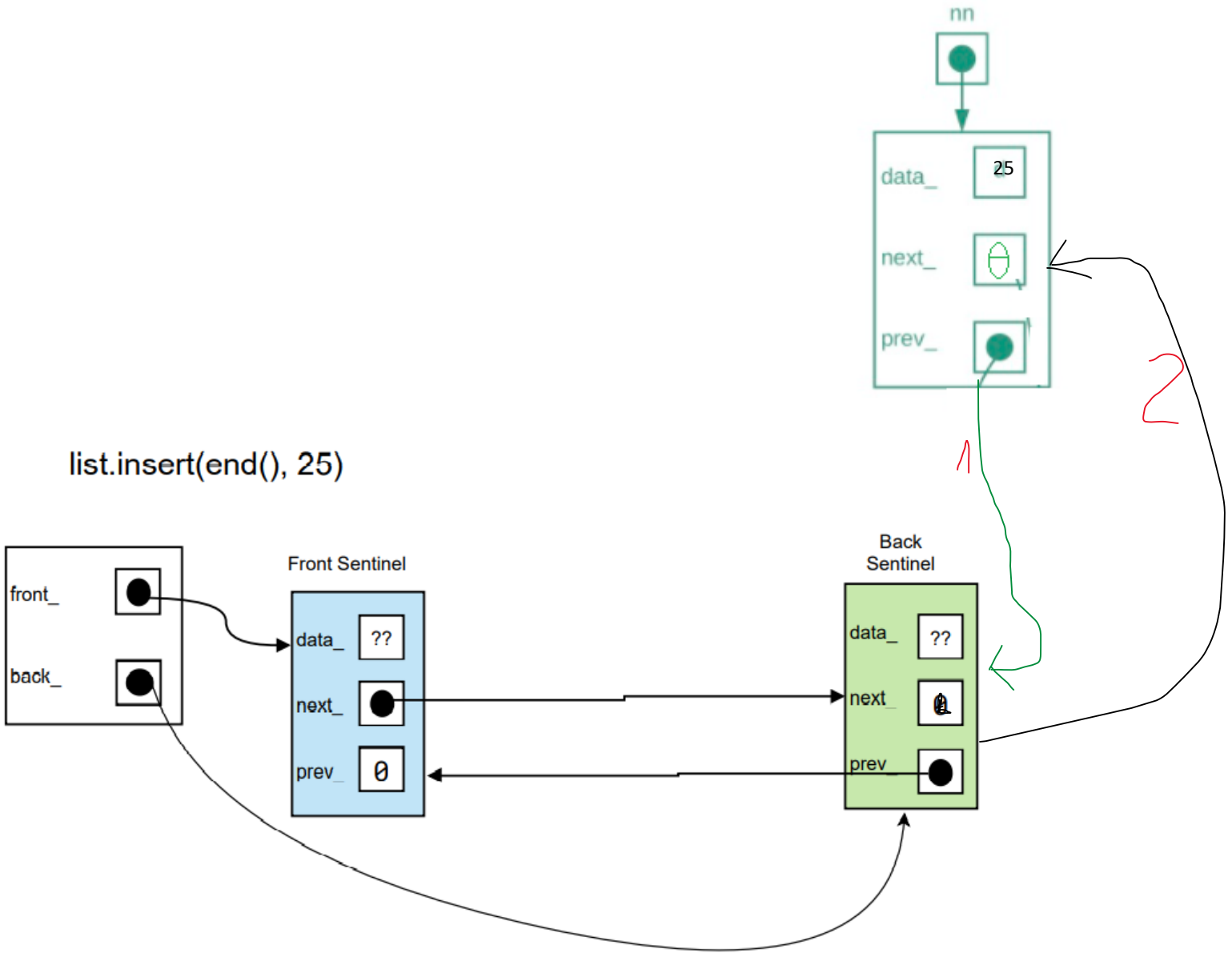
list.insert(loc, 25)




```
Node *temp = front_;

// loc is iterator => loop through to find the node at loc position
while(front != null && front->data != *loc){
                front = front -> next;
}
// now front is at loc position
Step 1: Create new node
Node *nn = new Node (25, front -> previous, front);

Step 2: Make the previous pointer of the front sentinel point to
the new node
front -> previous = nn;

Step 3: Set the next pointer of the front's previous (loc's
previous) sentinel to the new node.
front -> previous -> next = nn; (3)

// After inserting, front comes back to the first node
front = temp
```

Step 1: Node *nn = Node(25, back_->prev_, back_);
Step 2: back_->prev_=nn;
Step 3: back_->prev_->next_ = nn;

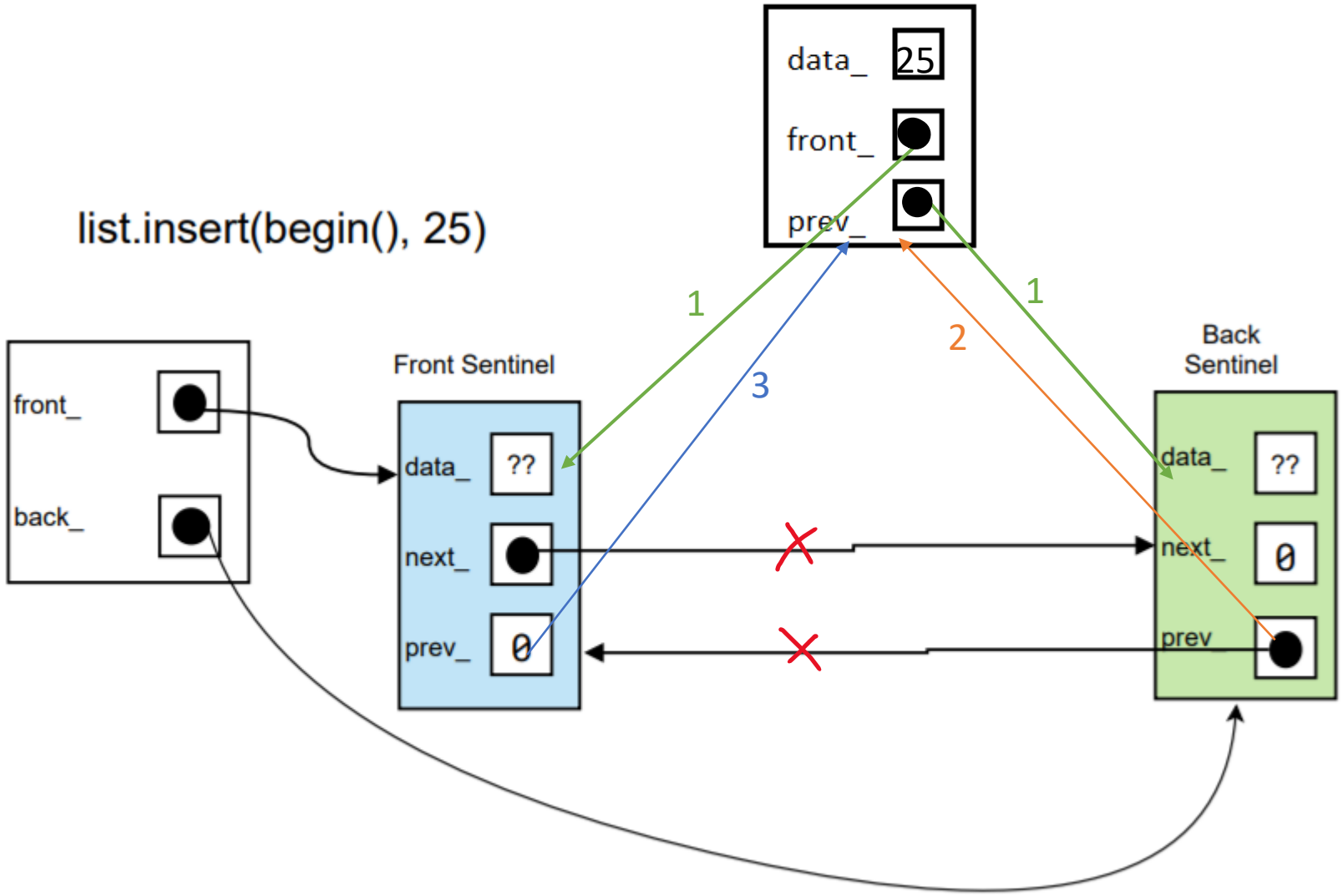




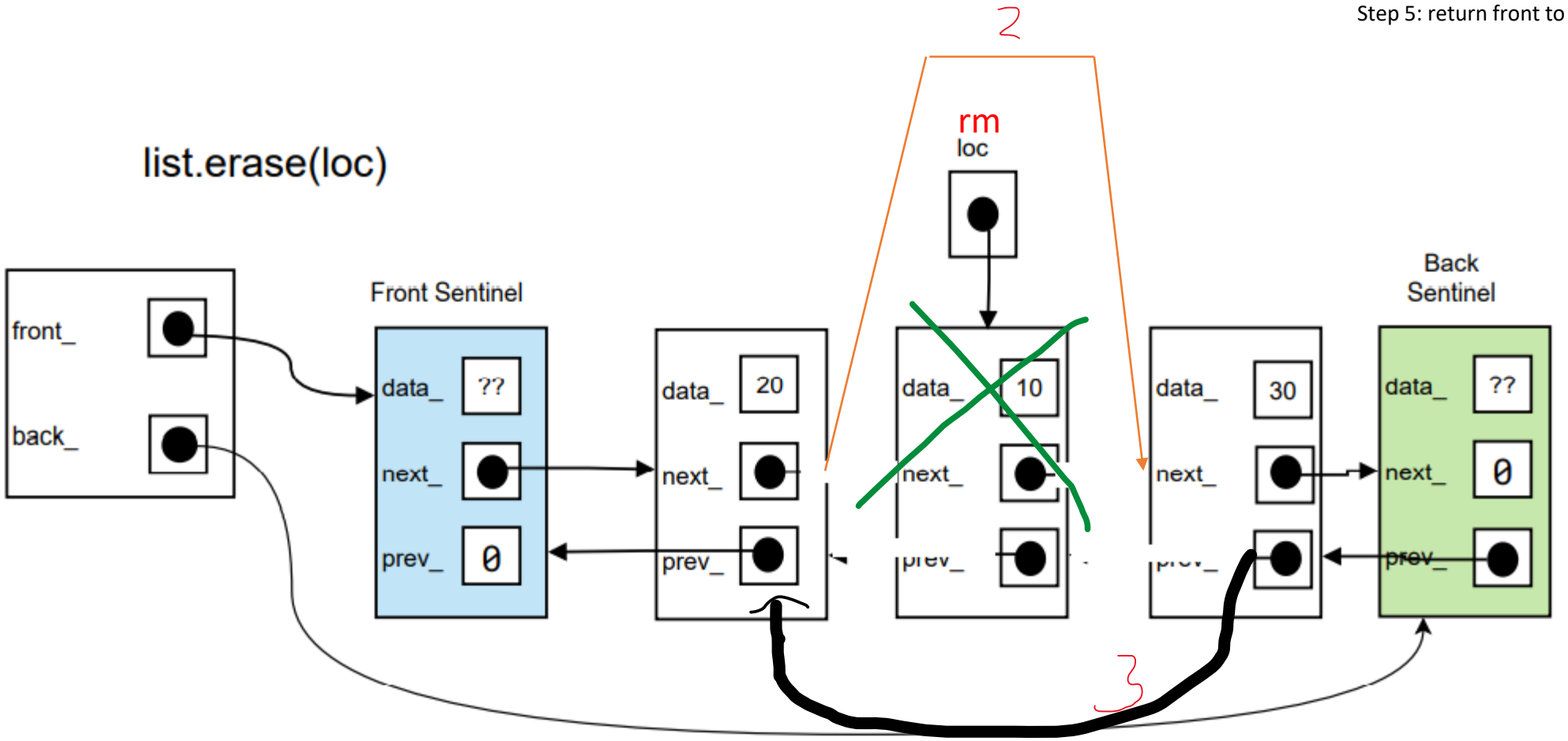
Step 1: Create new node, next node is nullptr and previous node is back sentinel.
Node *nn = new Node (25, front -> next, nullptr);

Step 2: Set the next pointer of the back sentinel to the new node.
front -> next -> next = nn; (2)

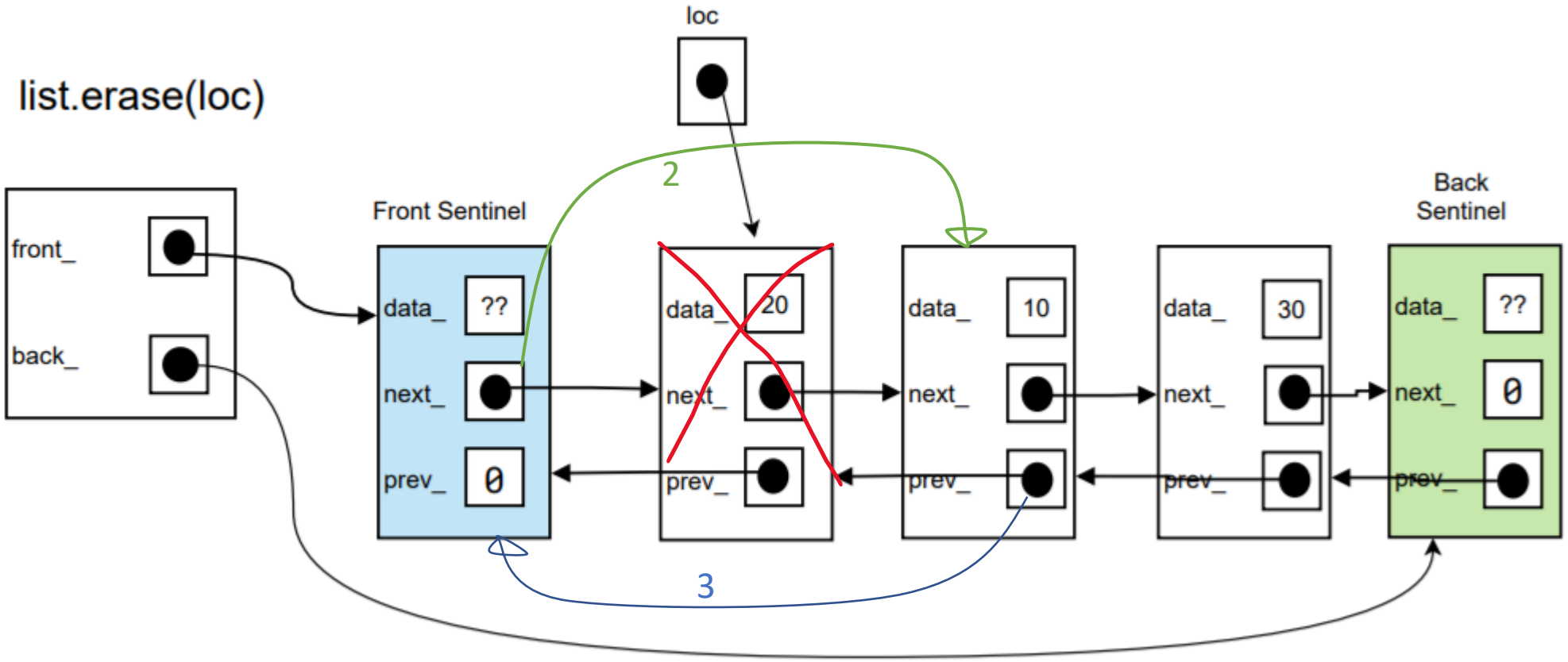
Step 1: Node *nn=new Node(25, back_, front_)
Step 2: front_->next_= nn;
Step 3: back_->prev_=nn;



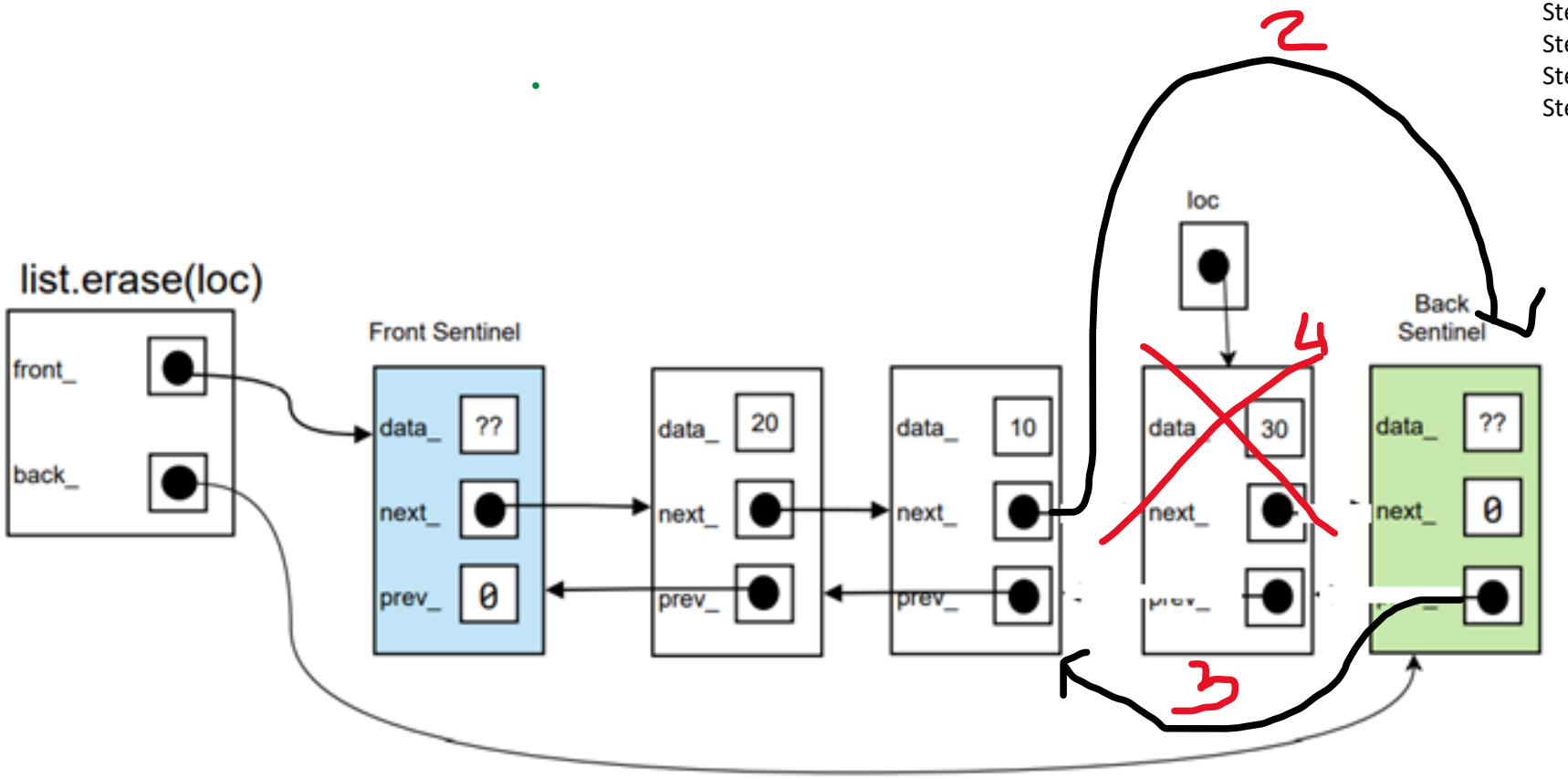
```
Node *temp = front_;

// loc is iterator => loop through to find the node at loc position
while(front != null && front->data != *loc){
                front = front -> next;
}
// now front is at loc position
Step 1: Node *rm = front
Step 2: front -> previous -> next = rm -> next;
Step 3: front -> next -> previous = rm -> previous;
Step 4: deallocate the node we are moving: delete rm
Step 5: return front to the first node: front = temp;
```



Step 1: Check empty list
If(front_->next_ != back_) → Step 2
Step 2: assume, loc is node will be removed.
front_->next_ = loc->next_
Step 3: front_->next_ now is the node next to loc.
front_->next_->prev_=loc->prev_
Step 4: deallocate loc.
delete loc;



```
Node *temp = front_;

// loc is iterator => loop through to find the node at loc position
while(front != null && front->data != *loc){
                front = front -> next;
}
// now front is at loc position
Step 1: Node *rm = front
Step 2: front -> previous -> next = rm -> next;
Step 3: front -> next -> previous = rm -> previous;
Step 4: deallocate the node we are moving: delete rm
Step 5: return front to the first node: front = temp;
```



Step 1: loop list from Front to Back Sentinel.

loop list from Front to Back Sentinel inside the first loop(Bubble Sort)

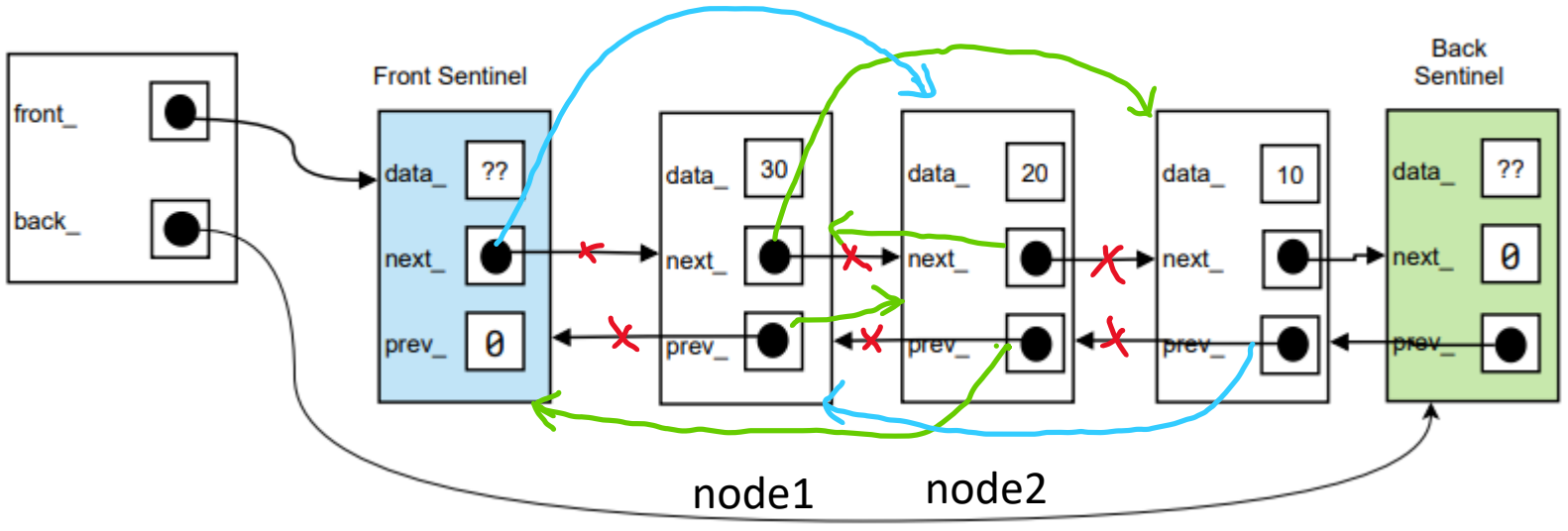
call node1 for first node and node2 for the next_ of node1.

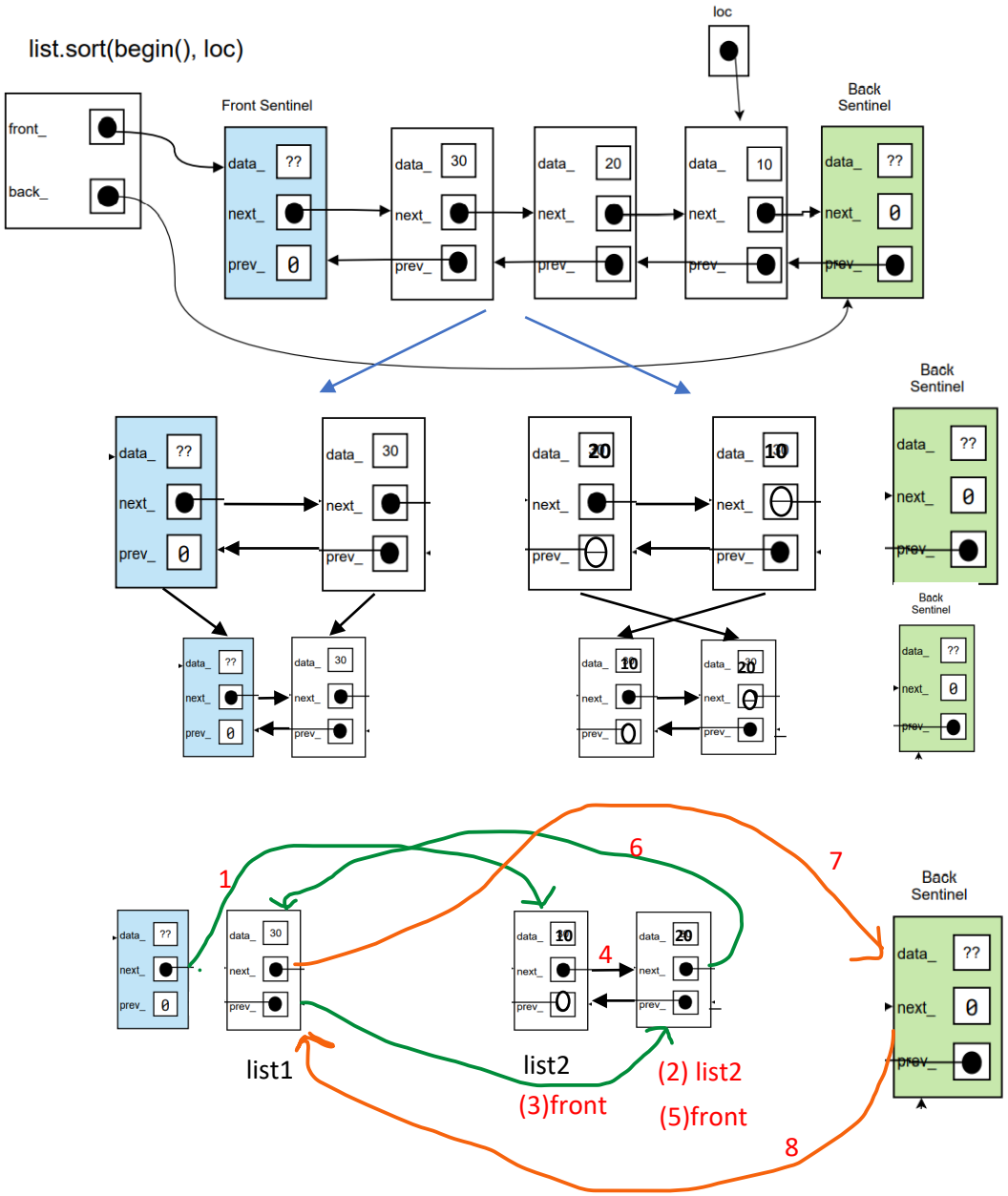
Step 2: if node1_data > node2_data → swap node1 and node2. (Green)

Step 3: change the destinate of node front node1 and node after node2 (Blue)

Step 3: loop until it reach null(end of list)

list.sort(begin(), end())





Step 1: Sublist from begin() to loc

Step 2: Divide the sublist into subproblems.
Recursively split the original list into two halves. The split continues until there is only one node in the linked list. To split the list into two halves, we find the middle of the linked list using the Fast and Slow pointer approach

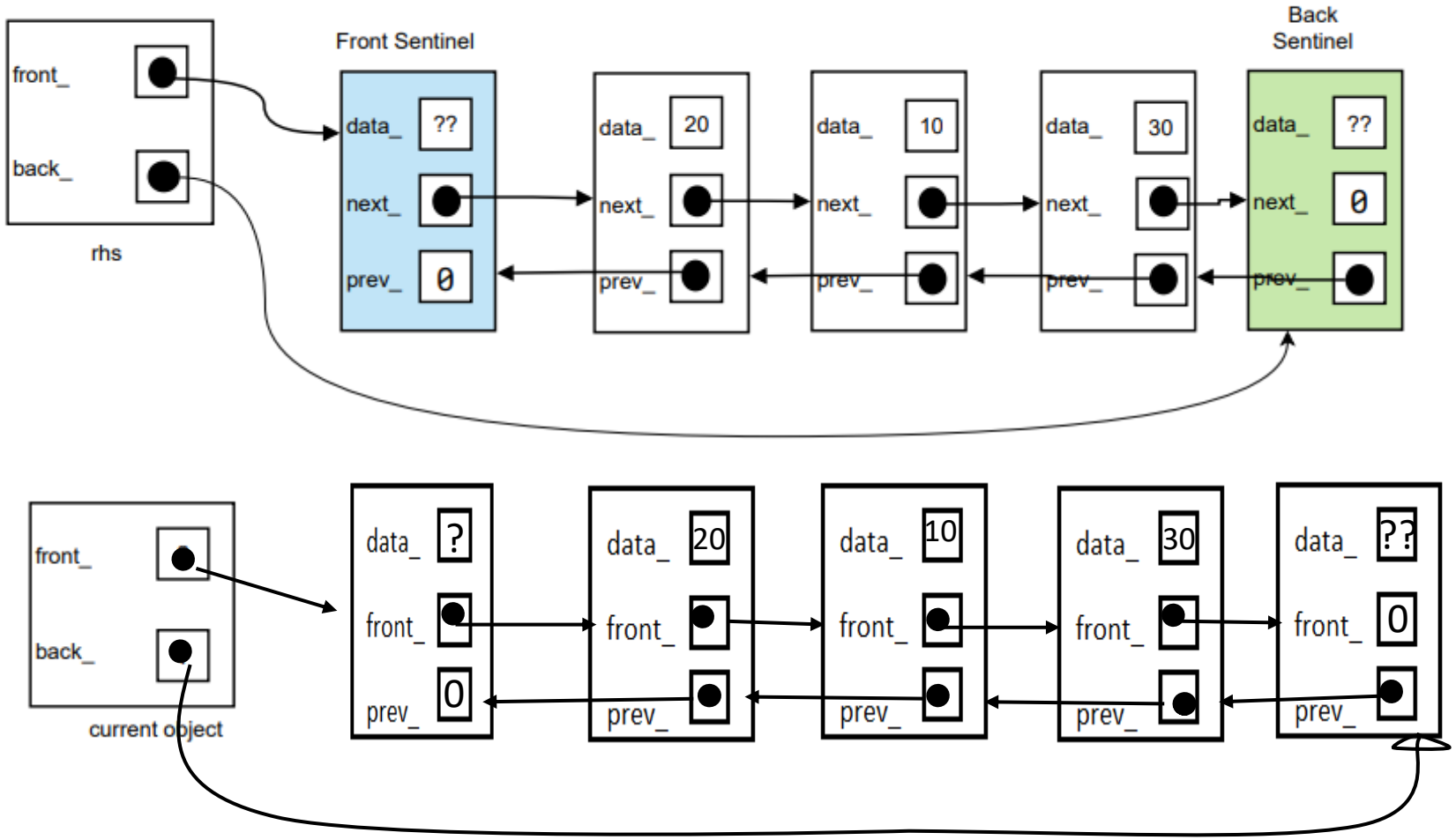
Step 3: Recursively sort each sublist and combine it into a single sorted list. The process continues until we get the original list in sorted order.

```
while(list1  && list2){
        if (list1->val < list2->val) {
            front->next = list1;
            list1 = list1->next;
          } else {
            front->next = list2;      (1), (4)
            list2 = list2->next;      (2)
          }
          front = front->next;        (3), (5)
       }
 if(list1) front->next = list1;       (6)
 else front->next = list2;
}
```

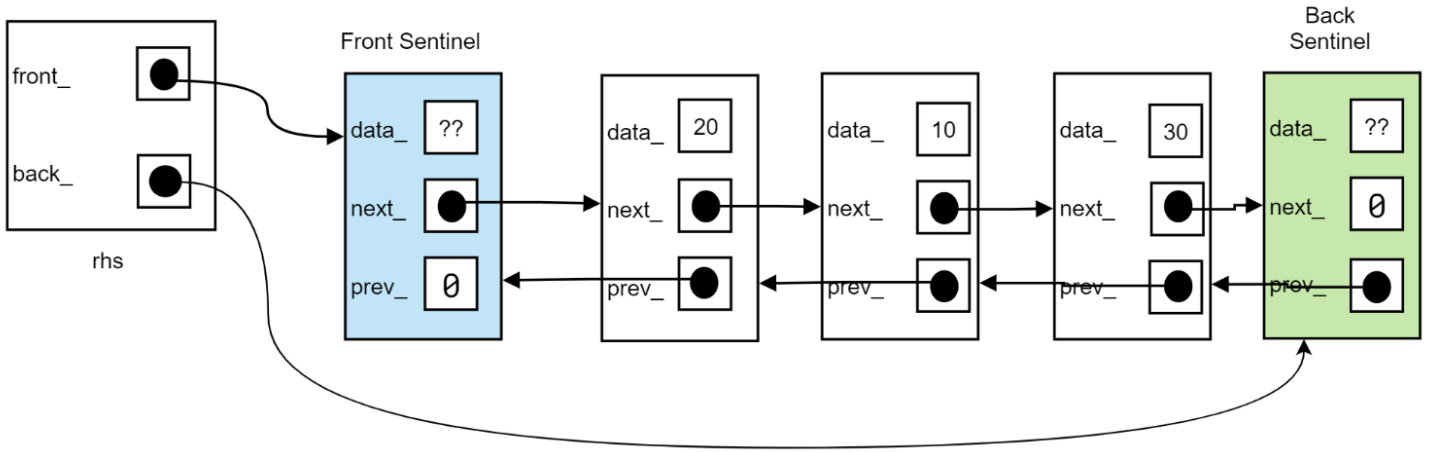
Step 4: Merge front to the rest of the node
front->next = loc ->next (7)
loc->next->previous = front (8)

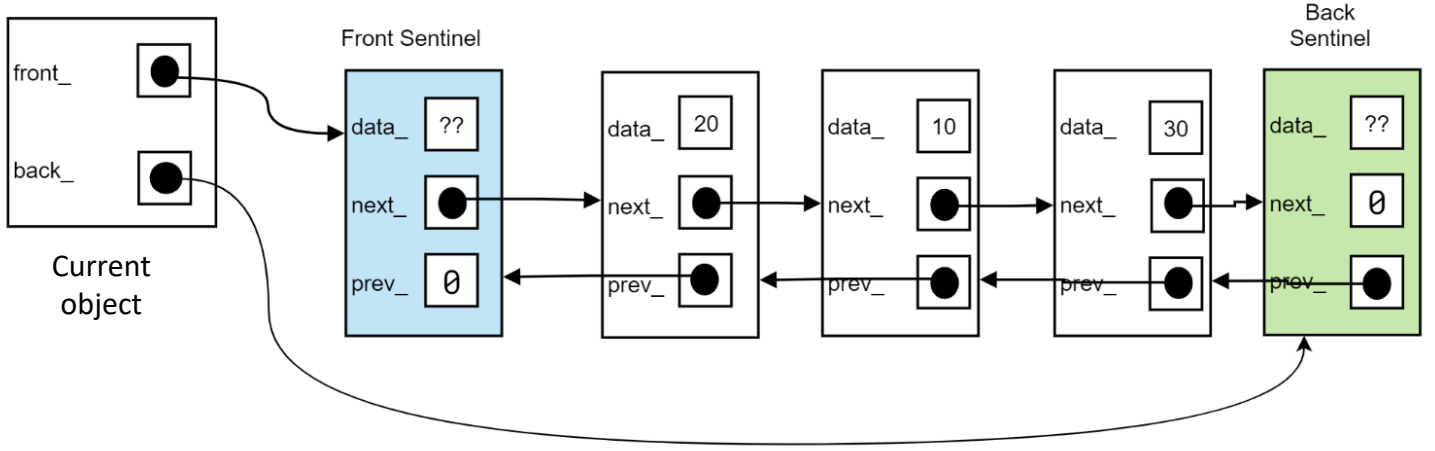
Step 1: create another Empty linked list
Step 2: loop from Front to Back Sentinel
Step 3: create new Node with the data from original list and push_front to the new list just was created
Step 4: Loop until the end.

copy constructor - alter rhs and/or current object as appropriate

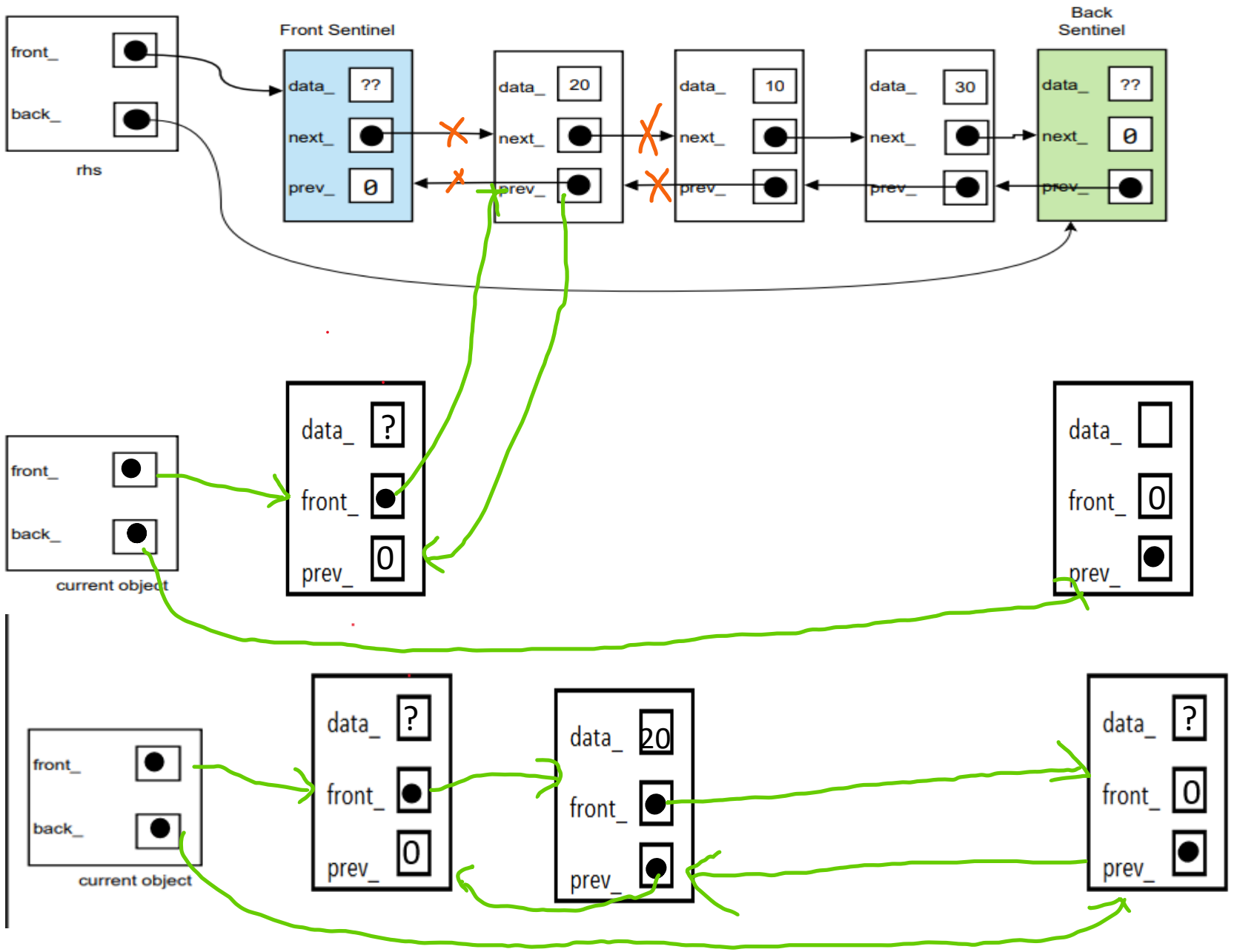


copy assignment operator - alter rhs and/or current object as appropriate



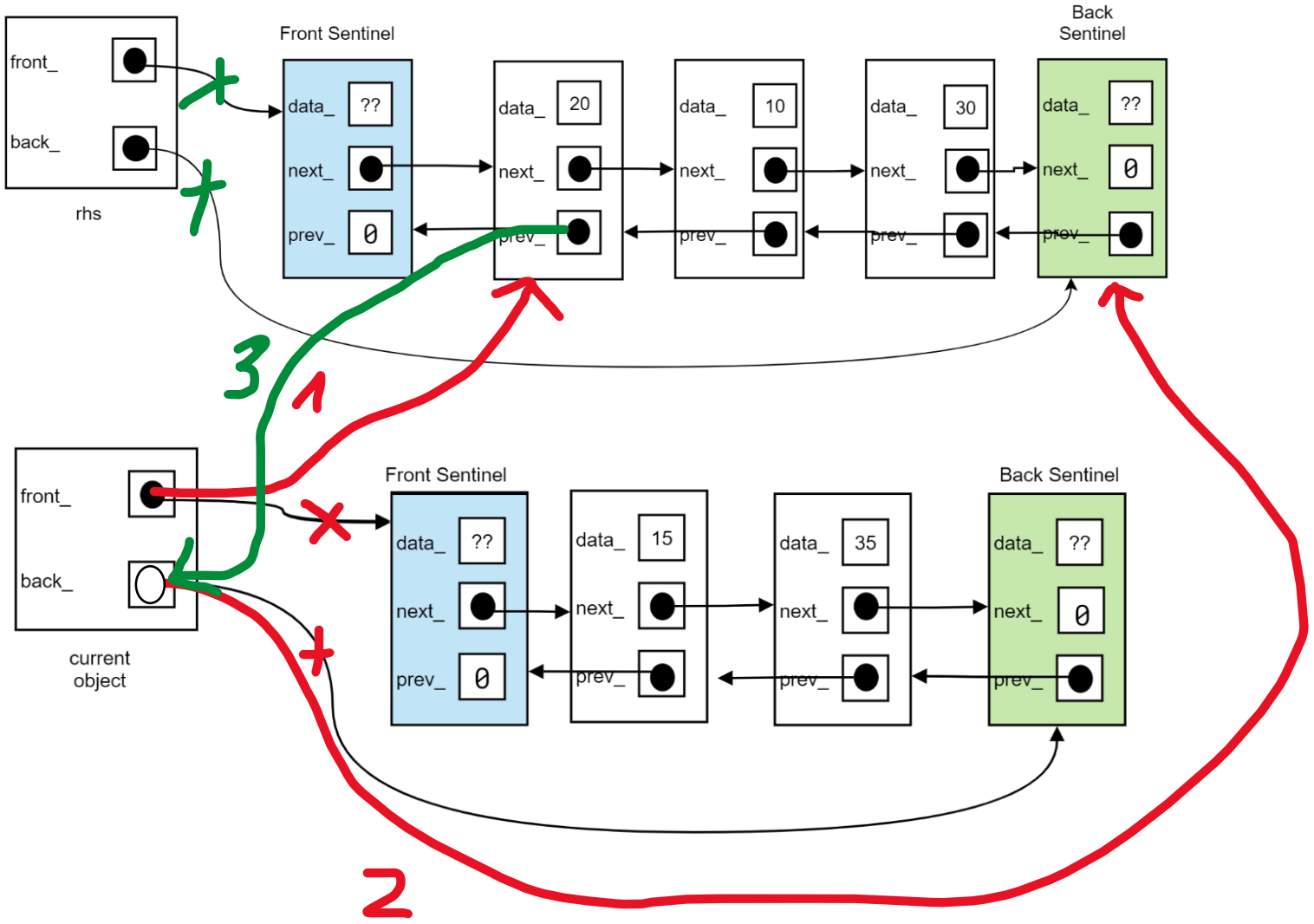


move constructor - alter rhs and/or current object as appropriate



Step 1: Create another empty linked list
Step 2: Loop from Front to Back Sentinel
push the node from original List
to new list
Step 3: Change next_, prev_ of node and front, back Sentinel (push_front()).
Step 4: Loop until the end

move assignment operator - alter rhs and/or current object as appropriate



Step 1: let the current object point points to the sentinels in rhs
Step 2: let current object points to the back
Step 3:cut the original rhs