ECE36800 Data structures

Linked lists
Chapter 3 (pp. 90-108)

Lu Su

School of Electrical and Computer Engineering
Purdue University

Slides Courtesy: Prof. Cheng-Kok Koh

Overview

- List description
- Singly-linked lists
  - Primitive operators
  - Implementations
- Circular list
- Doubly-linked list

- Reference pages: pp. 90-108

List

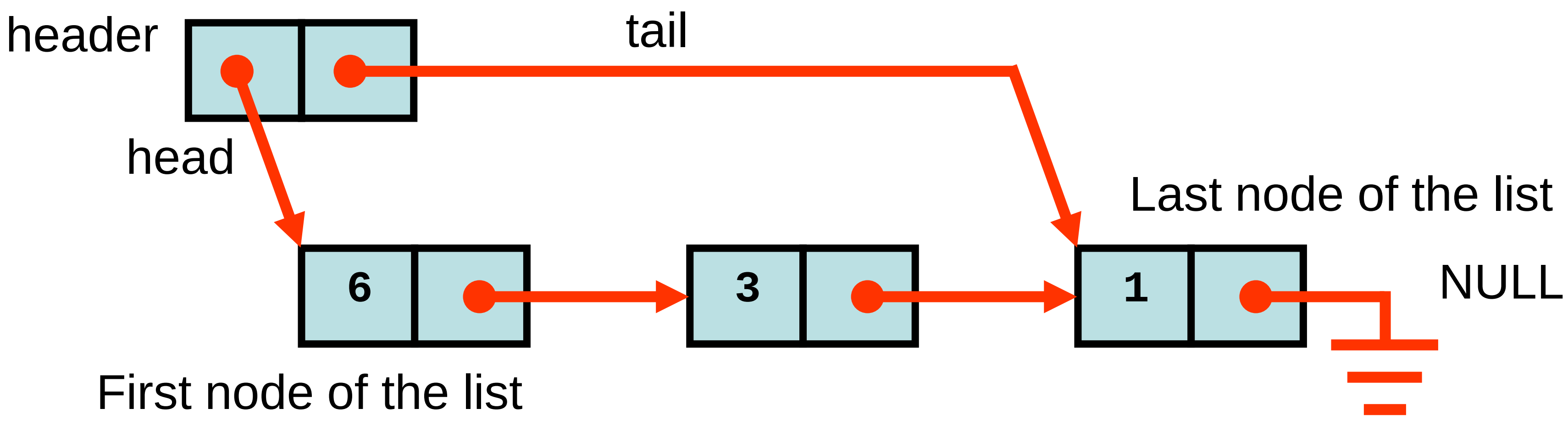
- A linear collection of items that can be inserted and removed at any position
- Each item is typically called a node, which contains a data field and an address field (typically named as “next”) to point to the next item (node) in the list
- The next field links the nodes into a linked-list
  - List == singly-linked list
- Lists and arrays are linear collections of items
  - The linear order in an array is maintained by the index and the linear order in a list is imposed by the next field

Lists vs. Arrays

- Array is more memory efficient (as there is no need to maintain the next field), but requires a contiguous block of memory
- Items in an array can be read and overwritten at any position in O(1) if you know the index, but insertion or removal of an item may take more than O(1) even if you know the index
  - Must also know the size of the array to avoid going out of bound
  - As long as an array is dynamically allocated, it is possible to change the size of the array (subject to availability of contiguous memory for the re-allocation of memory)
- Nodes in a list are not stored in contiguous locations
- Not all nodes in a list can be accessed in constant time, but insertion or removal of an item can take O(1) if you know the address of the node before the point of insertion or deletion
  - Can easily shrink or grow a linked list (subject to availability of memory)

List representation

- Node: a box (container) made up of two boxes, one for the data field and one for the next field, which stores the address pointing to the next node in the list
- Head (absolutely necessary): stores the address of the first node in the list
- Tail (optional): stores the address of the last node in the list
- Header (auxiliary, optional): contains the head and the tail addresses and possibly other information



List implementation (recall ECE26400)

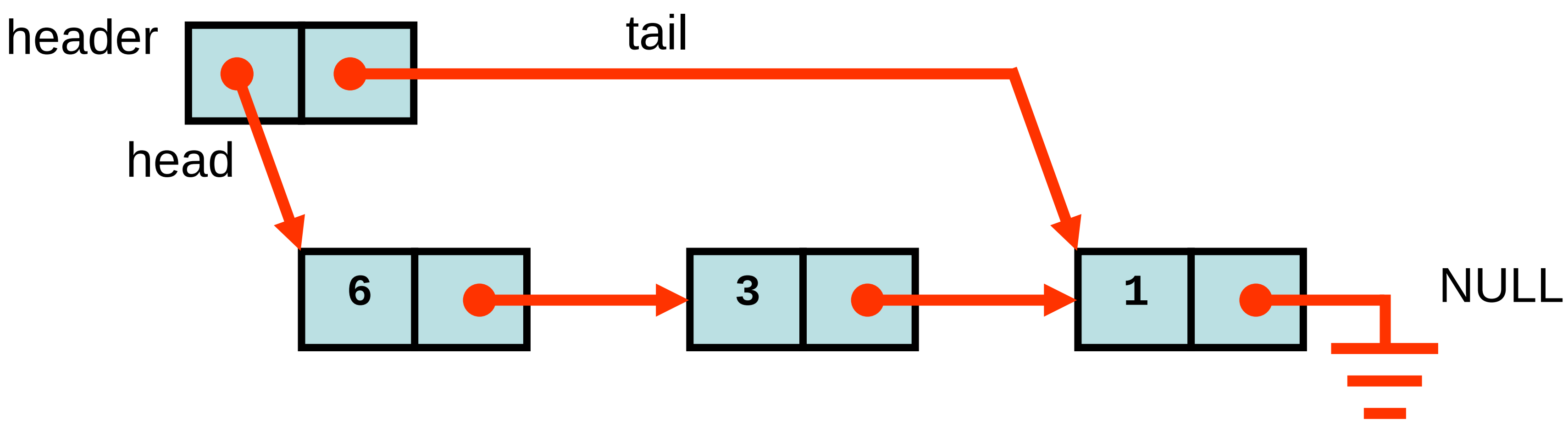
- Assume that the data field stores int type

```
typedef int Info_t;

typedef struct _Node
{
  Info_t data;
  struct _Node *next;
} Node;

typedef struct _Header
{
  Node *head;
  Node *tail;
} Header;
```

Primitive operations

- Empty: return true/false
- First: return address to first (head) node
- Last: return address to last (tail) node
- Insert at head: insert as the first node of list, update head address
- Insert at tail: insert as the last node of list (if we have tail address), update tail address
- Remove at head: remove the first node of list, update head address
- Next: return address of the next node for a given node (address) if node is valid
- Info: return data stored at a given node (address) if node is valid
- All O(1), assuming that malloc and free are both O(1)
- Remove at tail: remove the last node of list, O(n), n is the number of nodes in the list

Remove at Tail




```
Node *prev = NULL;
Node *curr = head;
while (curr != tail) {
    prev = curr;
    curr = curr->next;
}
prev->next = NULL;
tail = prev;
return curr;
```

When list is empty:

```
if (head == NULL) return NULL;
```

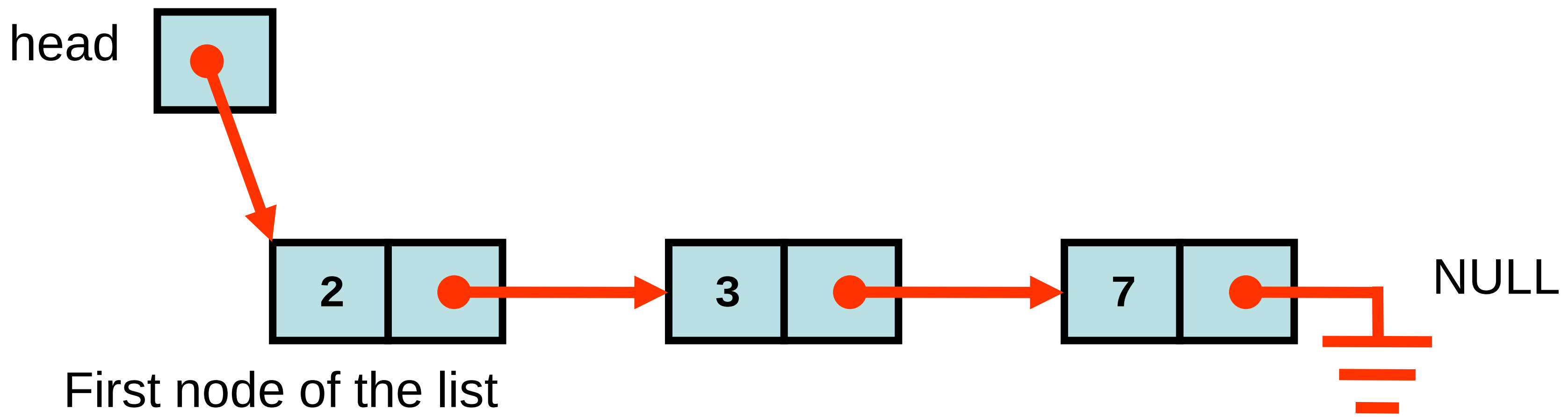
When list has only one node:

```
if (head == tail) {
    Node *curr = head;
    head = tail = NULL;
    return curr;
}
```

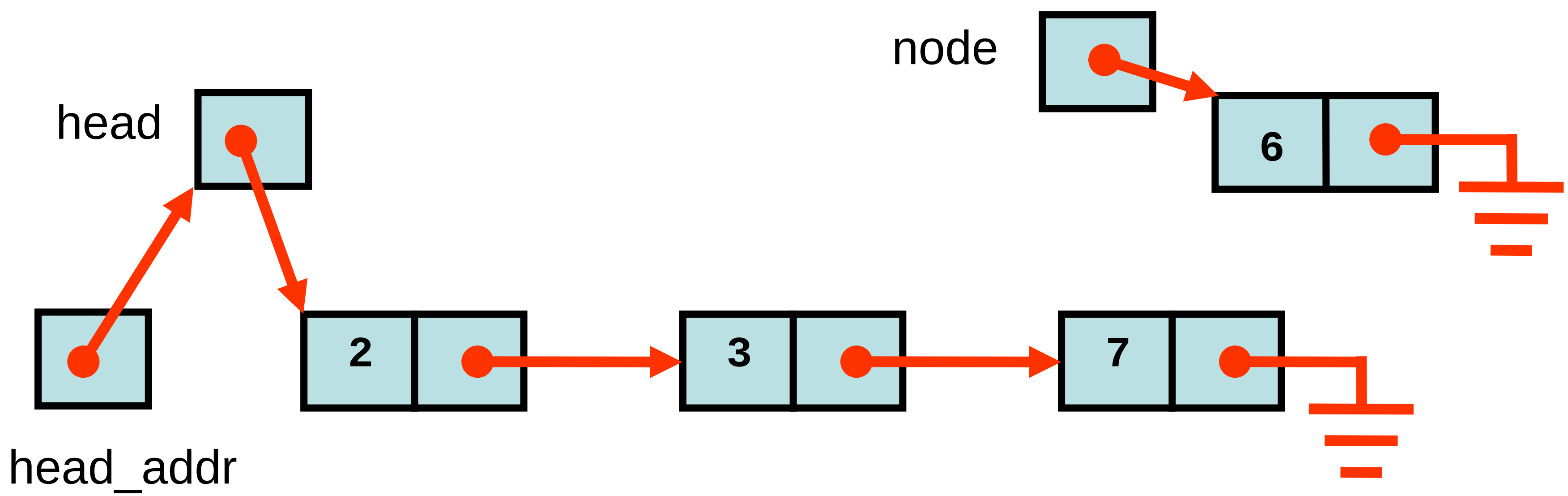
Inserting a node in ascending order (assume no need to keep track of tail)

```
void List_insert_in_order(Node **head_addr, Node
  *node) {

  Node dummy;
  dummy.next = *head_addr;
  Node *curr = dummy.next;
  Node *prev = &dummy;
  while (curr && curr->data < node->data) {
    prev = curr;
    curr = curr->next;
  }
  prev->next = node;
  node->next = curr;
  *head_addr = dummy.next;
}
```

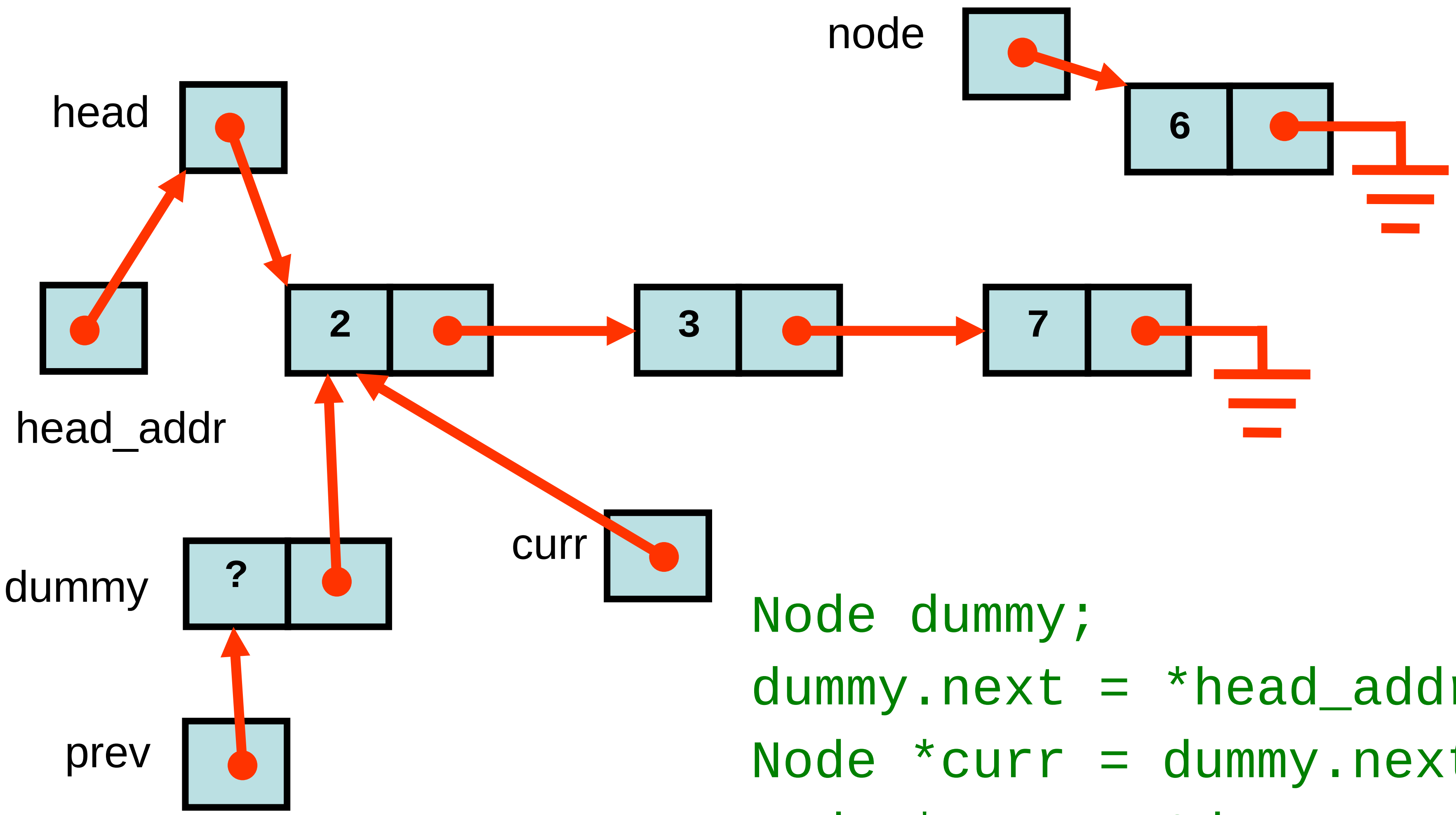
Inserting a node of 6




```
Call List_insert_in_order(&head, Node *node)
```

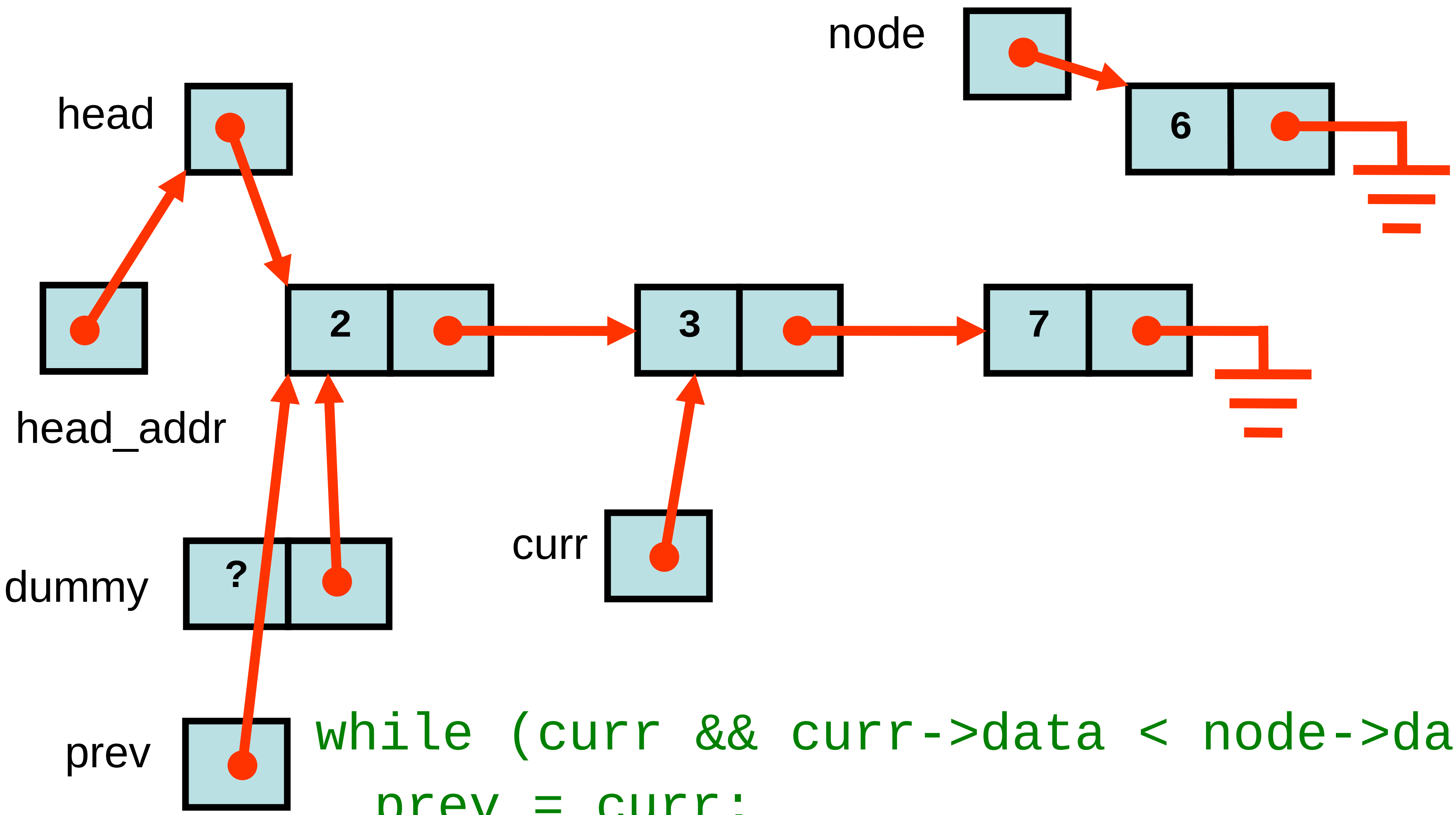


Inserting a node of 6



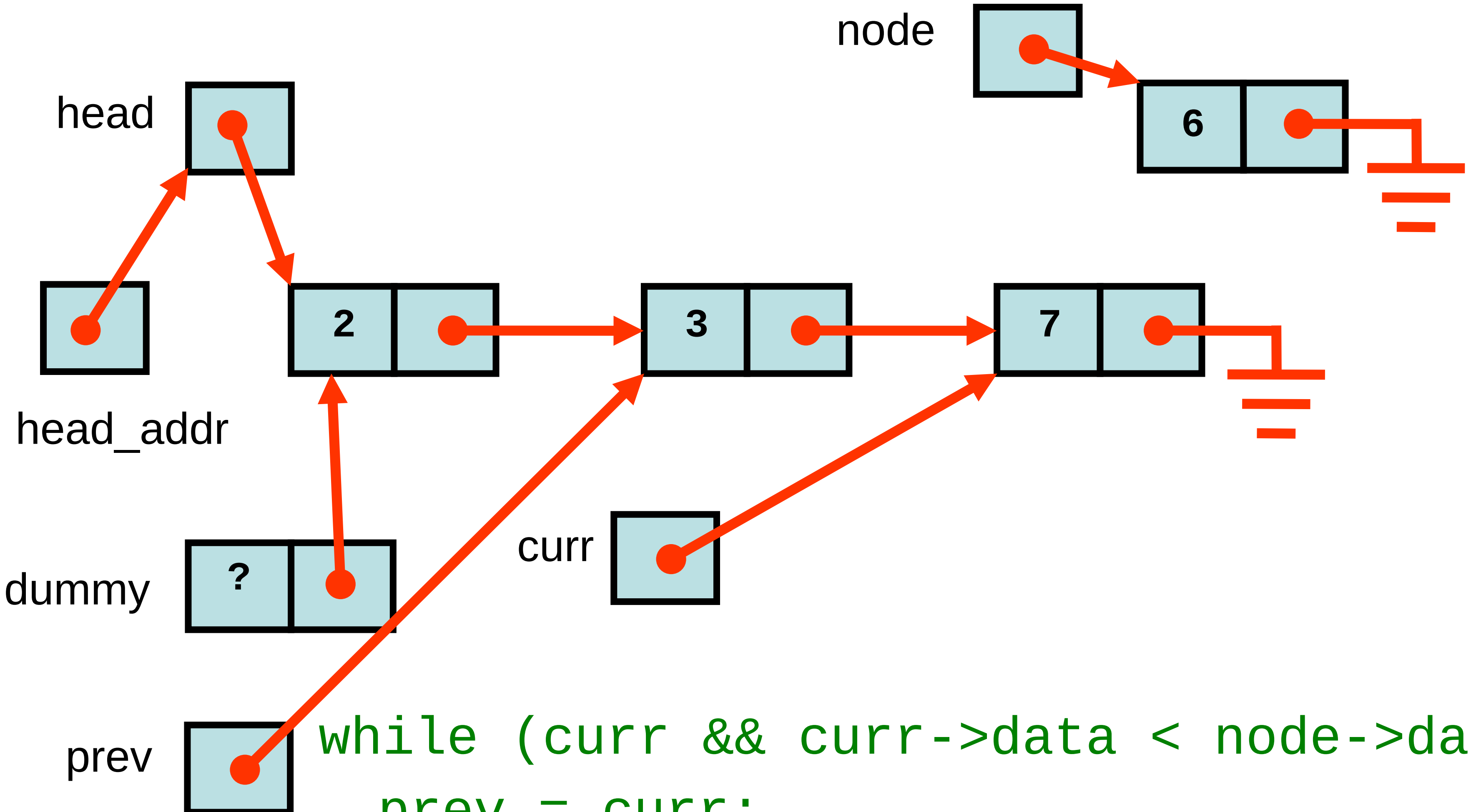

```
Node dummy;
dummy.next = *head_addr;
Node *curr = dummy.next;
Node *prev = &dummy;
```

Inserting a node of 6



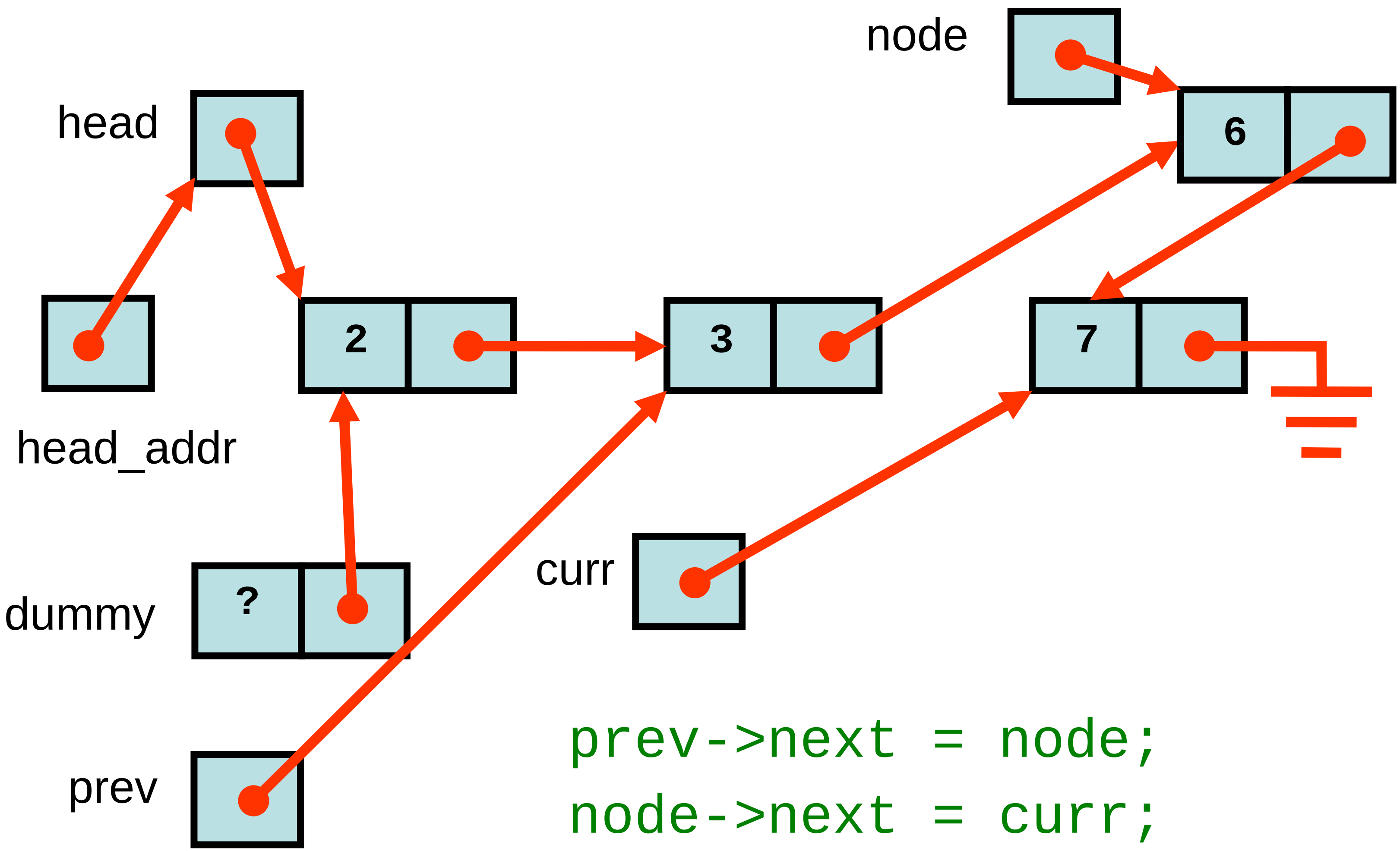

```
while (curr && curr->data < node->data) {
  prev = curr;
  curr = curr->next;
}
```

Inserting a node of 6



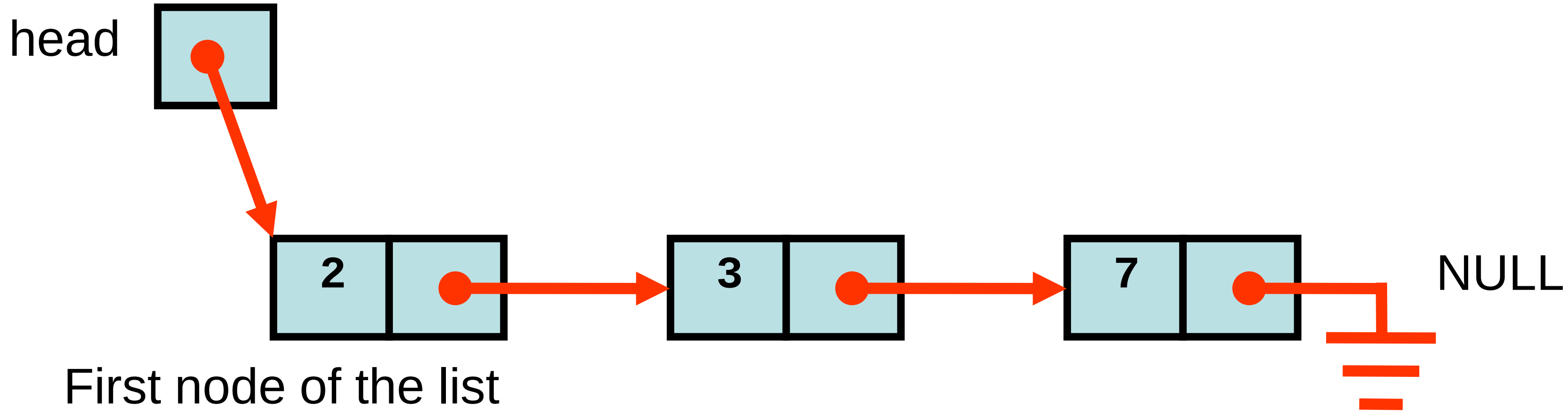

```
while (curr && curr->data < node->data) {
   prev = curr;
   curr = curr->next;
}
```

Inserting a node of 6

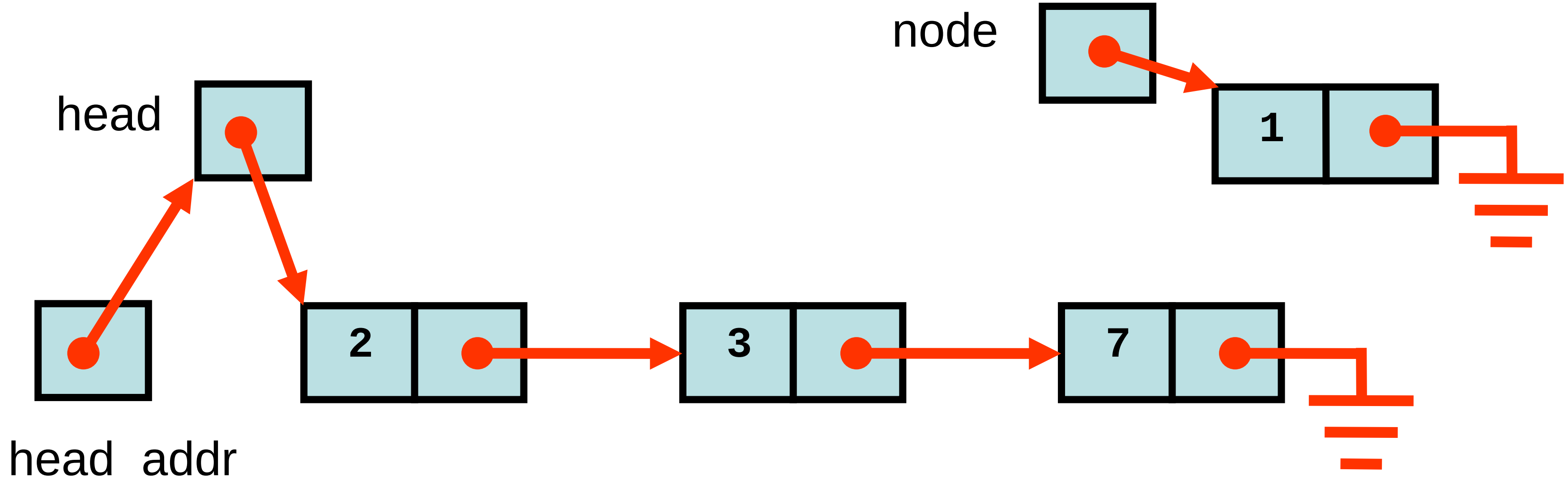



```
prev->next = node;
node->next = curr;
*head_addr = dummy.next;
```

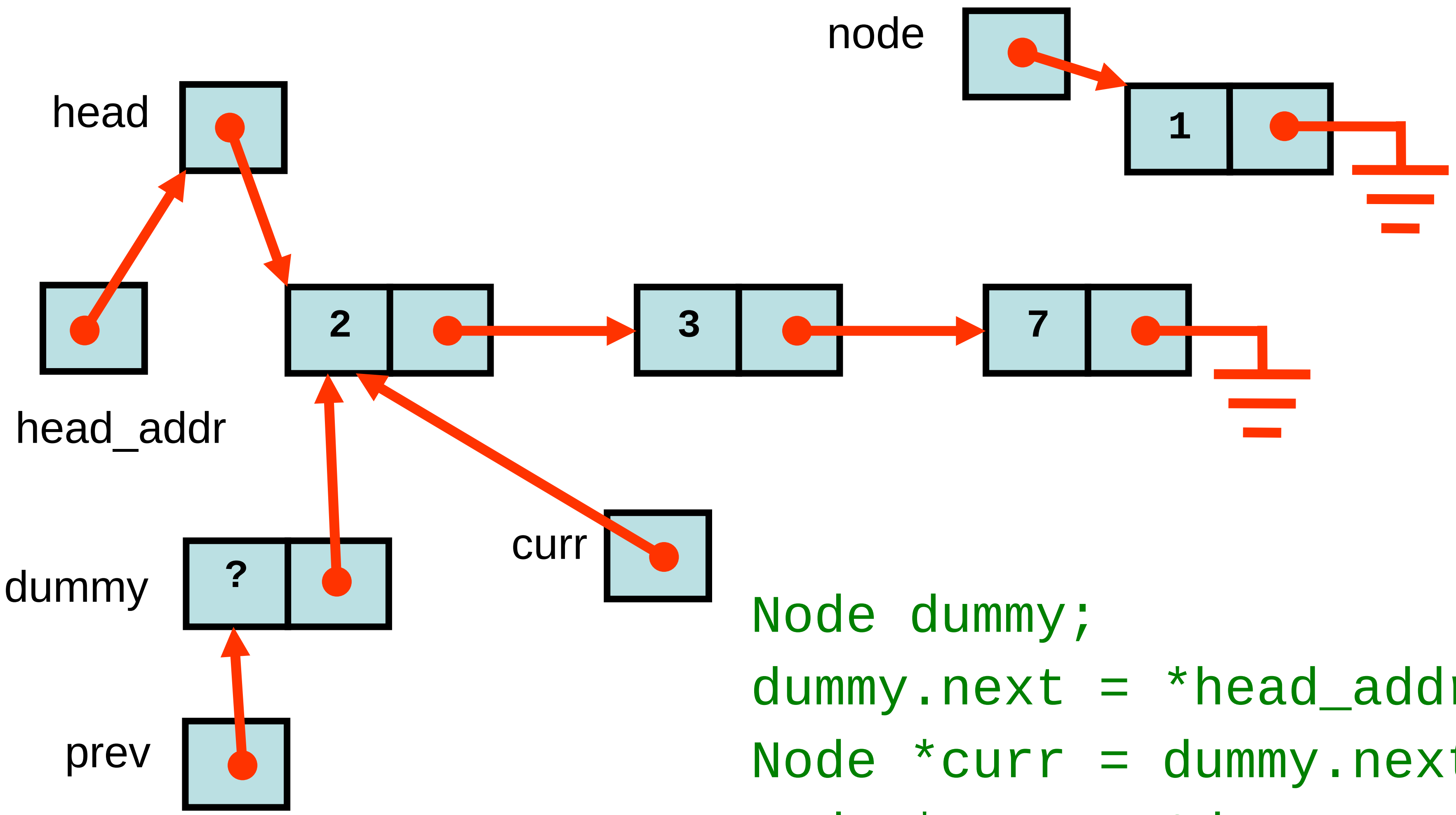
Inserting a node of 1



`Call List_insert_node_in_order(&head, node)`

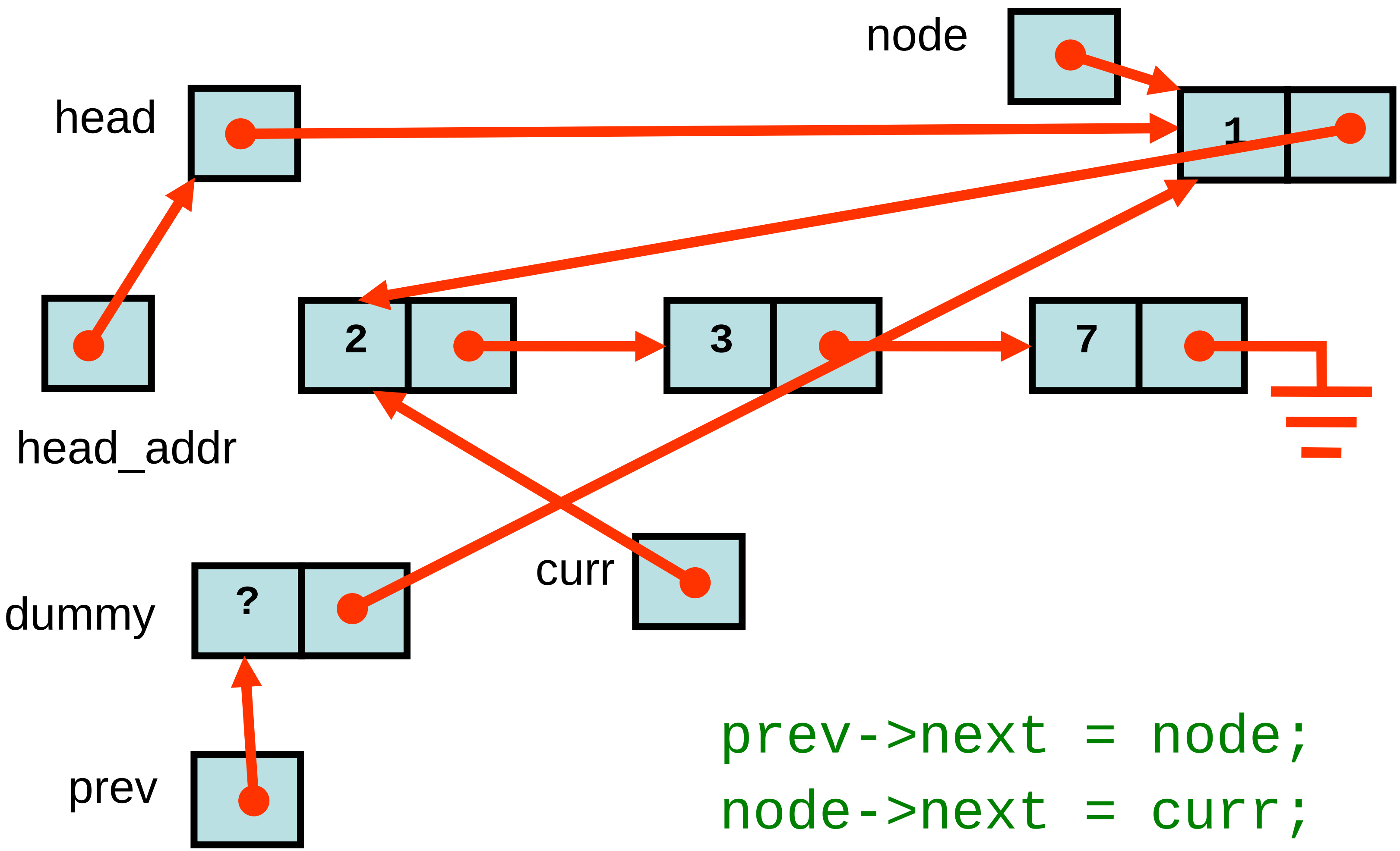


Inserting a node of 1




```
Node dummy;
dummy.next = *head_addr;
Node *curr = dummy.next;
Node *prev = &dummy;
```

Inserting a node of 1




```
prev->next = node;
node->next = curr;
*head_addr = dummy.next;
```

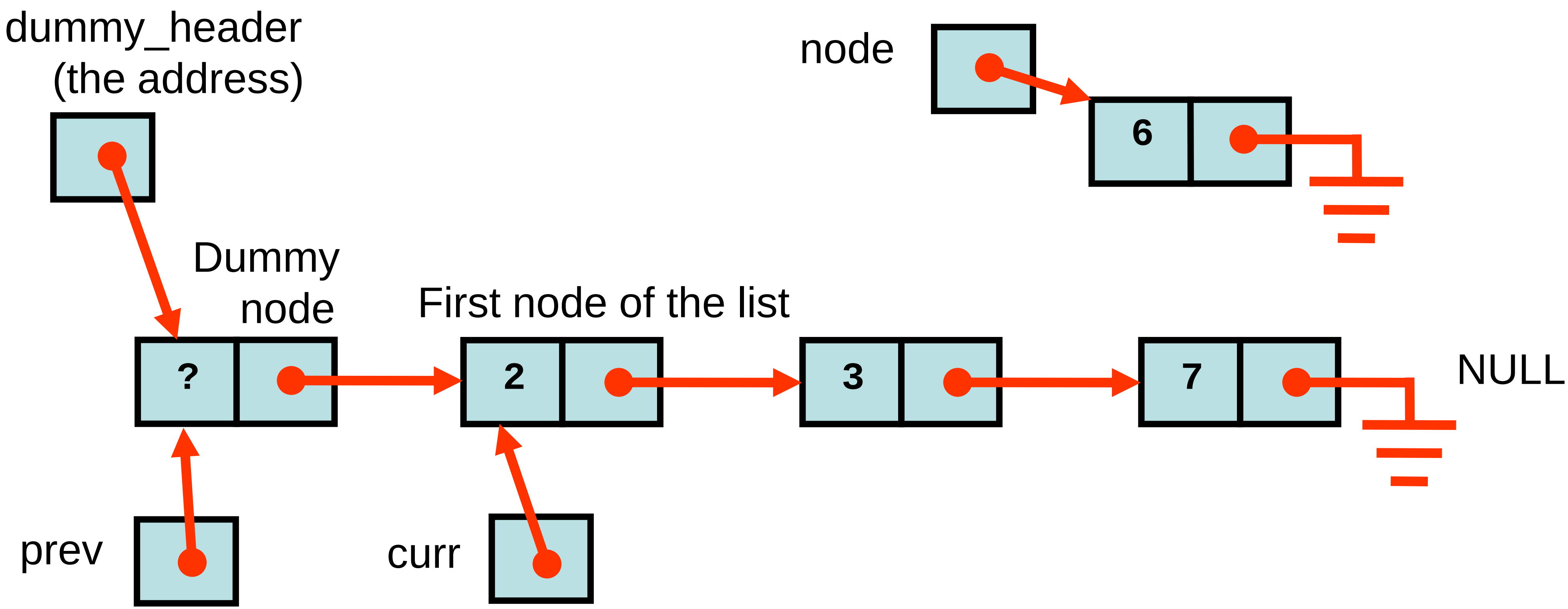
Inserting a node in ascending order (assume no need to keep track of tail)

```
void List_insert_in_order(Node **head_addr, Node
  *node) {

  Node dummy;
  dummy.next = *head_addr;
  Node *curr = dummy.next;
  Node *prev = &dummy;
  while (curr && curr->data < node->data) {
    prev = curr;
    curr = curr->next;
  }
  prev->next = node;
  node->next = curr;
  *head_addr = dummy.next;
}
```

Inserting a node in ascending order (assume dummy header is part of the linked list)

```
void List_insert_node_in_order(Node
  *dummy_header, Node *node) {
  Node *curr = dummy_header->next;
  Node *prev = dummy_header;
  while (curr && curr->data < node->data) {
    prev = curr;
    curr = curr->next;
  }
  prev->next = node;
  node->next = curr;
}
```

Inserting a node of 6

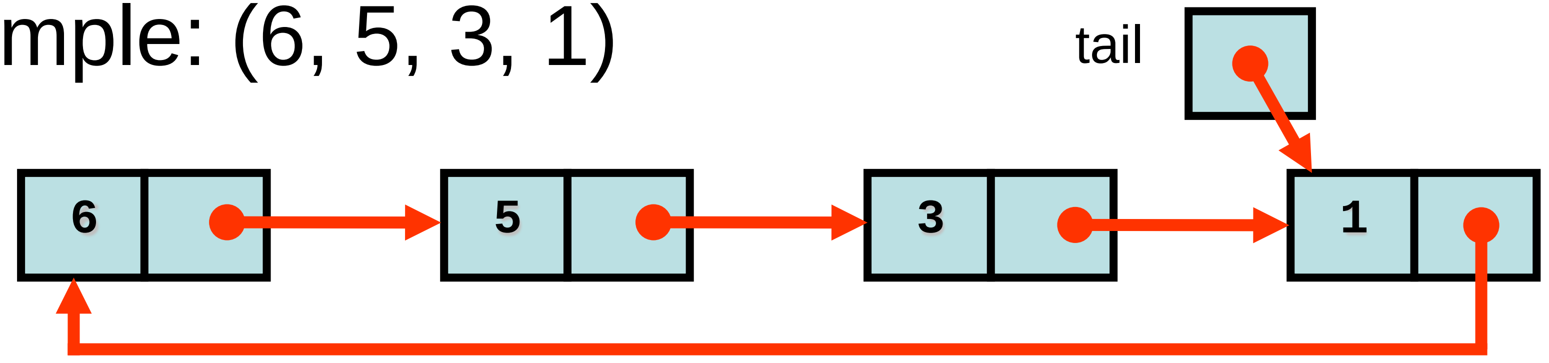



```
Call List_insert_in_order(dummy_header, node)
```

dummy_header is of type Node *

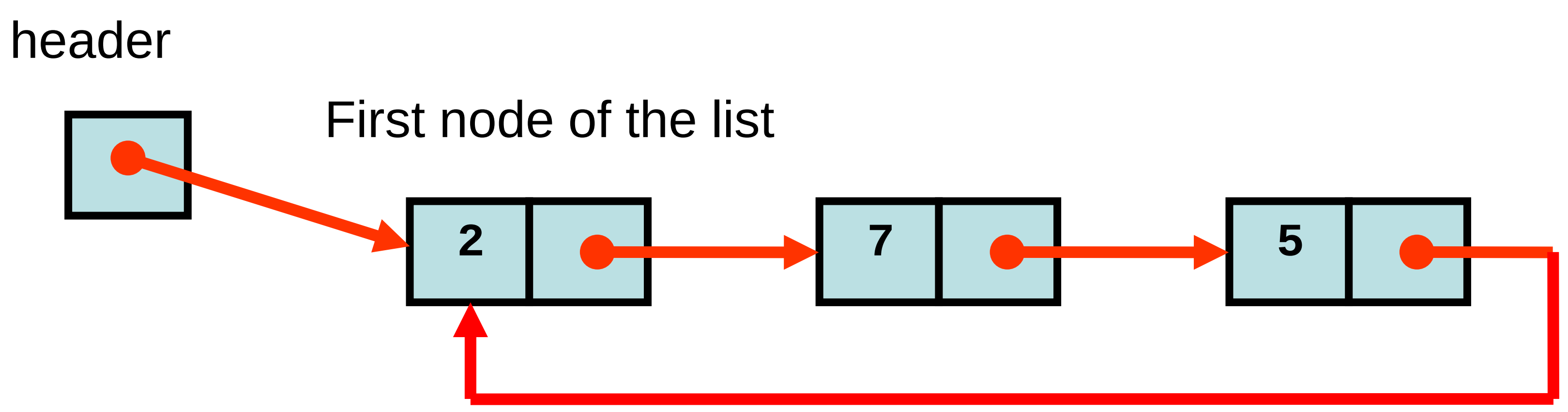
Circular linked lists

- Last node points to the first node
- Instead of storing head address, we maintain only the tail address
- Obtain the head address by using next(tail)
  - Of course, if there is no need to keep track of tail, can keep track of head directly
- Example: (6, 5, 3, 1)



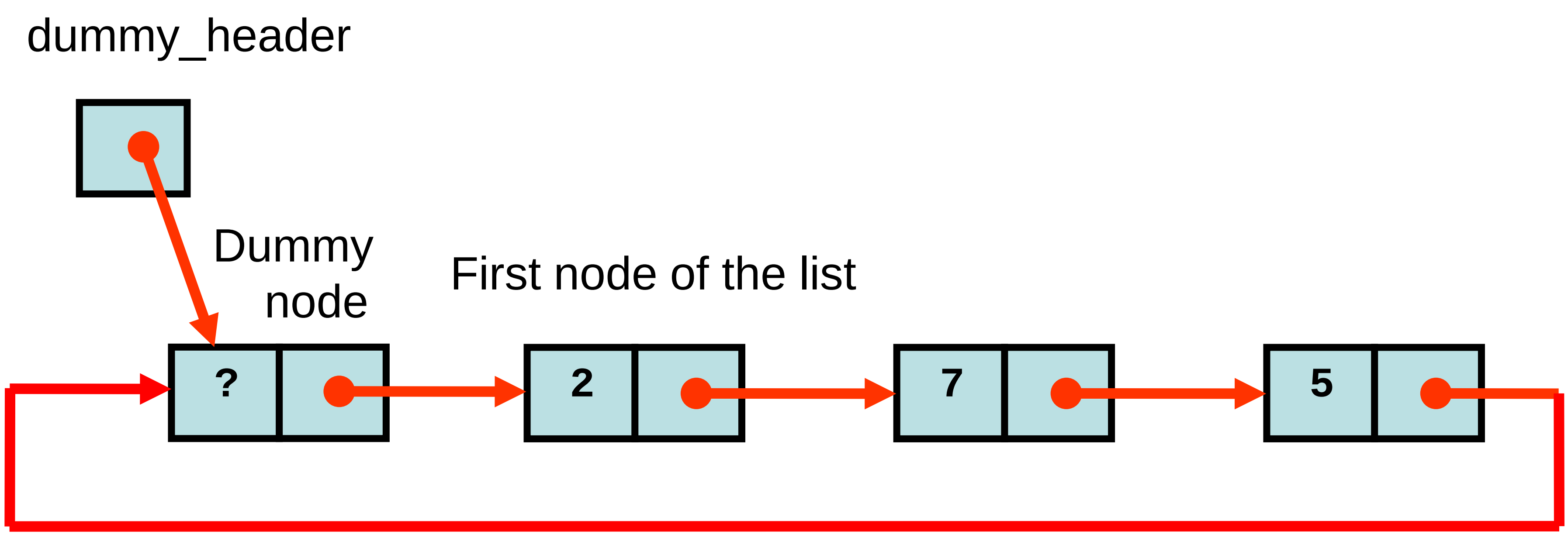
Search with sentinel

- Assume that there is no need to keep track of tail
- In a conventional search, must check whether an address points to a valid node before we check on the data stored in the node
- Use sentinel to avoid checking the validity of address



Search with sentinel

- Assume that there is no need to keep track of tail, and there is always a dummy header
  - dummy_header is of type Node *
- In a conventional search, must check whether an address points to a valid node before we check on the data stored in the node
- Use sentinel to avoid checking the validity of address



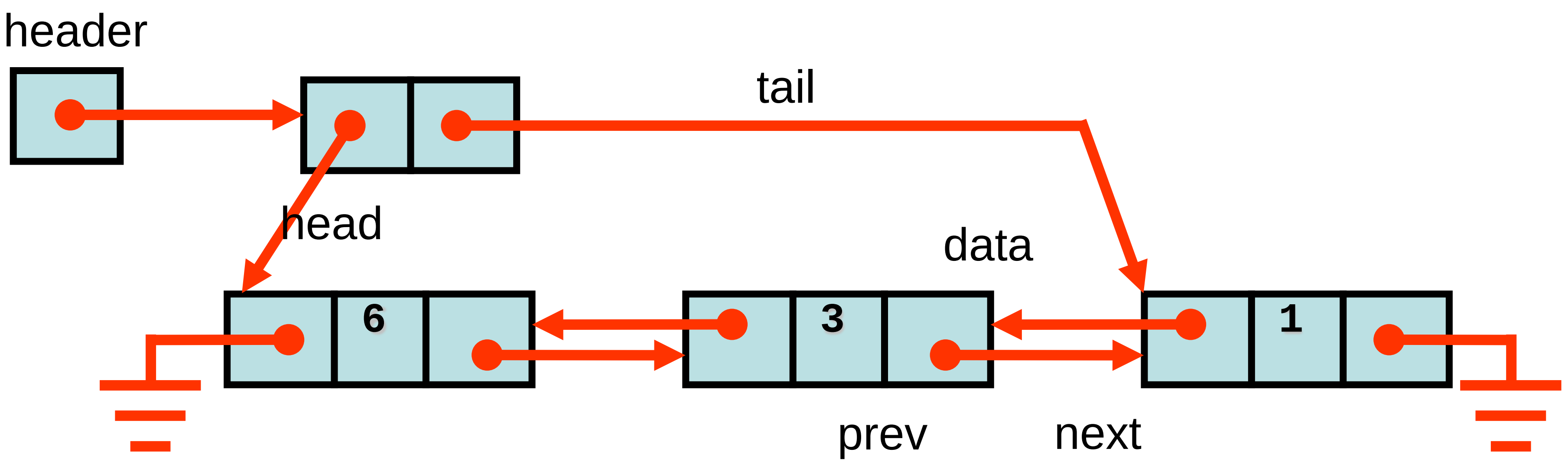
Search with sentinel

- Store the search key in the dummy header
- Search for the key, with the list traversal starting from the node after the dummy header

```
Node *List_search(Node *dummy_header, Info_t key)
{
  dummy_header->data = key;
  Node *curr = dummy_header->next;
  while (curr->data != key) {
    curr = curr->next;
  }
  return (curr != dummy_header) ? curr : NULL;
}
```

Doubly-linked lists

- Each node stores data, the address of the next node (in the “next” field), and the address of the previous node (in the “prev” field)
- Example list: (6, 3, 1) (header is of type Header *)



- 4 addresses to be updated for insertion and 2 addresses for deletion

Deleting from doubly-linked list

```
List-Delete(header, x):
  if x->prev != NULL
    (x->prev)->next ← x->next
  else
    header->head ← x->next

  if x->next != NULL
    (x->next)->prev ← x->prev
  else
    header->tail ← x->prev
```

Some other function has to free x

Doubly-linked lists vs. singly-linked lists

- Advantages of doubly-linked list:
  - Previous node available!
  - Can remove a node easily if address is known
  - Can insert a node easily if insertion location is known
- Disadvantages of doubly-linked list:
  - More memory
  - More overhead (management of links)

Doubly-linked list with sentinel

- A dummy header that represents "NULL" but has all fields of other list elements
  - dummy_header is of type Node *
- Effectively a circular doubly-linked list
- Example list: (6, 3, 1)

```
List-Delete(x):
   (x->prev)->next ← x->next
   (x->next)->prev ← x->prev
```