ICOM 4035 – Data Structures

Dr. Manuel Rodríguez Martínez
Electrical and Computer Engineering Department

Readings

- Chapter 17 of textbook: Linked Lists

What is missing?

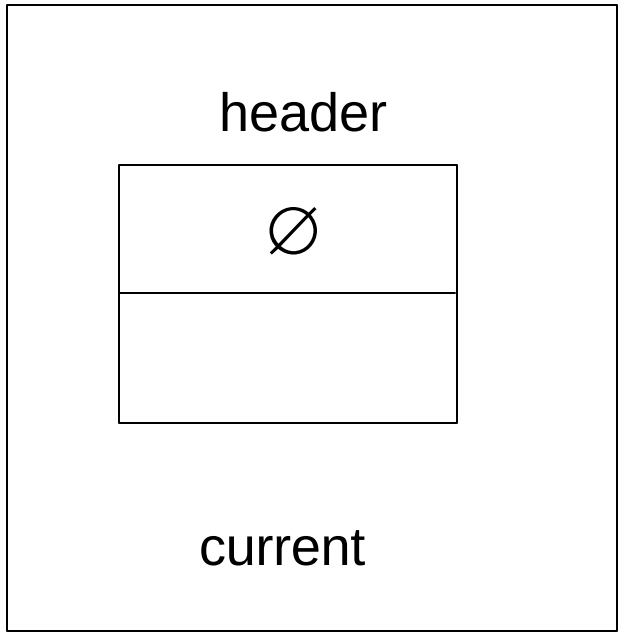
- We have not implement the following operations:
  - Copy constructor
  - Copy assignment operator
  - Find operator
  - First operator
  - Insert after position
- Reason: They all need to have an entity that provide a position anywhere in the list.
  - This position cannot be a pointer
  - This position will be implemented as a list iterator

List Iterator

- List iterator is a helper class for the linked list class.
- It provides the abstraction of a current position at a given node in the list.
- The iterator hides the pointer used to access the given node in the list.
- Instead, it provides method that can be used to
  - Access the data in a node.
  - Move to the next node from the current one.
- Iterator can be used to traverse the entire list without having access to any of the pointers.

List Iterator structure

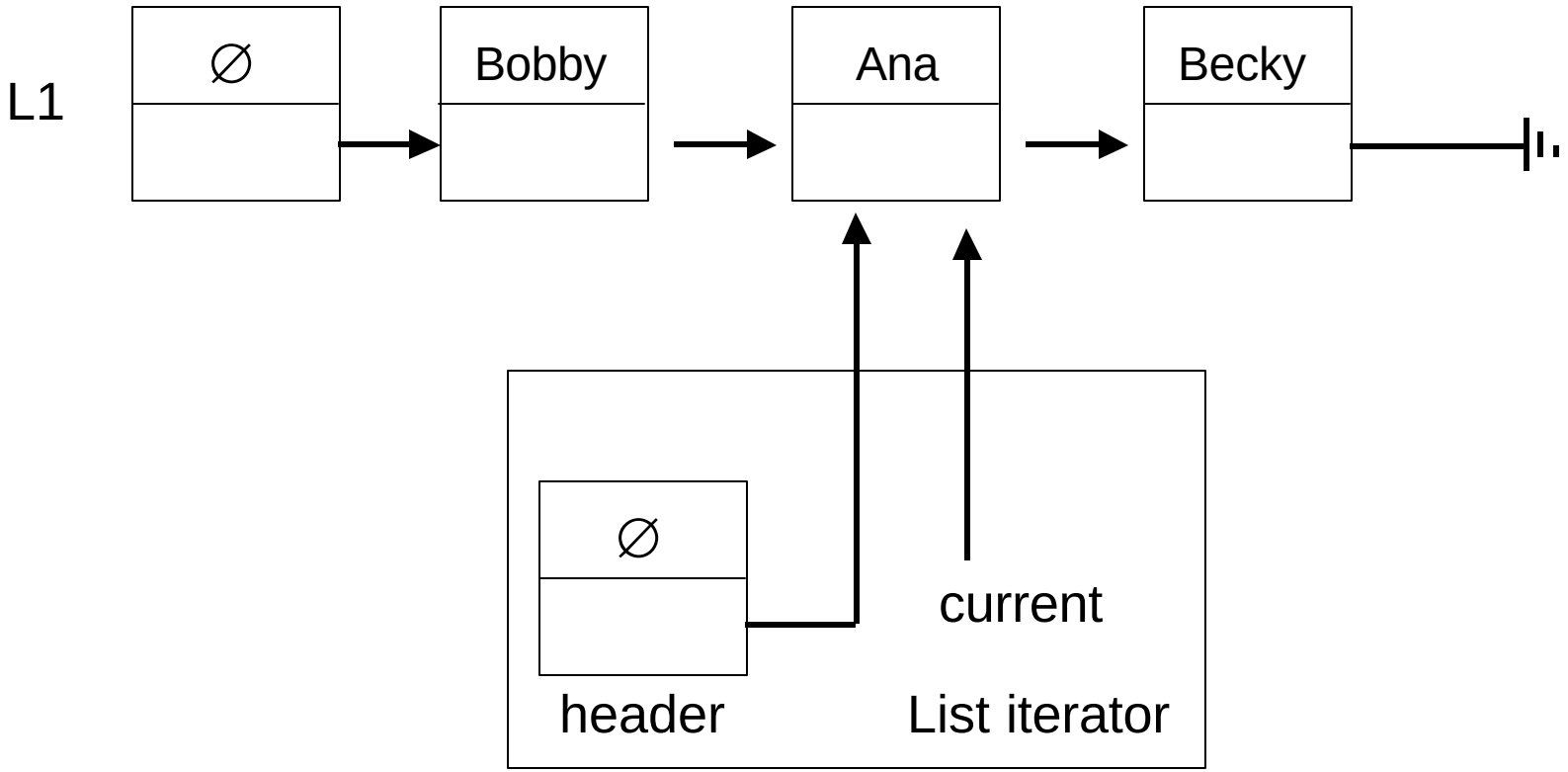
- List iterator has two private fields:
  - Header – point to a node with no data, whose next pointer points to the element on which the iterator is first stationed.
  - Current – points to the node in which the iterator is currently stationed. This value can change during execution.



List iterator

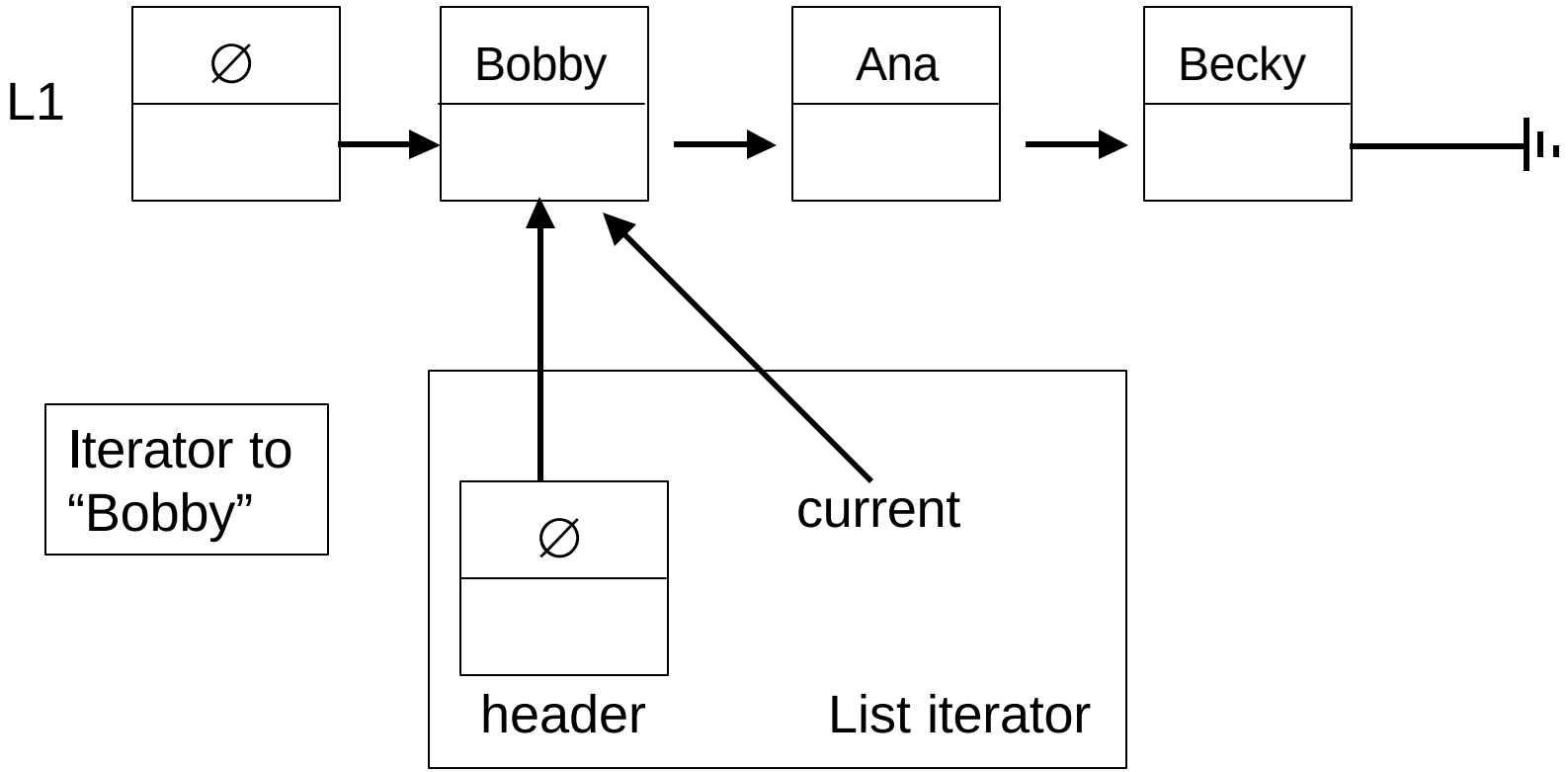
Example

- Here we have an iterator to the node with element Ana.



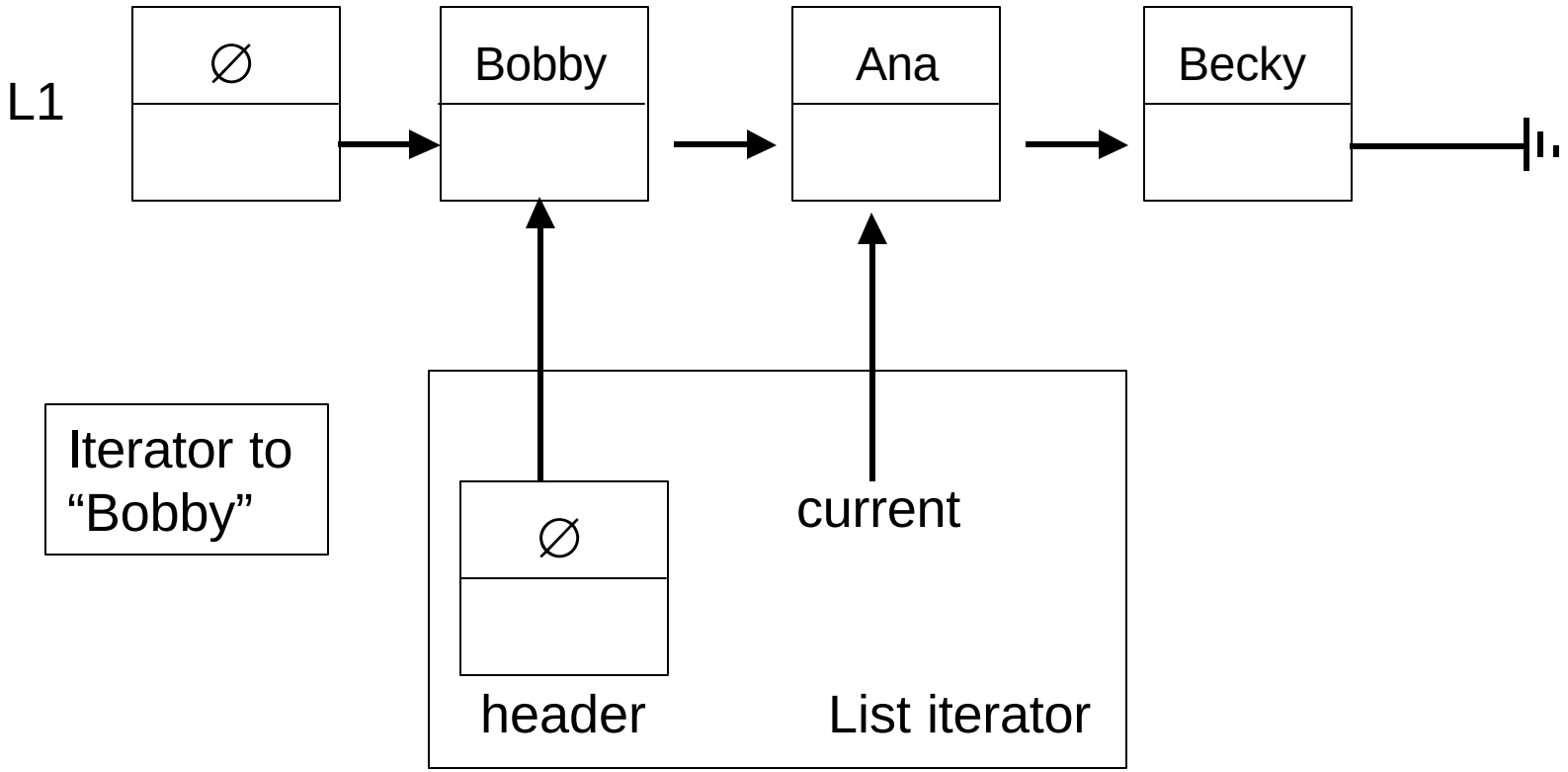
Iterator can traverse the list

- An iterator can be used to move over list by moving the current pointer.



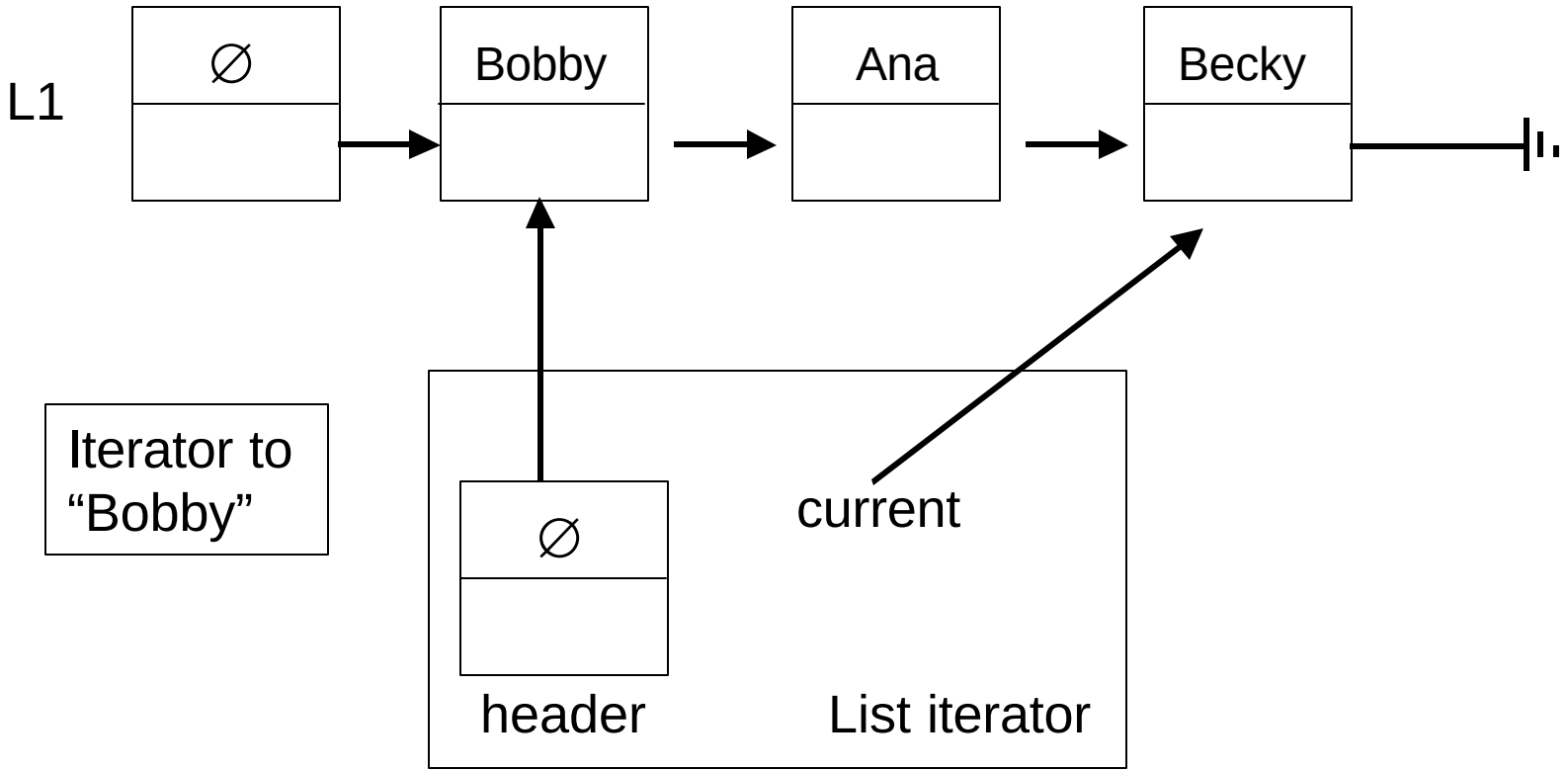
Iterator can traverse the list

- Move to next from the current (“Ana”)



Iterator can traverse the list

- Move to next from the current (“Becky”)



Iterator methods

- Get the code from class Web site.
- Constructors – private members so only linked list can call them (linked list is declared as friend class)
- has_data – indicates if current node has data
- get_data – returns data field from current node
- next – moves current position of iterator the next node from current one.
- reset – returns the iterator current position to its original position
- insert – add a new element after the current position in the iterator.

Constructor

- Creates a new header, and makes it point to node. Current is also set to this node.

```
template <typename ListData>
list_iterator<ListData>::list_iterator
(Node<ListData> *p = NULL){
    header = new Node<ListData>;
    header->next = p;
    current = p;
}
```

Has Data

- Iterator has data if current is not null.

```
template <typename ListData>
bool list_iterator<ListData>::has_data() const {
    return current != NULL;
}
```

Get Data

- Returns data at current node

```
template< typename ListData>
ListData& list_iterator<ListData>::get_data() {
    assert(has_data());
    return current->data;
}
```

Next method

- Make current move one spot to the next of current.

```
template <typename ListData>
void list_iterator<ListData>::next(){
    if (current != null){
        current = current->next;
    }
}
```

Reset method

- Brings the iterator back to the original node to which it began pointing to.

```
template <typename ListData>
void list_iterator<ListData>::reset(){
    current = header->next;
}
```

Insert method

- Adds an element after current (non-NULL) position.

```
template <typename ListData>
void list_iterator<ListData>::insert
(const ListData& obj){
    Node<ListData> *temp = NULL;
    if (current != NULL){
            temp = new Node<ListData>;
            temp->data = obj;
            temp->next = current->next;
            current->next = temp;
    }
}
```

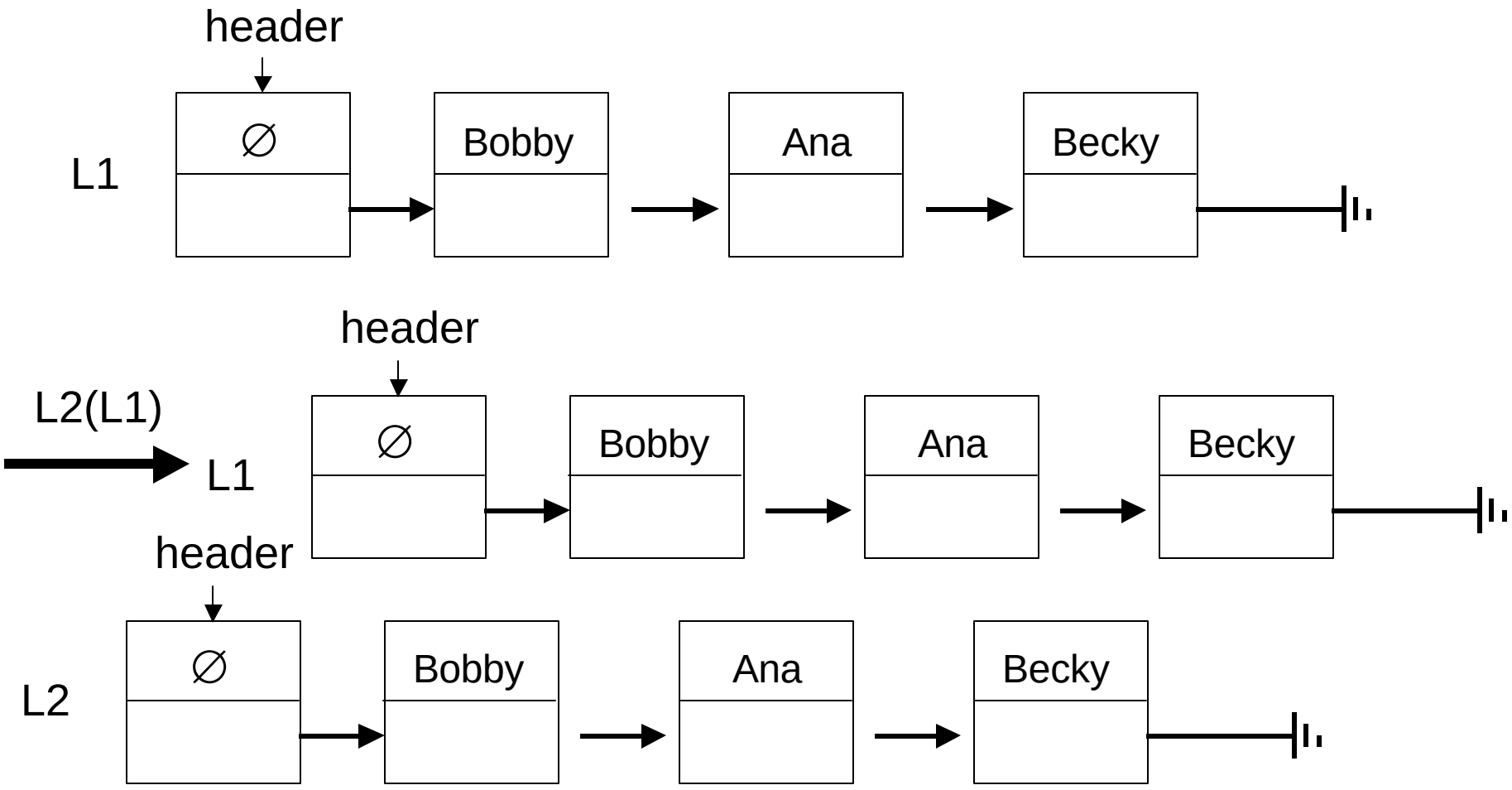
Linked List: missing method

- Now we can use the list iterator to implement the missing methods from the linked list iterator.

- Note: An iterator instance must be associated with a given linked list instance.

Copy constructor

Makes a deep copy (element by element) of a list L1 into a new list L2.



Copy constructor

```
template <typename ListData>
Linkedlist<ListData>::linkedlist(const linkedlist<ListData>& L){
    Node<ListData> *temp1=NULL, *new_node = NULL;
    header = new Node<ListData>;
    header->next = NULL;
    for (temp1 = header->next, list_iterator<ListData> iter = L.first();
          iter.has_data(); temp1 = temp1->next, iter.next()) {
          new_node = new Node<ListData>;
          new_node->data = iter.get_data();
          new_node->next = NULL;
          temp1->next = new_node;
          ++list_size;
    }
}
```

Copy Assignment

```
template <typename ListData>
const& linkedlist<ListData> linkedlist<ListData>::operator
    (const linkedlist<ListData>& L){
    Node<ListData> *temp1=NULL, *new_node = NULL;
    if (this != &L){
            make_empty();
            for (temp1 = header->next, list_iterator<ListData> iter = L.first();
                 iter.has_data(); temp1 = temp1->next, iter.next()) {
                        new_node = new Node<ListData>;
                        new_node->data = iter.get_data();
                        new_node->next = NULL;
                        temp1->next = new_node;
                        ++list_size;
            }
    } return *this;
}
```

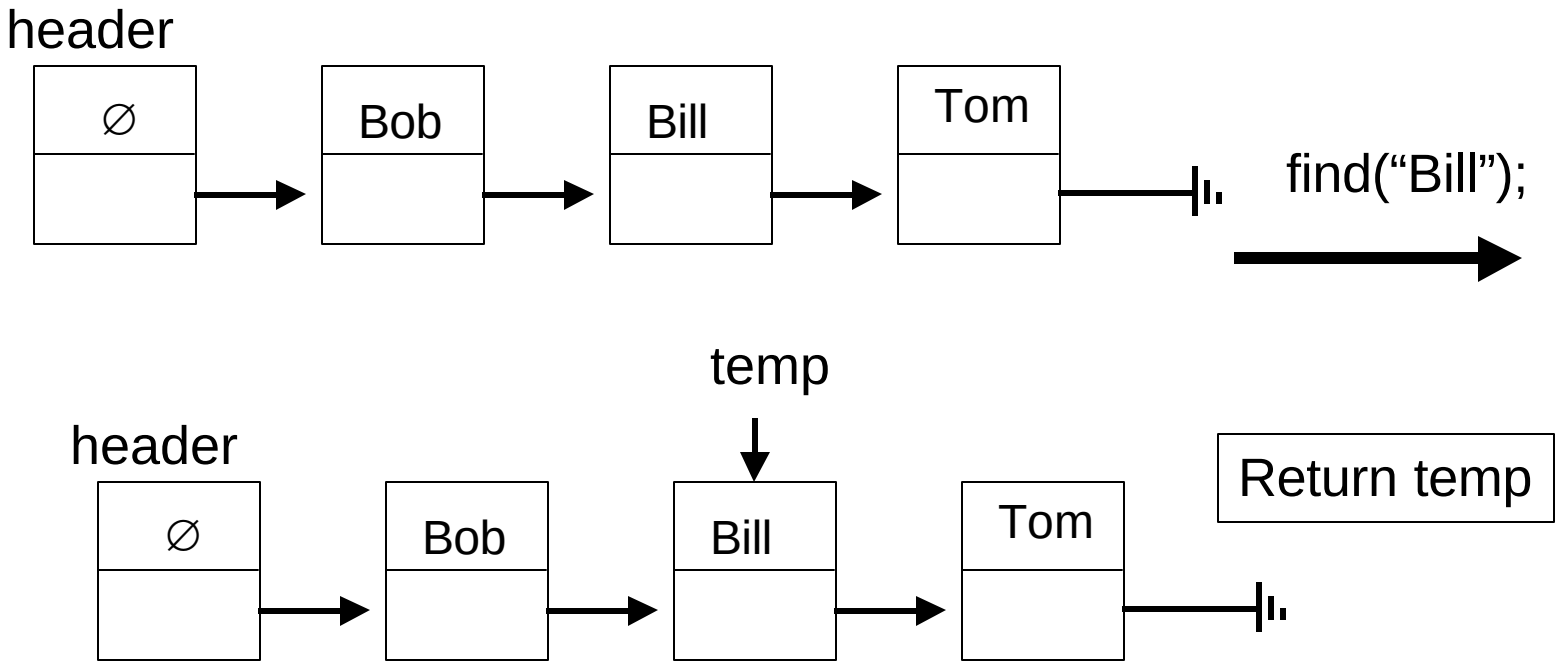
Insert after a position

- Adds a new element after a position in the list (a pointer to a node)
  - Obtained from either find() or first()

```
template <typename ListData>
void linkedlist<ListData>::insert(const ListData& obj,
                                  list_iterator<ListData>& iter){
    Node<ListData> *temp = NULL;
    temp = new Node<ListData>; // creates the new node
    temp->data = obj; // adds the data to it
    iter.insert(temp); // insert after current iterator position
    ++list_size;
}
```

Find operation

- Returns a pointer to the position at which an object obj is located. Returns empty iterator if the element is not in the list.



Find operation

```
template <typename ListData>
list_iterator<ListData> linkedlist<ListData>::find
    (const Object& obj) const{
    Node<ListData> *temp=NULL;

    for (temp = header->next; temp != next;
            temp = temp->next){
            if (temp->data == obj){
                    return list_iterator<ListData>(temp);
            }
    }
    // not found
    return list_iterator<ListData>(NULL);
}
```

First operation

- Returns position (iterator) to the first element in the list.

```
template <typename ListData>
list_iterator<ListData> linkedlist<ListData>::first
    () const{
    return list_iterator<ListData>(header->next);
}
```