ICOM 4035 – Data Structures

Dr. Manuel Rodríguez Martínez
Electrical and Computer Engineering Department

Readings

- Chapter 17 of textbook: Linked Lists

Linked Lists

- A linked list is a container used to store elements sequentially.
- Unlike the array (and the vector) the linked list does not need to have a fixed size.
  - Instead the linked list can be grown as much as necessary during run time.
- But unlike arrays, the memory allocated for array elements is not contiguous, so the cost of accessing in the linked list is larger.

Basics of linked list

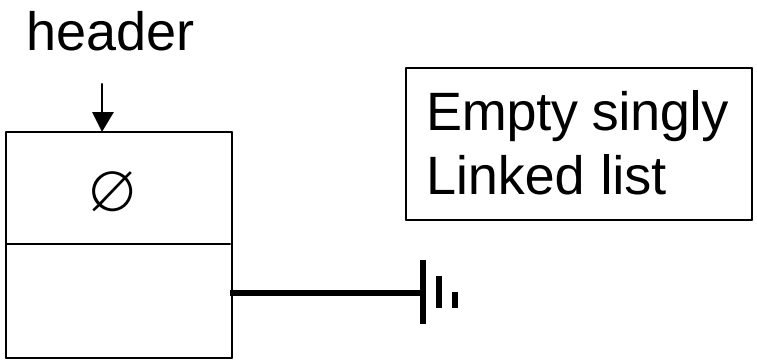
- The basic form the linked list is called the singly linked-list
- It stores elements of given type Object in a “node”.
- Each node points to the next elements in the list, except for the last element which points to the value NULL.
- The list has a special element called the header that stores no data.
- The header points to the first element in the list, or to the value NULL if the list is empty.

Singly linked list node

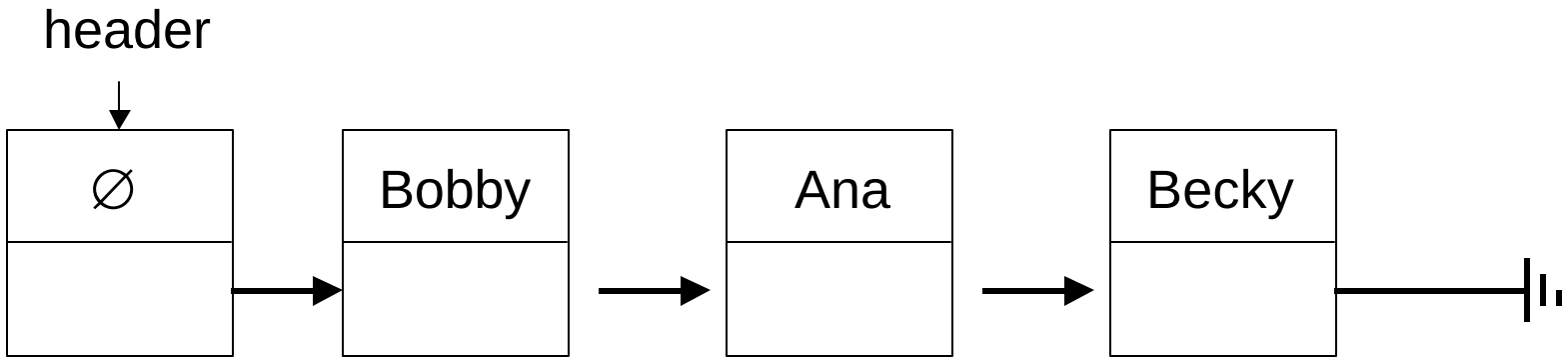
- // Make object be a string

```
typedef Object string;
// define a node for the linked list
struct Node {
    // data field
    Object data;
    // pointer to the next node in the list
    Node *next;
};
```

Sample linked lists



Empty singly Linked list



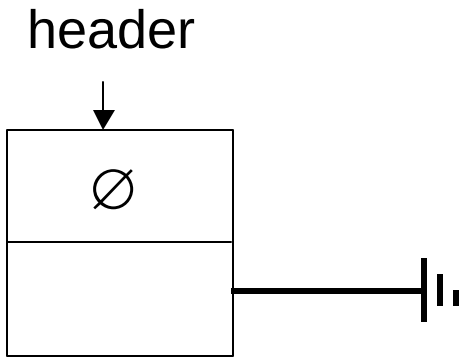
Singly linked list with three elements

Linked list interface

```
class linkedlist{
    public:
        linkedlist();
        linkedlist(cons linkedlist &L);
        ~linkedlist();
        const linkedlist& operator= (const linkedlist &L);
        bool is_empty() const;
        void insert(const Object& obj);
        bool erase(const Object& obj);
        Node *find(const Object& obj) const;
        Node *first() const;
        void make_empty() const;
        int size();
    private:
        Node *header;
        int list_size;
};
```

Constructor

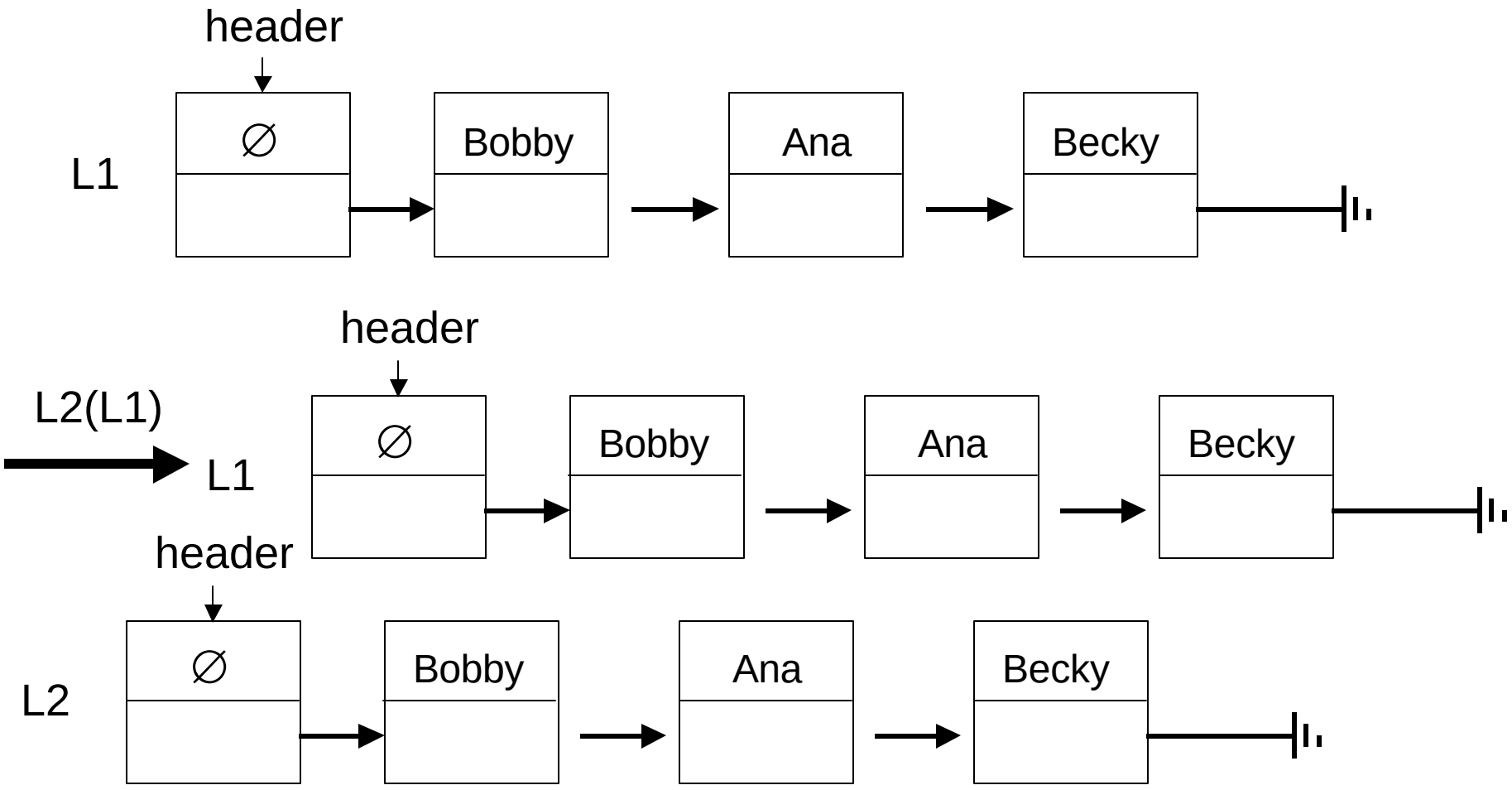
- Makes an empty list.




```
linkedlist::linkedlist(){
    header = new Node;
    header->next = NULL;
    list_size = 0;
}
```

Copy constructor

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



Copy constructor

```
linkedlist::linkedlist(const linkedlist& L){
    Node *temp1=NULL, *temp2=NULL, *new_node = NULL;
    header = new Node;
    header->next = NULL;
    for (temp1 = header, temp2  = L.first(); temp2 != NULL;
            temp1 = temp1->next, temp2 = temp2->next) {
            new_node = new Node;
            new_node->data = temp2->data;
            new_node->next = NULL;
            temp1->next = new_node;
            ++list_size;
    }
}
```

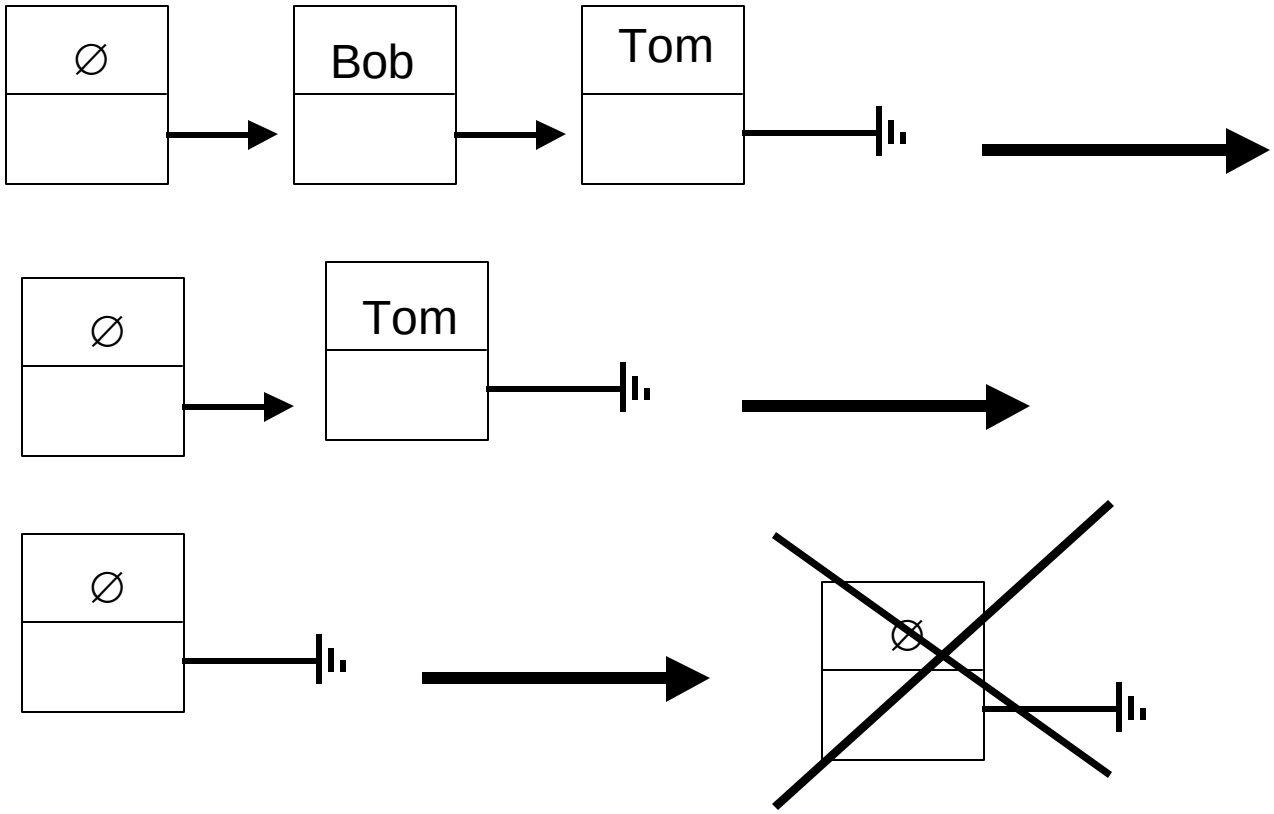
Copy assignment

- Pretty much like copy constructor

```
const linkedlist& linkedlist::linkedlist(const linkedlist& L){
    Node *temp1=NULL, *temp2=NULL, *new_node = NULL;
    if (this != &L){
            header = new Node;
            header->next = NULL;
            for (temp1 = header, temp2  = L.first(); temp2 != NULL;
                        temp1 = temp1->next, temp2 = temp2->next) {
                        new_node = new Node;
                        new_node->data = temp2->data;
                        new_node->next = NULL;
                        temp1->next = new_node;
                        ++list_size;
            }
    }
    return *this;}
```

Destructor

- Removes all elements from the list and then deletes the header.

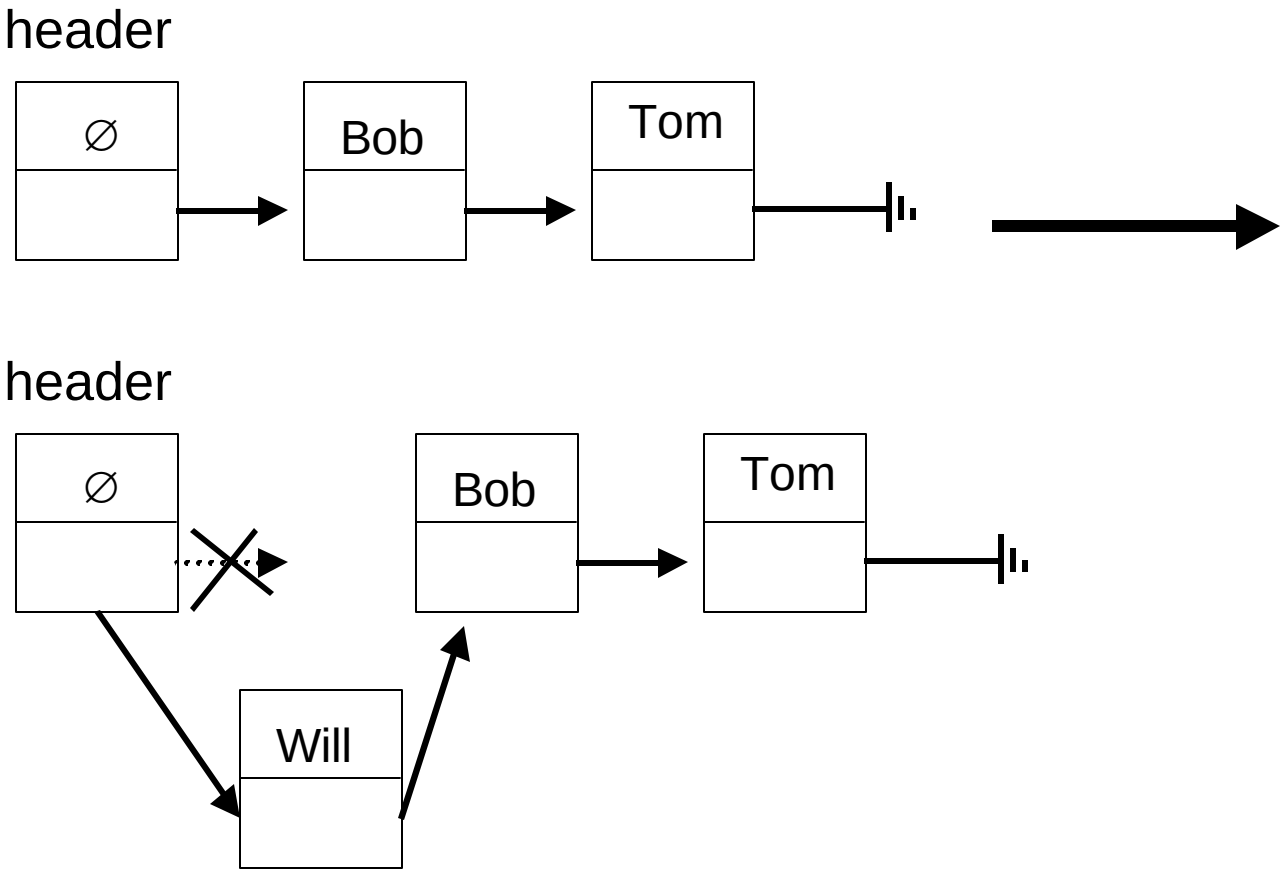


Destructor

```
linkedlist::~linkedlist(){
    // remove all the stuff from the list
    make_empty();
    // remove the header
    delete header;
}
```

Insert operation

- Adds a new element after the header of the list



Insert operation

```
void linkedlist::insert(const Object& obj){
    temp = new Node; // creates the new node
    temp->data = obj; // adds the data to it
    temp->next = header->next; // adds it to the list
    header->next = temp; // make header point to it
    ++list_size; // increase list size
}
```

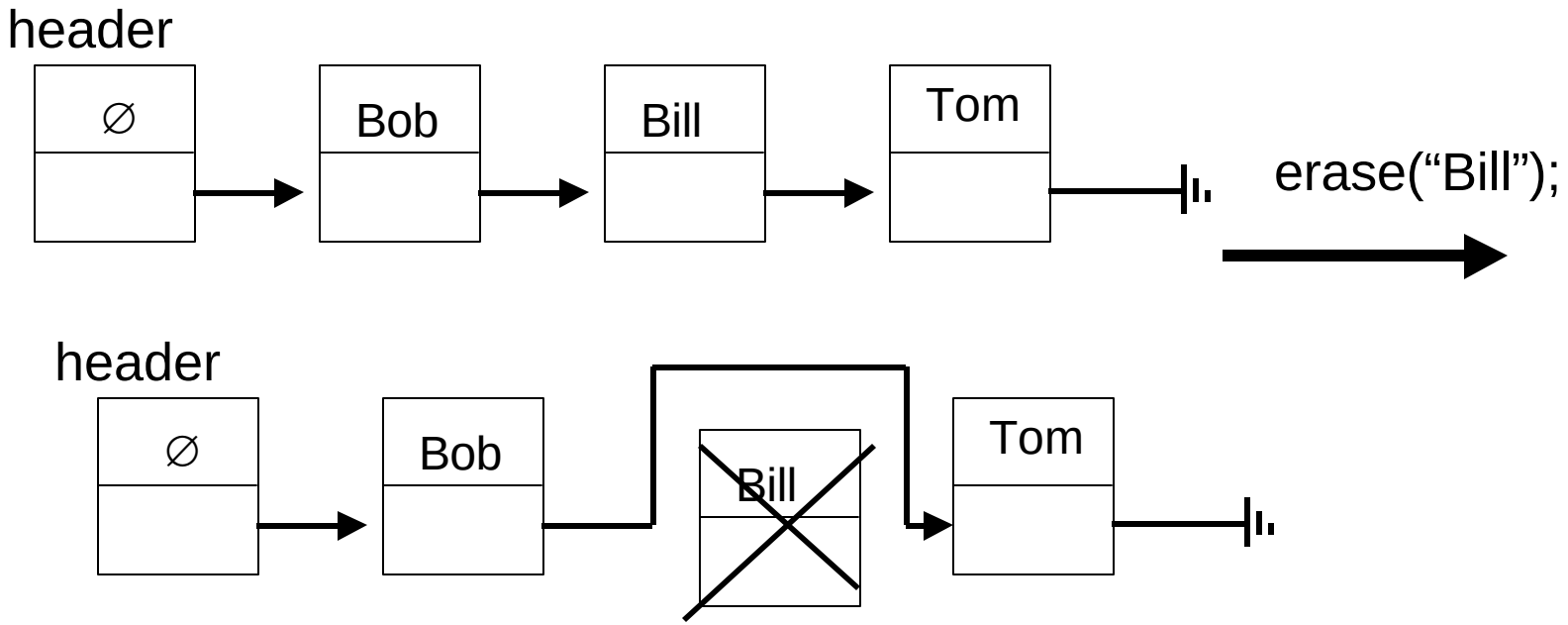
Insert after a position

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

```
void linkedlist::insert(const Object& obj, Node *pos){
    assert(pos != NULL);
    temp = new Node; // creates the new node
    temp->data = obj; // adds the data to it
    temp->next = pos->next; // adds it to the list
    pos->next = temp; // make pos point to it
    ++list_size;
}
```

Erase operation

- Removes the first occurrence of an object from the list.
- Returns true if the element is deleted, false if not found.

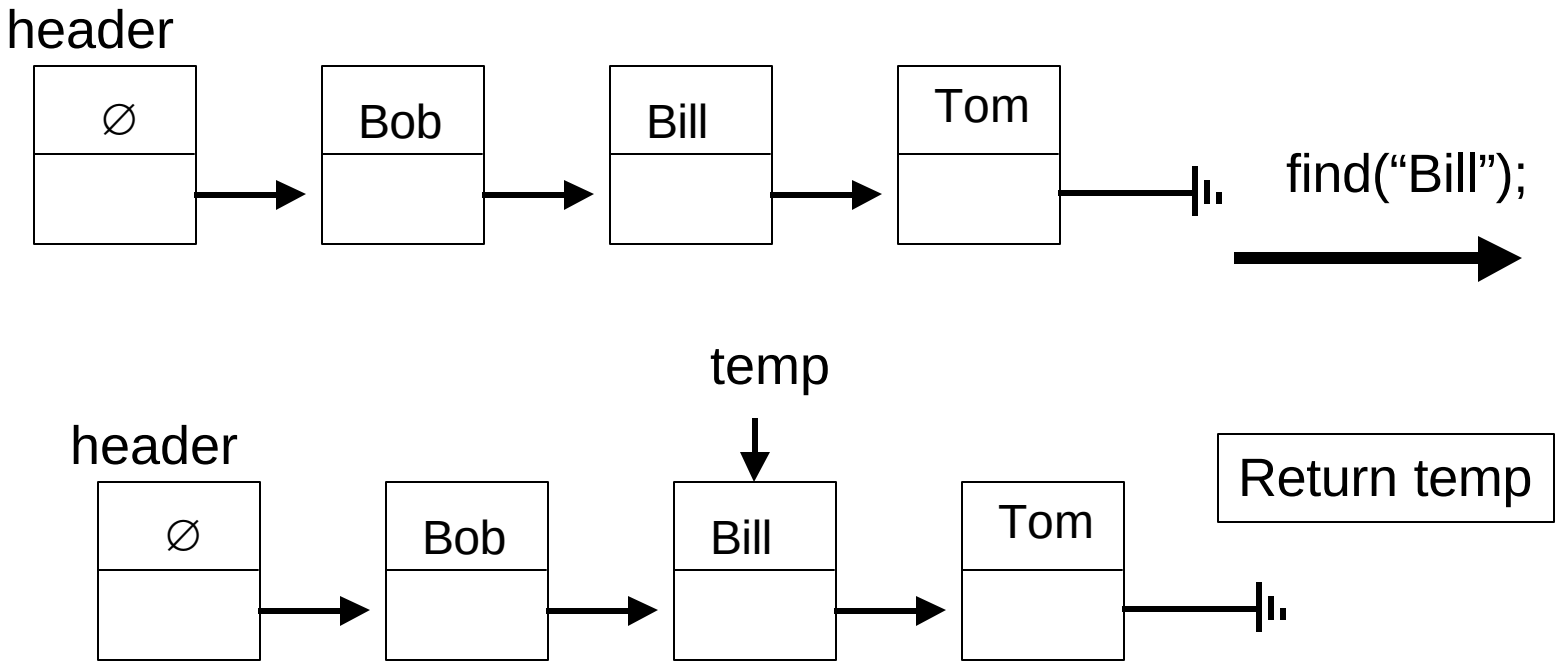


Erase operation

```
bool linkedlist::erase(const Object& obj){
    Node *temp1 = NULL, *temp2 = NULL;
    for (  temp1 = header->next, temp2 = header;
           temp1 != NULL;
           temp1 = temp1->next, temp2 = temp2->next){
           if (temp1->data == obj){
                  // remove from the list
                  temp2->next = temp1->next;
                  temp1->next = NULL;
                  delete temp1;
                  --list_size;
                  return true;
           }
    }
    return false;
}
```

Find operation

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



Find operation

```
Node* linkedlist::find(const Object& obj) const{
    Node *temp=NULL;

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

First operation

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

```
Node* linkedlist::first() const{
    return header->next;
}
```

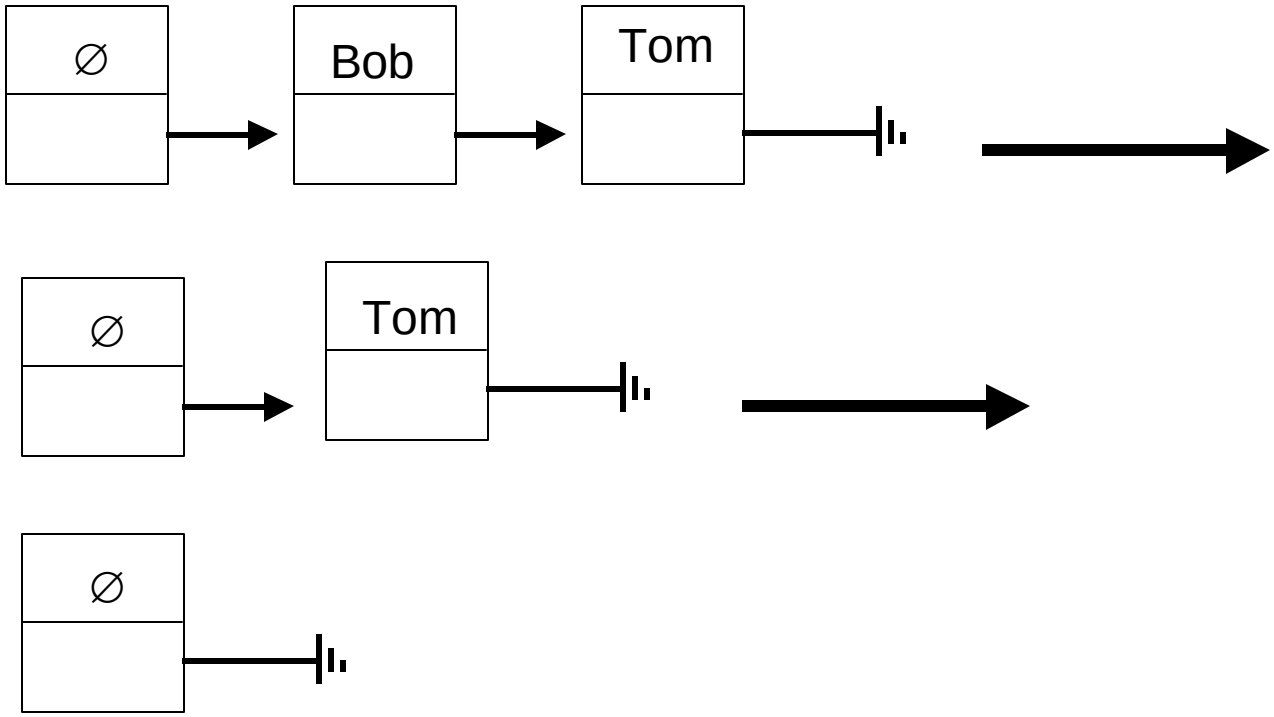
Is Empty operation

- Determines if the list is empty
  - If header->next is null

```
bool linkedlist::is_empty() const {
    return header->next == null;
}
```

Make Null operation

- Erase all elements from the list until is becomes an empty list.



Make Null operation

```
void linkedlist::make_empty() {
    while (!is_empty()){
        erase(header->next);
    }
}
```

Size operation

- Returns the size of the list

```
int linkedlist::size() const {
    return list_size;
}
```