ICOM 4035 – Data Structures

Dr. Manuel Rodríguez Martínez
Electrical and Computer Engineering Department

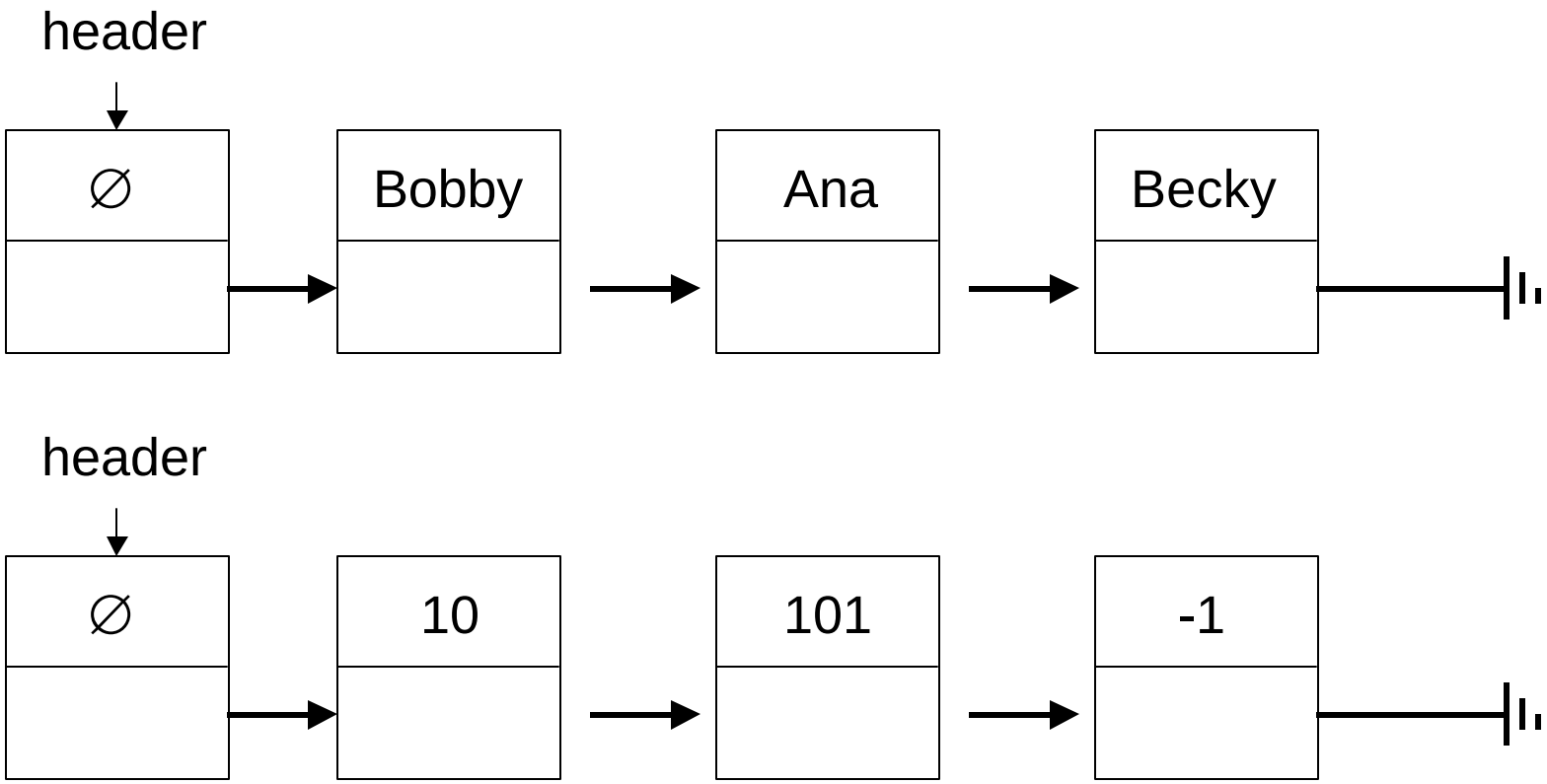
Readings

- Chapter 17 of textbook: Linked Lists

Limitations in first Linked List version

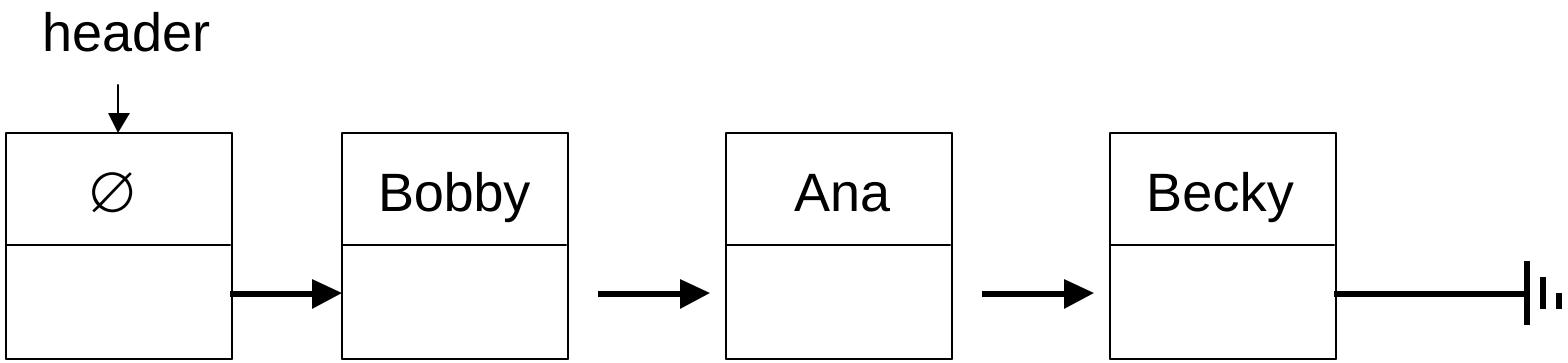
- The first linked list version had two main drawback
  - Need to re-define the Object data type for each different type of data object that we want to store in the list.
    - Linked list behavior does not change, whether we store integers of strings.
    - Could we have a generic linked list that is used for all types?
  - Returning pointers to positions in the linked list is a bad idea.
    - People can delete these pointers, breaking the linked list
    - Can we return some object that provides access to the nodes in the list, but without giving the pointer.

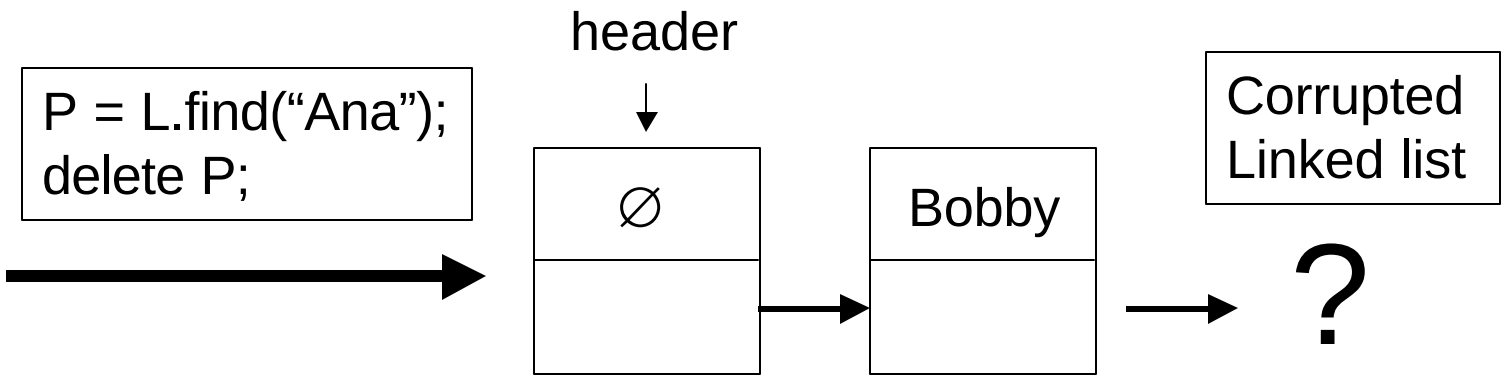
Limitation 1: fixed Object type



No difference between string list and integer list

Limitation 2: Returning pointers





By deleting the P pointer, half the list is lost

Solutions

- Implement the linked list as generic container class using templates.
  - Parameterized the type for the element to be used in the linked list.
  - Write the linked list code once, and keep re-using (“recycling”) it.
- To return positions in the list, don’t return pointers to the node, but return objects holding those pointers as private members. This is called an iterator.
  - It allows you to move over the nodes in the list, one after the other.
    - Iterate over the nodes in the list.

C++ Templates

- A template provides a pattern for implementing an algorithm whose basic behavior is independent of the data types of its input.
- Example: Swap operation.

Integer swap:

```
void swap(int& a, int&b){
    int temp = a;
    a = b;
    b= temp
}
```

Character swap:

```
void swap(char& a, char&b){
    char temp = a;
    a = b;
    b= temp
}
```

A Swap pattern

- As we can see swap is the same regardless of the types being swapped.
- Template provide mechanism to create generic swap function.
  - A swap pattern
  - Specific type use is a parameter of the template.
  - Swap template function:

```
template <typename Type>
void swap(Type& a, Type& b){
    Type temp = a;
    a = b;
    b = temp
}
```

A Swap pattern (Cont)

- In the template declaration, the Type parameter specifies the data type to be used in the template.
- We can now call swap in many different ways, without re-writing code, or without defining the Object type.

```
char c1 = ‘a’, c2 = ‘b’;
int num1 = -1, num2 = 20;
swap(c1, c2);
swap(num1, num2);
```

A note about templates

- Templates are not actual function instances, but rather a “recipe” for making functions based on the parameterized types.
- The compiler will bind specific data types as parameters to the templates during compilation.
- Then it will generate actual code, using the pattern provide by the template to create a function instance for that type.
- Compile time errors with templates are complex to understand, because they are given based on the code generated by the compiler.

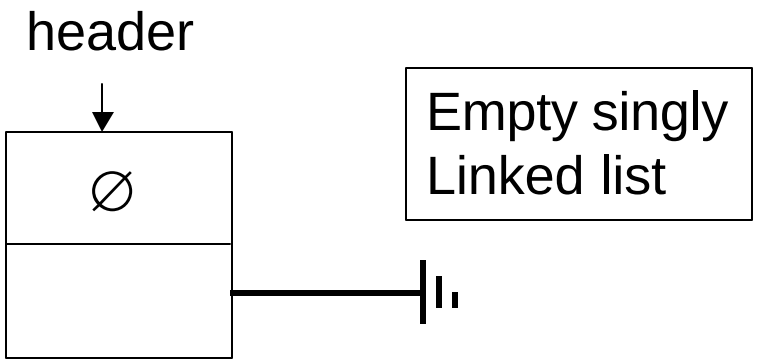
Linked list and templates

- We can implement a linked list as container class that uses templates.
- The type of data element stored in a node of the list is the parameter to the template.
- This will be a general purpose linked list that be used in any situation.

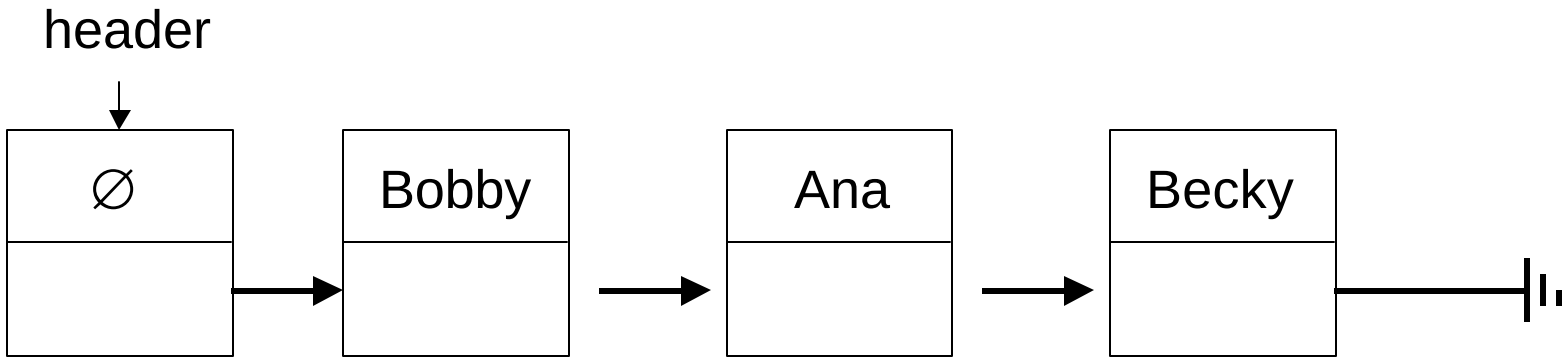
Singly linked list node

```
// define a node for the linked list
// node stores “generic” data type.
template <typename ListData>
struct Node {
    // data field
    ListData data;
    // pointer to the next node in the list
    Node *next;
};
```

Sample linked lists



Empty singly Linked list



Singly linked list with three elements

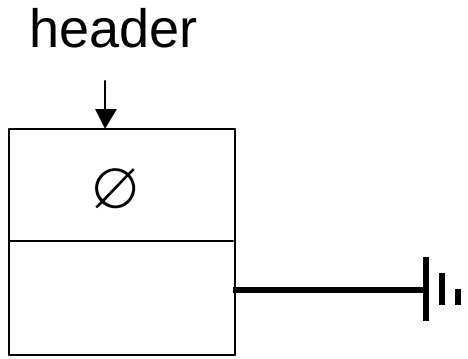
Linked list interface

- Same as before, but using templates and return iterators from find() and first() methods.

- Available from class web page.

Constructor

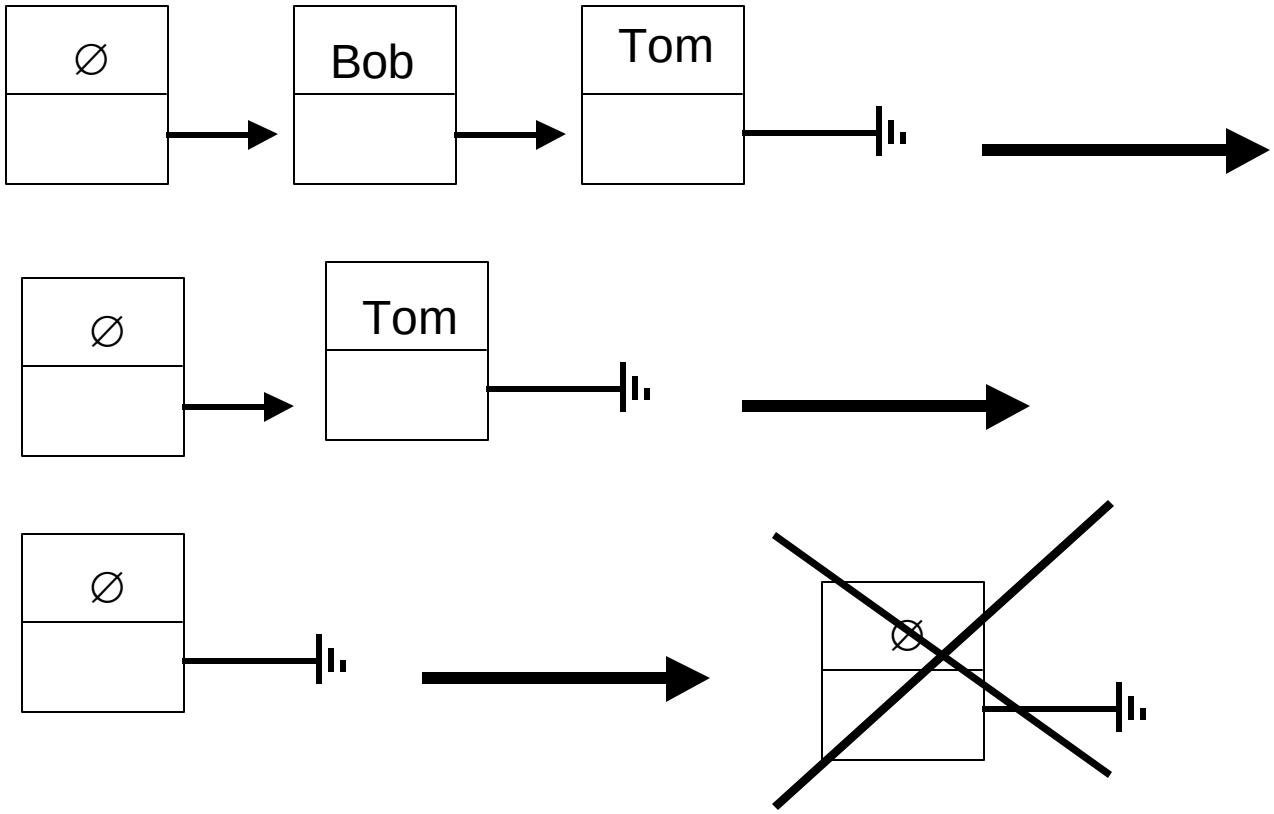
- Makes an empty list.




```
template <typename ListData>
Linkedlist<ListData>::linkedlist(){
    header = new Node<ListData>; // node of ListData
    header->next = NULL;
    list_size = 0;
}
```

Destructor

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

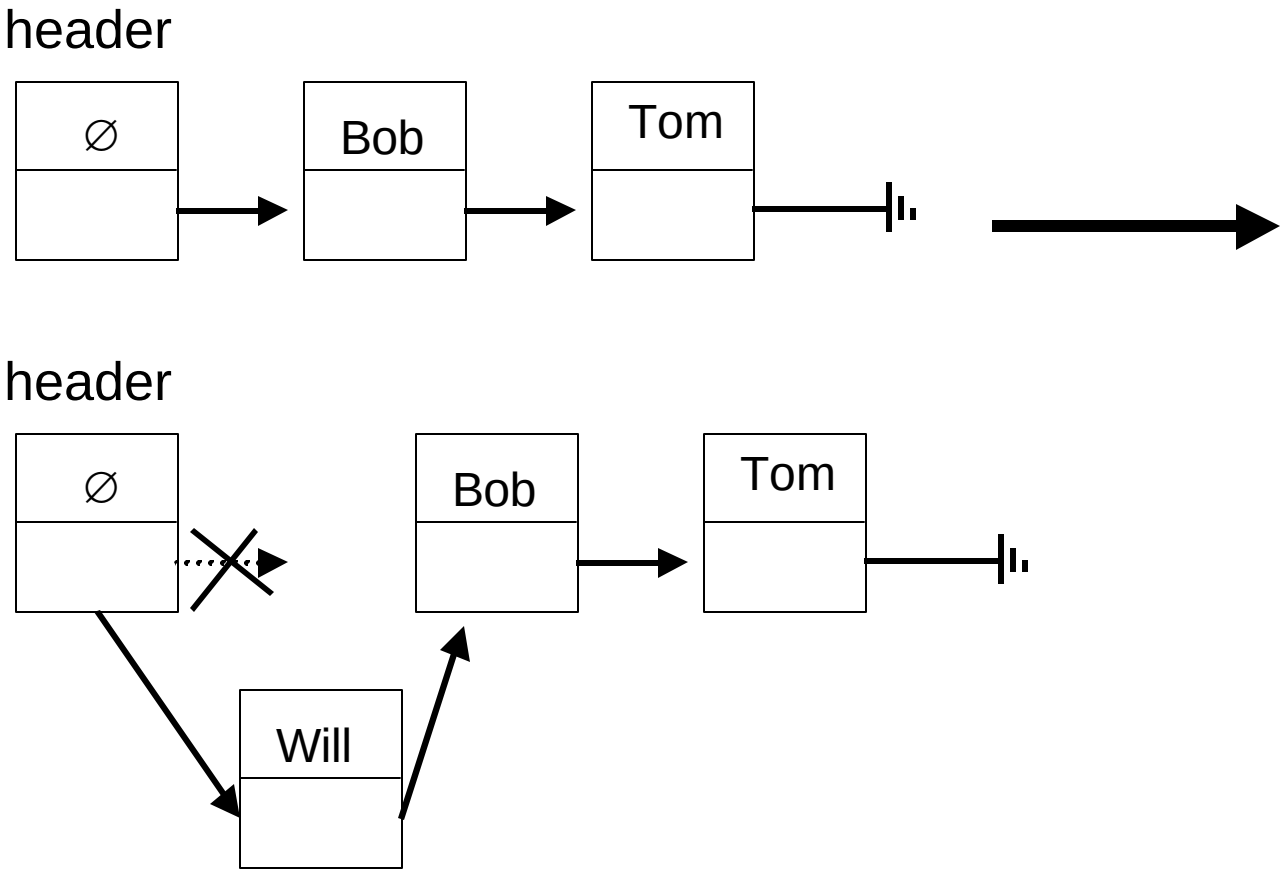


Destructor

```
template <typename ListData>
Linkedlist<ListData>::~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

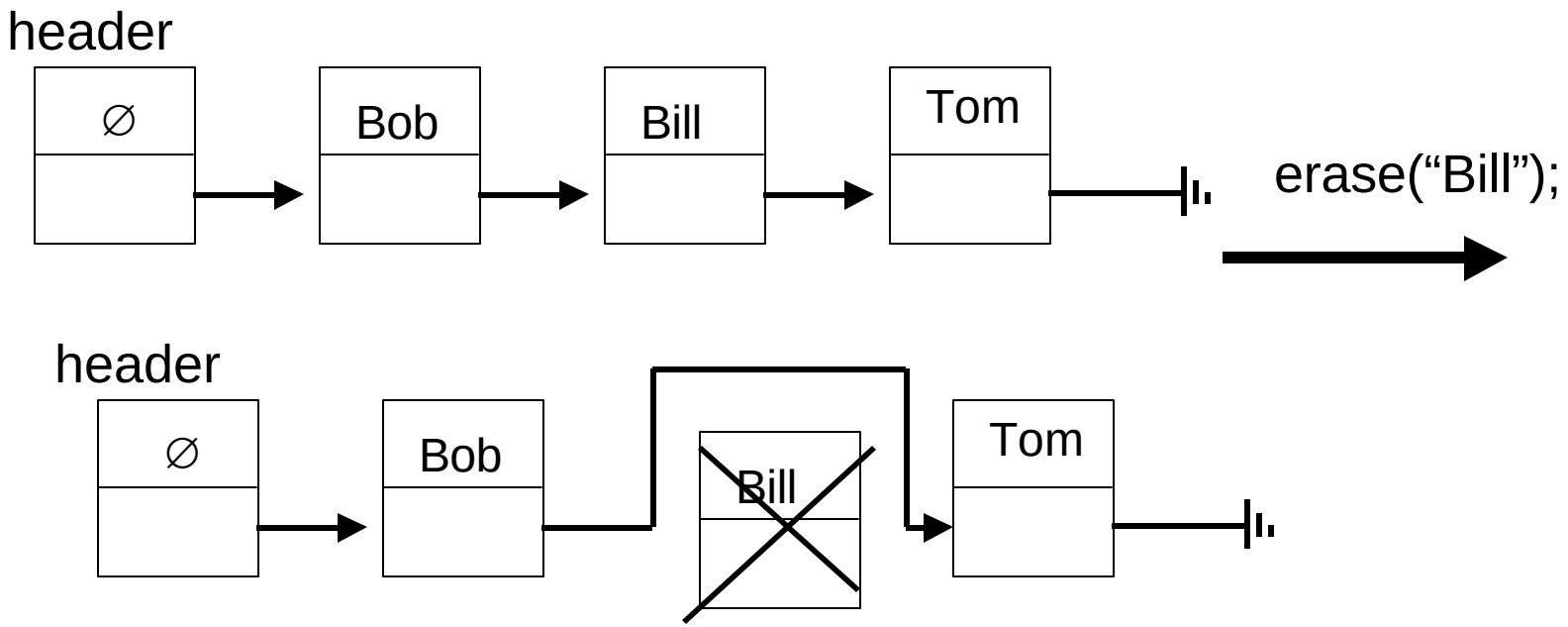


Insert operation

```
template <typename ListData>
void linkedlist::insert(const ListData& obj){
    temp = new Node<ListData>; // 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
}
```

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

```
template <typename ListData>
bool linkedlist<ListData>::erase(const ListData& obj){
    Node<ListData> *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; // if we get here, the element obj was not in the list.
}
```

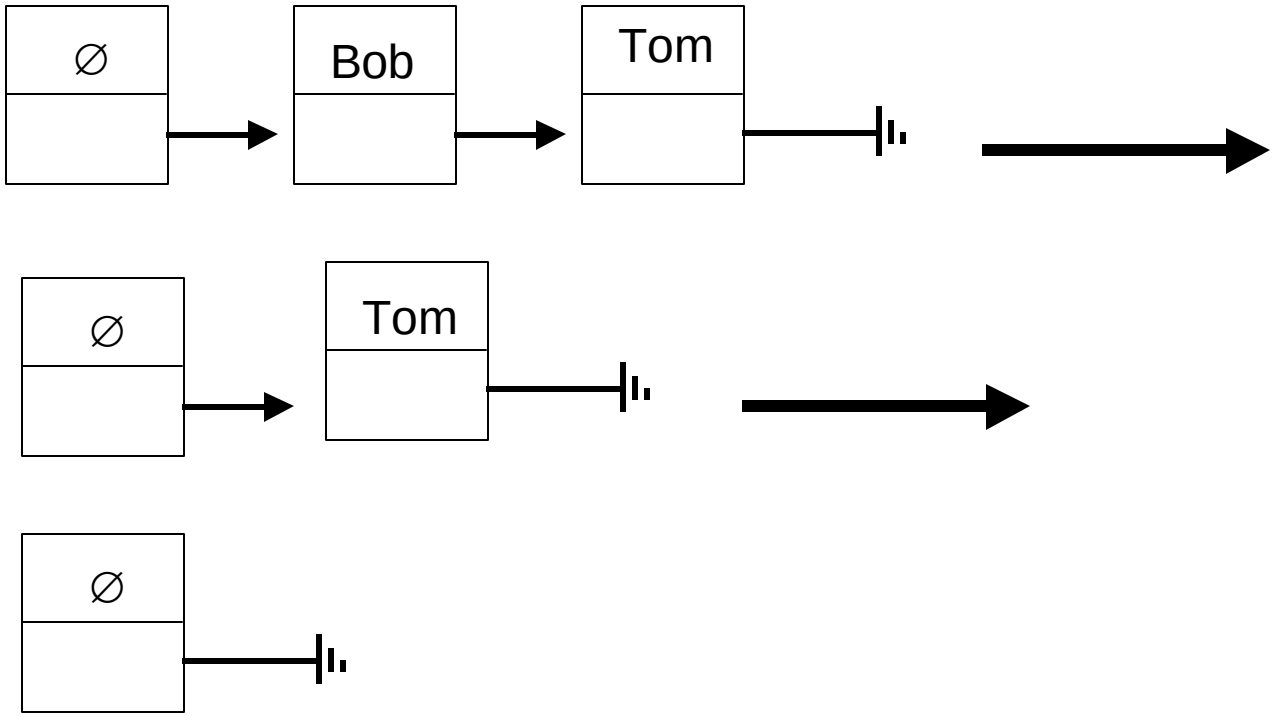
Is Empty operation

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

```
template <typename ListData>
bool linkedlist<ListData>::is_empty() const {
    // test is header->next is the null pointer.
    return header->next == null;
}
```

Make Null operation

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



Make Null operation

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

Size operation

- Returns the size of the list

```
template <typename ListData>
int linkedlist<ListData>::size() const {
    return list_size;
}
```