SINGLY LINKED LIST: IMPLEMENTATION

How would you implement these methods?

```
// returns the element at the front
const string& StringLinkedList::front() const {
    return head->elem;
}

// returns "true" if the list is empty
bool StringLinkedList::empty() const {
    return head == NULL;
}

// destructor; clears the memory
StringLinkedList::~StringLinkedList() {

    while (!empty()) removeFront();

}
```

SINGLY LINKED LIST: IMPLEMENTATION

A singly linked list of integers:

```
class IntNode {
    private:
       int elem;
       IntNode* next;
       friend class IntLinkedList;
    };

class IntLinkedList {
    public:
       IntLinkedList(): head(NULL) {};
       ~IntLinkedList();
       bool empty() const;
       const int& front() const;
       void addFront(const int& e);
       void removeFront();
    private:
       IntNode* head;
    };
```

A singly linked list of double:

```
class DoubleNode {
    private:
       double elem;
       DoubleNode* next;
       friend class DoubleLinkedList;
    };

class DoubleLinkedList {
    public:
       DoubleLinkedList(): head(NULL) {};
       ~DoubleLinkedList();
       bool empty() const;
       const double& front() const;
       void addFront(const double& e);
       void removeFront();
    private:
       DoubleNode* head;
    };
```

How can you avoid writing a new version for every type? **Use a template!**

SINGLY LINKED LIST: TEMPLATE

How would you create a template version of these classes?

```
class IntNode {
    private:
       int elem;
       IntNode* next;
       friend class IntLinkedList;
    };


class IntLinkedList {
    public:
       IntLinkedList(): head(NULL) {};
       ~IntLinkedList();
       bool empty() const;
       const int& front() const;
       void addFront(const int& e);
       void removeFront();
    private:
       IntNode* head;
    };
```

```
template <typename E>
class Node {
    private:
       E elem;
       Node<E>* next;
       friend class LinkedList<E>;
    };

template <typename E>
class LinkedList {
    public:
       LinkedList(): head(NULL) {};
       ~LinkedList();
       bool empty() const;
       const E& front() const;
       void addFront(const E& e);
       void removeFront();
    private:
       Node<E>* head;
    };
```

SINGLY LINKED LIST: TEMPLATE

How would you create a template version of these classes?

```
const int& IntLinkedList::front() const{ return head->elem; }
```

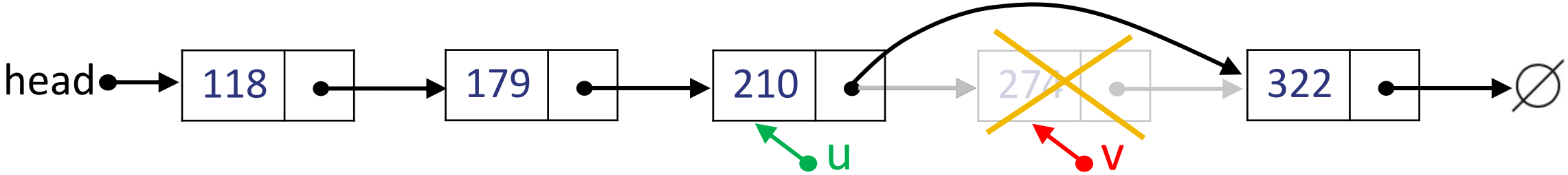
```
template <typename E>
const E& LinkedList::front() const{ return head->elem; }
```

```
void IntLinkedList::addFront(const int& e){
    IntNode* v = new IntNode;
    v->elem = e;
    v->next = head;
    head = v;  }
```

```
template <typename E>
void LinkedList::addFront(const E& e){
    Node<E>* v = new Node<E>;
    v->elem = e;
    v->next = head;
    head = v;   }
```

SINGLY LINKED LIST: DELETING A NODE

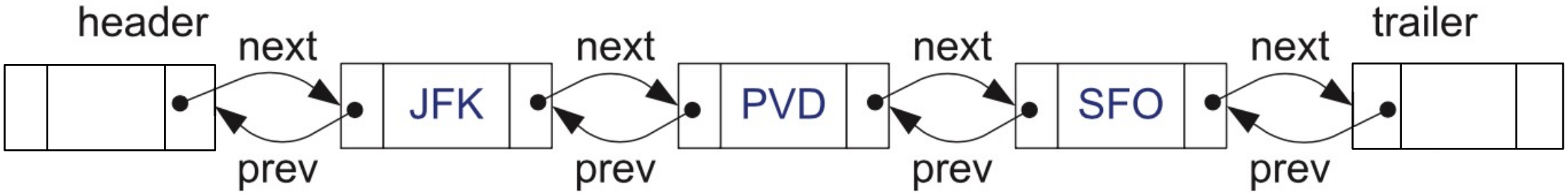
- We've seen how to delete the front of the list, but how do we delete a node, v, that isn't at the front?



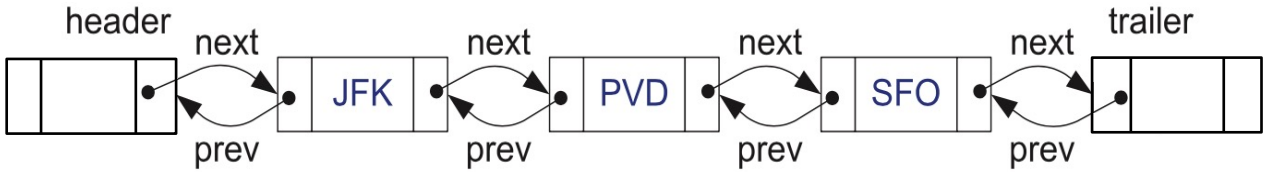
- As can be seen, to delete a node, v, we need to update the `next` pointer of the node, u, that comes before v
- Locating u requires traversing the entire list, starting from the head and going forward until we find the node whose `next` points to v.
- ***What is the problem with that?*** This is inefficient, especially if the list is long!
- This can be avoided using another linked list called a "**doubly-linked list**"!

DOUBLY LINKED LIST

- A node in a doubly linked list stores two pointers:
  - *next:* points to the next node in the list
  - *prev:* points to the previous node in the list
- It has two "dummy" nodes that **store no elements**:
  - *header*: comes before the first element
  - *trailer*: comes after the last element



IMPLEMENTATION



Implementation of a **node**:

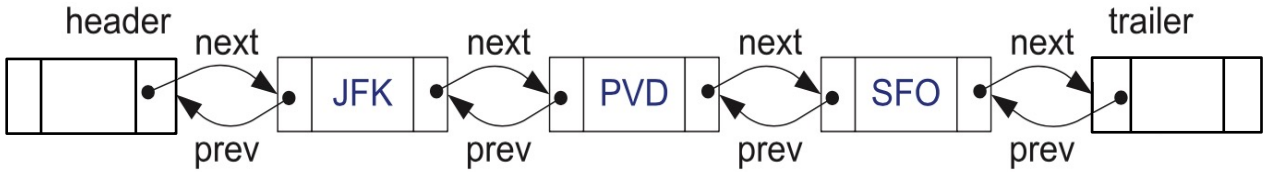
- To simplify the code, instead of defining templates, from now on we will often use the keyword `typedef`, which allows us to choose a name for a type.
- Here, the type `string` is called `Elem`.

```
typedef string Elem; // define the type Elem to be string (can be any data type)
class DNode { // An object of this class is one node
private:
        Elem elem;    // element of the type defined above
        DNode* prev;  // previous node in the list
        DNode* next;  // next node in the list
        friend class DLinkedList; /* DlinkedList is the class of the list
};
```

- If you want to implement a template, use the following before the class definition:

```
template <typename Elem>
```

IMPLEMENTATION

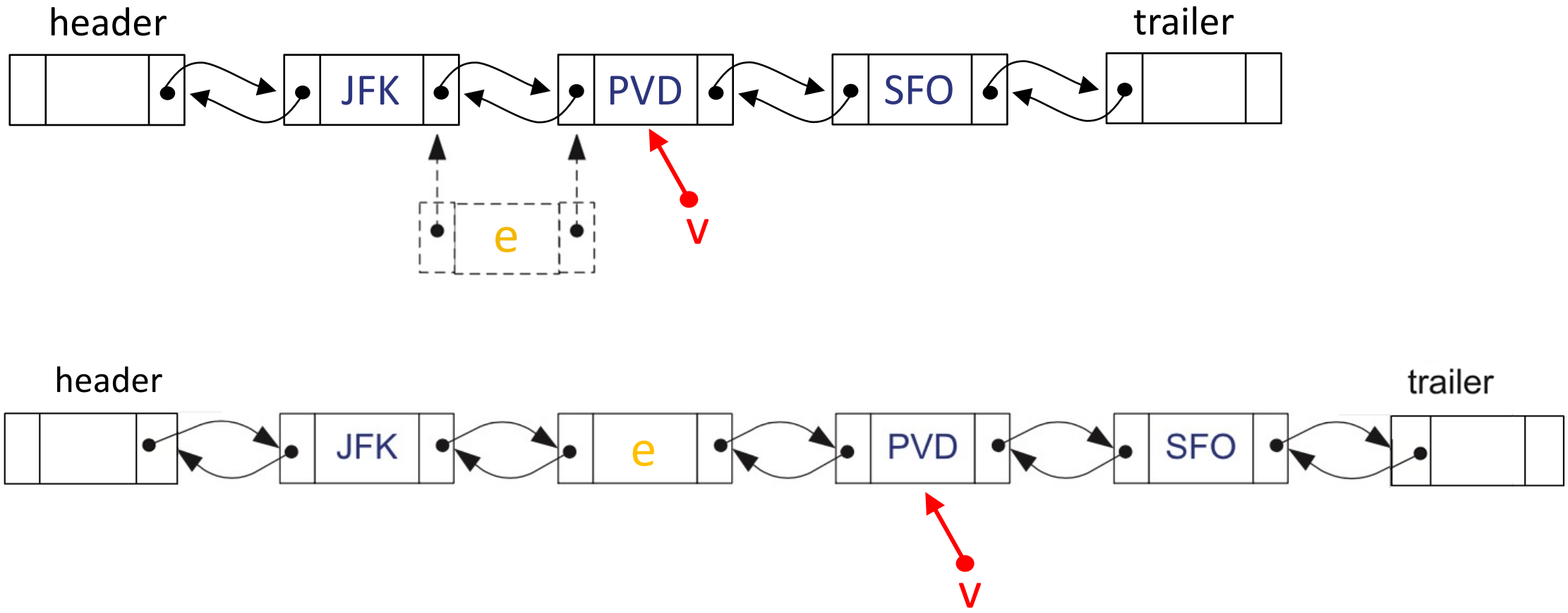


Implementation of a **class**:

```
class DLinkedList { // An object of this class is doubly linked list
public:
        DLinkedList();  // constructor; initializes an empty list
        ~DLinkedList(); // destructor; clears the memory in which the list is stored
        bool empty() const;  // returns "true" if the list is empty
        const Elem& front() const; // returns the element at the front of the list
        const Elem& back() const; // returns the element at the end of the list
        void addFront(const Elem& e); // puts "e" in a node, and adds it to the front
        void addBack(const Elem& e); // puts "e" in a node, and adds it to the end
        void removeFront(); // deletes the node at the front of the list
        void removeBack(); // deletes the node at the end of the list
private:
        DNode* header; // a pointer to the node containing the first element
        DNode* trailer; // a pointer to the node containing the last element
protected:
        void add(DNode* v, const Elem& e); //put e in a node & insert it before v
        void remove(DNode* v); // delete the node that v points to
};
```

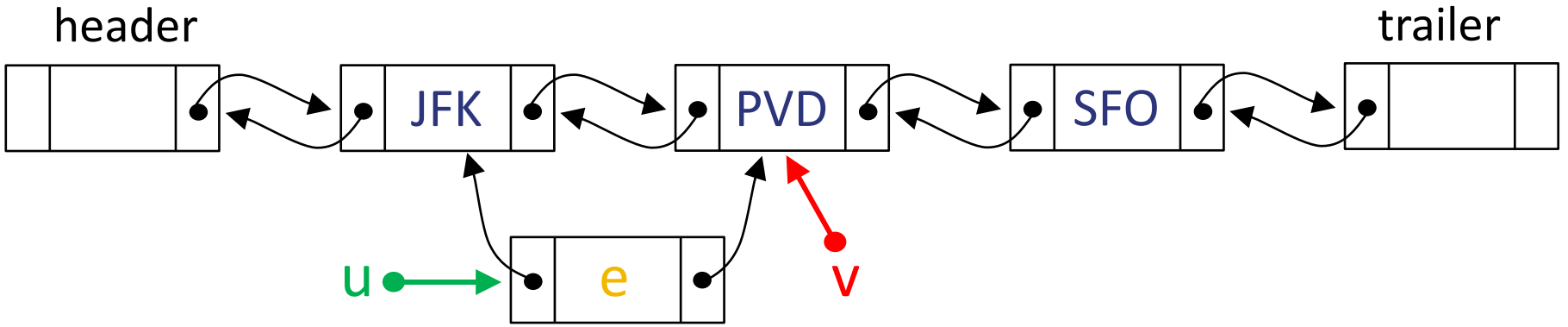
DOUBLY LINKED LIST: INSERTION

Example of how to insert element e just before a node v



DOUBLY LINKED LIST: INSERTION

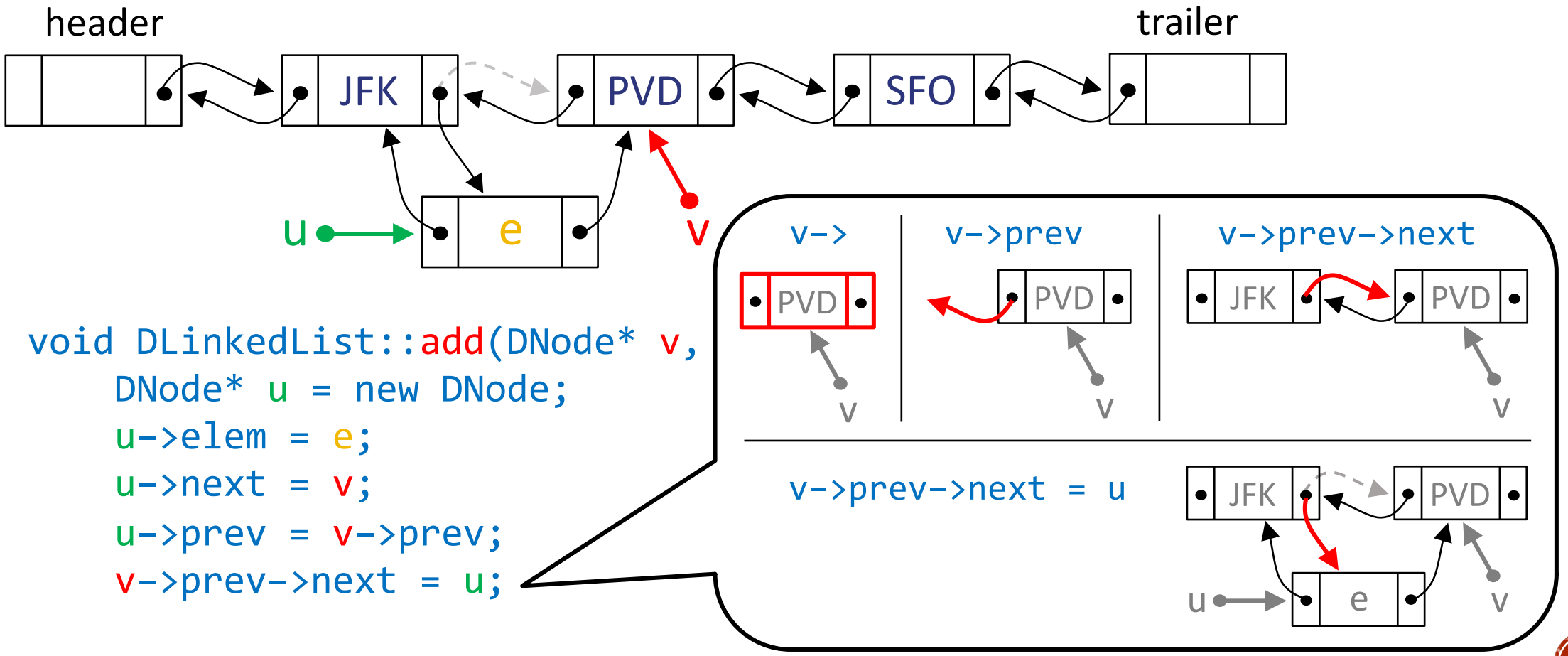
How would you implement the insertion of element e just before v?




```
void DLinkedList::add(DNode* v, const Elem& e) {
    DNode* u = new DNode;
    u->elem = e;
    u->next = v;
    u->prev = v->prev;


}
```

DOUBLY LINKED LIST: INSERTION

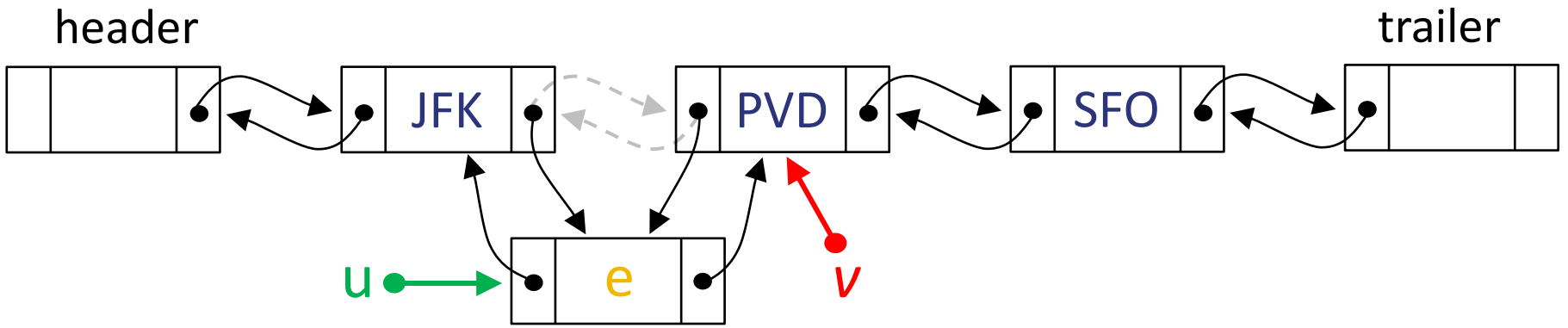
How would you implement the insertion of element e just before v?



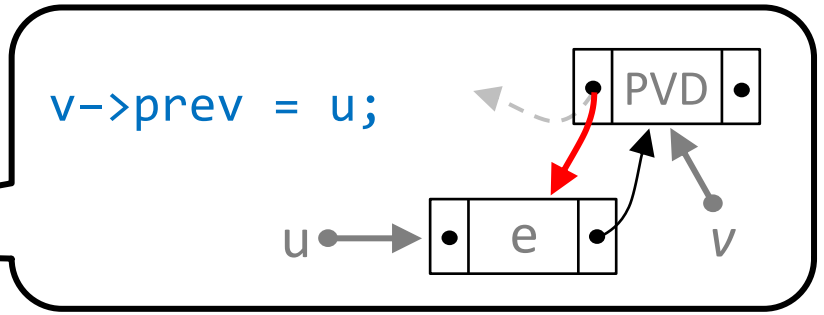

```
void DLinkedList::add(DNode* v,
    DNode* u = new DNode;
    u->elem = e;
    u->next = v;
    u->prev = v->prev;
    v->prev->next = u;

}
```

DOUBLY LINKED LIST: INSERTION

How would you implement the insertion of element e just before v?

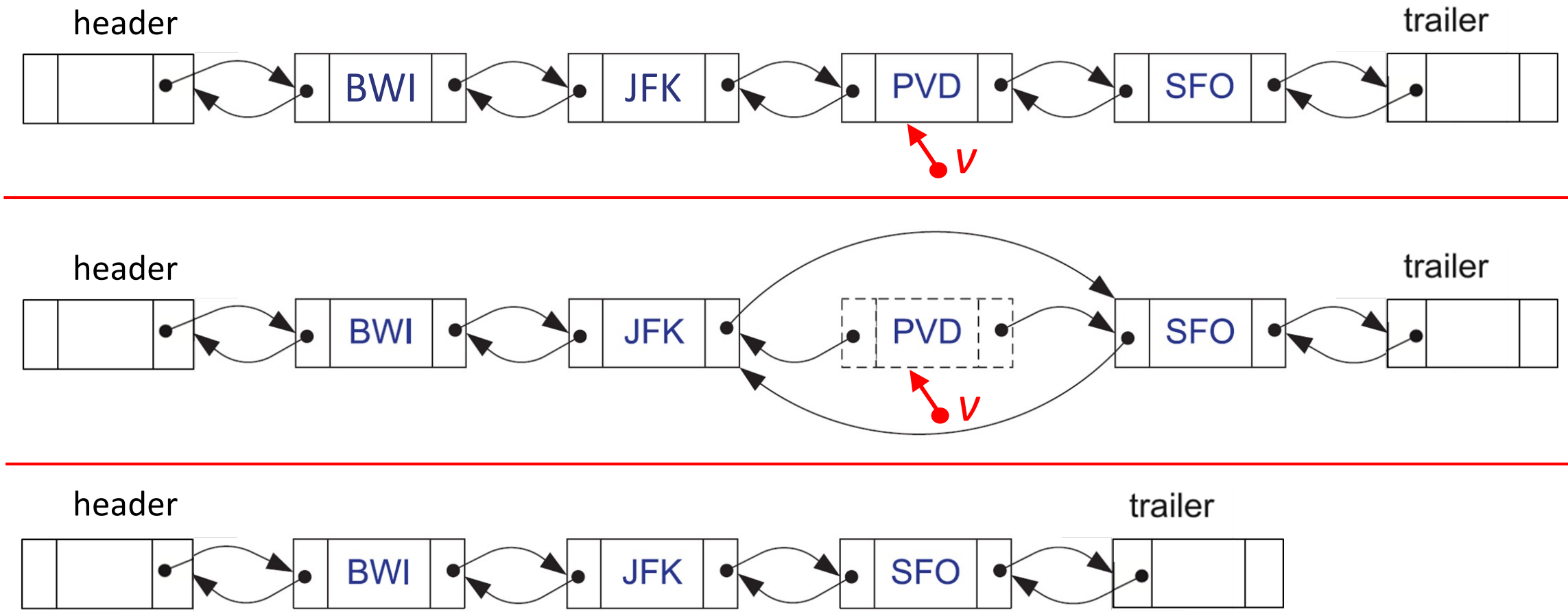



```
void DLinkedList::add(DNode* v, const Elem& e) {
    DNode* u = new DNode;
    u->elem = e;
    u->next = v;
    u->prev = v->prev;
    v->prev->next = u;
    v->prev = u;
}
```



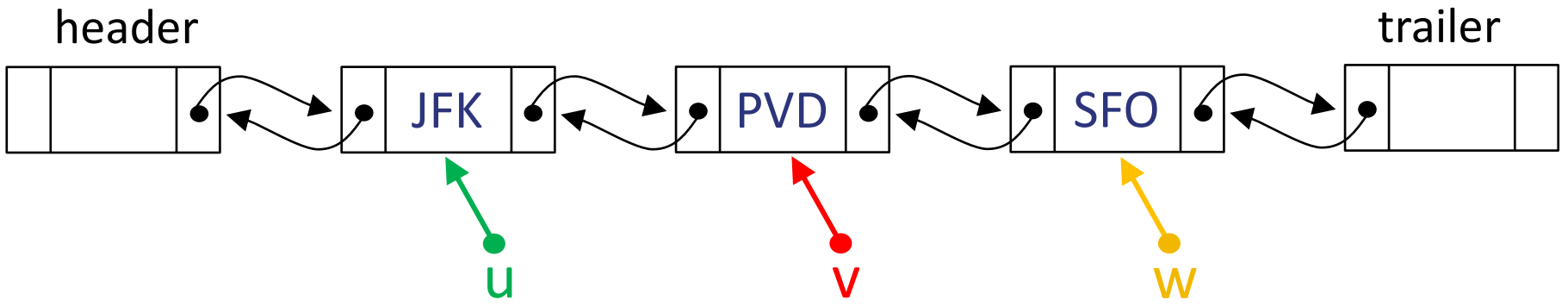
DOUBLY LINKED LIST: DELETION

Example of how to delete a node, *v*:



DOUBLY LINKED LIST: DELETION

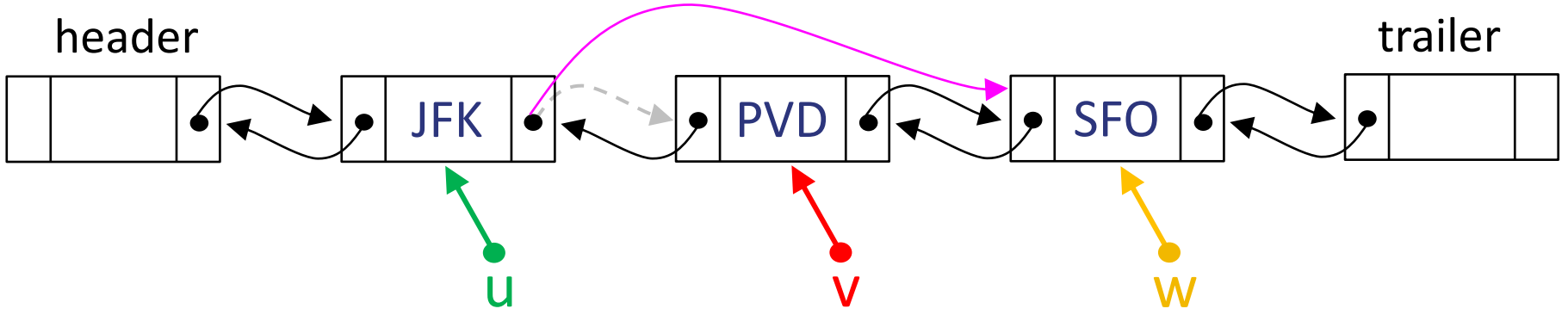
How would you implement the removal of node v?




```
void DLinkedList::remove(DNode* v) {
    DNode* u = v->prev;
    DNode* w = v->next;



}
```

DOUBLY LINKED LIST: DELETION

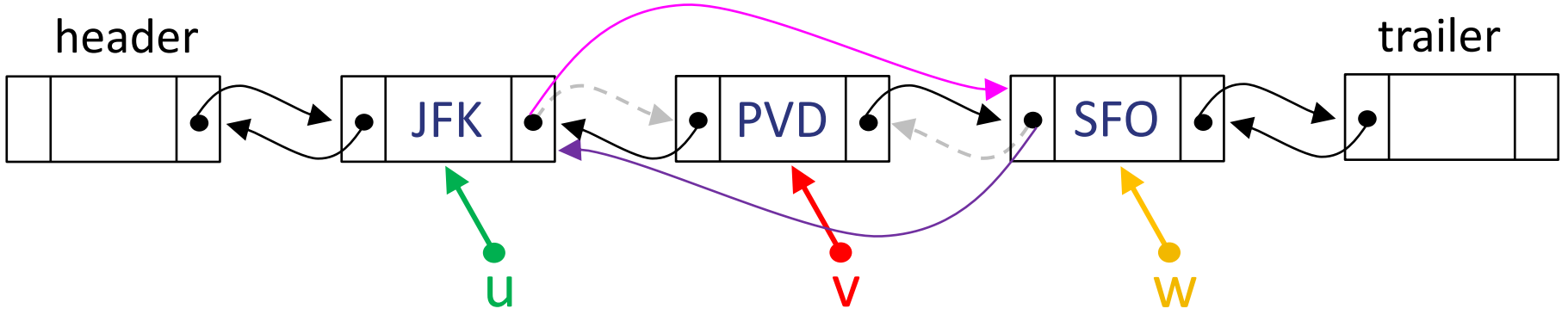
How would you implement the removal of node v?




```
void DLinkedList::remove(DNode* v) {
    DNode* u = v->prev;
    DNode* w = v->next;
    u->next = w;


}
```

DOUBLY LINKED LIST: DELETION

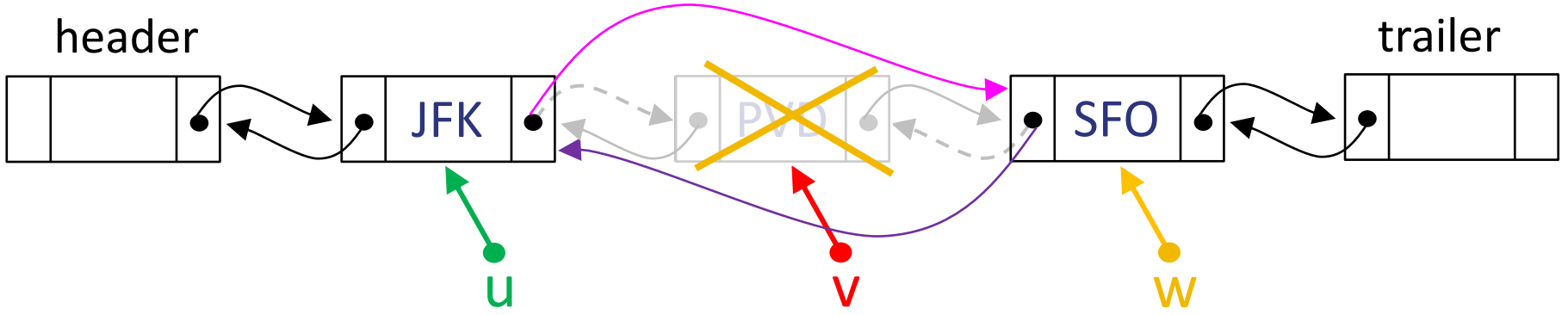
How would you implement the removal of node v?




```
void DLinkedList::remove(DNode* v) {
    DNode* u = v->prev;
    DNode* w = v->next;
    u->next = w;
    w->prev = u;

}
```

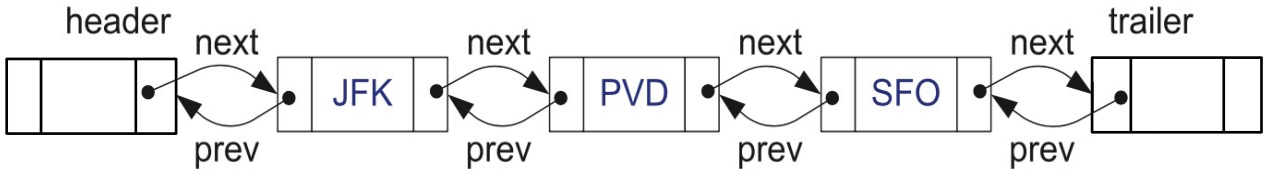
DOUBLY LINKED LIST: DELETION

How would you implement the removal of node v?



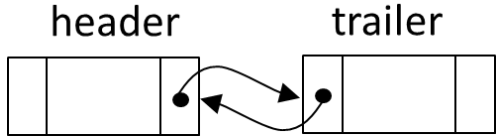

```
void DLinkedList::remove(DNode* v) {
    DNode* u = v->prev;
    DNode* w = v->next;
    u->next = w;
    w->prev = u;
    delete v;
}
```

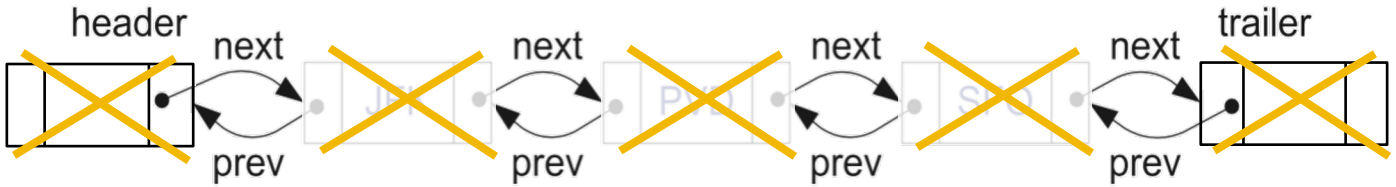
IMPLEMENTATION



How would you implement the constructor and the destructor?

```
DLinkedList::DLinkedList() { // constructor; creates an empty list
    header = new DNode;
    trailer = new DNode;
    header->next = trailer;
    trailer->prev = header;
}
DLinkedList::~DLinkedList() { // destructor; clears the memory
    while (!empty()) removeFront();
    delete header;
    delete trailer;
}
```

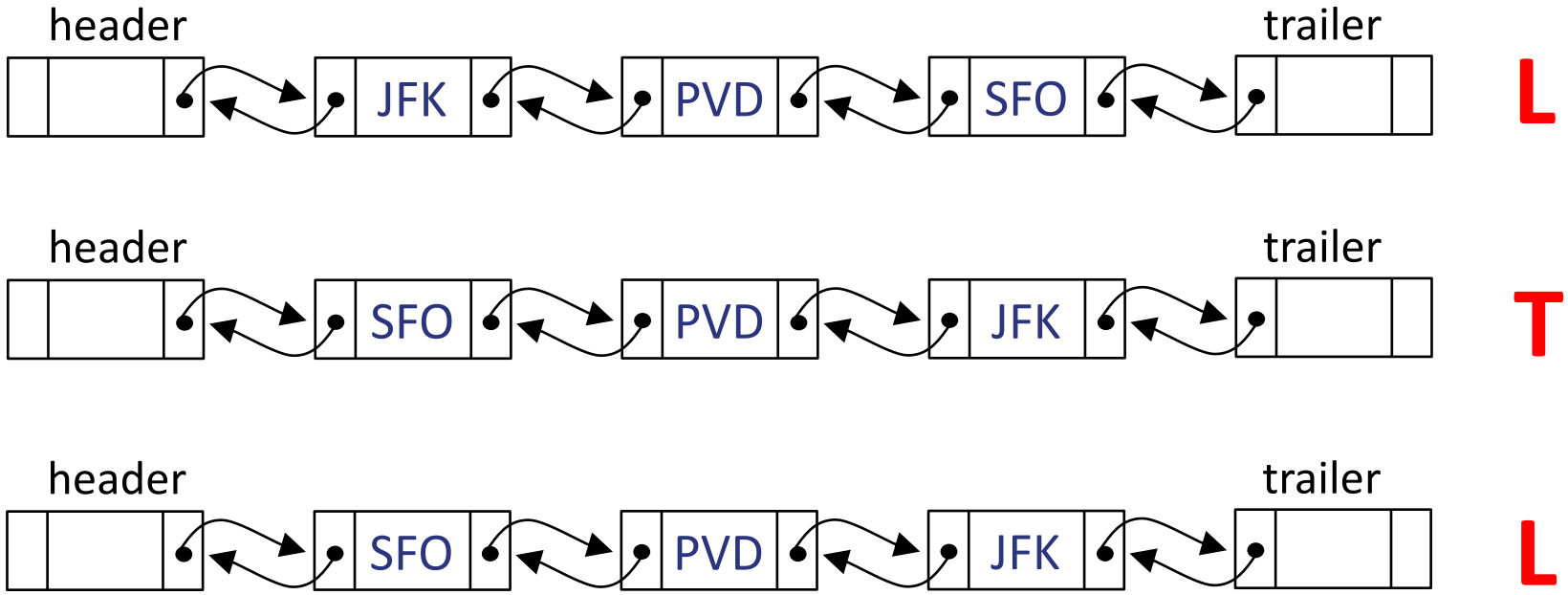




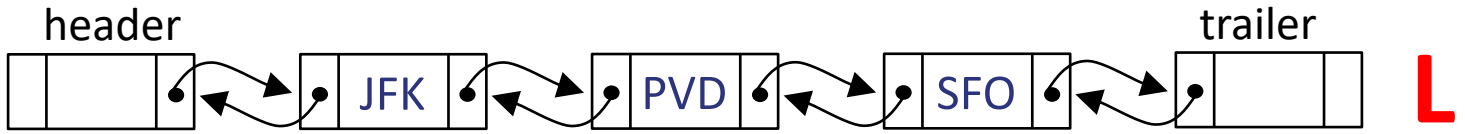
REVERSING A DOUBLY-LINKED LIST

One way to reverse the elements of a doubly linked list, **L**, is as follows:

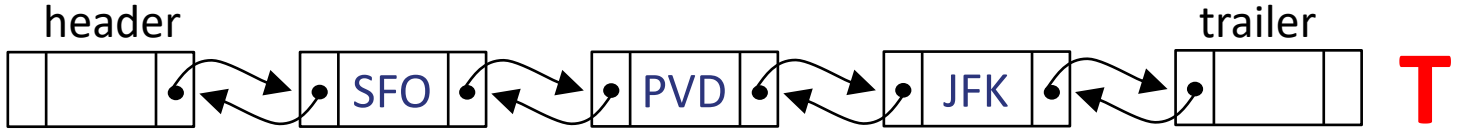
- Step 1: **Copy** the contents of **L in reverse order** into a temporary list, **T**
- Step 2: **Copy** the contents of **T** back into **L** (**but NOT in reverse order**)



REVERSING A DOUBLY-LINKED LIST

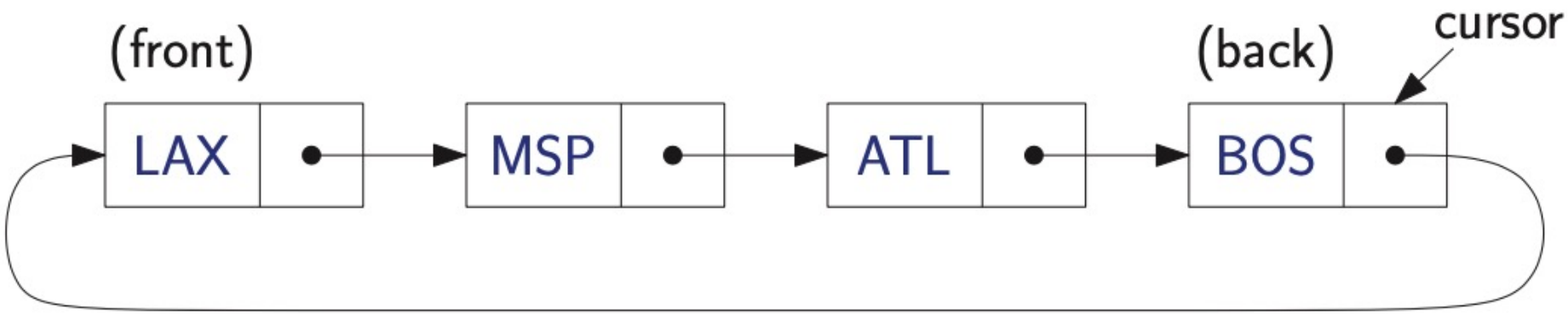



```
void listReverse(DLinkedList& L) { // reverses the list

    DLinkedList T; // T is a temporary list

    while (!L.empty()) { // copy L to T, but in reverse order
        string s = L.front(); // s is the element in the node at the front
        T.addFront(s);
        L.removeFront();
    }

    while (!T.empty()) { // copy T back to L
        string s = T.front(); // s is the element in the node at the front
        T.removeFront();
        L.addBack(s);
    }
}
```



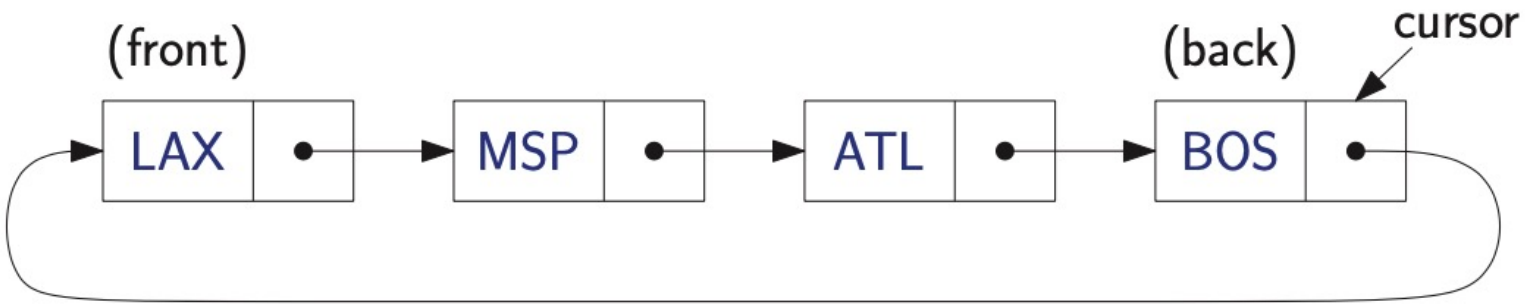
CIRCULARLY LINKED LIST

- Each node has one pointer, just like Singly Linked lists
- The nodes are linked into a ***cycle***
- There is a special pointer called **cursor**
  - **back** = The node that **curser** points to at a certain time.
  - **front** = The node right after the one that **curser** points to.
  - Although a circularly linked list does not have a beginning and an end, it helps to **imagine that cursor points to the end of the list**.



CIRCULARLY LINKED LIST: IMPLEMENTATION

```
typedef string Elem; // define the type Elem to be string (can be any data type)

class CNode { // An object of this class is one node
private:
    Elem elem;   // the element stored in the node
    CNode* next; // the next node in the list
    friend class CircleList; // Now the list can access the private members of the node
};
class CircleList { // An object of this class is a circularly linked list
public:
    CircleList();  // constructor; creates an empty list
    ~CircleList(); // destructor; clears the memory
```

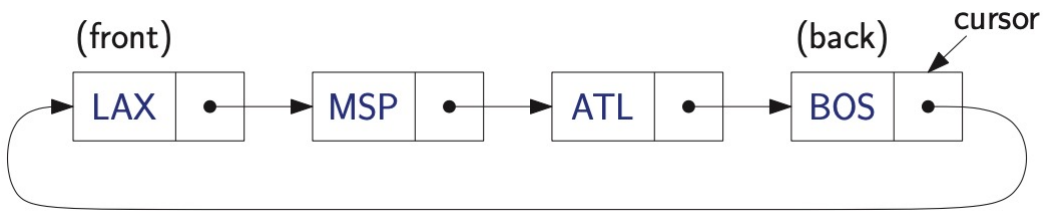



```
private:
    CNode* cursor;
}
```

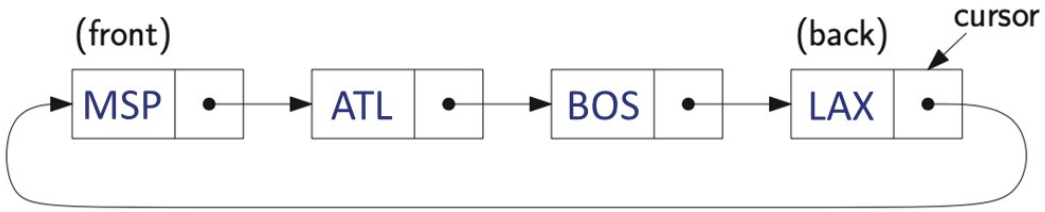
CIRCULARLY LINKED LIST: IMPLEMENTATION

```
typedef string Elem; // define the type Elem to be string (can be any data type)

class CNode { // An object of this class is one node
private:
    Elem elem;   // the element stored in the node
    CNode* next; // the next node in the list
    friend class CircleList; // Now the list can access the private members of the node
};
class CircleList { // An object of this class is a circularly linked list
public:
    CircleList();  // constructor; creates an empty list
    ~CircleList(); // destructor; clears the memory
    void advance(); // moves cursor one step forward in the list
```

before calling "advance()":



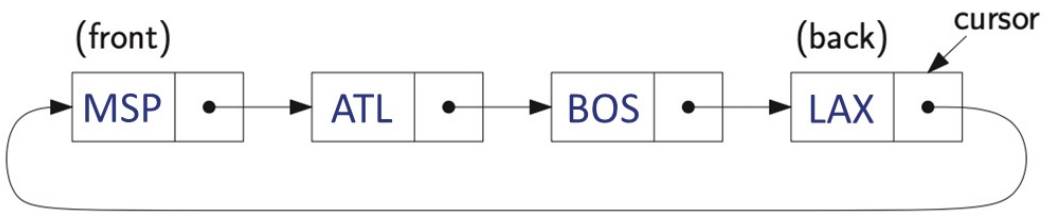
after calling "advance()":




```
private:
    CNode* cursor;
}
```

CIRCULARLY LINKED LIST: IMPLEMENTATION

```
typedef string Elem; // define the type Elem to be string (can be any data type)

class CNode { // An object of this class is one node
private:
    Elem elem;   // the element stored in the node
    CNode* next; // the next node in the list
    friend class CircleList; // Now the list can access the private members of the node
};
class CircleList { // An object of this class is a circularly linked list
public:
    CircleList();  // constructor; creates an empty list
    ~CircleList(); // destructor; clears the memory
    void advance(); // moves cursor one step forward in the list
    const Elem& front() const; /* returns the element in the node that is right after
                                  the one that curser points to */
    const Elem& back() const; // returns the element in the node that curser points to
    bool empty() const; // returns "true" if the list is empty
```




```
private:
    CNode* cursor;
}
```

CIRCULARLY LINKED LIST: IMPLEMENTATION

```
typedef string Elem; // define the type Elem to be string (can be any data type)

class CNode { // An object of this class is one node
private:
    Elem elem;   // the element stored in the node
    CNode* next; // the next node in the list
    friend class CircleList; // Now the list can access the private members of the node
};
class CircleList { // An object of this class is a circularly linked list
public:
    CircleList();  // constructor; creates an empty list
    ~CircleList(); // destructor; clears the memory
    void advance(); // moves cursor one step forward in the list
    const Elem& front() const; /* returns the element in the node that is right after
                                  the one that curser points to */
    const Elem& back() const; // returns the element in the node that curser points to
    bool empty() const; // returns "true" if the list is empty
    void remove();  // deletes the node at the front of the list, i.e., the node right
                    after the one that curser points to. */
    void add(const Elem& e); // Puts e in a new node at the front of the list
private:
    CNode* cursor;
}
```

CIRCULARLY LINKED LIST: IMPLEMENTATION

Implementation of some methods (the rest are in the textbook):

```
// returns "true" if the list is empty
bool CircleList::empty() const
{ return cursor == NULL; }

// returns the element in the node that curser points to
const Elem& CircleList::back() const
{ return cursor->elem; }

// returns the element in the node right after the one that curser points to
const Elem& CircleList::front() const
{ return cursor->next->elem; }

// moves cursor one step forward in the list
void CircleList::advance()
{ cursor = cursor->next; }
```