

# CHAPTER 3: LINKED LISTS & RECURSION

1

2

# LINKED LISTS

# STORING HIGHEST SCORES

- Suppose you want to build a software that stores the **highest 10 scores** in a game
- One way to do so is using arrays in which every element is an object that has two data members:
  1. The player's name
  2. The player's score
- Suppose that, so far, 6 players played your game. The array may look like this:

Mike	Rob	Paul	Anna	Rose	Jack				
1105	750	720	660	590	510				

0    1    2    3    4    5    6    7    8    9

# STORING HIGHEST SCORES

- Now, suppose that a new player, **Jill**, plays the game and scores **740**.  
**How will you update the list?**

Mike	Rob	Paul	Anna	Rose	Jack				
1105	750	720	660	590	510				

0    1    2    3    4    5    6    7    8    9

- You can **shift** the players, one by one, to create an empty slot, and then add Jill's object in this slot:

The diagram illustrates the process of inserting a new player into an array. A blue box labeled "Jill" is shown above a blue box labeled "740". A dashed arrow points from this box down to the second slot of a horizontal array. The array consists of two rows of ten cells each, indexed from 0 to 9 below it. The first six slots contain the names and scores of existing players: Mike (1105), Rob (750), Paul (720), Anna (660), Rose (590), and Jack (510). The remaining four slots are empty. Dashed arrows point from the seventh slot of the top row to the eighth, the eighth to the ninth, and the ninth to the tenth, indicating the rightward shift of each player's position to accommodate the new entry.

Mike	Rob		Paul	Anna	Rose	Jack			
1105	750		720	660	590	510			

0    1    2    3    4    5    6    7    8    9

- The list would now look like this:

Mike	Rob	Jill	Paul	Anna	Rose	Jack			
1105	750	740	720	660	590	510			

0    1    2    3    4    5    6    7    8    9

# STORING HIGHEST SCORES

- Suppose we want to **remove** the object of, say, **Paul** from the list.  
**How will you update the list?**

Mike	Rob	Jill	Paul	Anna	Rose	Jack			
1105	750	740	720	660	590	510			

0    1    2    3    4    5    6    7    8    9

- You can **shift** the players one by one, just like we did earlier:

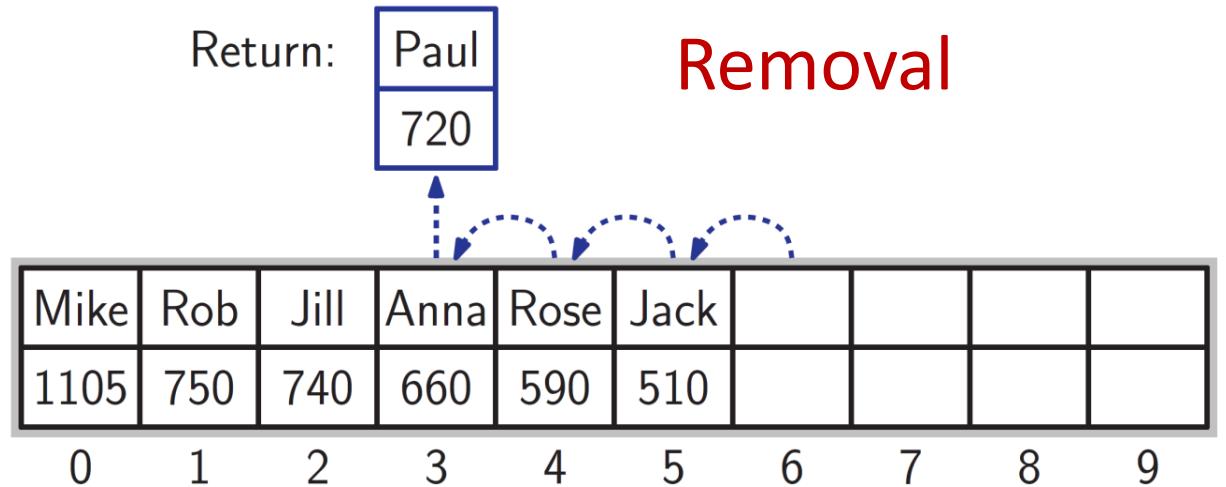
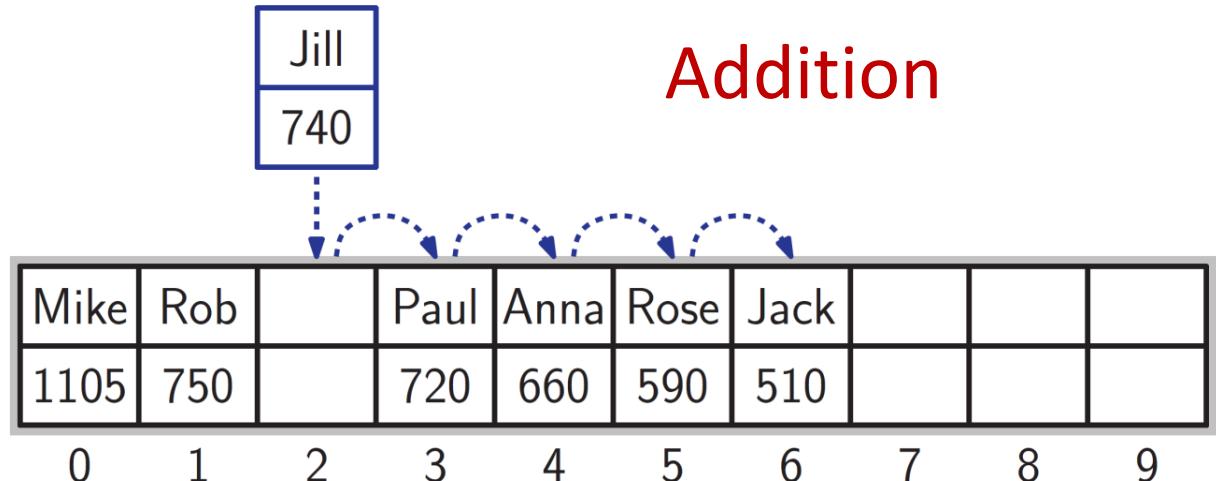
Return: **Paul**  
720

The diagram illustrates the process of shifting elements in an array to remove an element. A blue box labeled "Return: Paul" contains the value "720". A dashed arrow points from this box down to the cell containing "720" in the original array. From this cell, a dashed arrow points right, followed by a series of dashed arrows that curve around the remaining elements in the array, indicating they are being shifted left to fill the gap created by the removal of the element at index 3.

Mike	Rob	Jill	Anna	Rose	Jack				
1105	750	740	660	590	510				

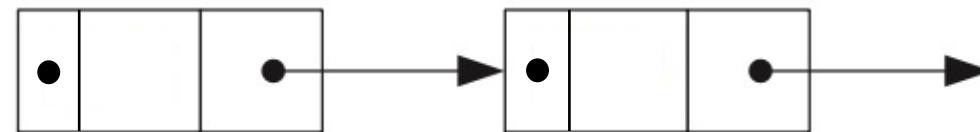
0    1    2    3    4    5    6    7    8    9

- Here is a **summary** of addition and removal.
- Do you think this is **computationally efficient?**
- **No!** because the shifting involves too many operations!
- **Example:** if the list contained 500,000 entries, then depending on the location of the element that is being added or removed, **you may end up shifting hundreds of thousands of elements**, just to make a tiny update to the list!
- **Solution:** Use a “Linked list”!



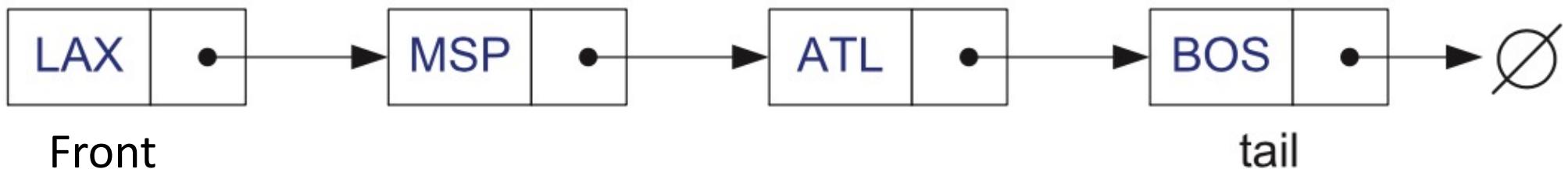
# LINKED LISTS

- A linked list is a data structure consisting of a chain of nodes
- Each node stores **data element**, in addition to at least one **pointer** that points to the next node in the list



# SINGLY LINKED LIST

- The first node = the node with the “front” element
- The last node = whose “*next*” points to NULL



- *No predetermined fixed size*, unlike arrays!
- It is called “**singly linked list**” because each node stores a single link (pointer)!

# IMPLEMENTATION

Let's see how to implement these classes



```
class StringNode { // An object of this class is one node
private:
    string elem;                                // element value
    StringNode* next;                            // pointer to the next node
friend class StringLinkedList; // Now, an object of StringLinkedList can access
};                                              // the node's private members "elem" and "next"

class StringLinkedList { // An object of this class is a singly linked list
private:
    StringNode* head; // this is a pointer that points to the front
public:
    StringLinkedList():head(NULL){}; // constructor; sets head = NULL
    ~StringLinkedList();           // destructor; clears the memory
    bool empty() const;           // returns "true" if the list is empty
    const string& front() const;   // returns the element at the front
    void addFront(const string& e); // puts "e" in a node, and adds it to the front
    void removeFront();           // deletes the node at the front of the list
};
```

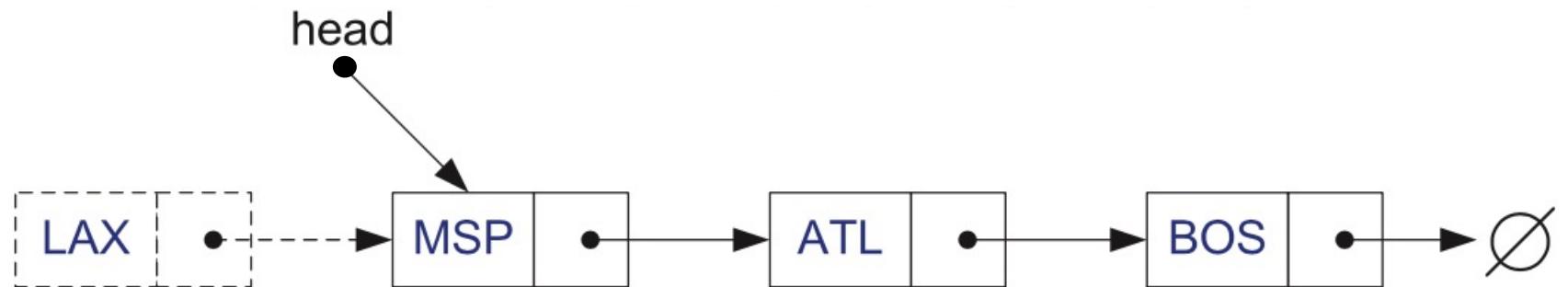
The head is implemented as the pointer that points to the first node

# SINGLY LINKED LIST: INSERTION TO THE FRONT

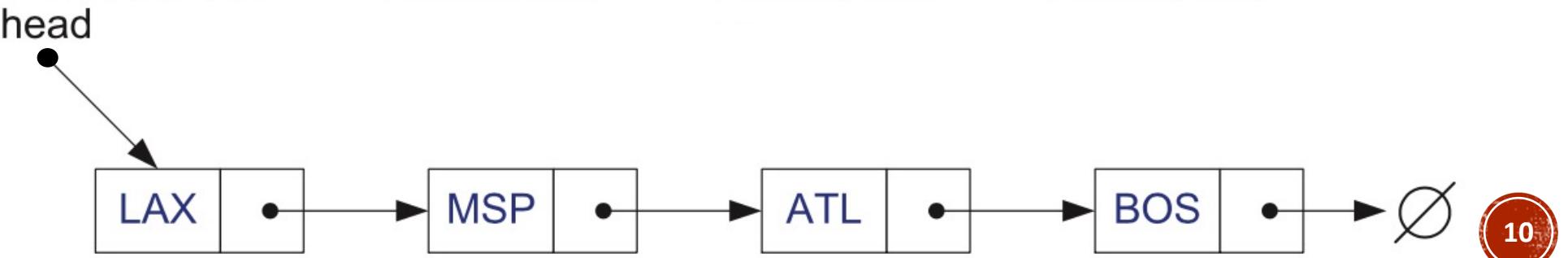
Before insertion



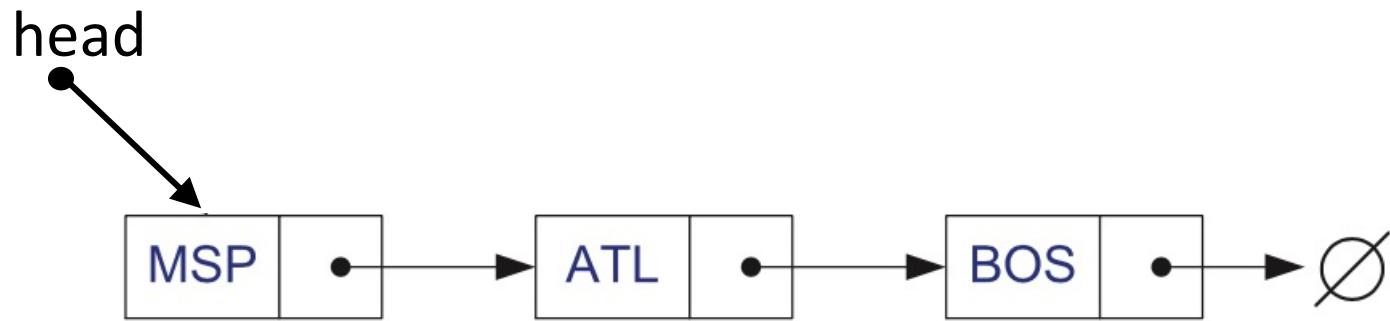
During insertion



After insertion



# SINGLY LINKED LIST: INSERTION TO THE FRONT

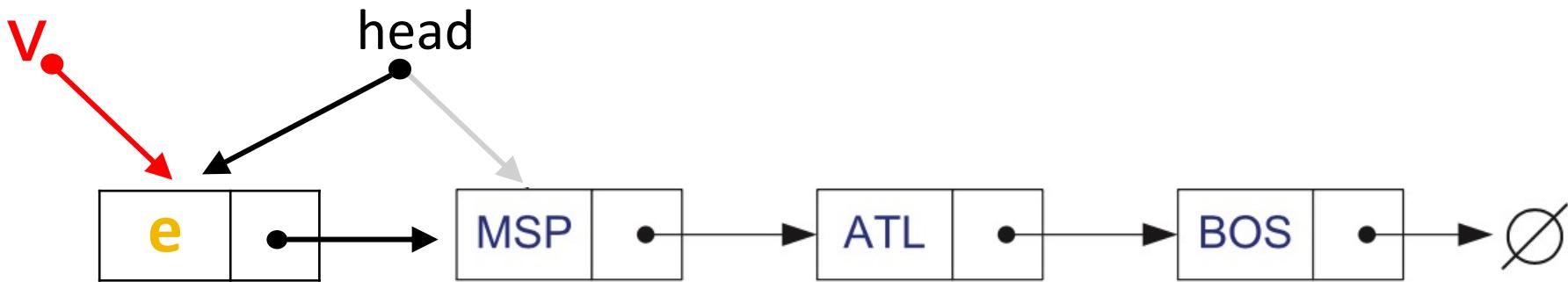


How would you implement this?

```
void StringLinkedList::addFront(const string& e) {
```

```
}
```

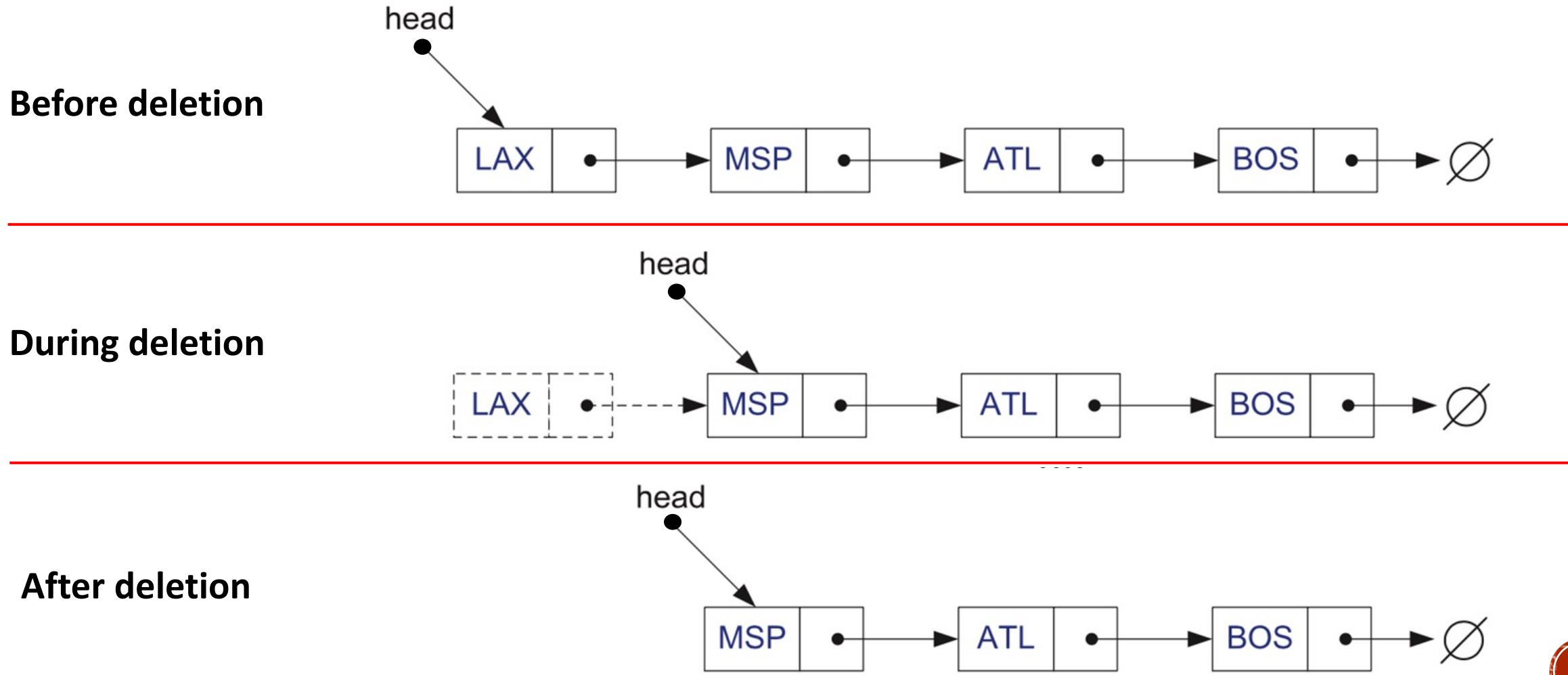
# SINGLY LINKED LIST: INSERTION TO THE FRONT



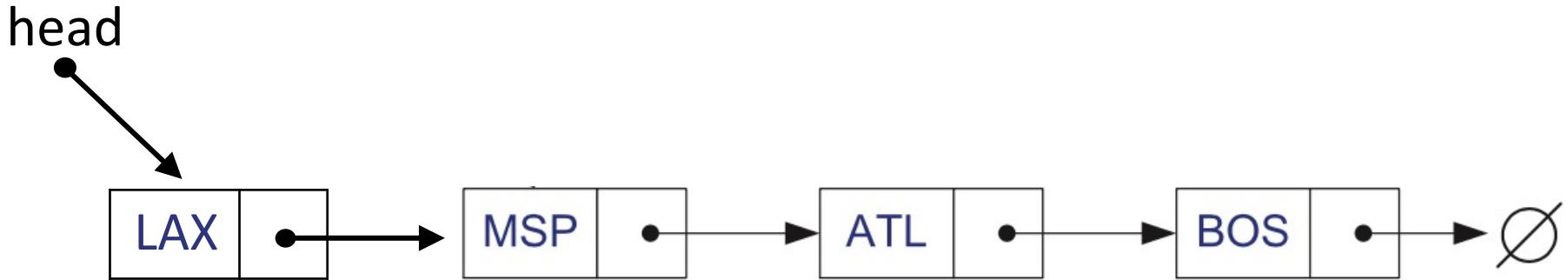
How would you implement this?

```
void StringLinkedList::addFront(const string& e) {  
    StringNode* v = new StringNode; // create new node, "v"  
    v->elem = e; // store "e" in the node "v"  
    v->next = head; // Now, "v" point to the front of the list  
    head = v; // set "v" to be the new head  
}
```

# SINGLY LINKED LIST: DELETION TO THE FRONT



# SINGLY LINKED LIST: DELETION TO THE FRONT

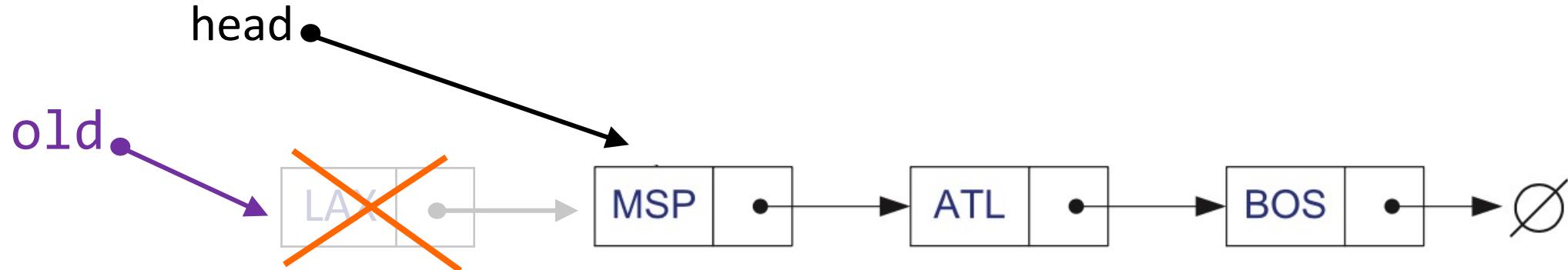


How would you implement this?

```
void StringLinkedList::removeFront() {
```

```
}
```

# SINGLY LINKED LIST: DELETION TO THE FRONT



How would you implement this?

```
void StringLinkedList::removeFront() {  
    StringNode* old = head; //save current front (pointed to by head)  
    head = old->next;      //skip over old front  
    delete old;            //delete the old front  
}
```