MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

# 11 Linked List Structures

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

February 7, 2022

# Administration

- The midterm will be in-person
  - Location:  PGCLL B138
  - Date: Monday, February 14, 2022
  - Time: 1:30 PM - 3:30 PM

- The university policy is that we are back fully in person as of Feb 7th. If someone feels unsafe coming to campus to take a test, they can apply for an exemption for in person learning through student accessibility services, though these will only be approved if appropriate medical documentation is provided.

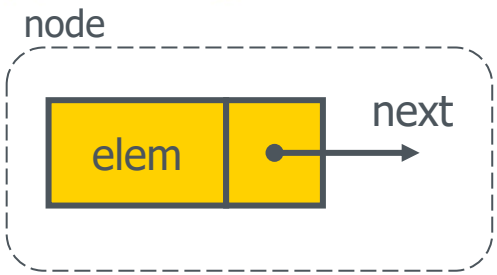
- I will make an announcement about the sources that you can study for the mid-term.

# Review Singly Linked List - AddFront

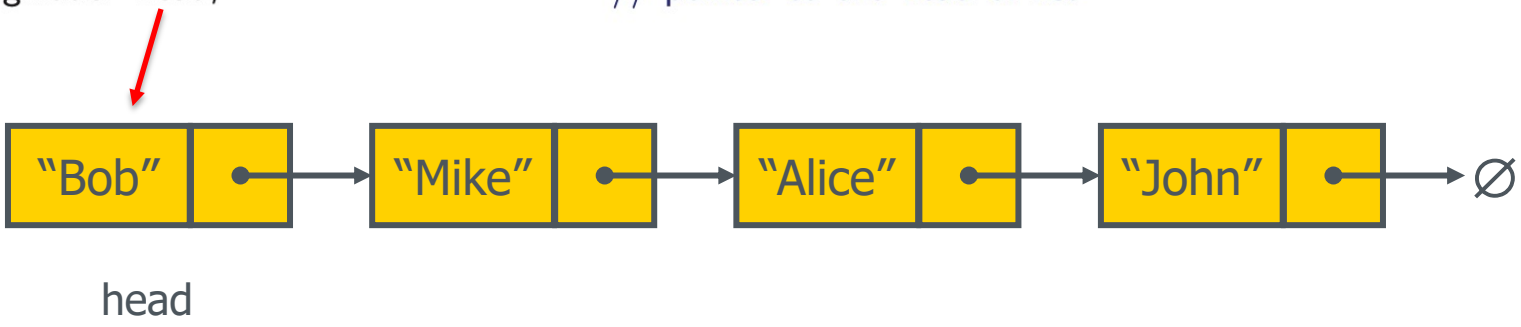
- For storing strings only!

```
class StringNode {                    // a node in a list of strings
private:
  string elem;                        // element value
  StringNode* next;                   // next item in the list

  friend class StringLinkedList;      // provide StringLinkedList access
};
```



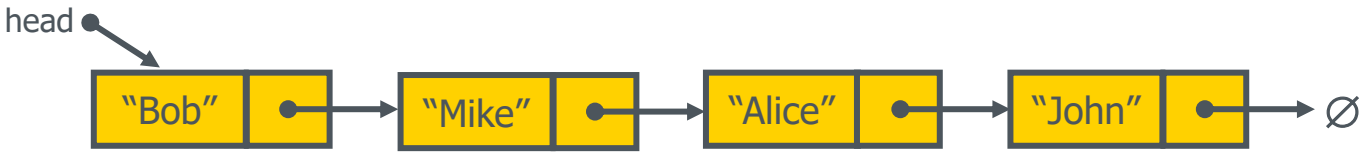

```
class StringLinkedList {              // a linked list of strings
public:
  StringLinkedList();                 // empty list constructor
  ~StringLinkedList();                // destructor
  bool empty() const;                 // is list empty?
  const string& front() const;        // get front element
  void addFront(const string& e);     // add to front of list
  void removeFront();                 // remove front item list
private:
  StringNode* head;                   // pointer to the head of list
};
```



# Singly Linked List - addFront

- Insert element at the head of the singly linked list

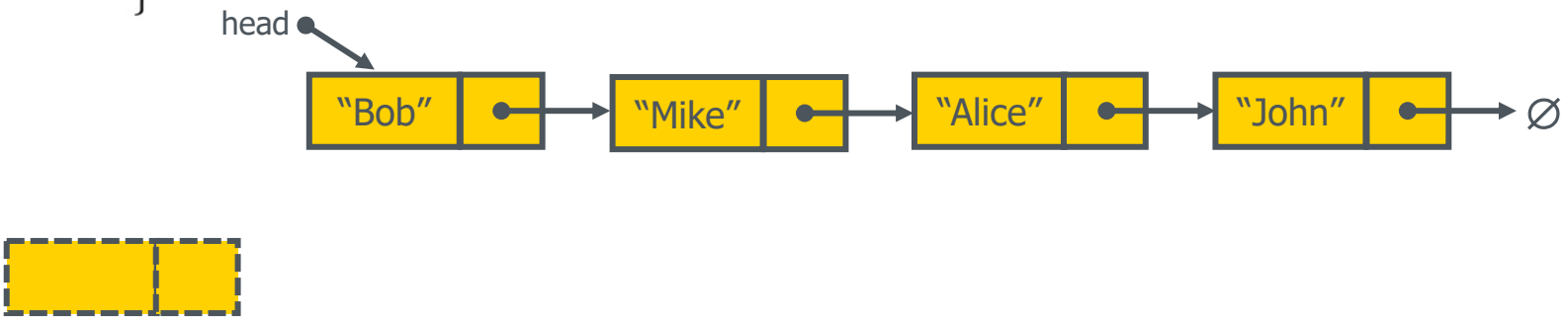
```
void StringLinkedList::addFront(const string& e) {  // add to front of list
  StringNode* v = new StringNode;                   // create new node
  v->elem = e;                                      // store data
  v->next = head;                                   // head now follows v
  head = v;                                         // v is now the head
}
```



# Singly Linked List - addFront

- Insert element at the head of the singly linked list

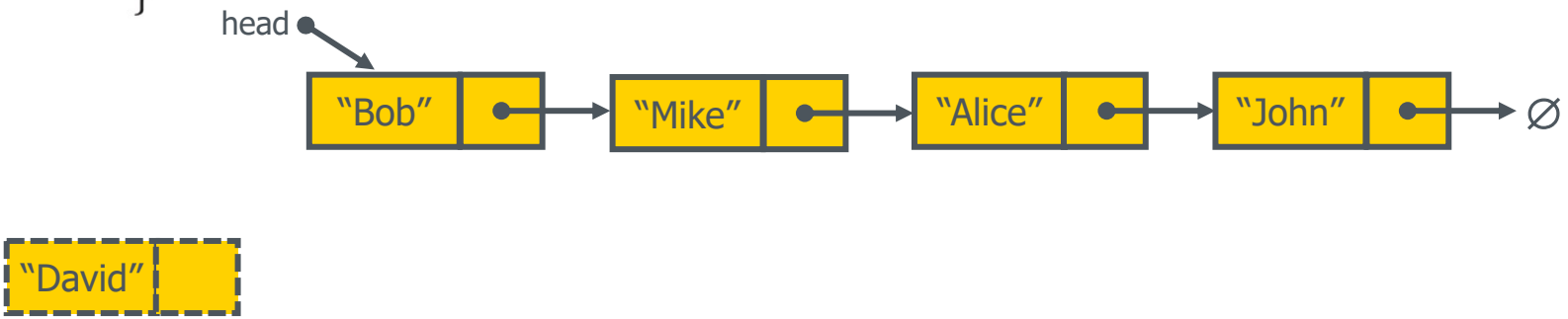
```
void StringLinkedList::addFront(const string& e) {  // add to front of list
  StringNode* v = new StringNode;                   // create new node
  v->elem = e;                                      // store data
  v->next = head;                                   // head now follows v
  head = v;                                         // v is now the head
}
```



# Singly Linked List - addFront

- Insert element at the head of the singly linked list

```
void StringLinkedList::addFront(const string& e) {   // add to front of list
  StringNode* v = new StringNode;                    // create new node
  v->elem = e;                                       // store data
  v->next = head;                                    // head now follows v
  head = v;                                          // v is now the head
}
```

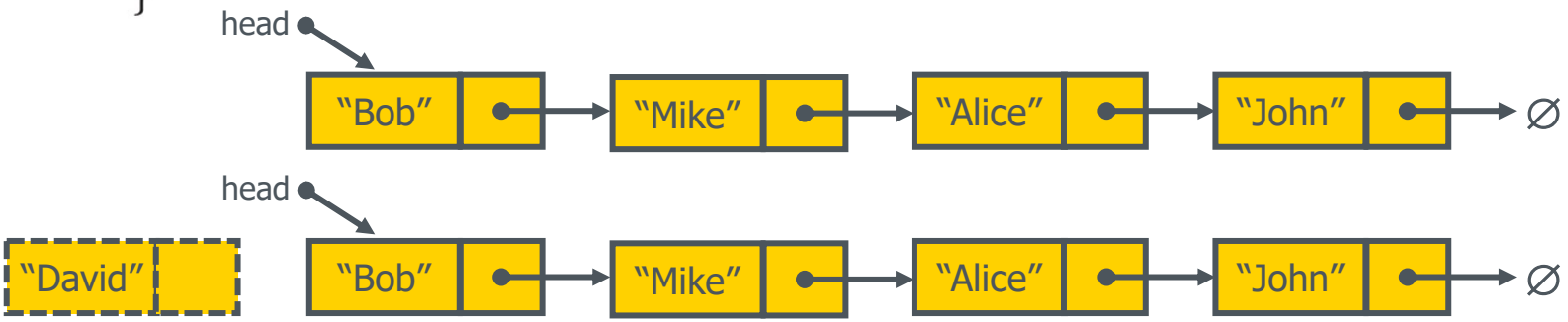



- Notice friendship!

# Singly Linked List - addFront

- Insert element at the head of the singly linked list

```
void StringLinkedList::addFront(const string& e) {   // add to front of list
  StringNode* v = new StringNode;                    // create new node
  v->elem = e;                                       // store data
  v->next = head;                                    // head now follows v
  head = v;                                          // v is now the head
}
```

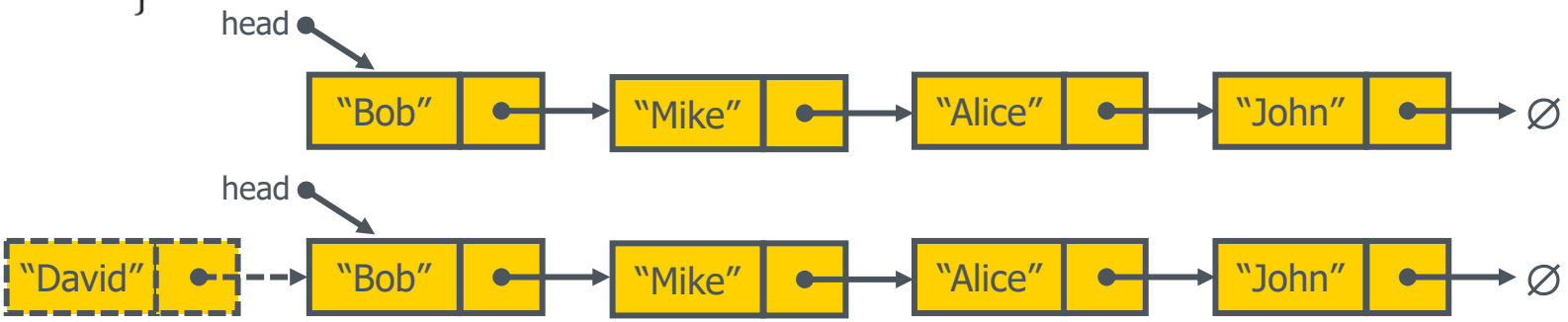



- Notice friendship!

# Singly Linked List - addFront

- Insert element at the head of the singly linked list

```
void StringLinkedList::addFront(const string& e) {   // add to front of list
  StringNode* v = new StringNode;                    // create new node
  v->elem = e;                                       // store data
  v->next = head;                                    // head now follows v
  head = v;                                          // v is now the head
}
```

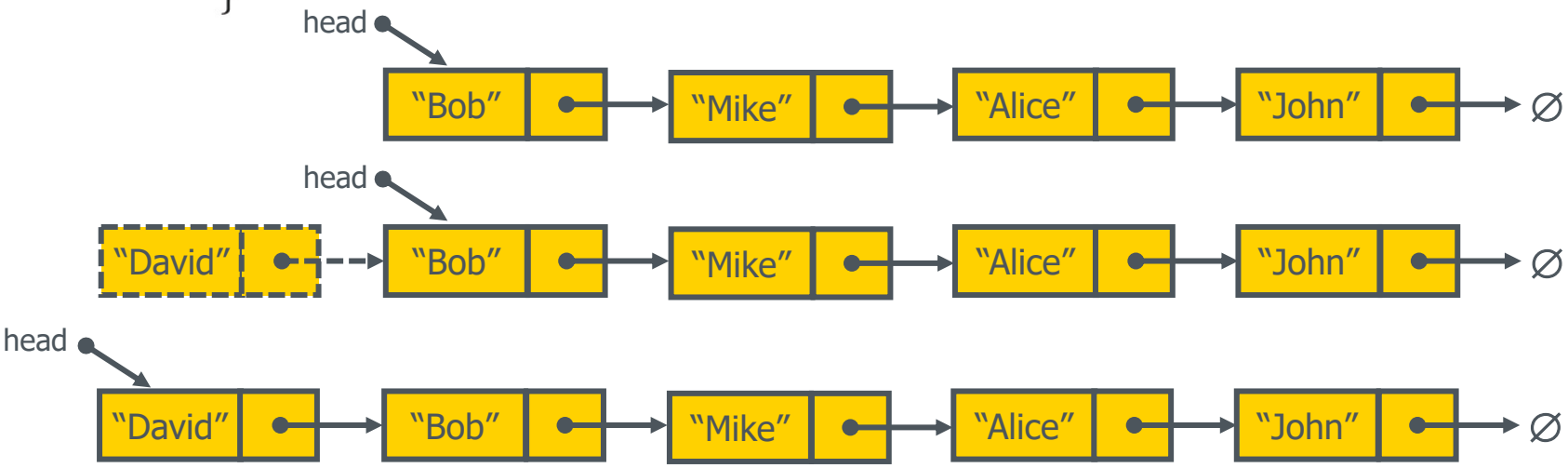



- Notice friendship!

# Singly Linked List - addFront

- Insert element at the head of the singly linked list

```
void StringLinkedList::addFront(const string& e) {   // add to front of list
  StringNode* v = new StringNode;                    // create new node
  v->elem = e;                                       // store data
  v->next = head;                                    // head now follows v
  head = v;                                          // v is now the head
}
```



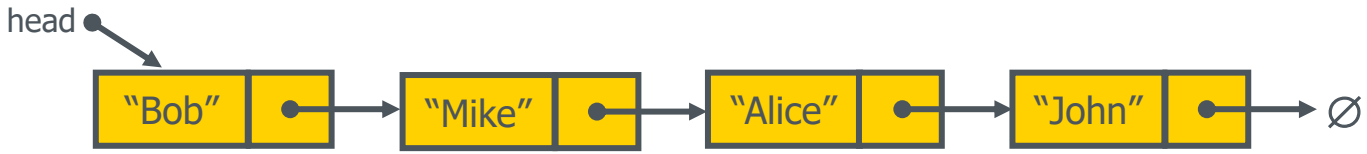

- Notice friendship!

# Singly Linked List - removeFront

- Remove an element from the head of the singly linked list

```
void StringLinkedList::removeFront() {
  StringNode* old = head;
  head = old->next;
  delete old;
}
```

```
// remove front item
// save current head
// skip over old head
// delete the old head
```

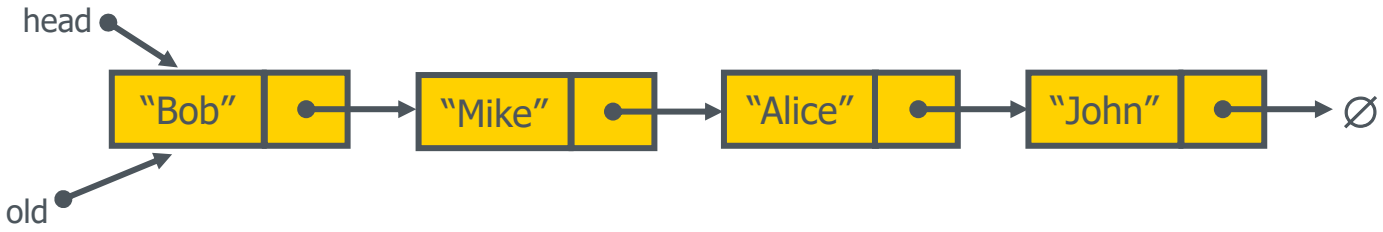


# Singly Linked List - removeFront

- Remove an element from the head of the singly linked list

```
void StringLinkedList::removeFront() {
  StringNode* old = head;
  head = old->next;
  delete old;
}
```

```
// remove front item
// save current head
// skip over old head
// delete the old head
```

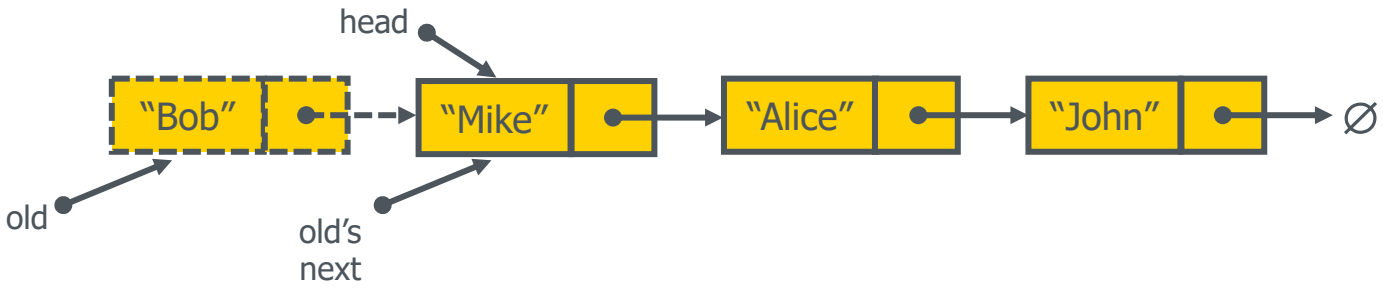


# Singly Linked List - removeFront

- Remove an element from the head of the singly linked list

```
void StringLinkedList::removeFront() {
  StringNode* old = head;
  head = old->next;
  delete old;
}
```

```
// remove front item
// save current head
// skip over old head
// delete the old head
```

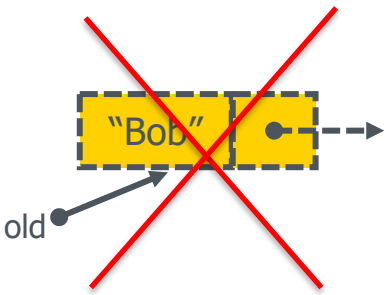


# Singly Linked List - removeFront

- Remove an element from the head of the singly linked list

```
void StringLinkedList::removeFront() {
  StringNode* old = head;
  head = old->next;
  delete old;
}
```

```
// remove front item
// save current head
// skip over old head
// delete the old head
```

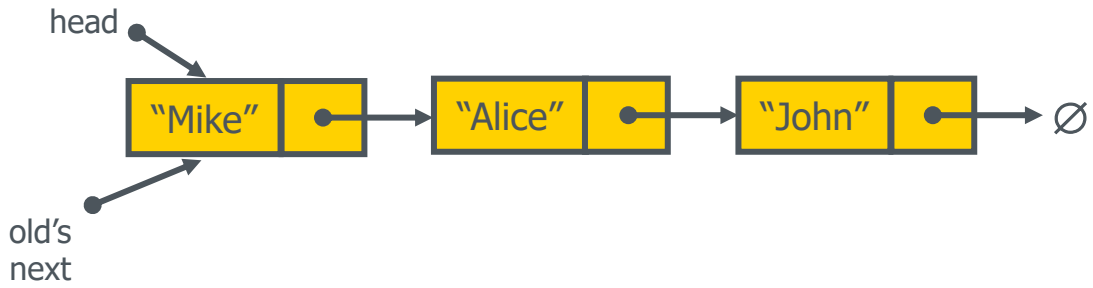

- Avoid memory leak!

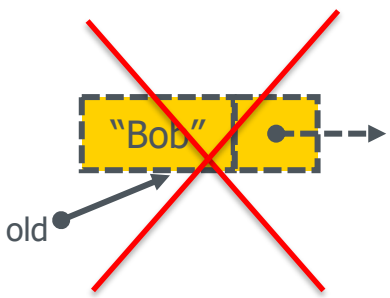
# Singly Linked List - removeFront

- Remove an element from the head of the singly linked list

```
void StringLinkedList::removeFront() {
  StringNode* old = head;
  head = old->next;
  delete old;
}
```

```
// remove front item
// save current head
// skip over old head
// delete the old head
```

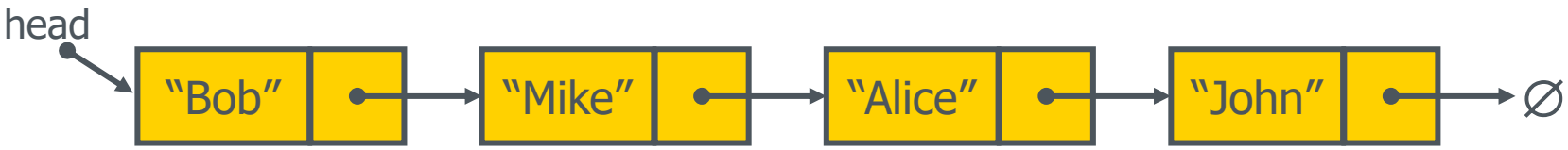



- Avoid memory leak!

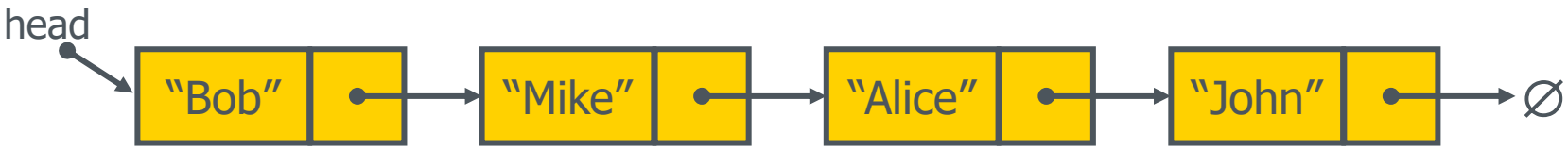
# Other Operations on Singly Linked Lists

- How to insert at the tail?



# Other Operations on Singly Linked Lists

- How to insert at the tail?




  - Allocate a new node
  - Insert new element
  - Have new node point to Null
  - Traverse to find the old last node (how?) and have it to point to new node

# Generic Singly Linked List

- We assumed elements were strings, now we want arbitrary element types
- It is easy using C++'s **Template** mechanism

Before

```
class StringNode {                  // a node in a list of strings
private:
  string elem;                      // element value
  StringNode* next;                 // next item in the list

  friend class StringLinkedList;    // provide StringLinkedList access
};
```

After

```
template <typename E>
class SNode {                       // singly linked list node
private:
  E elem;                           // linked list element value
  SNode<E>* next;                   // next item in the list
  friend class SLinkedList<E>;      // provide SLinkedList access
};
```

# Generic Singly Linked List

```
class StringLinkedList {                  // a linked list of strings
public:
  StringLinkedList();                     // empty list constructor
  ~StringLinkedList();                    // destructor
  bool empty() const;                     // is list empty?
  const string& front() const;            // get front element
  void addFront(const string& e);         // add to front of list
  void removeFront();                     // remove front item list
private:
  StringNode* head;                       // pointer to the head of list
};
```

Before

```
template <typename E>
class SLinkedList {                       // a singly linked list
public:
  SLinkedList();                          // empty list constructor
  ~SLinkedList();                         // destructor
  bool empty() const;                     // is list empty?
  const E& front() const;                 // return front element
  void addFront(const E& e);              // add to front of list
  void removeFront();                     // remove front item list
private:
  SNode<E>* head;                         // head of the list
};
```

After

# Generic Singly Linked List

## Before

```
StringLinkedList::StringLinkedList()
  : head(NULL) { }


bool StringLinkedList::empty() const
  { return head == NULL; }

const string& StringLinkedList::front() const
  { return head->elem; }


StringLinkedList::~StringLinkedList()
  { while (!empty()) removeFront(); }

void StringLinkedList::addFront(const string& e) {
  StringNode* v = new StringNode;
  v->elem = e;
  v->next = head;
  head = v;
}

void StringLinkedList::removeFront() {
  StringNode* old = head;
  head = old->next;
  delete old;
}
```

## After

```
template <typename E>
SLinkedList<E>::SLinkedList()
  : head(NULL) { }

template <typename E>
bool SLinkedList<E>::empty() const
  { return head == NULL; }

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

template <typename E>
SLinkedList<E>::~SLinkedList()
  { while (!empty()) removeFront(); }

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

template <typename E>
void SLinkedList<E>::removeFront() {
  SNode<E>* old = head;
  head = old->next;
  delete old;
}
```

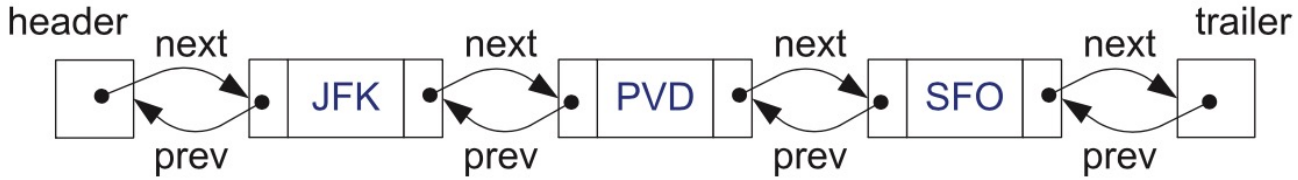
## Usage

```
SLinkedList<string> a;     // list of strings
a.addFront("MSP");
// ...
SLinkedList<int> b;        // list of integers
b.addFront(13);
```

# Limitations of Singly Linked Lists

- Not easy to remove an element at the tail (or any other node)
  - We don't have a quick way to access to the node immediately preceding the one we want to delete!
  - Some operations need the tree traversal from beginning to the end


- A better Data Structure
  - Doubly Linked List
    - Offers traversing in both directions
    - Quick update operations (insert or remove at any given position)



# Administration

- The first mid-term's material will be up to this slide!

# Doubly Linked Lists

- Allows traversing in both directions (forward and reverse)
- Each node stores
  - element
  - link to the next node
  - link to the previous node
- Sentinel nodes
  - Dummy **header** node
  - Dummy **trailer** node

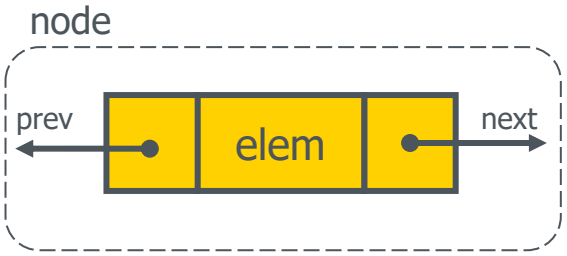

∅

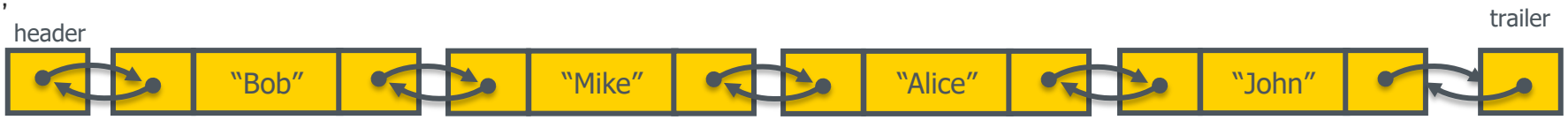
# Doubly Linked List C++ Classes Declaration

- Notice typedef!
  - Generic, like templates

```
typedef string Elem;                    // list element type
class DNode {                           // doubly linked list node
private:
  Elem elem;                            // node element value
  DNode* prev;                          // previous node in list
  DNode* next;                          // next node in list
  friend class DLinkedList;             // allow DLinkedList access
};
```

```
class DLinkedList {                     // doubly linked list
public:
  DLinkedList();                        // constructor
  ~DLinkedList();                       // destructor
  bool empty() const;                   // is list empty?
  const Elem& front() const;            // get front element
  const Elem& back() const;             // get back element
  void addFront(const Elem& e);         // add to front of list
  void addBack(const Elem& e);          // add to back of list
  void removeFront();                   // remove from front
  void removeBack();                    // remove from back
private:                                // local type definitions
  DNode* header;                        // list sentinels
  DNode* trailer;
protected:                              // local utilities
  void add(DNode* v, const Elem& e);    // insert new node before v
  void remove(DNode* v);                // remove node v
};
```



# Doubly Linked List Definitions

- Constructor
- is Empty?
  - header and trailer pointing each other
- Return front and back elements
- Dynamic memory allocation
  - We need destructor
- Destructor
  - remove nodes until list is empty

```
DLinkedList::DLinkedList() {                // constructor
  header = new DNode;                       // create sentinels
  trailer = new DNode;
  header->next = trailer;                   // have them point to each other
  trailer->prev = header;
}

bool DLinkedList::empty() const             // is list empty?
  { return (header->next == trailer); }

const Elem& DLinkedList::front() const      // get front element
  { return header->next->elem; }

const Elem& DLinkedList::back() const       // get back element
  { return trailer->prev->elem; }

DLinkedList::~DLinkedList() {               // destructor
  while (!empty()) removeFront();           // remove all but sentinels
  delete header;                            // remove the sentinels
  delete trailer;
}
```

# Doubly Linked List - Add Element

- add() is protected
  - Utility function, why?

```
                                                    // insert new node before v
void DLinkedList::add(DNode* v, const Elem& e) {
  DNode* u = new DNode; u->elem = e; // create a new node for e
  u->next = v;                       // link u in between v
  u->prev = v->prev;                 // ...and v->prev
  v->prev->next = v->prev = u;
}

void DLinkedList::addFront(const Elem& e) // add to front of list
  { add(header->next, e); }

void DLinkedList::addBack(const Elem& e) // add to back of list
  { add(trailer, e); }
```

# Doubly Linked List - Remove Element

- remove() is protected
  - Utility function

```
void DLinkedList::remove(DNode* v) {      // remove node v
  DNode* u = v->prev;                     // predecessor
  DNode* w = v->next;                     // successor
  u->next = w;                            // unlink v from list
  w->prev = u;
  delete v;
}

void DLinkedList::removeFront()           // remove from font
  { remove(header->next); }

void DLinkedList::removeBack()            // remove from back
  { remove(trailer->prev); }
```

# Questions?