MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

12 Linked Lists - Continued

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

February 9, 2022

Administration

- The midterm will be in-person
  - Location: PGCLL B138
  - Date: Monday, February 14, 2022
  - Time: 1:30 PM - 2:30 PM **Mid-Term 1's duration: ONE HOUR**

- 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.

- Read the announcement about resources that you can study for the mid-term.
- My Office Hour:
  - Today at 15:00 in **ITB-159** in-person (or virtually using teams as usual)

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

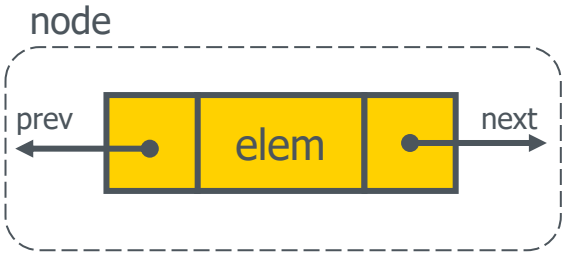
∅

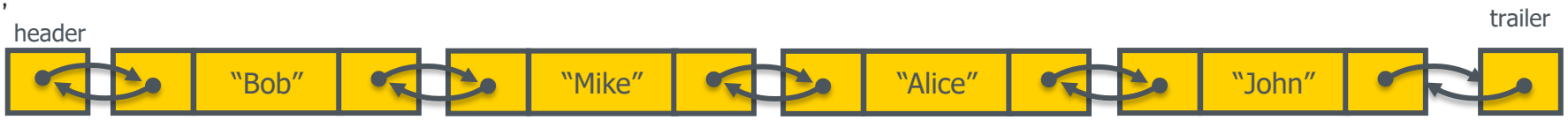
Doubly Linked List C++ Classes Declaration

- 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

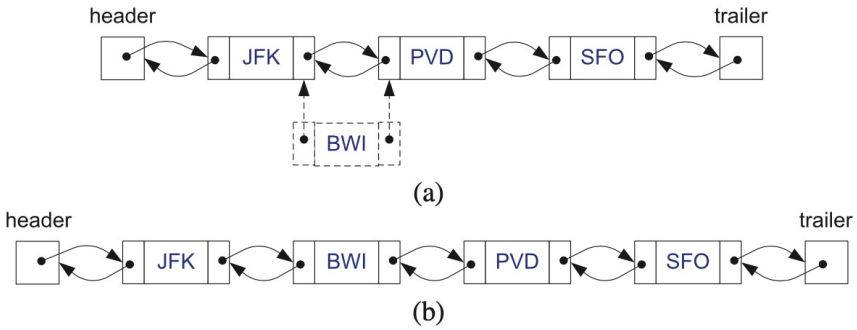
- Constructor
- is Empty?
  - header and trailer pointing each other
- Return front and back elements
- Dynamic memory allocation
  - We need destructor
- Destructor
  - remove nodes until list is empty

```
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?

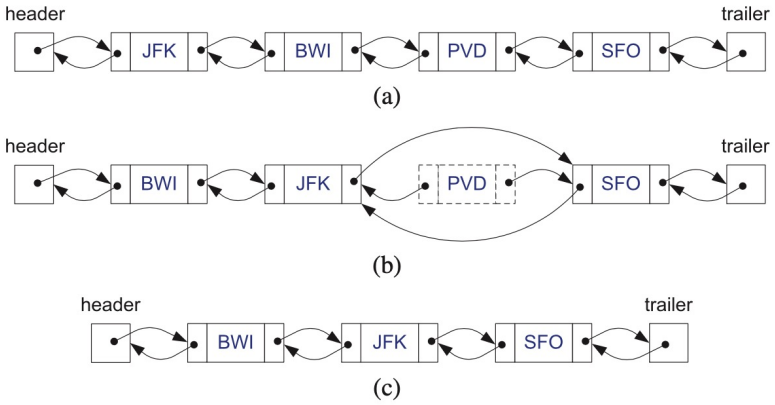
```
                                                    // insert new node before v
void DLinkedList::add(DNode* v, const Elem& e) {
  DNode* u = new DNode; u->elem = e;                // create a new node for e
  u->next = v;                                      // link u in between v
  u->prev = v->prev;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



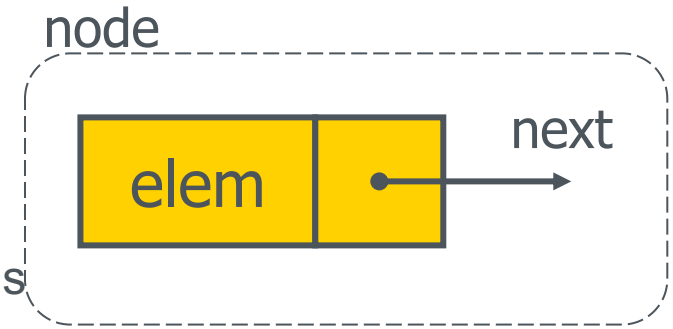
Doubly Linked List - Reverse

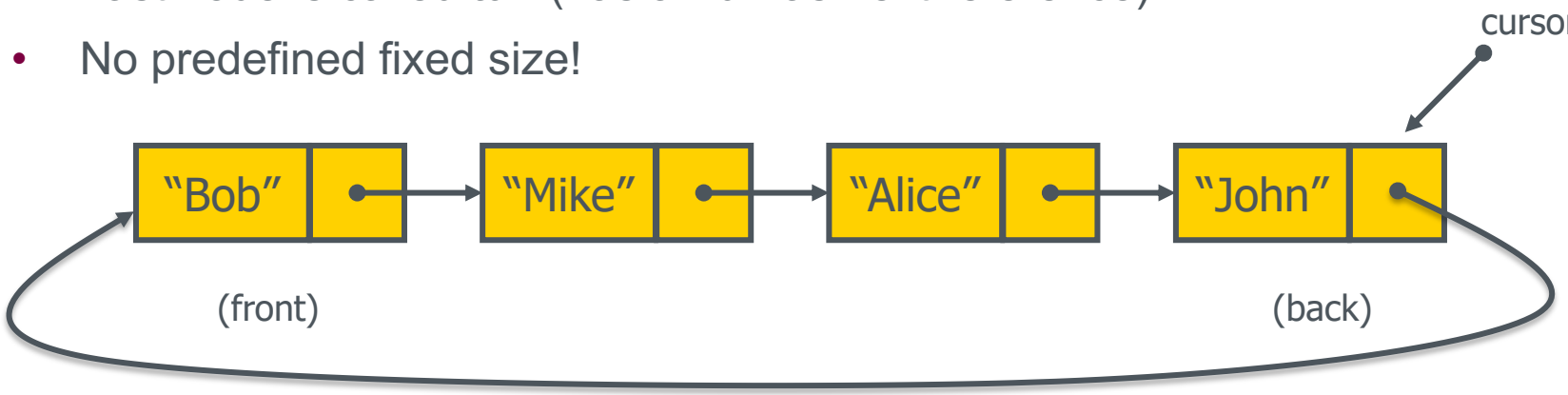
- This is not a function declared in the class
- See the tutorial of week 4 for a possible implementation in the class

```
void listReverse(DLinkedList& L) {          // reverse a list
  DLinkedList T;                            // temporary list
  while (!L.empty()) {                      // reverse L into T
    string s = L.front();  L.removeFront();
    T.addFront(s);
  }
  while (!T.empty()) {                      // copy T back to L
    string s = T.front();  T.removeFront();
    L.addBack(s);
  }
}
```

Circularly Linked List

- A variant of Singly Linked List
  - Rather than having a **head** or a **tail**, it forms a cycle
- Same node structure as Singly Linked List
- Cursor
  - A *virtual* starting node
  - This can be varying as we perform operations
- First node is called **head**
- Last node is called **tail** (has a **null** as next reference)
- No predefined fixed size!



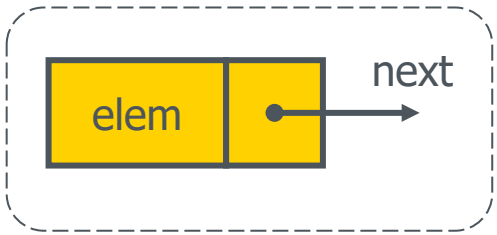


Circularly Linked List C++ Classes Declaration

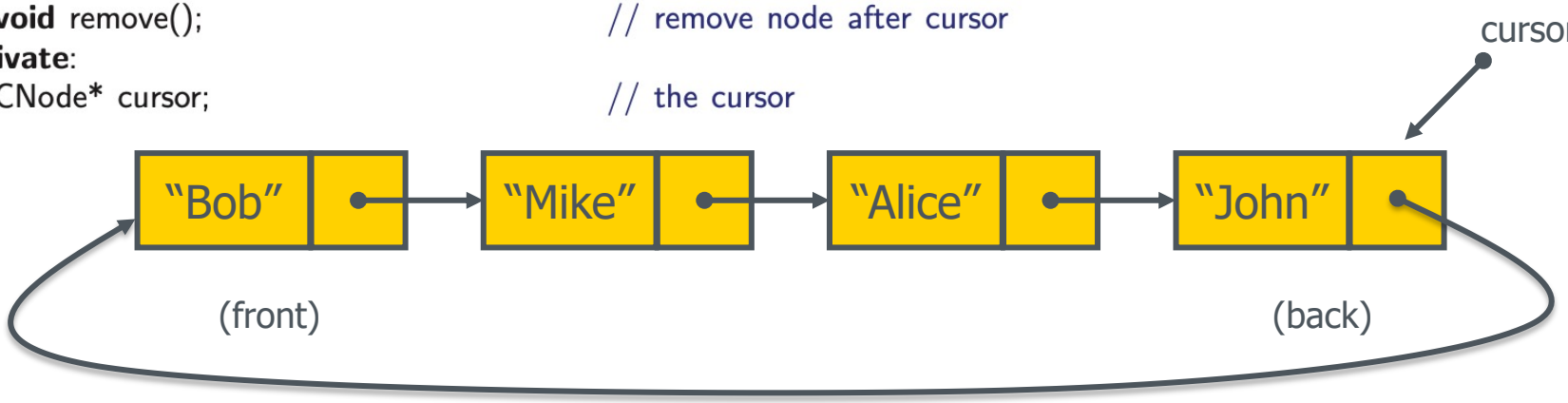
- Notice typedef!
  - Generic, like templates

```
typedef string Elem;                  // element type
class CNode {                         // circularly linked list node
private:
  Elem elem;                          // linked list element value
  CNode* next;                        // next item in the list

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

node

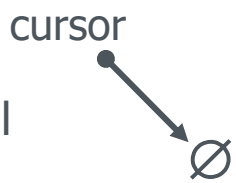



```
class CircleList {                    // a circularly linked list
public:
  CircleList();                       // constructor
  ~CircleList();                      // destructor
  bool empty() const;                 // is list empty?
  const Elem& front() const;          // element at cursor
  const Elem& back() const;           // element following cursor
  void advance();                     // advance cursor
  void add(const Elem& e);            // add after cursor
  void remove();                      // remove node after cursor
private:
  CNode* cursor;                      // the cursor
};
```

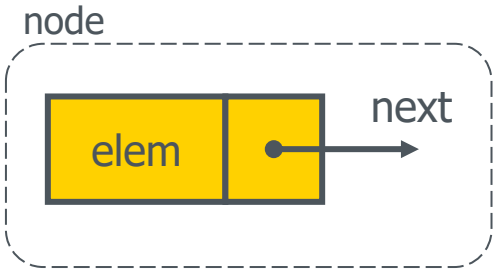


Circular Linked List Definitions

- Constructor
  - Set cursor to Null
- Destructor
  - remove nodes until list is empty



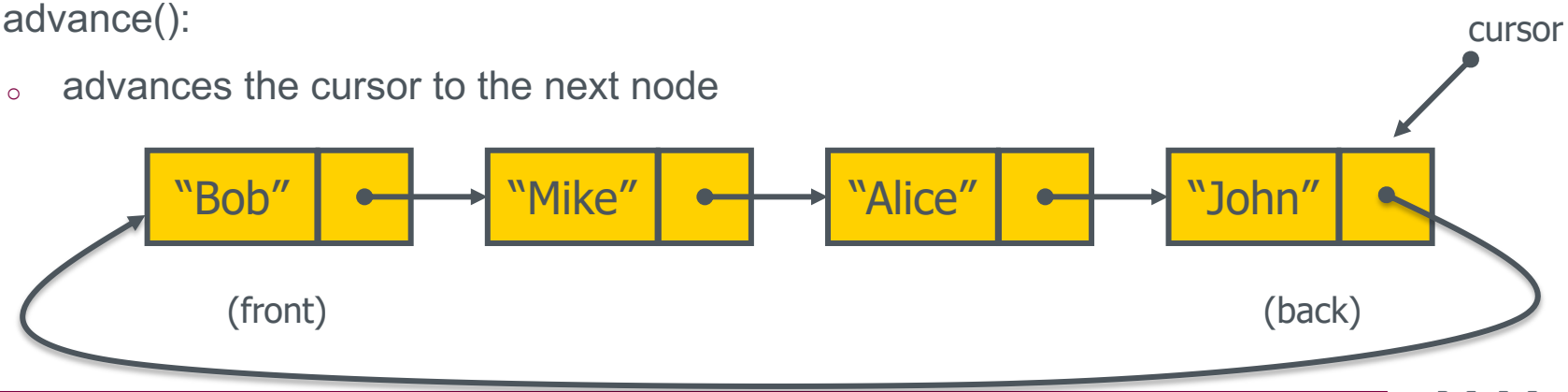

```
CircleList::CircleList()                    // constructor
  : cursor(NULL) { }
CircleList::~CircleList()                   // destructor
  { while (!empty()) remove(); }
```



Circular Linked List Definitions

- Constructor
  - Set cursor to Null
- Destructor
  - remove nodes until list is empty
- is Empty?
  - check if cursor is Null
- Return **front** element
  - return element immediately after cursor
- Return **back** element
  - return element referenced by cursor
- advance():
  - advances the cursor to the next node

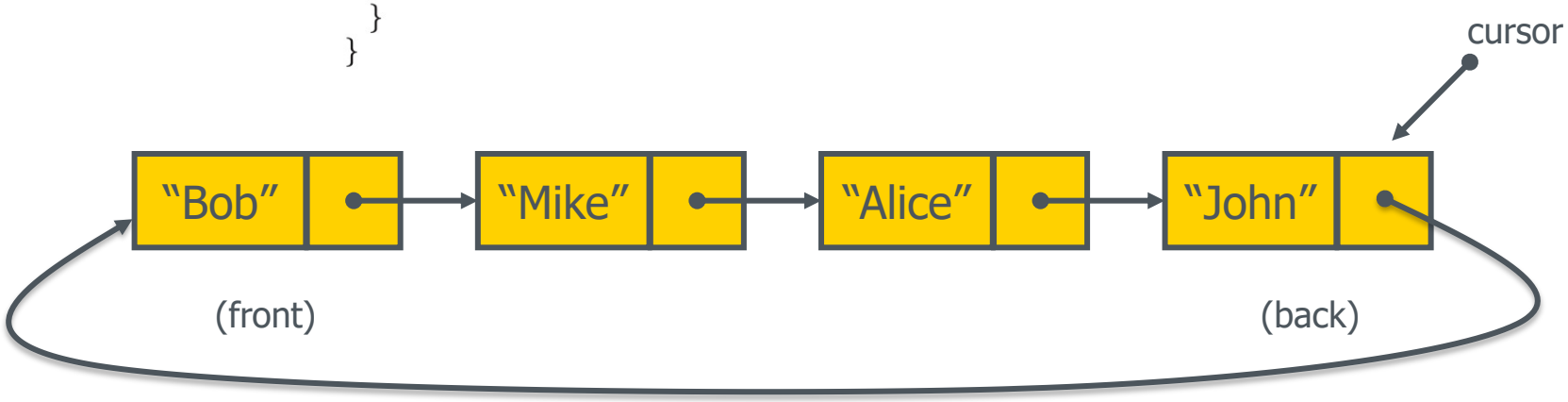
```
CircleList::CircleList()                    // constructor
  : cursor(NULL) { }
CircleList::~CircleList()                   // destructor
  { while (!empty()) remove(); }

bool CircleList::empty() const              // is list empty?
  { return cursor == NULL; }
const Elem& CircleList::back() const        // element at cursor
  { return cursor->elem; }
const Elem& CircleList::front() const       // element following cursor
  { return cursor->next->elem; }
void CircleList::advance()                  // advance cursor
  { cursor = cursor->next; }
```



Circular Linked List - add

- add an element
  - Insert a new node with element **e** immediately after the cursor; if the list is empty, then this node becomes the cursor and its next pointer points to itself.

```
void CircleList::add(const Elem& e) {      // add after cursor
  CNode* v = new CNode;                    // create a new node
  v->elem = e;
  if (cursor == NULL) {                    // list is empty?
    v->next = v;                           // v points to itself
    cursor = v;                            // cursor points to v
  }
  else {                                   // list is nonempty?
    v->next = cursor->next;                // link in v after cursor
    cursor->next = v;
  }
}
```

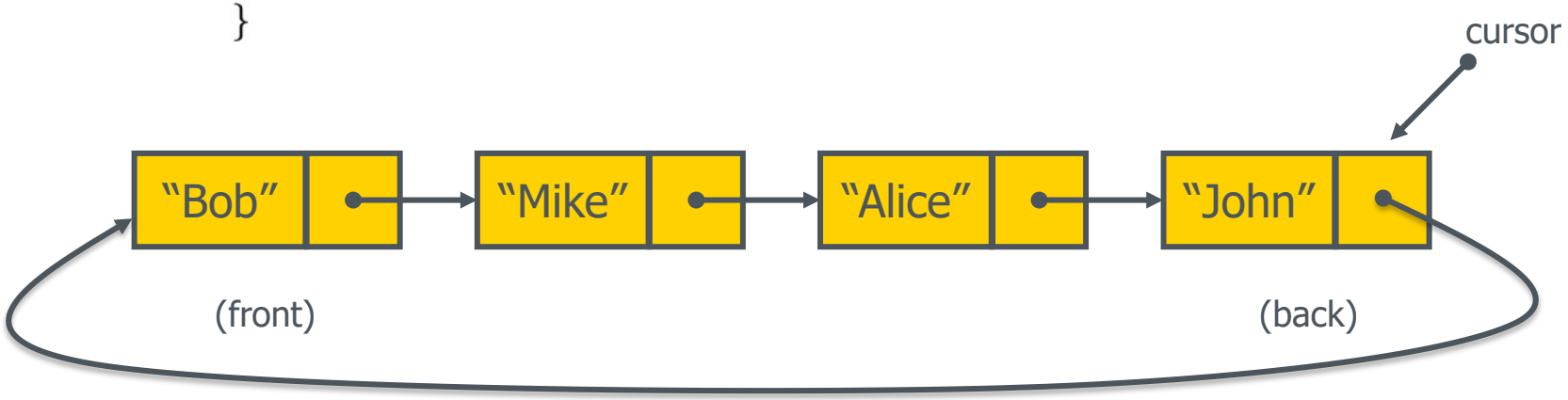


Circular Linked List - remove

- remove the element after cursor
  - Remove the node immediately after the cursor (not the cursor itself, unless it is the only node); if the list becomes empty, the cursor is set to null.

```
void CircleList::remove() {
  CNode* old = cursor->next;
  if (old == cursor)
    cursor = NULL;
  else
    cursor->next = old->next;
  delete old;
}
```

```
// remove node after cursor
// the node being removed
// removing the only node?
// list is now empty

// link out the old node
// delete the old node
```



Circularly Linked List - An Example

- A music playlist!

```
int main() {
  CircleList playList;                  // []
  playList.add("Stayin Alive");         // [Stayin Alive*]
  playList.add("Le Freak");             // [Le Freak, Stayin Alive*]
  playList.add("Jive Talkin");          // [Jive Talkin, Le Freak, Stayin Alive*]

  playList.advance();                   // [Le Freak, Stayin Alive, Jive Talkin*]
  playList.advance();                   // [Stayin Alive, Jive Talkin, Le Freak*]
  playList.remove();                    // [Jive Talkin, Le Freak*]
  playList.add("Disco Inferno");        // [Disco Inferno, Jive Talkin, Le Freak*]
  return EXIT_SUCCESS;
}
```

Questions?