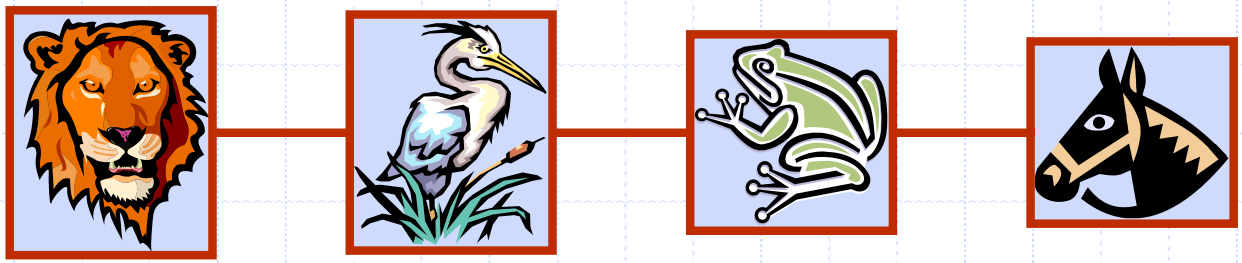
EE 205, Yung Yi
Lecture 2: Array and Linked Lists



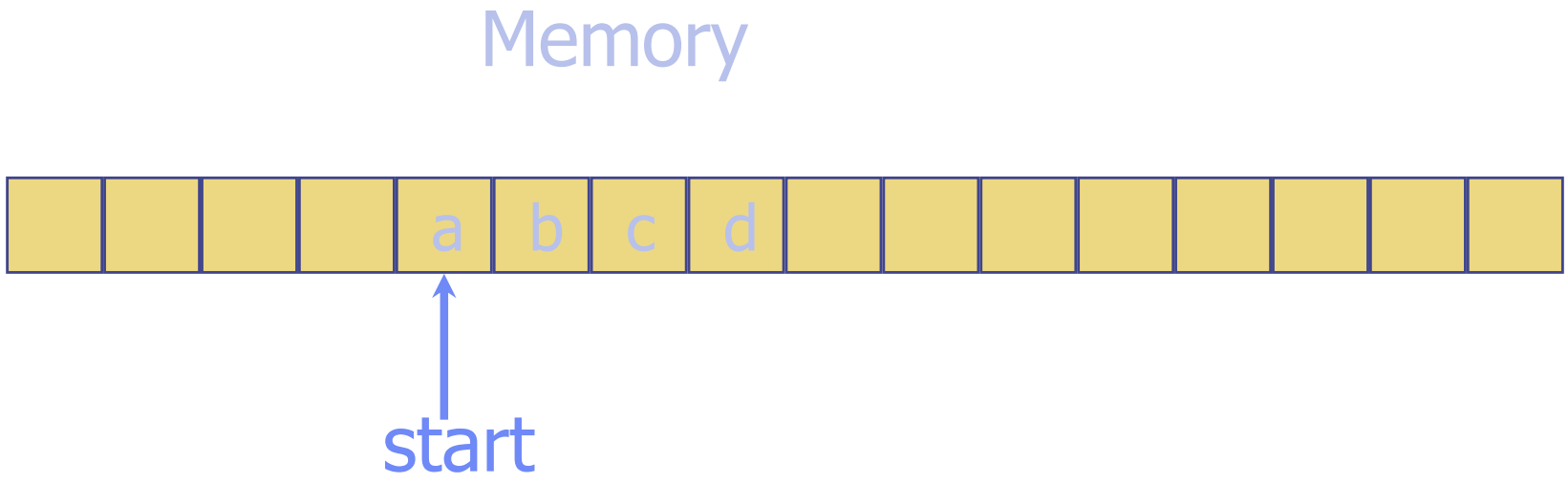
Overview and Reading

- Reading: Chapters: 3.1, 3.2, and 3.3

- Basic Elementary Data Structures

- Array
- Linked Lists
  - Singly linked lists
  - Doubly linked lists
  - Circular linked lists
- These are used for more advanced data structures later

Array (§ 3.1)



- Storing data in a sequential memory locations
- Access each element using integer index
- Very basic, popular, and simple
- int a[10]; int *a = new int(10);

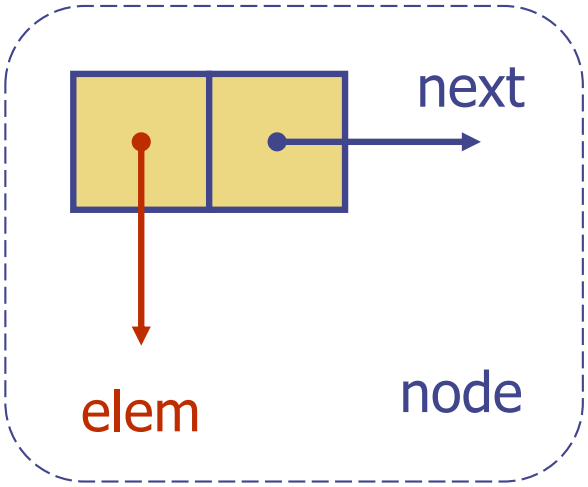
Array: Problems

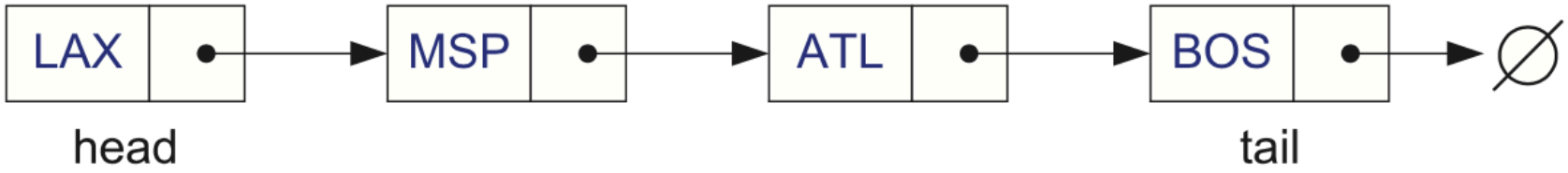
- New insertion and deletion: difficult
  - Need to shift to make space for insertion
  - Need to fill empty positions after deletion

- Why don’t we connect all elements just “logically” not “physically”?
  - Linked List

Singly Linked List (§ 3.2)

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node



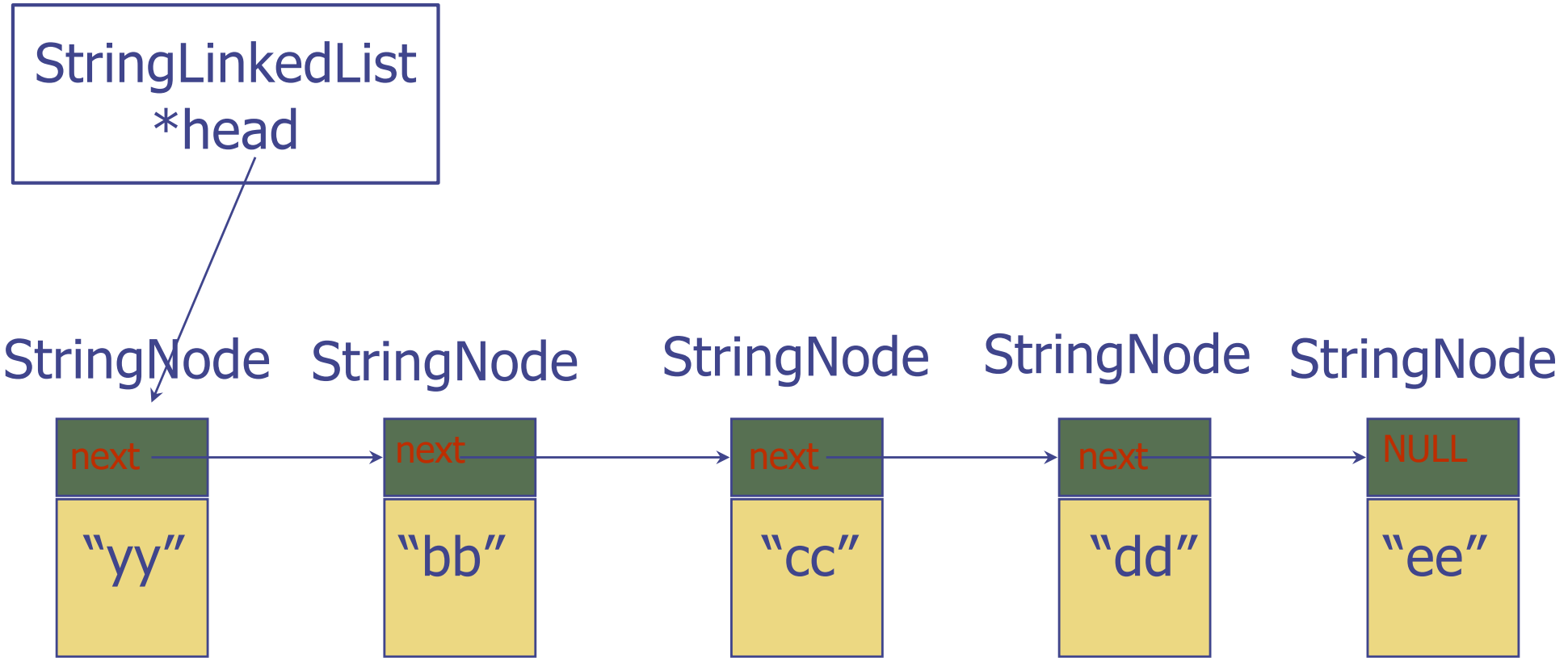


Example: Linked list of strings

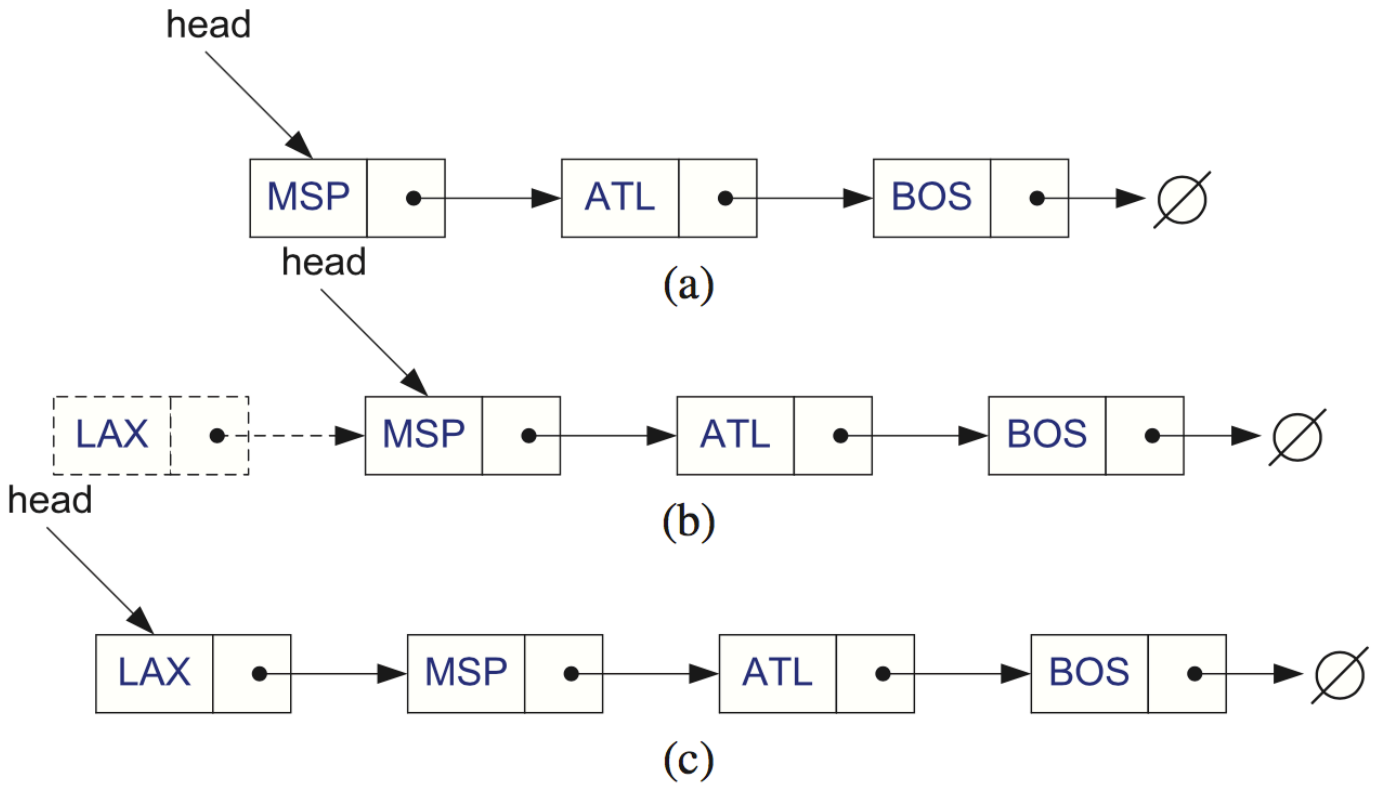
```
class StringNode {                          // a node in a list of strings
private:
  string elem;                              // element value
  StringNode* next;                         // next item in the list

  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 of Strings: Picture

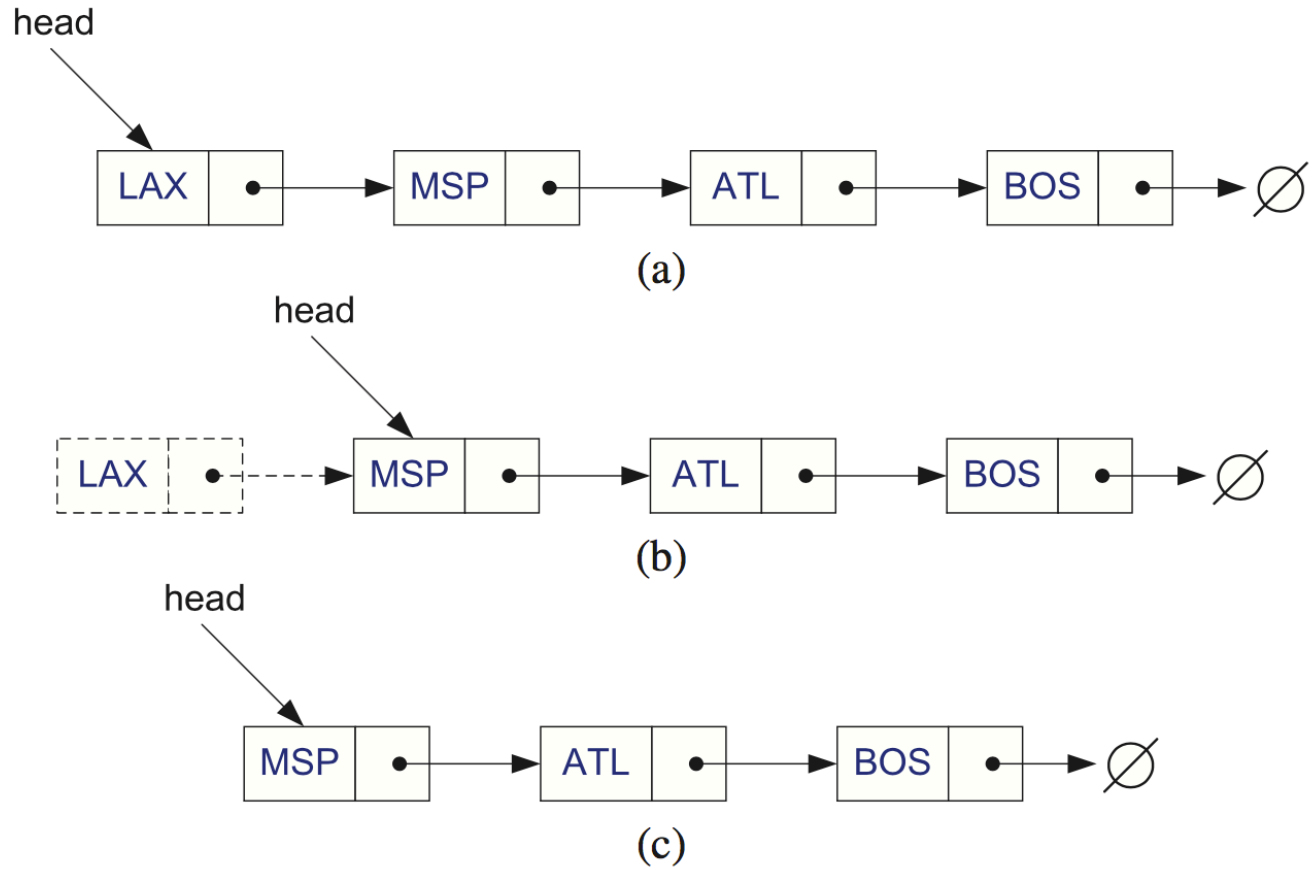


Inserting at the Head



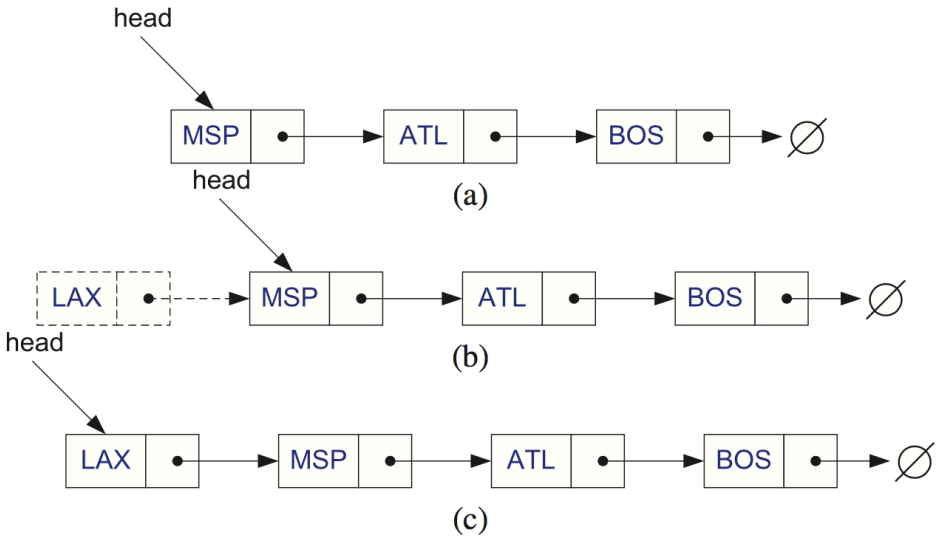
1. Allocate a new node
2. Insert a new element
3. Have the new node point to the old head
4. Update head to point to new node

Removing at the Head

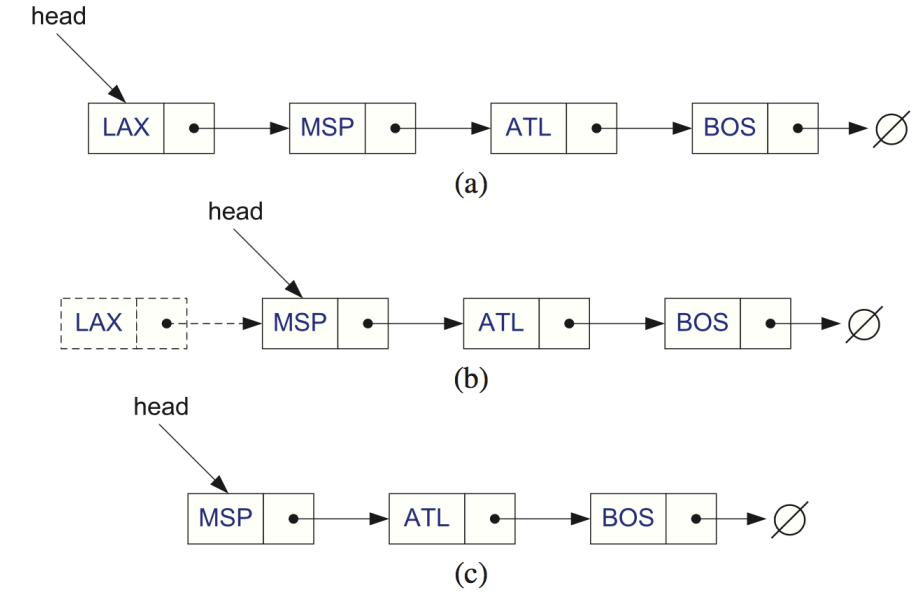


1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node (typically done by calling "delete" in C++)

Let's make codes




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




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

Inserting at the Tail and Removing at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node

Insertion at the tail

1. …
2. …
3. …
4. …

Removal at the tail

”Generic” Singly Linked Lists: Template

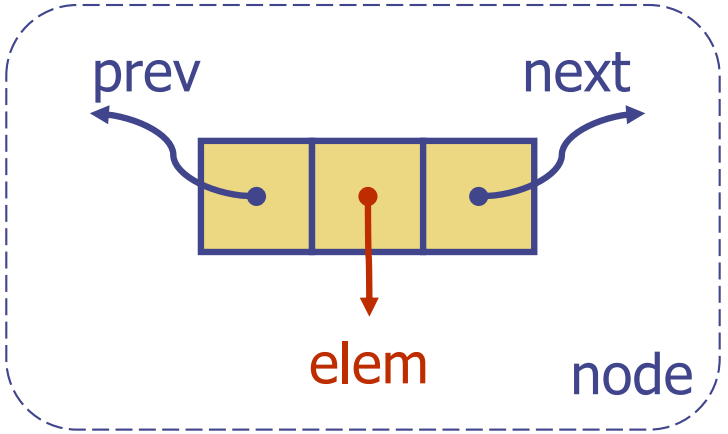
```
template <typename E>
class SNode {
private:
  E elem;
  SNode<E>* next;
  friend class SLinkedList<E>;
};
```

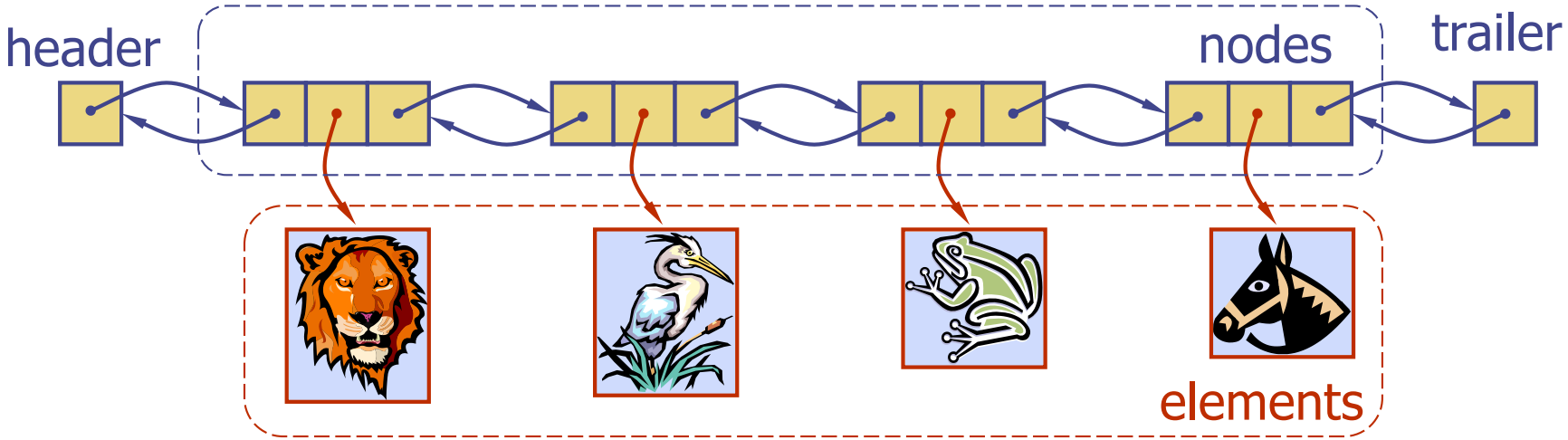
```
template <typename E>
class SLinkedList {
public:
  SLinkedList();
  ~SLinkedList();
  bool empty() const;
  const E& front() const;
  void addFront(const E& e);
  void removeFront();
private:
  SNode<E>* head;
};
```

See the implementation code of member functions in the text (page 122)

Doubly Linked List (§ 3.3)

- Singly Linked List
  - Not easy to remove an elem. at the tail (or any other node)
- Trailer: Dummy sentinel
- Previous link





C++ Implementation: Class Design

```
typedef string Elem;
class DNode {
private:
  Elem elem;
  DNode* prev;
  DNode* next;
  friend class DLinkedList;
};
```

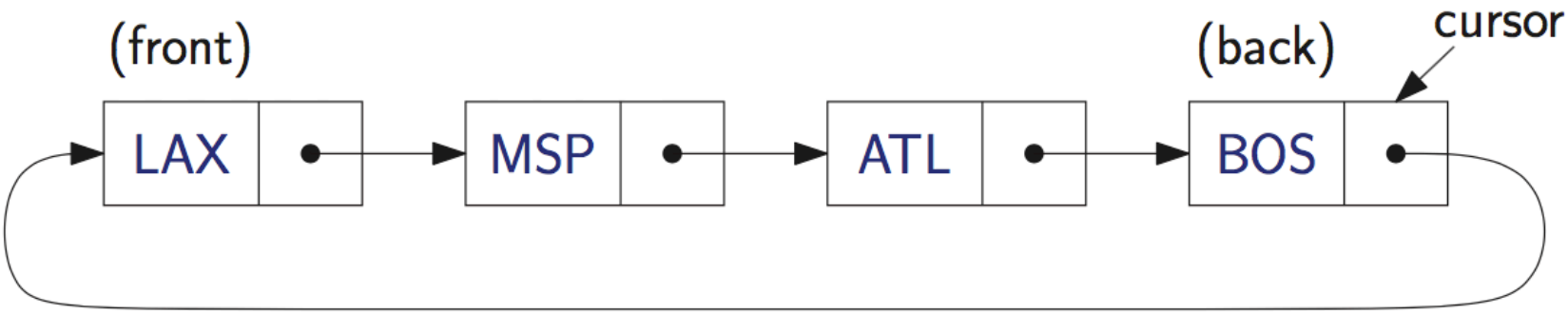
```
class DLinkedList {
public:
  DLinkedList();
  ~DLinkedList();
  bool empty() const;
  const Elem& front() const;
  const Elem& back() const;
  void addFront(const Elem& e);
  void addBack(const Elem& e);
  void removeFront();
  void removeBack();
private:
  DNode* header;
  DNode* trailer;
protected:
  void add(DNode* v, const Elem& e);
  void remove(DNode* v);
};
```

Constructor and Destructor (Don't forget!)

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

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

Circular Linked List (§ 3.3)

- A kind of Singly Linked List
- Rather than having a head or a tail, it forms a cycle
- Cursor
  - A virtual starting node
  - This can be varying as we perform operations



C++ Implementation

```
typedef string Elem;
class CNode {
private:
  Elem elem;
  CNode* next;

  friend class CircleList;
};
```

```
class CircleList {
public:
  CircleList();
  ~CircleList();
  bool empty() const;
  const Elem& front() const;
  const Elem& back() const;
  void advance();
  void add(const Elem& e);
  void remove();
private:
  CNode* cursor;
};
```

What is advance()?

Summary

- Array and Lists
  - A simple data structure to store multiple elements (of the same type)

- Array
- Singly Linked Lists
- Doubly Linked Lists
- Circular Linked Lists

- Key Question
  - For each of the operations, how efficiently does each data structure perform the operation?

Questions?