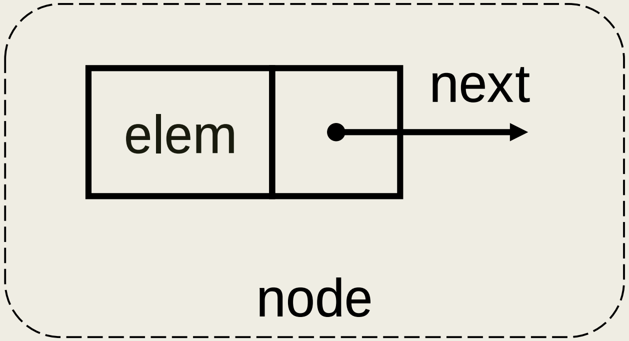
# CH3. LINKED LIST AND RECURSION

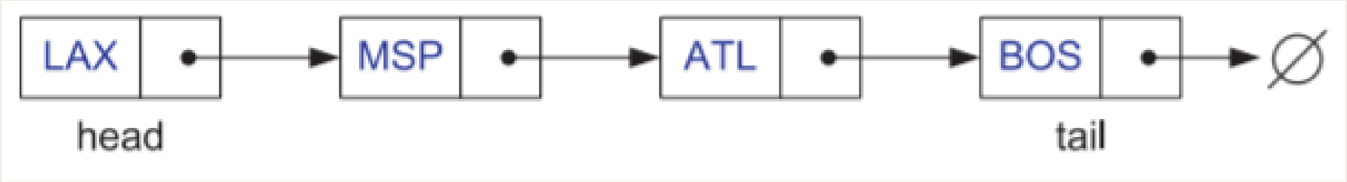
CSED233 Data Structure

Prof. Hwanjo Yu
POSTECH

# 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*



```
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
};
```

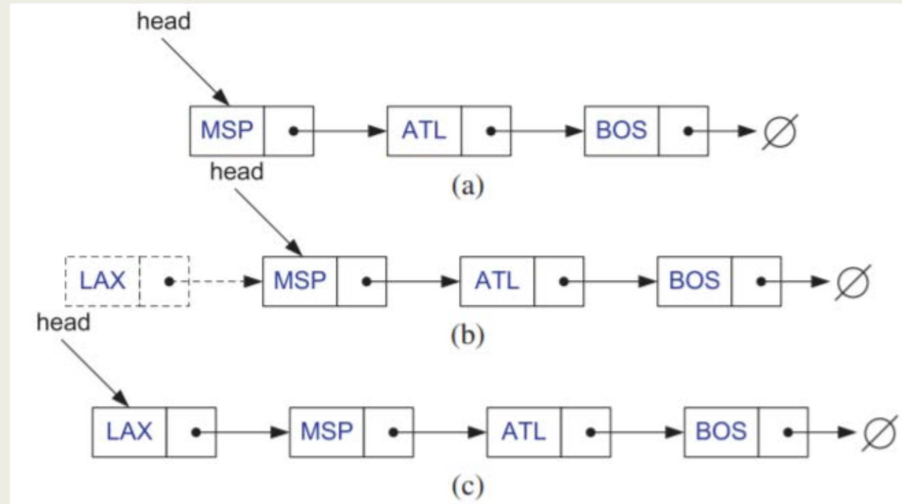
**Code Fragment 3.13:** A node in a singly linked list of strings.

```
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
};
```

**Code Fragment 3.14:** A class definition for a singly linked list of strings.

# Inserting at the Head

1. Create a new node
2. Set its *elem* value
3. Set its *next* link to the current head
4. Update head to point to new node

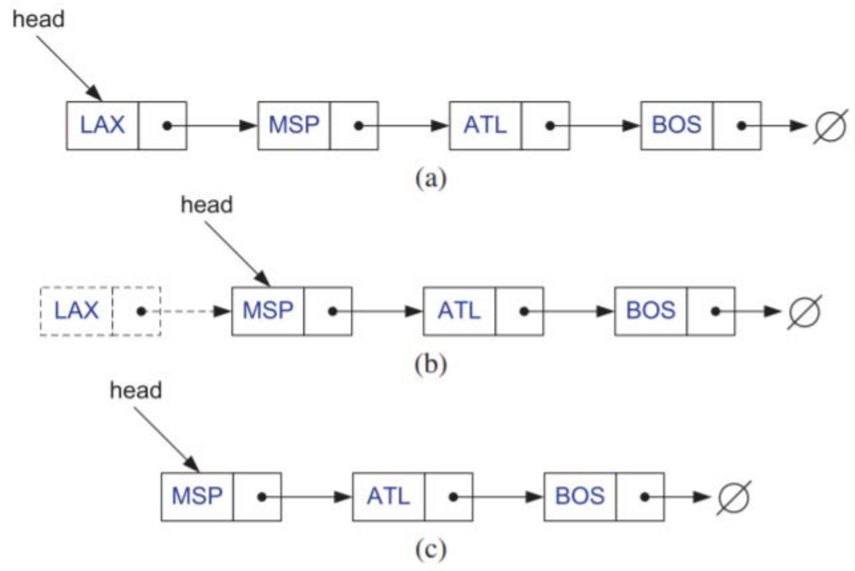



```
void StringLinkedList::addFront(const string& e) {   // add to front of list
  StringNode* v = new StringNode;                    // create new node
  v->elem = e;                                       // store data
  v->next = head;                                    // head now follows v
  head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

**Code Fragment 3.16:** Insertion to the front of a singly linked list.

# Removing at the Head

1. Update head to point to next node in the list
2. Remove the former first node




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

**Code Fragment 3.17:** Removal from the front of a singly linked list.

# Generic Singly Linked List

```
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;
};
```

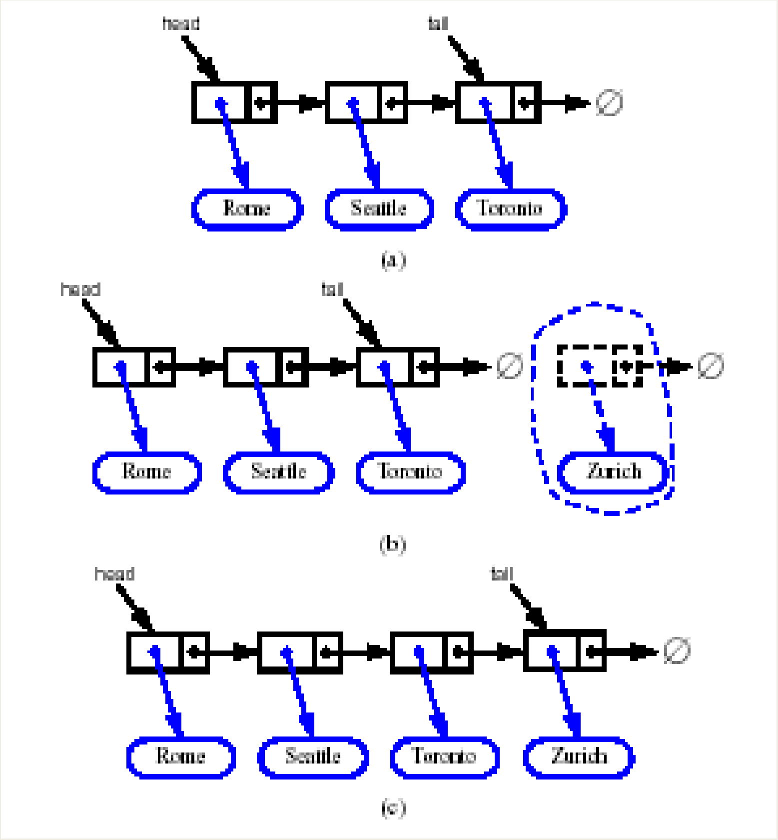
```
template <typename E>
void SLinkedList<E>::addFront(const E& e) {
  SNode<E>* v = new SNode<E>;
  v->elem = e;
  v->next = head;
  head = v;
}

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

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

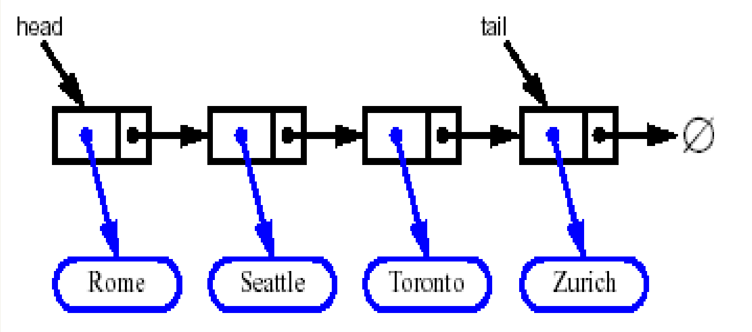
# Inserting at the Tail (need to have "tail")

1. Create a new node
2. Set its elem value
3. Set its next link to NULL
4. Have old last node point to new node
5. Update tail to point to new node



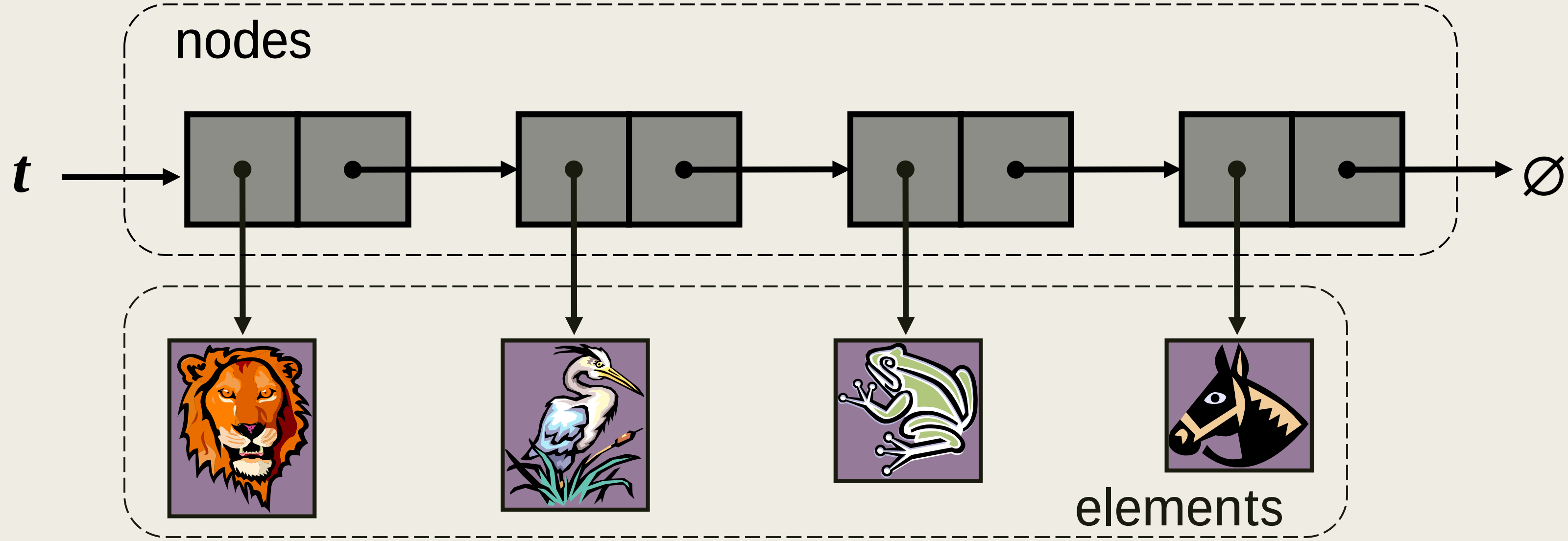
# Removing at the Tail (need to have "tail")

- Removing at the tail of a singly linked list is not efficient! Why?
- There is no constant-time way to update the tail to point to the previous node



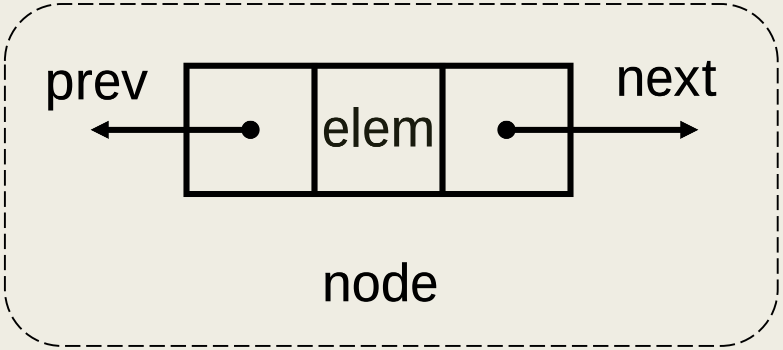
# Stack as a Linked List (§ 5.1.3)

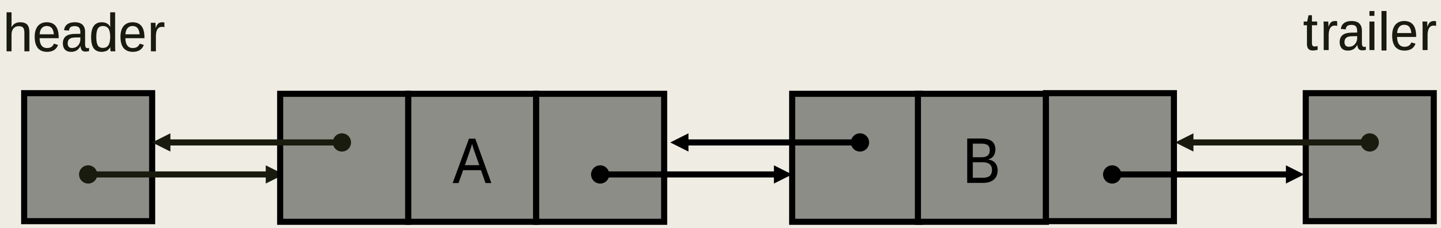
- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $\boldsymbol{O(n)}$ and each operation of the Stack ADT takes $\boldsymbol{O}(1)$ time



# Doubly Linked List (§ 3.3)

- Each node stores
  - *link to the prev node*
  - *element*
  - *link to the next node*



```
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
};
```

**Code Fragment 3.22:** C++ implementation of a doubly linked list node.

```
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
};
```

**Code Fragment 3.23:** Implementation of a doubly linked list class.

# Inserting

```
                                          // 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); }
```

**Code Fragment 3.26:** Inserting a new node into a doubly linked list. The protected utility function add inserts a node $z$ before an arbitrary node $v$. The public member functions addFront and addBack both invoke this utility function.

# Removing

```
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); }
```

**Code Fragment 3.27:** Removing a node from a doubly linked list. The local utility function remove removes the node *v*. The public member functions removeFront and removeBack invoke this utility function.

# Circularly Linked List and List Reversal (§ 3.4)

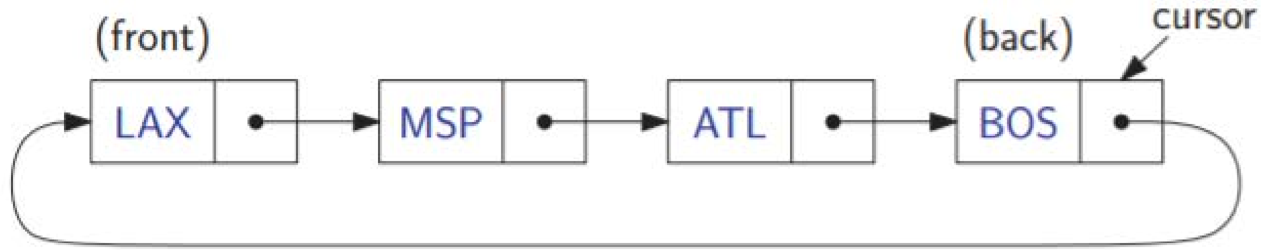



**Figure 3.15:** A circularly linked list. The node referenced by the cursor is called the back, and the node immediately following is called the front.

```
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);
  }
}
```

**Code Fragment 3.35:** A function that reverses the contents of a doubly linked list *L*.

# Recursion (§ 3.5)

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

```
int recursiveFactorial(int n) {              // recursive factorial function
  if (n == 0) return 1;                      // basis case
  else return n * recursiveFactorial(n-1);   // recursive case
}
```

**Code Fragment 3.36:** A recursive implementation of the factorial function.

# Linear Recursion (§ 3.5.1)

**Algorithm** LinearSum$(A, n)$:

*Input:* A integer array $A$ and an integer $n \geq 1$, such that $A$ has at least $n$ elements

*Output:* The sum of the first $n$ integers in $A$

```
if n = 1 then
    return A[0]
else
    return LinearSum(A, n − 1) + A[n − 1]
```

**Code Fragment 3.38:** Summing the elements in an array using linear recursion.

**Algorithm** ReverseArray$(A, i, j)$:

*Input:* An array $A$ and nonnegative integer indices $i$ and $j$

*Output:* The reversal of the elements in $A$ starting at index $i$ and ending at $j$

```
if i < j then
    Swap A[i] and A[j]
    ReverseArray(A, i + 1, j − 1)
return
```

**Code Fragment 3.39:** Reversing the elements of an array using linear recursion.

# Binary Recursion (§ 3.5.2)

- Fibonacci numbers
  - *0, 1, 1, 2, 3, 5, 8, 13, ..*

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_i &= F_{i-1} + F_{i-2} \quad \text{for } i > 1
\end{aligned}
$$

**Algorithm** BinaryFib($k$):
  ***Input:*** Nonnegative integer $k$
  ***Output:*** The $k$th Fibonacci number $F_k$
  **if** $k \leq 1$ **then**
    **return** $k$
  **else**
    **return** BinaryFib($k-1$) + BinaryFib($k-2$)

**Code Fragment 3.42:** Computing the $k$th Fibonacci number using binary recursion.

# Fibonacci execution

- Let $n_k$ be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.
- Note that $n_k$ at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

# A Better Fibonacci Algorithm

- Use linear recursion instead

  ***Algorithm*** *LinearFibonacci(k):*
  
  ***Input:*** *A nonnegative integer k*
  
  ***Output:*** *Pair of Fibonacci numbers* $(F_k, F_{k-1})$
  
  ***if*** *k = 1* ***then***
  
  ***return*** *(k, 0)*
  
  ***Else***
  
  *(i, j) = LinearFibonacci(k – 1)*
  
  ***return*** *(i +j, i)*

- LinearFibonacci makes k–1 recursive calls