



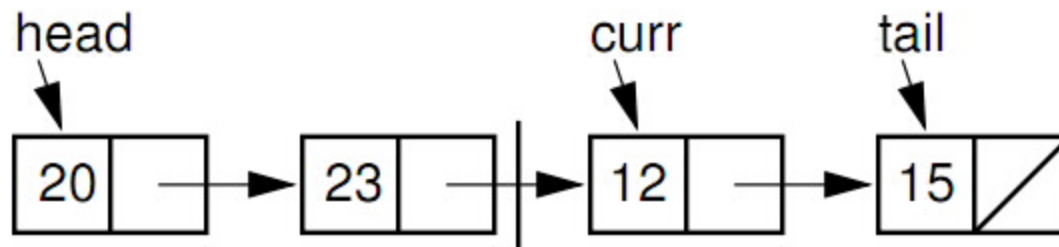
# Unit 4 Singly Linked List

College of Computer Science, CQU

# Introduction

---

- ❑ **Array**  
**successive items locate a fixed distance**
- ❑ **disadvantage**
  - data movements during insertion and deletion
  - waste space in storing n ordered lists of varying size
- ❑ **possible solution**
  - linked list

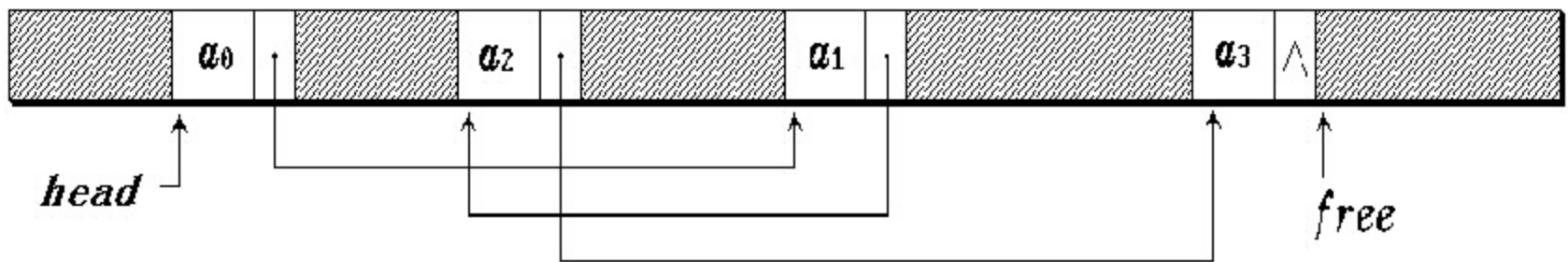
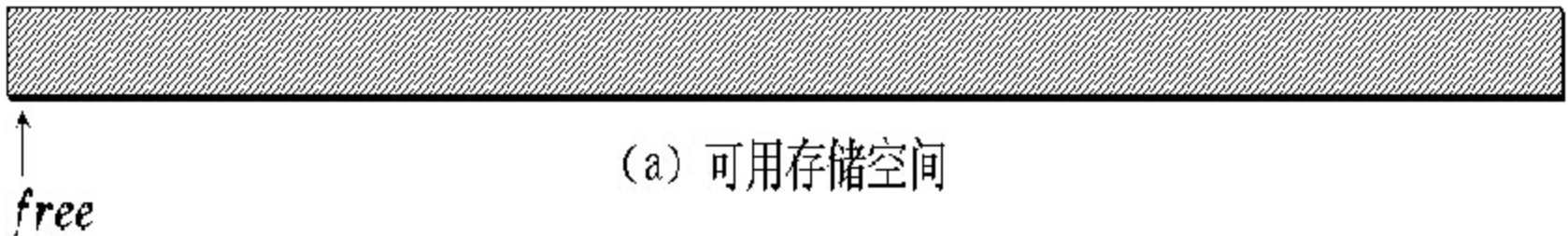


# Linked List

---

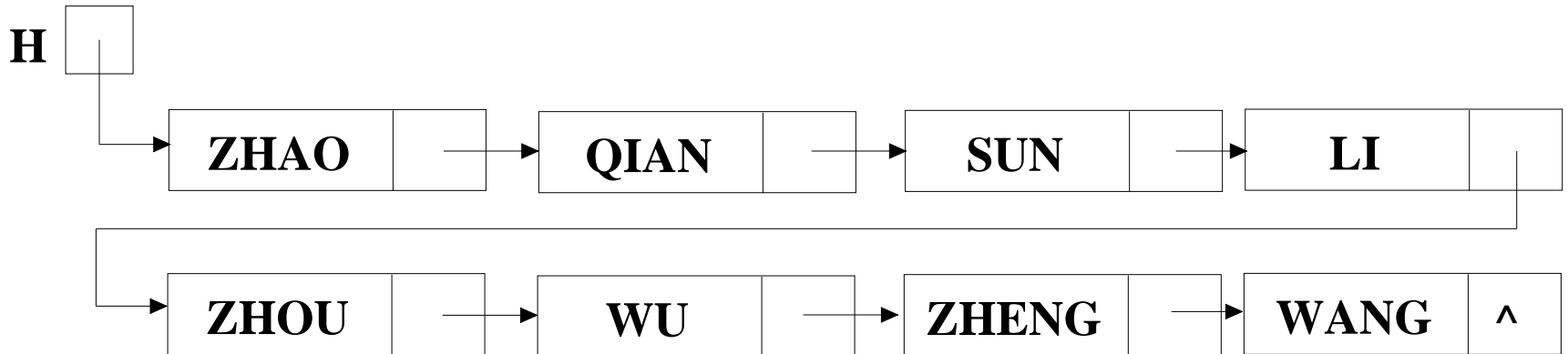
- ❑ A linked list is made up of a series of objects, called the nodes of the list.
- ❑ The linked list uses **dynamic memory allocation**

# Singly linked list storage mapping



# Liear list (ZHAO,QIAN,SUN,LI,ZHOU,WU,ZHENG,WANG)

	adress	data	next
	1	LI	43
	7	QIAN	13
	13	SUN	1
	19	WANG	NULL
	25	WU	37
	31	ZHAO	7
	37	ZHENG	19
	43	ZHOU	25



# Singly linked Link List(one-way list)

---

**// Singly linked list node**

```
template <typename E> class Link {  
public:
```

```
    E element;    // Value for this node
```

```
    Link *next;    // Pointer to next node in list
```

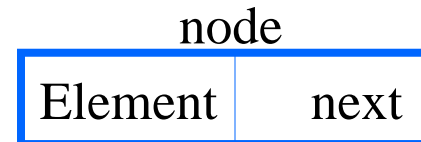
```
    // Constructors
```

```
    Link(const E& elemval, Link* nextval =NULL)
```

```
    { element = elemval; next = nextval; }
```

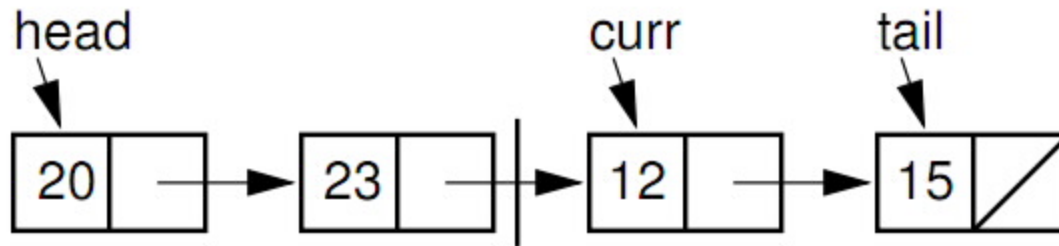
```
    Link(Link* nextval =NULL) { next = nextval; }
```

```
};
```



# Singly linked Link List

---



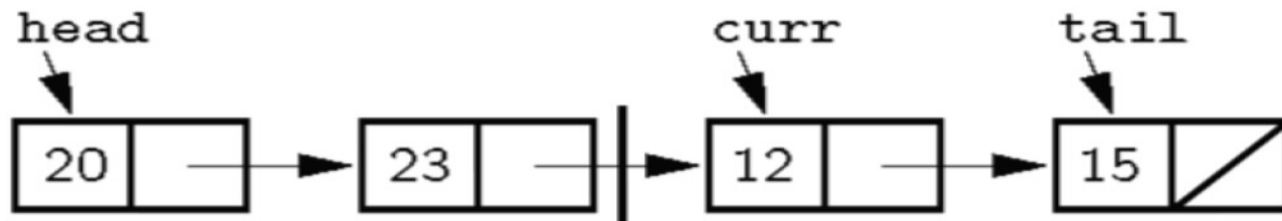
**head:** a pointer point to the list's first node.

**tail:** a pointer is kept to the last link of the list.

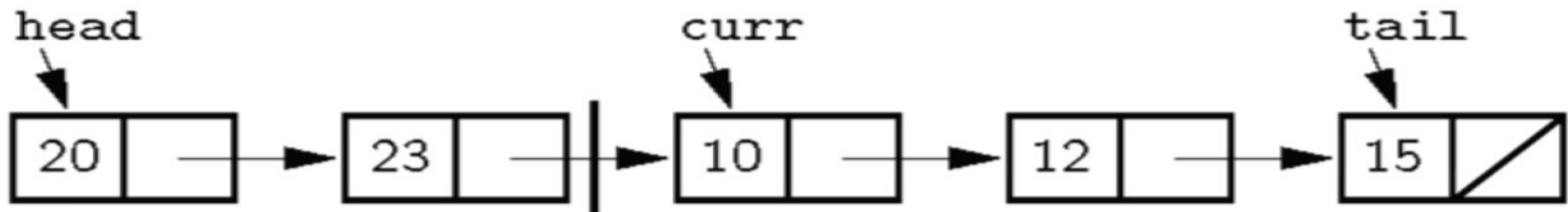
**curr:** a pointer indicate the current element.

**cnt:** the length of the list

# Curr Pointer



(a)

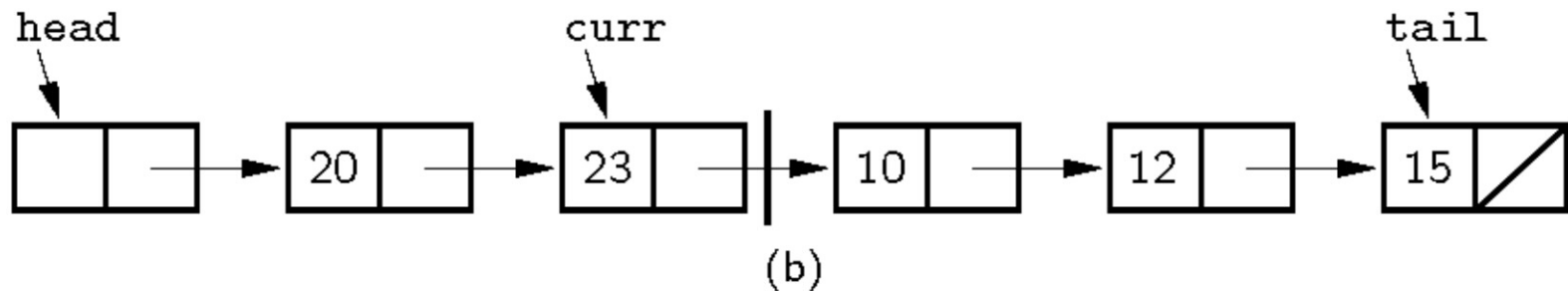
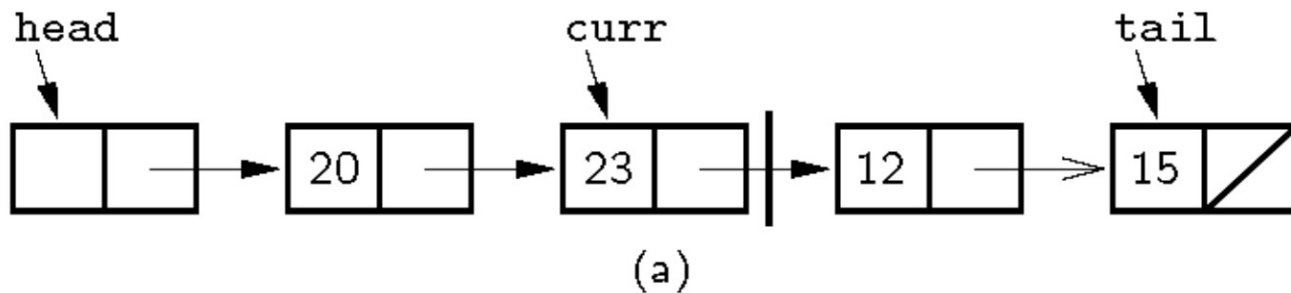


(b)

- ❑ A faulty linked-list implementation where curr points directly to the current node.

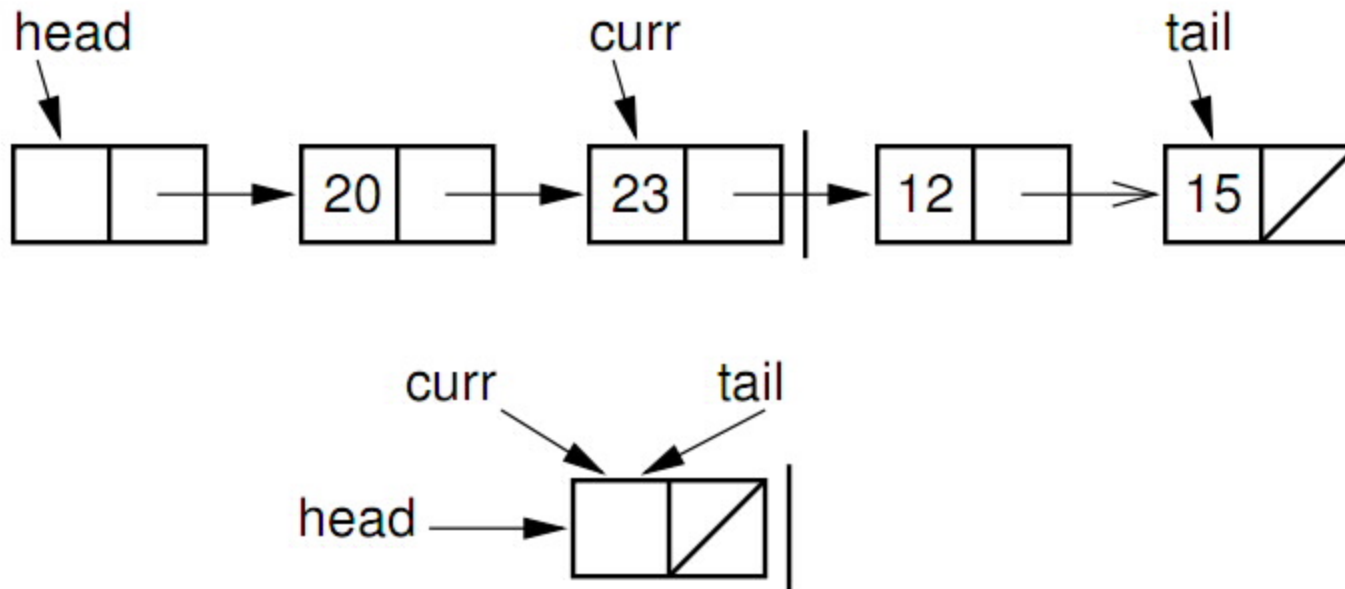


# Curr Pointer



- ▣ **Curr pointing one node head of the current element.**

# Header Node



**header node:** an additional node before the first element node of the list.

The header node saves coding effort because we no longer need to consider special cases for empty lists or when the current position is at one end of the list.

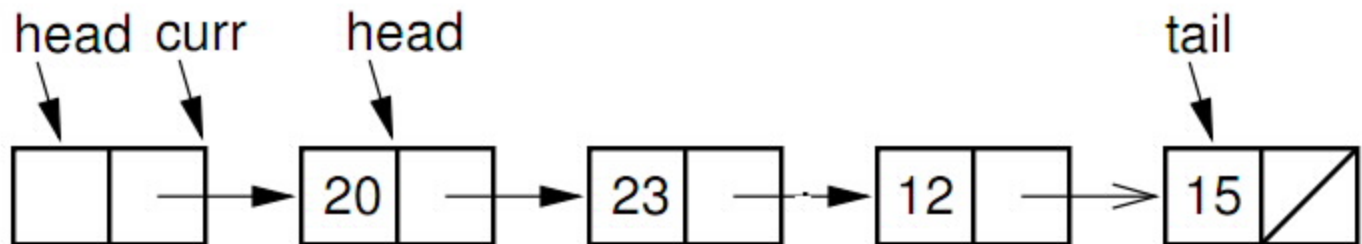
# Linked List Class (1)

---

- ❑ `template <typename E> class LList: public List<E> {`
- ❑ `private:`
- ❑ `Link<E>* head;       // Pointer to list header`
- ❑ `Link<E>* tail;       // Pointer to last element`
- ❑ `Link<E>* curr;       // Access to current element`
- ❑ `int cnt;               // Size of list`
- ❑ `void init() {       // Initialization helper method`
- ❑ `curr = tail = head = new Link<E>;`
- ❑ `cnt = 0; }`

# Linked List Class (2)

- ❑ `void removeall() { // Return link nodes to free store`
- ❑ `while(head != NULL) {`
- ❑ `curr = head;`
- ❑ `head = head->next;`
- ❑ `delete curr;`
- ❑ `}`
- ❑ `}`



# Linked List Class (3)

---

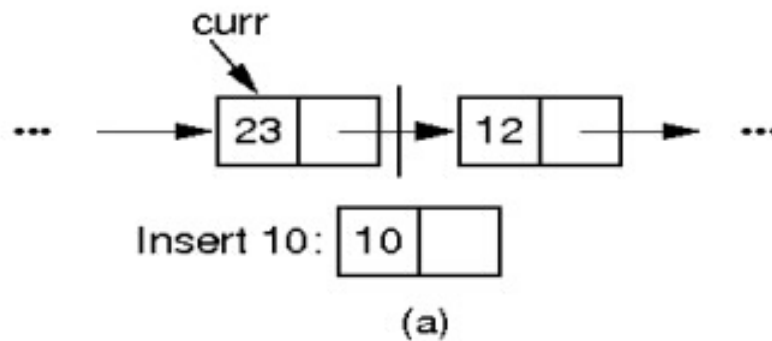
- public:
- `LList(int size=defaultSize) { init(); }` // Constructor
- `~LList() { removeall(); }` // Destructor
- `void print() const;` // Print list contents
- `void clear() { removeall(); init(); }` // Clear list

# Insertion

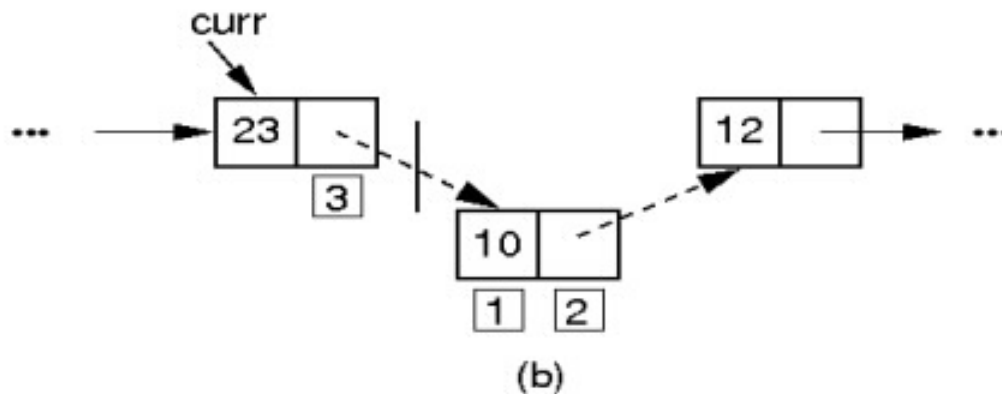
---

- ❑ Inserting a new element is **a three-step process**:
- ❑ **First**, the new list node is created and the new element is stored into it.
- ❑ **Second**, the next field of the new list node is assigned to point to the current node (the one after the node that curr points to).
- ❑ **Third**, the next field of node pointed to by curr is assigned to point to the newly inserted node.

# Insertion



$\text{curr} \rightarrow \text{next} = \text{new}$   
 $\text{Link}\langle E \rangle(\text{it}, \text{curr} \rightarrow \text{next});$



□ **The linked list insertion process.**

# Insertion

---

- `// Insert "it" at current position`
- `void insert(const E& it) {`
- `curr->next = new Link<E>(it, curr->next);`
- `if (tail == curr) tail = curr->next; // New tail`
- `cnt++;`
- `}`

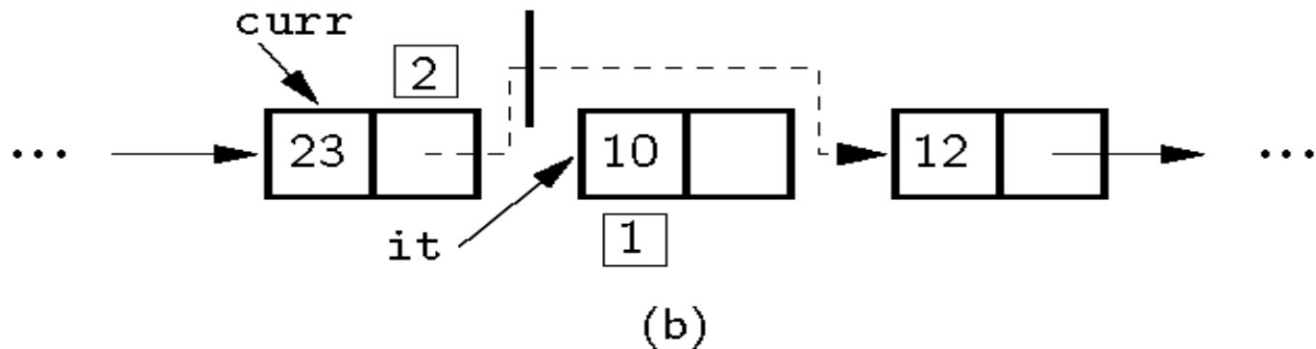
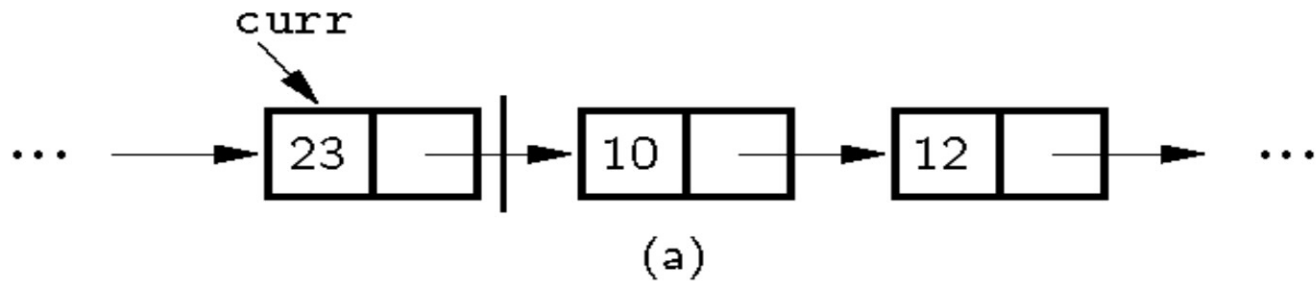


# Append

---

- `void append(const E& it) { // Append "it" to list`
- `tail = tail->next = new Link<E>(it, NULL);`
- `cnt++;`
- `}`

# Removal



▣ The linked list removal process.

# Remove

---

- ❑ `// Remove and return current element`
- ❑ `E remove() {`
- ❑ `Assert(curr->next != NULL, "No element");`
- ❑ `E it = curr->next->element;     // Remember value`
- ❑ `Link<E>* ltemp = curr->next;     // Remember link node`
- ❑ `if (tail == curr->next) tail = curr; // Reset tail`
- ❑ `curr->next = curr->next->next;     // Remove from list`
- ❑ `delete ltemp;                     // Reclaim space`
- ❑ `cnt--;                            // Decrement the count`
- ❑ `return it;`



# MoveToStart & MoveToEnd

---

- ❑ `void moveToStart()` // Place curr at list start
- ❑ `{ curr = head; }`
  
- ❑ `void moveToEnd()` // Place curr at list end
- ❑ `{ curr = tail; }`

# Prev

---

- ❑ `// Move curr one step left; no change if already at front`
- ❑ `void prev() {`
- ❑  `if (curr == head) return; // No previous element`
- ❑  `Link<E>* temp = head;`
- ❑  `// March down list until we find the previous element`
- ❑  `while (temp->next!=curr) temp=temp->next;`
- ❑  `curr = temp;`
- ❑ `}`

# Next / Length

---

- `// Move curr one step right; no change if already at end`
- `void next()`
- `{ if (curr != tail) curr = curr->next; }`
- `int length() const { return cnt; } // Return length`

# Get/Set Position

---

- ❑ `// Return the position of the current element`
- ❑ `int currPos() const {`
- ❑ `Link<E>* temp = head;`
- ❑ `int i;`
- ❑ `for (i=0; curr != temp; i++)`
- ❑ `temp = temp->next;`
- ❑ `return i;`
- ❑ `}`

# Get/Set Position

---

- `// Move down list to "pos" position`
- `void moveToPos(int pos) {`
- `Assert ((pos>=0)&&(pos<=cnt), "Position out of range");`
- `curr = head;`
- `for(int i=0; i<pos; i++) curr = curr->next;`
- `}`



# GetValue

---

- ❑ `const E& getValue() const { // Return current element`
- ❑ `Assert(curr->next != NULL, "No value");`
- ❑ `return curr->next->element;`
- ❑ `}`

# Comparison of Implementations

---

## Array-Based Lists:

- ❑ Insertion and deletion are  $O(n)$ .
- ❑ Prev and direct access are  $O(1)$ .
- ❑ Array must be allocated in advance.
- ❑ No overhead (结构性开销) if all array positions are full.

## Linked Lists:

- ❑ Insertion and deletion are  $O(1)$ .
- ❑ Prev and direct access are  $O(n)$ .
- ❑ Space grows with number of elements.
- ❑ Every element requires overhead.



# Space Comparison

---

***Array-Based Lists***

***DE***

**vs.**

***Linked Lists***

***$n(P + E)$***

***E***: Space for data value.

***P***: Space for pointer.

***D***: Number of elements in array.

***n*** : the number of elements currently in the list

**“Break-even” point:**

***$DE = n(P + E);$***

***$n = \frac{DE}{P + E}$***

# Space Example

---

- ❑ **Array-based list: Overhead is one pointer (4 bytes) per position in array – whether used or not.**
- ❑ **Linked list: Overhead is two pointers per link node**
  - one to the element, one to the next link
- ❑ **Data is the same for both.**
- ❑ **When is the space the same?**
  - When the array is half full

# Exercise

---

- **Write a function to merge two sorted linked lists.** The input lists have their elements in sorted order, from lowest to highest. The output list should also be sorted from lowest to highest. Your algorithm should run in linear time on the length of the output list.

# Freelists

---

- ❑ **The C++ free-store management operators `new` and `delete` are relatively expensive to use. System `new` and garbage collection are slow.**
- ❑ **Instead of making repeated calls to `new` and `delete`, the `Link` class can handle its own freelist.**
- ❑ **A freelist holds those list nodes that are not currently being used.**

# Freelists

---

- ❑ **When a node is deleted from a linked list, it is placed at the head of the freelist.**
- ❑ **When a new element is to be added to a linked list, the freelist is checked to see if a list node is available. If so, the node is taken from the freelist. If the freelist is empty, the standard new operator must then be called.**

# Approach to implement freelists

---

- ❑ **One approach would be to create two new operators to use instead of the standard free-store routines new and delete.**
- ❑ **This requires that the user's code, such as the linked list class implementation of Figure 4.8, be modified to call these freelist operators.**



# Approach to implement freelists

---

- ❑ **A second approach is to use C++ operator overloading to replace the meaning of new and delete when operating on Link class objects.**
- ❑ **In this way, programs that use the LList class need not be modified at all to take advantage of a freelist. Whether the Link class is implemented with freelists, or relies on the regular free-store mechanism, is entirely hidden from the list class user.**

# Link Class Extensions

---

- ❑ `// Singly linked list node with freelist support`
- ❑ `template <typename E> class Link {`
- ❑ `private:`
- ❑ `static Link<E>* freelist; // Reference to freelist head`
- ❑ `public:`
- ❑ `E element; // Value for this node`
- ❑ `Link* next; // Point to next node in list`



# Implementation for the Link class with a freelist

---

- **// Constructors**
- `Link(const E& elemval, Link* nextval =NULL)`
- `{ element = elemval; next = nextval; }`
- `Link(Link* nextval =NULL) { next = nextval; }`

# Implementation for the Link class with a freelist

---

- ❑ `void* operator new(size_t) { // Overloaded new operator`
- ❑ `if (freelist == NULL) return ::new Link; // Create space`
- ❑ `Link<E>* temp = freelist; // Can take from freelist`
- ❑ `freelist = freelist->next;`
- ❑ `return temp; // Return the link`
- ❑ `}`

# Implementation for the Link class with a freelist

---

- ❑ `// Overloaded delete operator`
- ❑ `void operator delete(void* ptr) {`
- ❑ `((Link<E>*)ptr)->next = freelist; // Put on freelist`
- ❑ `freelist = (Link<E>*)ptr;`
- ❑ `}`
- ❑ `};`

# Implementation for the Link class with a freelist

---

- `// The freelist head pointer is actually created here`
- `template <typename E>`
- `Link<E>* Link<E>::freelist = NULL;`

# Reference

---

▣ **P103-----P112**



---

-End-

