《数据结构与算法》课程组

重庆大学计算机学院

Data Structures & Algorithms

5

LINEAR LIST

Outline

5.1 Definition of ADT

5.2 Array-based List (Sequential List)

5.3 Singly Linked List

5.4 Freelist

5.5 Circularly Linked List

5.6 Doubly Linked List

5.7 Polynomial manipulation

5.1 Definitions of List ADT

Lists

Sorted list

<1, 3, 5, 6, 8, 9, 21, 24, 56, 77>

<98, 65, 43, 23, 11, 10, 9, 6, 5, 4, 2>

Unsorted list

<1, 6, 3, 9, 34, 30, 19, 8, 12, 44>

Terminology

- Length of list

  The number of elements currently stored is called the length of the list.

- Empty list

  A list is said to be empty when it contains no elements. the empty list would appear as <>.

- Order

  Order of a element is it's position in the list.

List Implementation Concepts

- Our list implementation will support the concept of a current position.

- Operations will act relative to the current position.

- Example: <20, 23 | 12, 15> to indicate the list of four elements, with the current position being to the right of the bar at element 12.

What operations should we implement?

List ADT

```
template <typename E> class List { // List ADT
private:
  void operator =(const List&) {}       // Protect assignment
  List(const List&) {}           // Protect copy constructor
public:
  List() {}          // Default constructor
  virtual ~List() {} // Base destructor

  // Clear contents from the list, to make it empty.
  virtual void clear() = 0;

  // Insert an element at the current location.
  // item: The element to be inserted
  virtual void insert(const E& item) = 0;
```

•

List ADT

```
// Append an element at the end of the list.
  // item: The element to be appended.
  virtual void append(const E& item) = 0;

  // Remove and return the current element.
  // Return: the element that was removed.
  virtual E remove() = 0;

  // Set the current position to the start of the list
  virtual void moveToStart() = 0;

  // Set the current position to the end of the list
  virtual void moveToEnd() = 0;
```

List ADT

```
// Move the current position one step left. No change  if already
at beginning.
  virtual void prev() = 0;
  // Move the current position one step right. No change  if
already at end.
  virtual void next() = 0;
  // Return: The number of elements in the list.
  virtual int length() const = 0;
  // Return: The position of the current element.
  virtual int currPos() const = 0;

  // Set current position. pos: The position to make current.
  virtual void moveToPos(int pos) = 0;
  // Return: The current element.
  virtual const E& getValue() const = 0;
};
```

List ADT Examples

List: <12 | 32, 15>

```
L.insert(99);
```

Result: <12 | 99, 32, 15>

Iterate through the whole list:

```
for (L.moveToStart(); L.currPos()<L.length(); L.next())
{
    it = L.getValue();
    doSomething(it);
}
```

List Find Function

```
//return True if k is in list L,
//false otherwise
bool find(List<int>& L, int k) {
    int it;
    for (L.moveToStart();  L.currPos()<L.length(); L.next())
    {
        it = L.getValue();
        if (k == it) return true;
     }
    return false;          // k not found
}
```

Physical Implementation

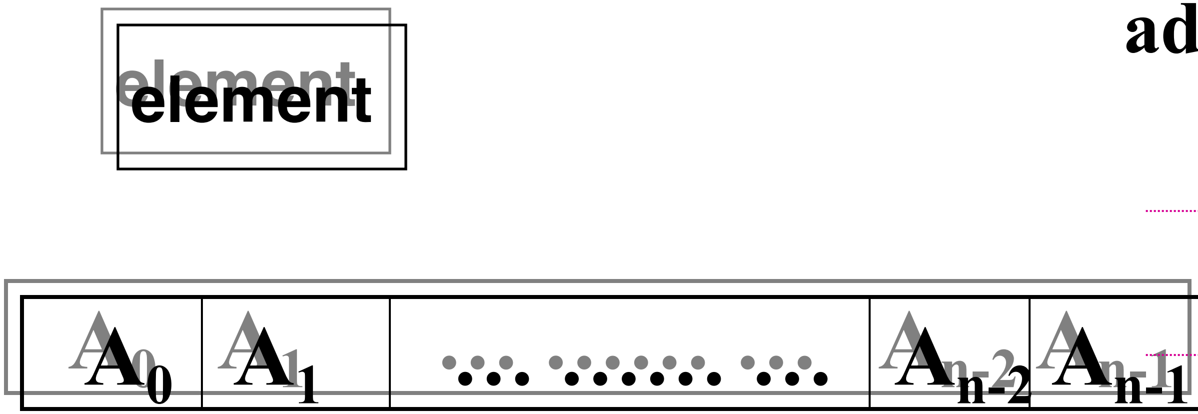
list physical implementation

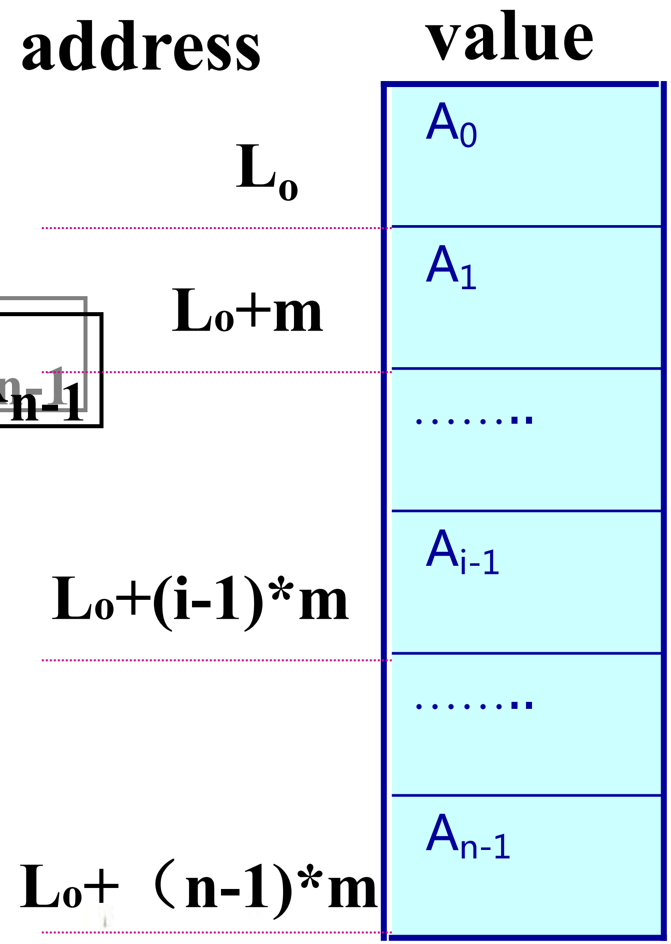
array-based list

linked list

5.2 Array-based List (Sequential List)

Array-Based List Implementation



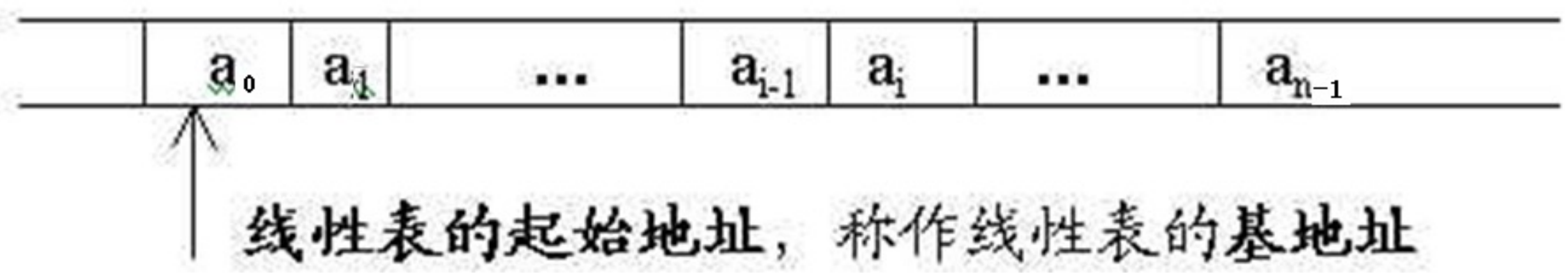


$Loc(a)=L_o+i*m$

Given some position in the list

Array-Based List Implementation

Array_Based List : The elements are stored in a consecutive storage area one by one



Notes ：

- With ordered pair $<a_{i-1}$ ， $a_i>$ to express "Storage is adjacent to" ， $loc(a_i) = loc(a_{i-1}) + c$
- Unnecessary to store logic relationship
- First data component location can decide all data elements locations

Array-Based List Class (1)

```
#include "list.h“
template <typename E> // Array-based list implementation
class AList : public List<E> {
private:
  int maxSize;        // Maximum size of list
  int listSize;       // Number of list items now
  int curr;           // Position of current element
  E* listArray;       // Array holding list elements
```

Array-Based List Class (2)

```
public:
  AList(int size=defaultSize)
  {   // Constructor
      maxSize = size;
      listSize = curr = 0;
      listArray = new E[maxSize];
  }

  ~AList() { delete [] listArray; }   // Destructor
```

Array-Based List Class (3)

```
void clear() {                         // Reinitialize the list
    delete [] listArray;               // Remove the array
    listSize = curr = 0;               // Reset the size
    listArray = new E[maxSize];  // Recreate array
 }
void moveToStart() { curr = 0; }
void moveToEnd() { curr = listSize; }
void prev() { if (curr != 0) curr--; }
void next()      { if (curr < listSize) curr++; }
```

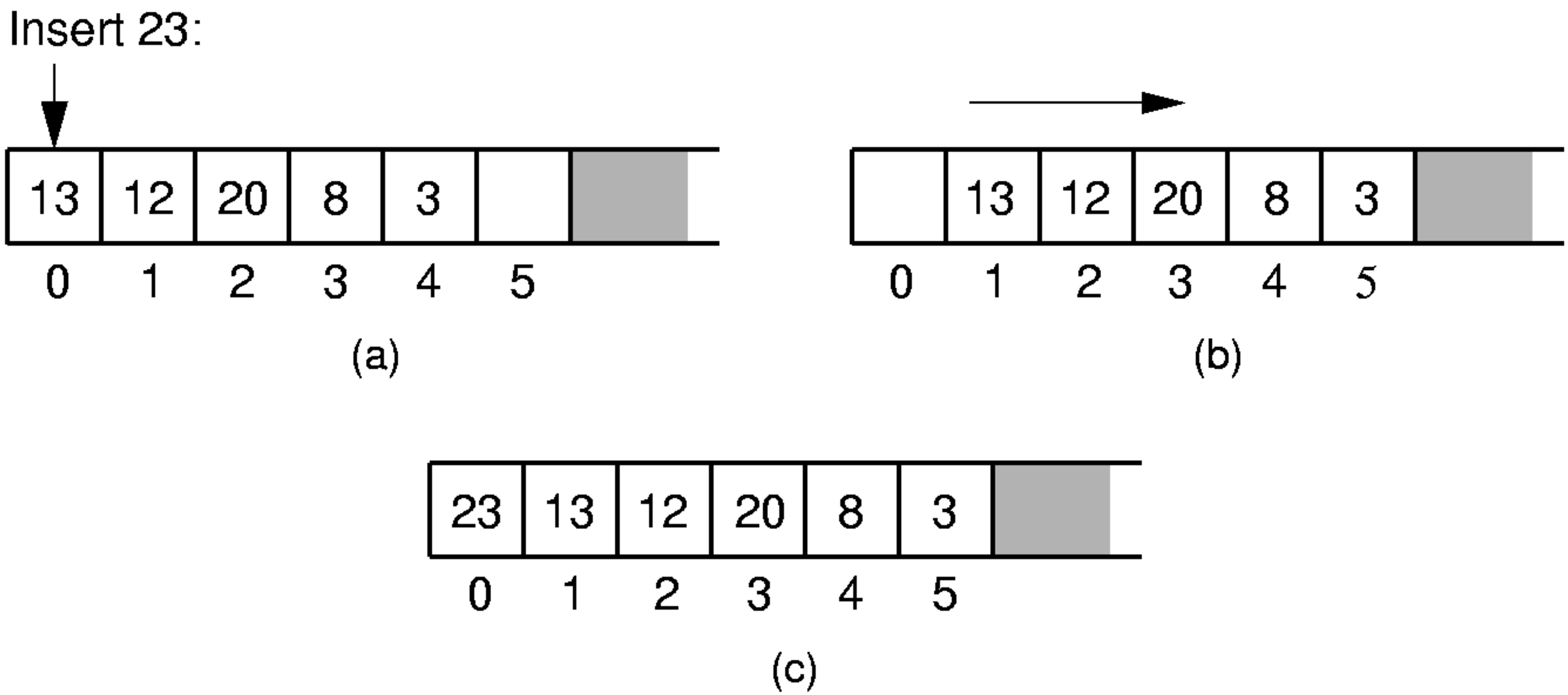
Array-Based List Class (4)

```
// Return list size
  int length() const  { return listSize; }
        // Return current position
  int currPos() const { return curr; }
        // Set current list position to "pos"
  void moveToPos(int pos) {
     Assert ((pos>=0)&&(pos<=listSize), "Pos out of
              range");
     curr = pos;
  }
```

Array-Based List Class (5)

```
// Return current element

const E& getValue() const
{
    Assert((curr>=0)&&(curr<listSize),"No current
            element");
    return listArray[curr];
}
```

Array-Based List Insert



Insert

```
// Insert "it" at current position
  void insert(const E& it) {
    Assert(listSize < maxSize, "List capacity
              exceeded");
    for(int i=listSize; i>curr; i--)  // Shift elements up
      listArray[i] = listArray[i-1];  //   to make room
    listArray[curr] = it;
    listSize++;                        // Increment list size
  }
```

Append

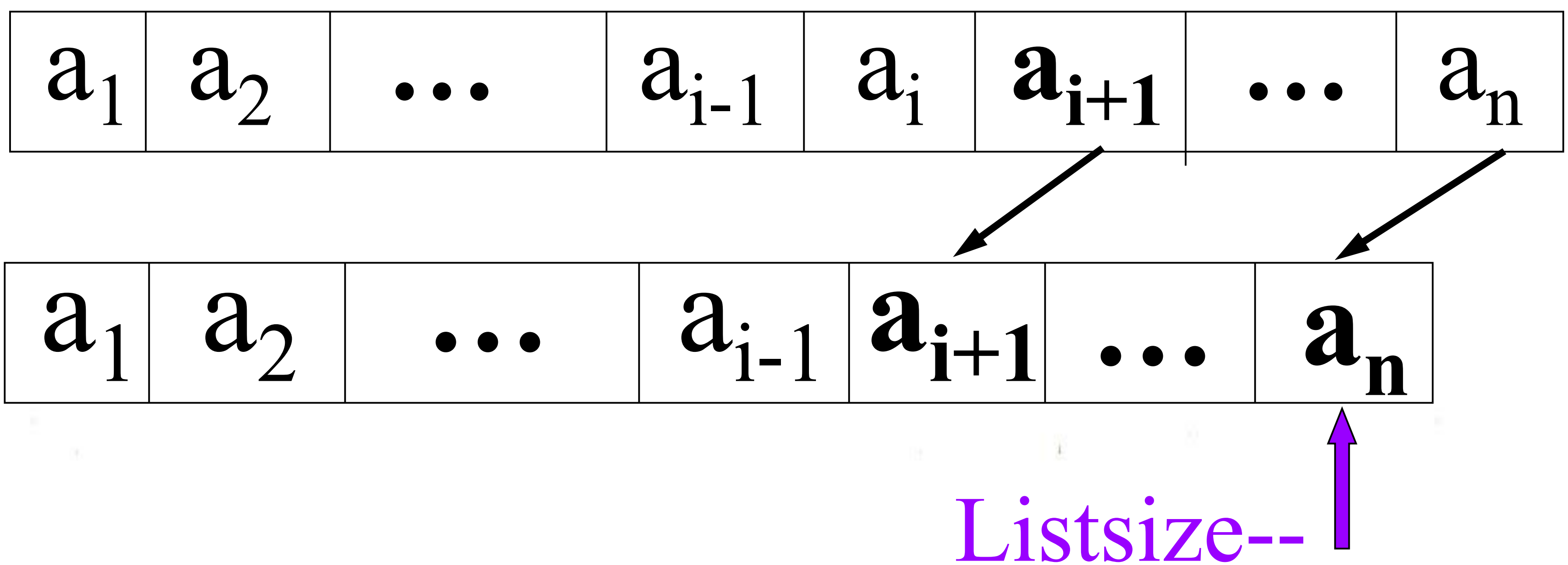
```
void append(const E& it) {      // Append "it"
    Assert(listSize < maxSize, "List capacity
                exceeded");
    listArray[listSize++] = it;
}
```

Remove

$<a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n>$ **change to**

$<a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n>$

$<a_{i-1}, a_i>, <a_i, a_{i+1}> \Rightarrow <a_{i-1}, a_{i+1}>$



Remove

```
// Remove and return the current element.
 E remove() {
    Assert((curr>=0) && (curr < listSize), "No element");
    E it = listArray[curr];          // Copy the element
    for(int i=curr; i<listSize-1; i++)  // Shift them down
        listArray[i] = listArray[i+1];
    listSize--;                      // Decrement size
    return it;
 }
```

Exercise：

Design an algorithm to reverse an sequential list $(a_1 a_2 \ldots\ldots a_{n-1} a_n) \rightarrow (a_n a_{n-1} \ldots\ldots a_2 a_1)$

Summing Up

- Advantages ：
  - Stores a collection of items contiguously.
    - Stores no relations
    - Access randomly
- Disadvantages ：
  - Need to shift many elements in the array whenever there is an insertion or deletion.
  - Need to allocate a fix amount of memory in advance.

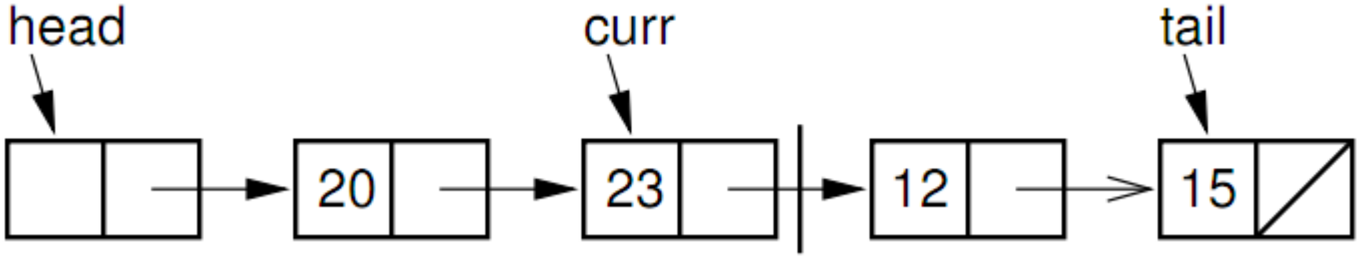
5.3 Singly Linked List

Introduction

- **Array**

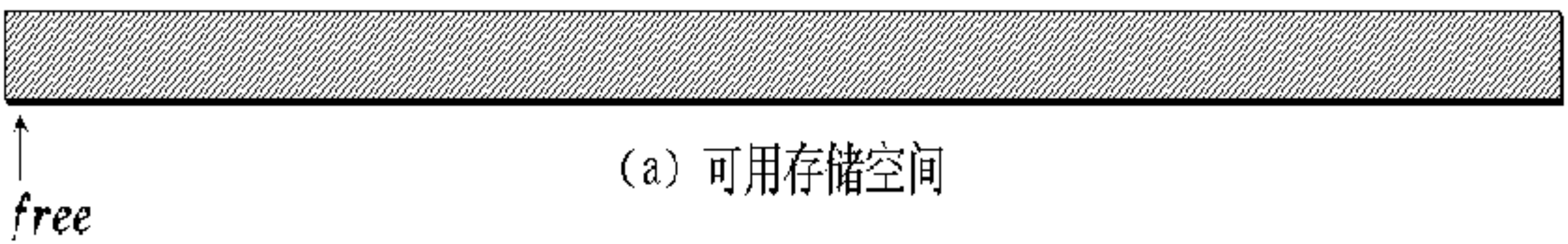
  successive items locate a fixed distance

- **disadvantage**
  - **data movements during insertion and deletion**
  - **waste space in storing n ordered lists of varying size**
- **possible solution**
  - **linked list**

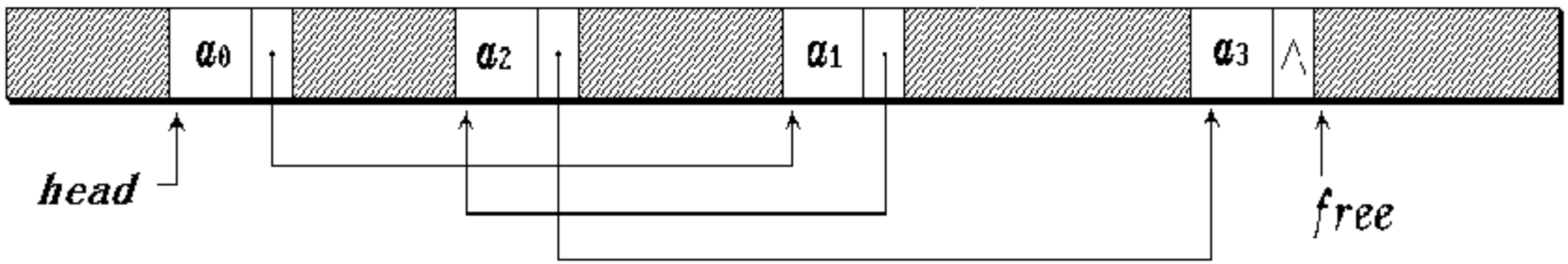


Linked List

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

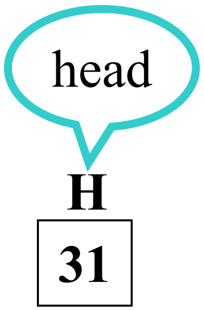


(a) 可用存储空间

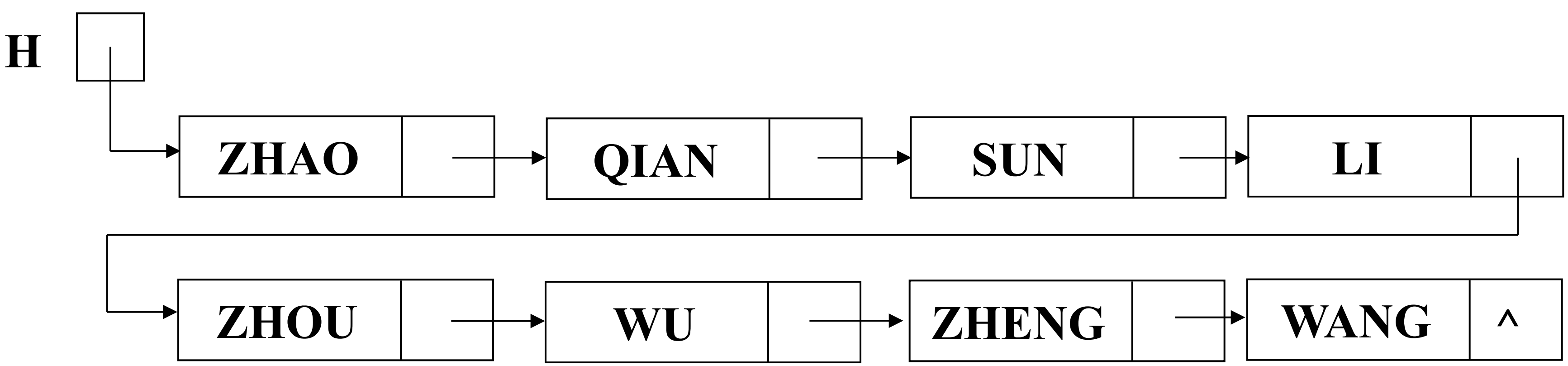


(b) 经过一段运行后的单链表结构

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

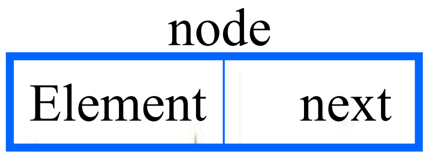


address	data	next
1	LI	43
7	QIAN	13
13	SUN	1
19	WANG	NULL
25	WU	37
31	ZHAO	7
37	ZHENG	19
43	ZHOU	25

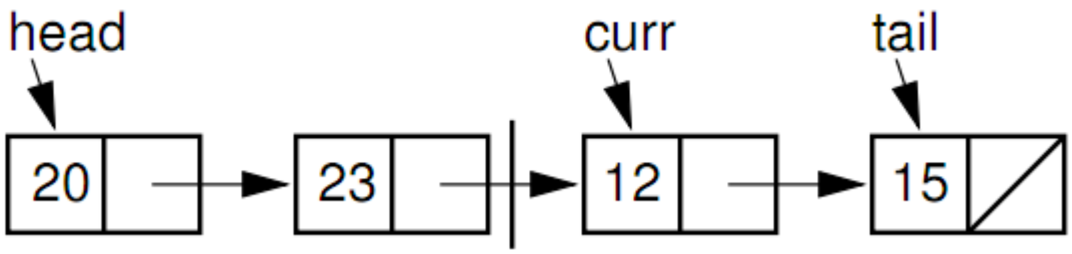


Singly Linked List (one-way list)

```
// 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 List (one-way list)



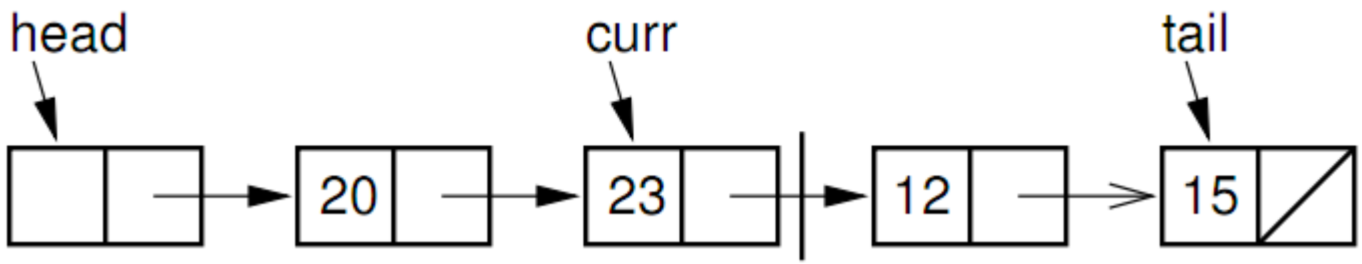
head: a pointer point to the list's first node.

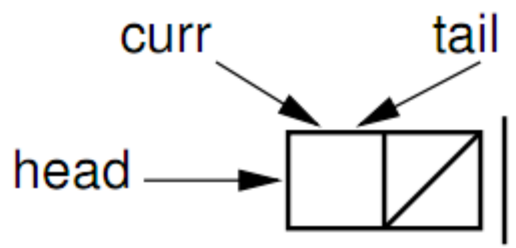
tail: a pointer is kept to the last link of the list.

curr: a pointer indicate the current element.

cnt: the length of the list

Singly Linked List (one-way list)





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

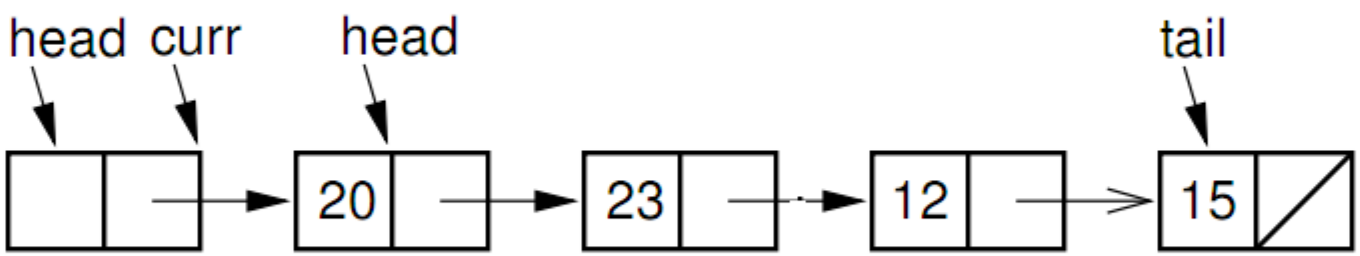
The header node saves coding effort because we no longer need to consider special cases for empty lists or when the current position is at one end of the list.

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() {        // Intialization helper method
    curr = tail = head = new Link<E>;
    cnt = 0;
}
```

Linked List Class (2)

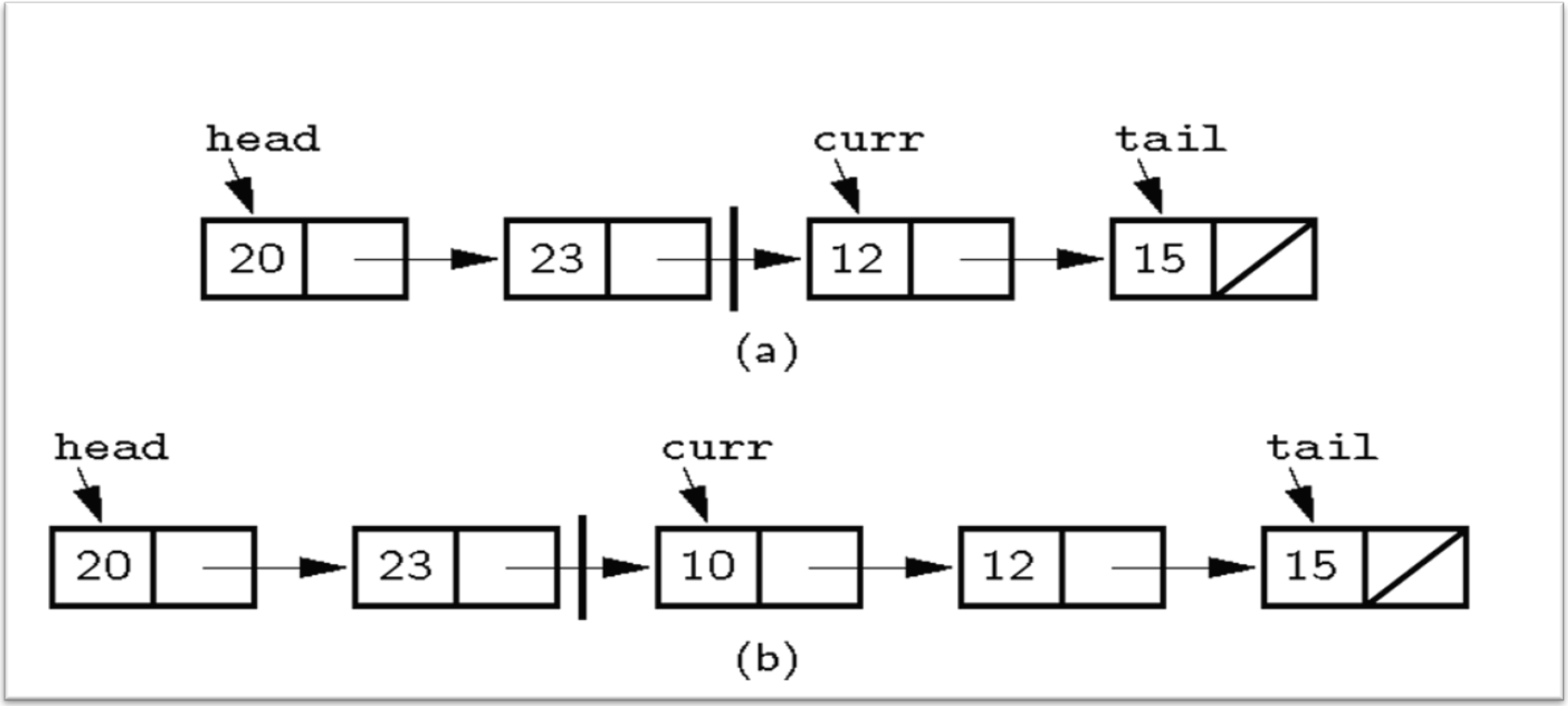
```
void removeall() {
            // Return link nodes to free store
    while(head != NULL) {
        curr = head;
        head = head->next;
        delete curr;
    }
}
```



Linked List Class (3)

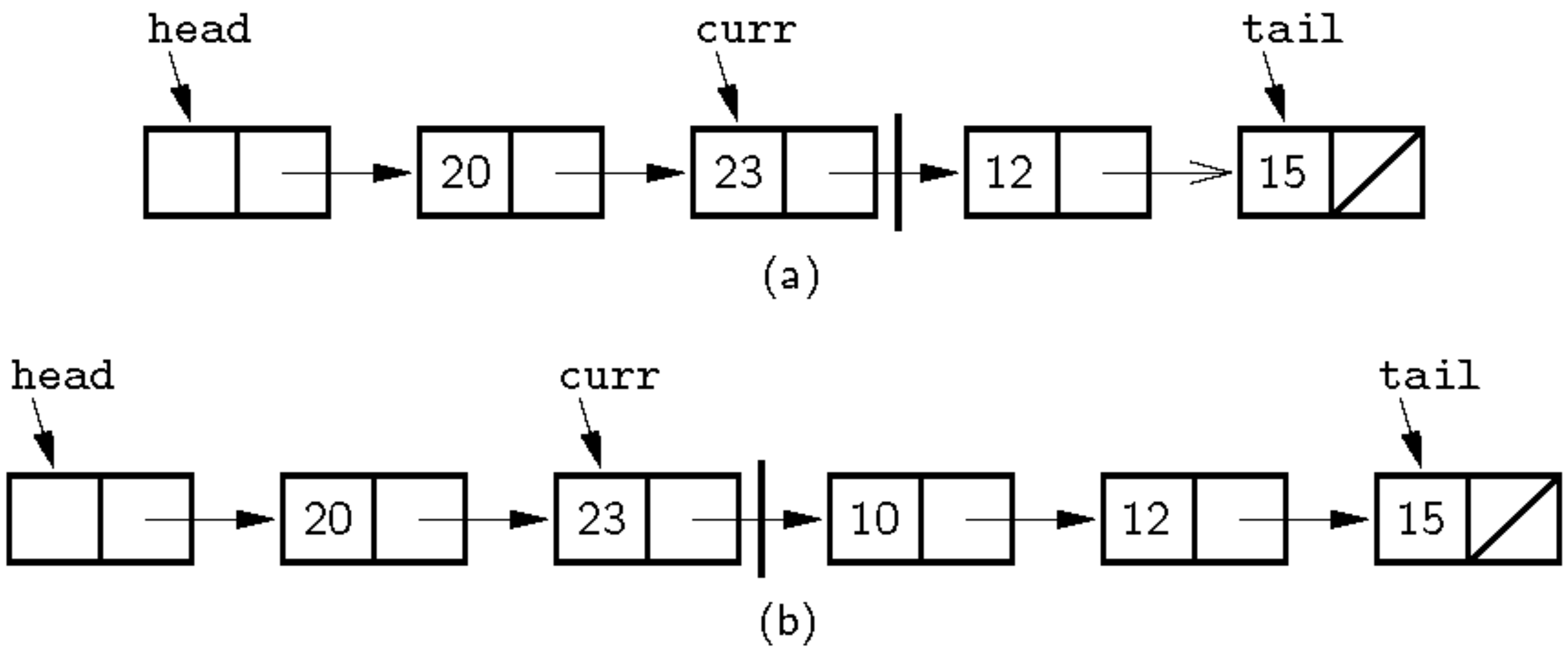
```
public:
  LList(int size=defaultSize) { init(); }    // Constructor
  ~LList() { removeall(); }                  // Destructor
  void print() const;                        // Print list contents
  void clear() { removeall(); init(); }      // Clear list
```

Insertion



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

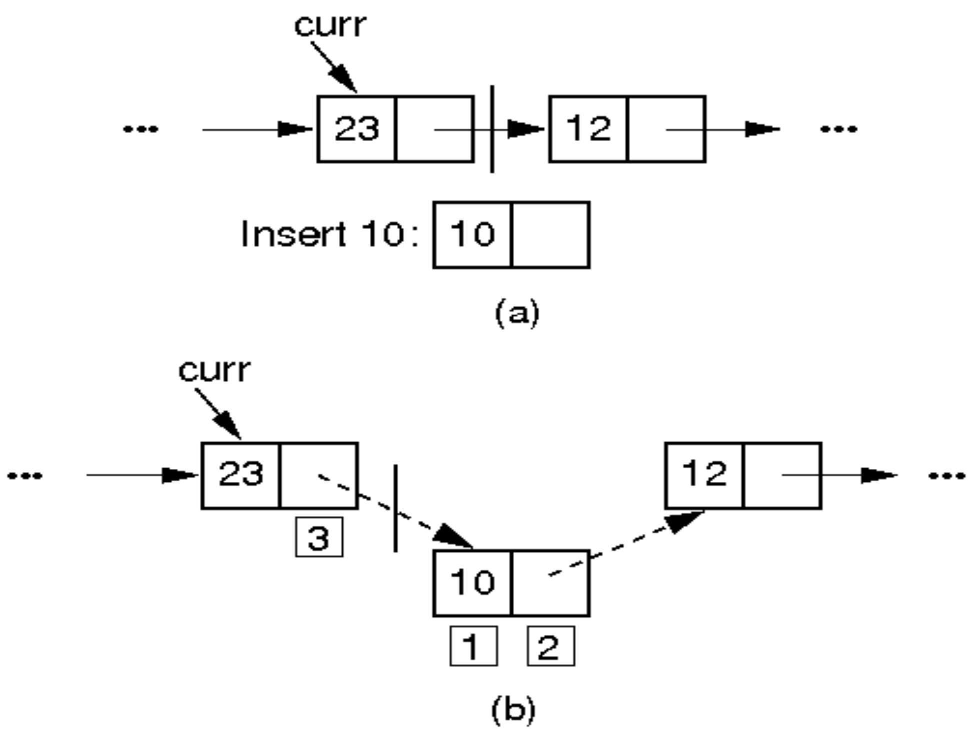
Insertion



- **Insertion using a header node, with curr pointing one node head of the current element.**

Insertion

- Inserting a new element is a three-step process:



① The new list node is created and the new element is stored into it.
② 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).
③ The next field of node pointed to by curr is assigned to point to the newly inserted node.

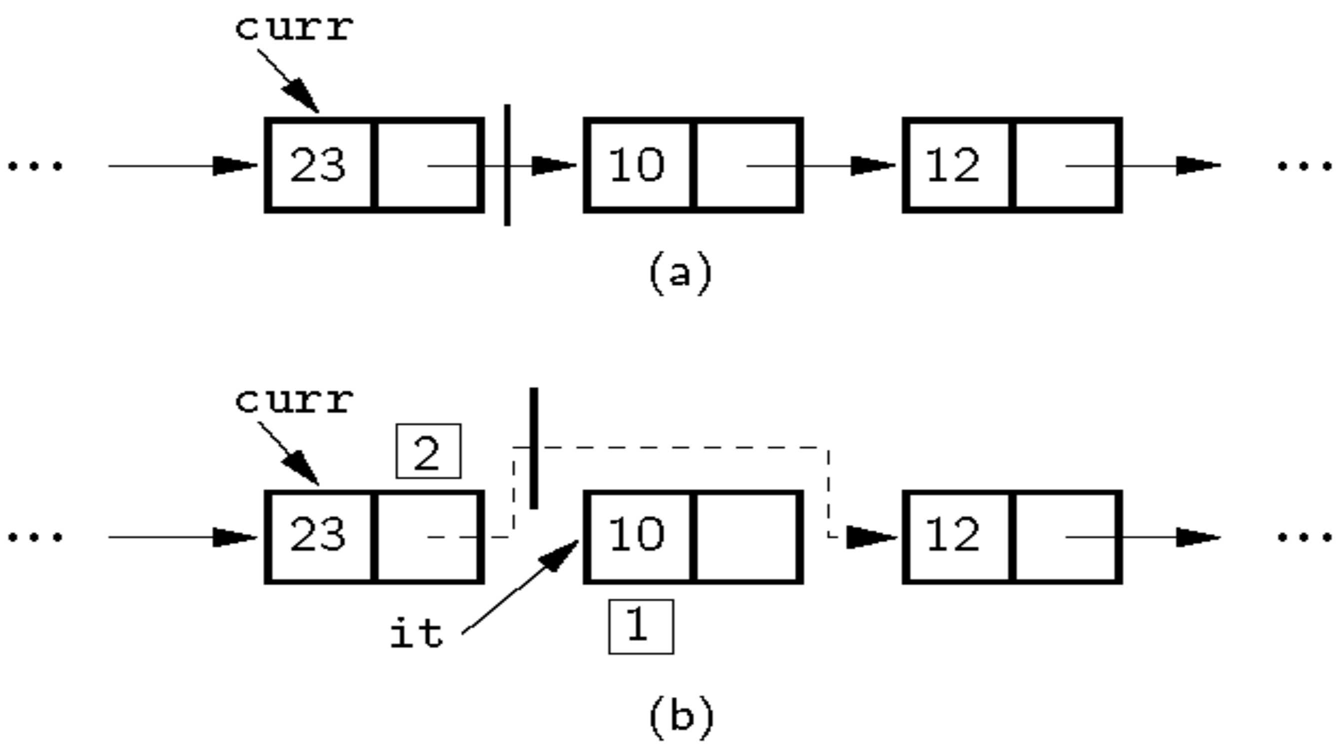
Insertion

```
// 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.

Removal

```
// 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 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

"Break-even" point D:

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

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

E: Space for each data value.
P: Space for each pointer.
n: Size of array.

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.

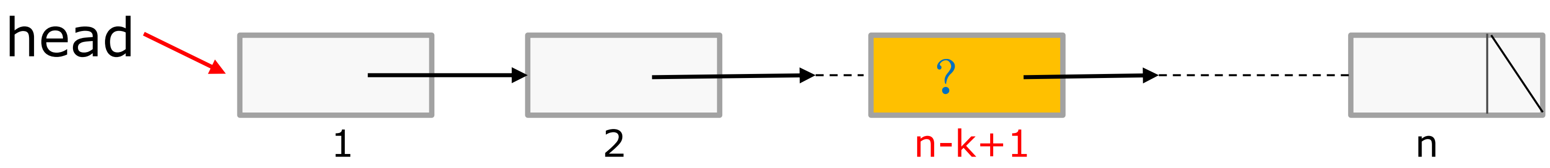
Exercise

```
void merge(LList<int> *p1,    LList<int> *p2)
{     p1->moveToStart();
      p2->moveToStart();
      while(p2->length() > 0)  {
                  // removing each node from p2 and inserting into p1
           if(  p1->currPos() == p1->length() ||   // p1->curr == p1->tail
                p1->getValue() >= p2->getValue() )
            {
                  p1->insert(p2->remove());
            }
            p1->next();
         }
}
```

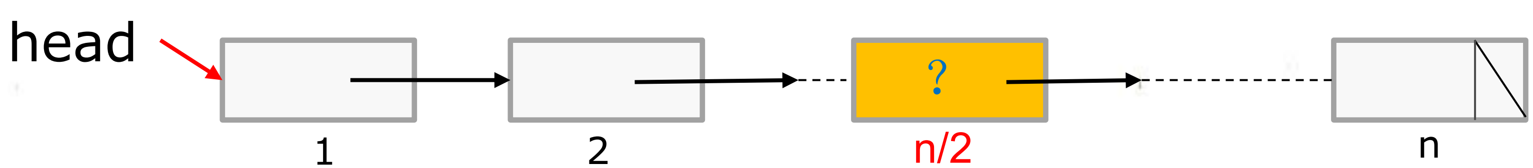
- Time complexity: O(n)
- Space complexity: O(1)

经典面试题（双指针应用）

- 长度为n的单链表const Link<E>* head, 输出其倒数第k个节点的值，时间复杂度O(n)



- 长度为n的单链表const Link<E>* head, 输出其中间节点的值，时间复杂度O(n)



5.4 Freelist

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

- A freelist holds those list nodes that are not currently being used.
- 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 head of 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 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 of Link class with freelist

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

Implementation of Link class with 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 of Link class with freelist

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

Implementation of Link class with freelist

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

5.5 Circularly Linked List

Circularly Linked Lists

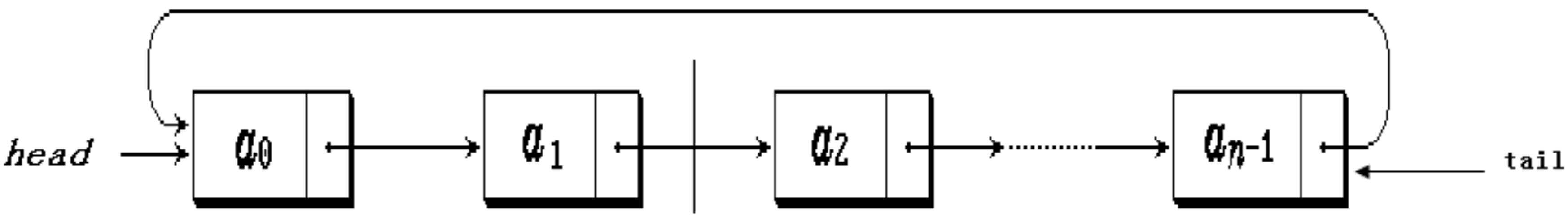
- **Singly Linked Lists**

  the last node contain a NULL pointer
- **Circularly Linked Lists**
  - **the last node contains a <u>pointer to the first node</u>**
- **Advantage**

  start from any node, can access the others.

Two structures of Circularly Linked Lists

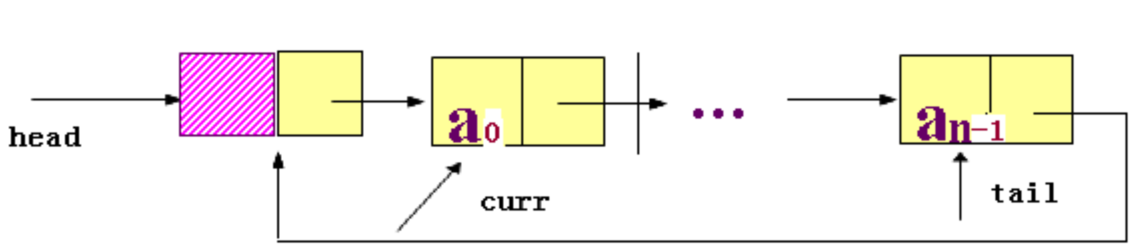
- without head node

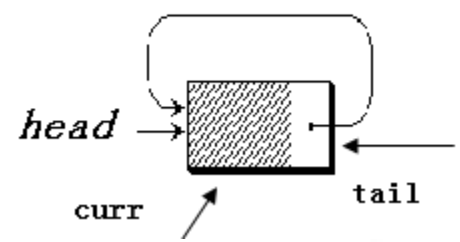


head =tail =curr = NIL

Empty list

- with head node





Empty list
Empty list

Application： Josehus problem

- A description of the problem are: number 1,2, ..., n of n individuals sitting around a circle clockwise, each holding a password (positive integer). Choose a positive integer beginning as a limit on the number of reported m, starting from the first person to start a clockwise direction from a report number, report the number of reported m stop. Who reported m out of line, his password as the new m value, in a clockwise direction from the next person he began to re-reported from a number, it goes on until all the people all of the columns so far. Design a program, according to the column order prints each number.

5.6 Doubly Linked Lists

Doubly Linked Lists

- **Singly Linked Lists**

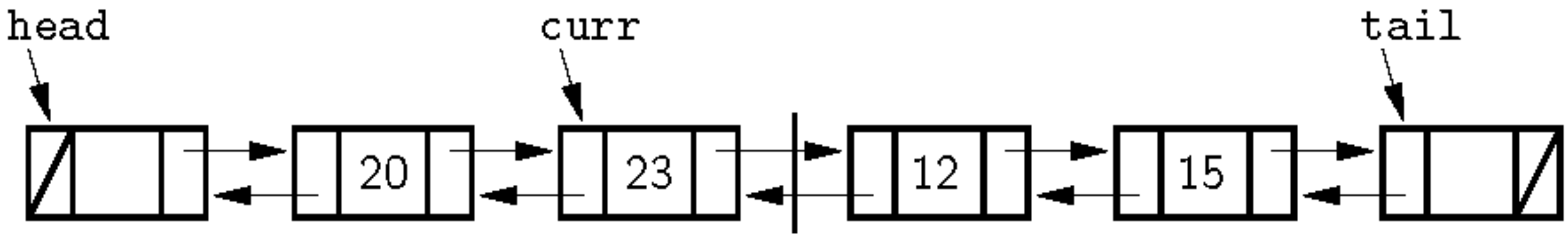
  The singly linked list allows for direct access from a list node only to the next node in the list.

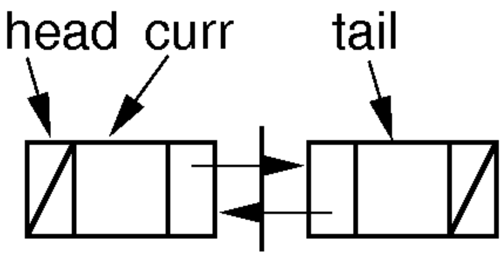
- **Doubly Linked Lists**
  - **A doubly linked list allows convenient access from a list node to the next node and also to the preceding node on the list.**

- **How to accomplish?**

The doubly linked list node accomplishes this in the obvious way by storing two pointers: one to the node following it (as in the singly linked list), and a second pointer to the node preceding it.

Doubly Linked Lists





empty list

Doubly Linked Lists

```
// Constructors
Link(const E& it, Link* prevp, Link* nextp) {
    element = it;
    prev = prevp;
    next = nextp;
}
Link(Link* prevp =NULL, Link* nextp =NULL) {
    prev = prevp;
    next = nextp;
}
```

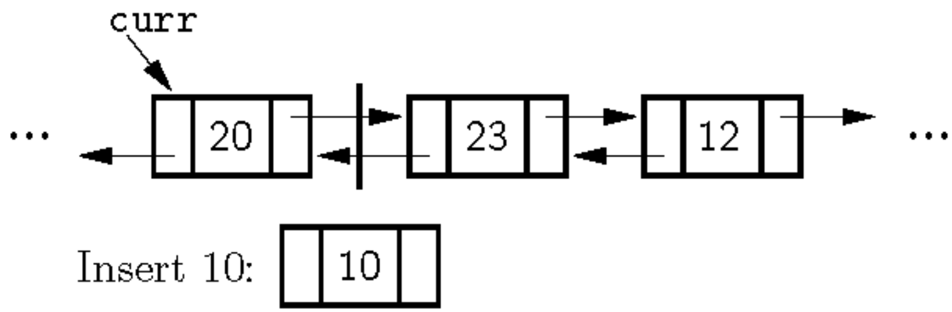
Doubly Linked Lists

```
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
}
```

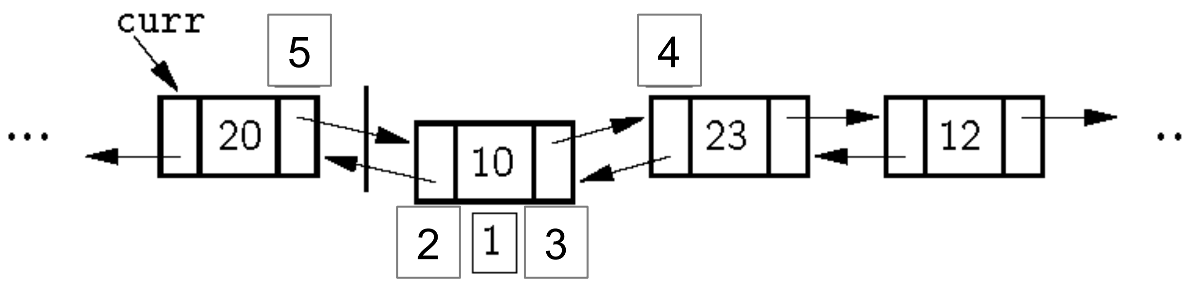
Doubly Linked Lists

```
  // Overloaded delete operator
  void operator delete(void* ptr) {
      ((Link<E>*)ptr)->next = freelist;  // Put on freelist
      freelist = (Link<E>*)ptr;
  }
};
// The freelist head pointer is actually created here
template <typename E>
Link<E>* Link<E>::freelist = NULL;
```

Doubly Linked Insert



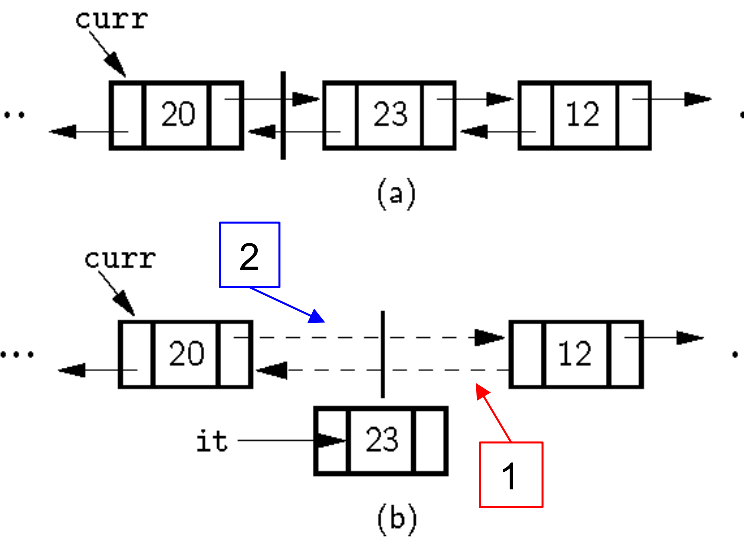
(a)



(b)

```
void insert(const E& it) {  // Insert "it" at current position
    curr->next = curr->next->prev = new Link<E>(it, curr, curr->next);

    cnt++;
}
```

Doubly Linked Remove




```
// Remove and return current element
E remove() {
    if (curr->next == tail)
        return NULL;
    E it = curr->next->element;
    Link<E>* ltemp = curr->next;
1   curr->next->next->prev = curr;
2   curr->next = curr->next->next;
                  // Remove from list
    delete ltemp;
    cnt--;
    return it;
}
```

Doubly Linked Append & Prev

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

// Move fence one step left; no change if left is empty
void prev() {
    if (curr != head)  // Can't back up from list head
        curr = curr->prev;
}
```

Doubly Linked List disadvantage

- The only disadvantage of the doubly linked list as compared to the singly linked list is the additional space used.

5.7 Polynomial Manipulation

Application: polynomial

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$
$$= \sum_{i=0}^{n} a_i x^i$$

Expressing the polynomial

- Express the linear list :

$$P = (p0, p1, \dots, \quad pn)$$

- It is also unsuitable to express the form like

$$S (X) = 1 + 3x^{10000}$$

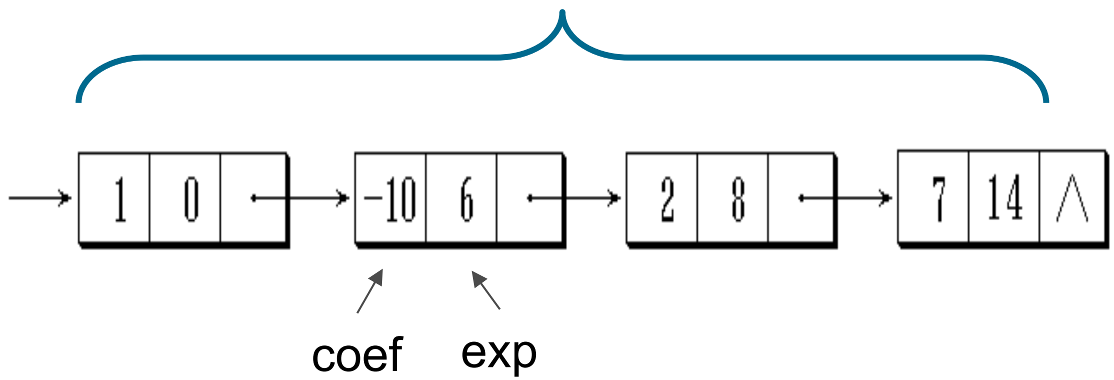
- Writing factor and index number

$$(\ (p1, e1)\ , (p2, e2), \cdots, (pm,em)\ )$$

How about the defects

The link expressing

$$1 - 10x^6 + 2x^8 + 7x^{14}$$



- Strong point is：

  The number of item of polynomial may rise dynamically，

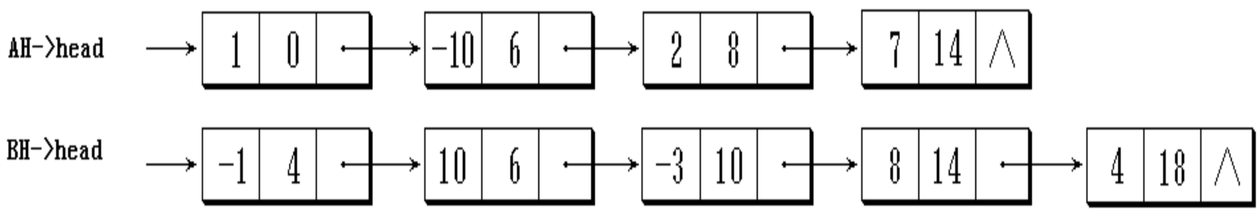
  It is convenient to insert, delete the element。

Polynomial node definition

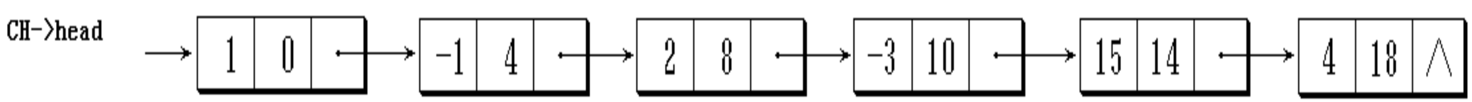
```
Class Term{
   public:
      int coef;
      int exp;

      Term(int c_t=1, int e_t=0)
       {
             coef = c_t;
             exp = e_t;
        }
       Term(const Term& t)
       {
             coef = t.coef;
             exp = t.exp;
        }
};
```

Polynomial adding to of chained lists

$$AH = 1 - 10x^6 + 2x^8 + 7x^{14}$$
$$BH = - x^4 + 10x^6 - 3x^{10} + 8x^{14} + 4x^{18}$$



(a) 两个相加的多项式



(b) 相加结果的多项式

```
LList<Term>* AddPoly(LList<Term> *AH,   LList<Term> *BH)
{
     AH->moveToStart();  BH->moveToStart();
     LList<Term>* CH = new LList<Term>;

    while( AH->currPos() < AH->length()  && BH->currPos() < BH->length() )
    {
          if( AH->getValue().exp == BH->getValue().exp) {
               if( AH->getValue().coef + BH->getValue().coef  !=  0 )
                    CH->append(Term(AH->getValue().coef + BH->getValue().coef,  AH->getValue().exp));
               AH->next();  BH->next();
          }else if( AH->getValue().exp < BH->getValue().exp ) {
               CH->append(AH->getValue());  AH->next();
          }else {
               CH->append(BH->getValue());  BH->next();
          }
    }  // end of while

    LList<Term>* TH = ( AH->currPos()<AH->length()? AH : BH );
    while( TH->currPos() < TH->length())
         { CH->append(TH->getValue());   TH->next();}
    return CH;
}
```

《数据结构与算法》课程组

重庆大学计算机学院

End of Chapter