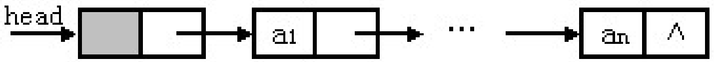
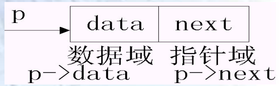
4.1.3 链表 (*Linked List*)

- 每个元素(表项)由结点构成。
- 结点可以不连续存储
- 表可扩充



用2个类表达链表

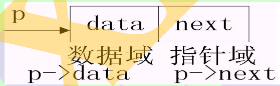


- **链表结点(node)类：Link**
- **链表(Linked list)类：*LList***

07:47 28

28

Link Class(node)



Dynamic allocation of new list elements.

```
// Singly-linked list node
template <class Elem> class Link {
public:
  Elem element; // Value for this node
  Link *next;   // Pointer to next node
  Link(const Elem& elemval,
       Link* nextval = NULL)
    { element = elemval;  next = nextval; }
  Link(Link* nextval = NULL)
    { next = nextval; }
};
```

构造函数

32 NULL

NULL

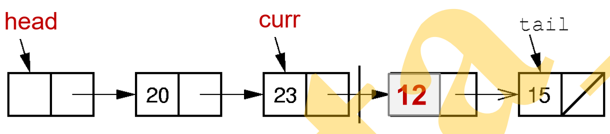
29

07:47

29

Linked List

- **3个指针＋1个整数可描述一个链表**
  - 头指针**head**/尾指针**tail**/当前指针**curr**
  - 结点个数 **cnt**



带无值头结点链表　**cnt=4**

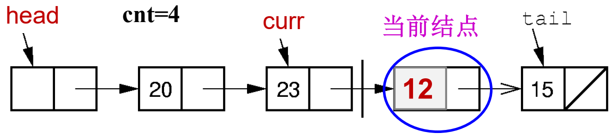
注意：无表满情况，只要系统允许，可一直插入
　　　有表空问题：cnt==0，此时不可再删除

30

07:47

30

带无值头结点链表的两点说明



① 头指针**head** 指向一个无**data**的结点，其下一个结点才是存放**list**中第一元素的结点

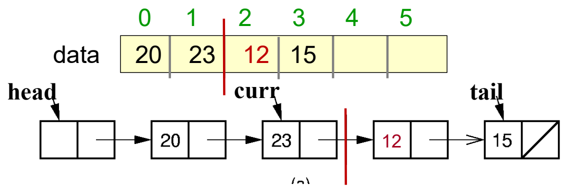
② 指针**curr** 指向当前结点的前一个结点

目的：简化 **insert**，**remove**操作

07:47 31

31

注意：当前元素/结点的*位置*----分割条之后的那个，不管是AList还是LList，插入，删除都是以它为基准



进一步体会 ADT(List) 与 Data structure (Alist, Llist)的区别与联系

List1.insert(12)；
List1.prev( );
List1.remove( )；
List1.currPos( );
l= List1.length( );

07:47

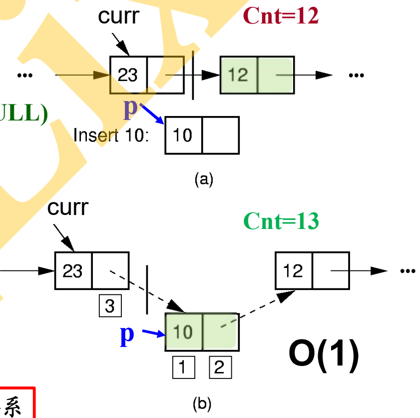
32

Linked list Insertion

S1:
p = new Link<Elem>(10,NULL)

S2: p->next = curr->next

S3: curr->next = p



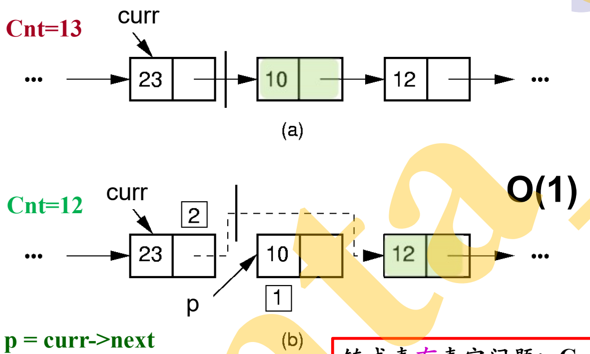
链式表无表满情况，只要系统允许，可一直插入。

33

07:47

33

Linked list Removal



S1: p = curr->next

S2: curr->next = p->next

链式表有表空问题：Cnt==0
此时不可再删除

07:47

34

34

链表各种操作的时间复杂度及对链表描述变量的影响

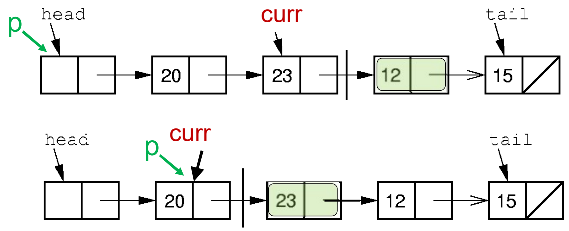
- 插入insert，删除remove
  - O(1)，cnt及链表中相关结点会改变
- 改变或获取当前位置
  - next，movetoStart，movetoEnd --- O(1)，curr指针会改变
  - prev，MovetoPos -----O(n) curr指针会改变
  - currPos -----O(n)
- 获得当前结点元素值—getValue
  - O(1)，返回 curr->next 所指结点的数据值
- 获得线性表长度---length
  - O(1)，返回 cnt 的值

35

07:47

35

Linked list prev




```
p=head;
while  p->next != curr      p = p->next;
curr=p;
```

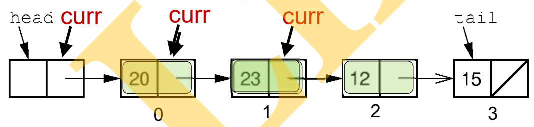
O(n)

36

07:47

36

Linked list MovetoPos



Ex1: L1.MovetoPos(2);

```
curr = head;
for(int i=0;i<k;i++)
   curr=curr->next;
```

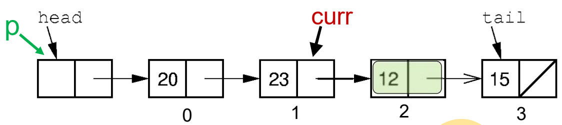
O(n)

37

07:47

37

Linked list CurrPos



O(n)

```
p=head; i=0;
while  p != curr   { p = p->next; i++; }
return i ;
```

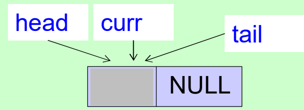
38

07:47

38

Linked List Class (1)

```
// Linked list implementation
template <class Elem> class Llist {
private:
  Link<Elem>* head; // Point to list header
  Link<Elem>* tail; // Pointer to last Elem
  Link<Elem>* curr; // Last element on left
  int cnt;        // Size of List
  void init() {     // Intialization routine
    curr = tail = head = new Link<Elem>;
    cnt = 0;
  }
```



39

07:47

39

Linked List Class (2)

```
void removeall() {
  while(head != NULL) {
      curr = head;
      head = head->next;
      delete curr;
    }
  }
public:
  LList( )
    { init(); }
  ~LList() { removeall(); }  // Destructor
  void clear() { removeall(); init(); }
```

构造函数

40

07:47

40

Linked List Class (3)

```
void moveToStart() { curr = head; }

void moveToEnd() { curr = tail }

void next() {
 // Don't move curr if right empty
 if (curr != tail)  curr = curr->next; }

int Length() const  { return cnt; }


Elem getValue() const {
  Assert(curr->next != NULL, "No value");
  return curr->next->element;
}
```

O(1)

41

07:47

41

Linked List Class (4)----insert/append

```
// Insert at front of right partition
void insert(const Elem& item) {
  Link<Elem>* temp = new Link<Elem>(item, curr->next);
  curr->next = temp;
  if (tail == curr) tail = curr->next;
  cnt++;
}

// Append Elem to end of the list
void append(const Elem& item) {
  Link<Elem>* temp = new Link<Elem>(item, NULL);
  tail->next = temp;
  tail = temp;
  cnt++;
}
```

O(1)

42

07:47

42

Linked List Class (5) ---Remove

```
//Remove and return first Elem in right partition
 Elem remove( ) {
 Assert (curr->next != NULL, "No element");
 // Remember link node
 Link<Elem>* ltemp = curr->next;
 curr->next = ltemp->next; // Remove
 Elem it = ltemp->element; //remember val
 if (tail == ltemp) // Reset tail
   tail = curr;
 delete ltemp;      // Reclaim space
 cnt--;
 return it;
}
```

O(1)

43

07:47

43

Linked List Class (6) ---Prev

```
// Move fence one step left;
// no change if left is empty
void prev() {
  Link<Elem>* temp = head;
  if (curr == head) return; //No prev
  while (temp->next != curr)
    temp = temp->next;
  curr = temp;
}
```

O(n)

07:47 44

44

Linked List Class (7) ---moveToPos

```
void moveToPos(int pos) {
   Assert ((pos >= 0) && (pos <= cnt), "Pos
out of range");
  curr = head;
  for(int i=0;i<pos;i++)
      curr=curr->next;
}
int currPos() const {
  Link<Elem> *temp=head;
  int i;
  for(i=0;curr!=temp;i++)
     temp=temp->next;
  return i;
}
};
```

O(n)

07:47 45

45

Linked List 应用举例

```
#include "LList.h"
......
voi main() {
    int a;
    LList <int>  myList;      //using the link-based list
    cout << myList.currPos() <<endl ;
    myList.insert(12);
    myList.insert(20);
    myList.insert(31);
    a=myList.remove( );   cout << a <<endl;
    myList.next( );   cout << myList.currPos( ) <<endl ;
    b=myList.getValue();  cout << b <<endl ;
}
```

07:47 46

46

LList.h (课件p29,p39-45)

```
// Singly-linked list node
template <class Elem> class Link {
public:
  Elem element; // Value for this node
  Link *next;   // Pointer to next node
  Link(const Elem& elemval,
      Link* nextval = NULL)
   { element = elemval;  next = nextval; }
  Link(Link* nextval = NULL)
   { next = nextval; }
};

template <class Elem> class LList {  // Linked list implementation
private:
  Link<Elem>* head;   // Point to list header
  Link<Elem>* tail;   // Pointer to last Elem
  Link<Elem>* curr;   // Last element on left
   ......
   ......
};
```

07:47 47

47

Space cost of AList and LList

AList

SC_{Alist}= D*E +3*4

E: Space for a data value
D: maxSize of list in a arrayed list

Llist

SC_{Llist}= (E+P)*n +3P

E: Space for a data value.
P: Space for a pointer.
n: Actual element number in a list

48

07:47

48

Space Comparison for Alist and LList

"Break-even" point/平衡点:

$$N = \frac{DE}{P + E}$$

If n>N arrayed based is better
If n<N linked list is better
If n=N both is OK!

E: Space for a data value.
P: Space for a pointer.
D: maxSize of list in a arrayed list
n: Actual element count in a list

49

07:47

49

Comparison of two Implementations of list

Time

Operation	AList	LList	Operation	AList	LList
insert	O(n)	O(1)	moveToStart	O(1)	O(1)
remove	O(n)	O(1)	moveToEnd	O(1)	O(1)
append	O(1)	O(1)	next	O(1)	O(1)
moveToPos	O(1)	O(n)	getValue	O(1)	O(1)
prev	O(1)	O(n)	length	O(1)	O(1)
currPos	O(1)	O(n)			

Space(byte)

Alist	LList
E*maxSize +3*4	(E+P)*listSize +3P+4

N* = *DE/ (P + E) n>N, Alist better, n<N Llist better

07:47

50

Comparison of two Implementations of list

Array-Based Lists:
- Insert and remove are O(*n*).
- Prev , currPos and moveToPos are O(1).
- Array must be allocated in advance.
- No overhead(额外空间) if all array positions are full.

E*maxSize +3*4

Linked Lists:
- Insertion and remove are O(1).
- Prev currPos and moveToPos are O(*n*).
- Space grows with number of elements.
- Every element requires overhead.

(E+P)*listSize +3P

51

07:47

51

```
// Insert at front of right partition
void insert(const Elem& item) {
  Link<Elem>* temp = new Link<Elem>(item, curr->next);
  curr->next = temp;
  if (tail == curr) tail = curr->next;
  cnt++;
}

// Append Elem to end of the list
void append(const Elem& item) {
  Link<Elem>* temp = new Link<Elem>(item, NULL);
   tail->next = temp;
   tail = temp;
  cnt++;
}

//Remove and return first Elem in right partition
  Elem remove( ) {
  Assert (curr->next != NULL, "No element");
  // Remember link node
  Link<Elem>* ltemp = curr->next;
  curr->next = ltemp->next; // Remove
  Elem it = ltemp->element; //remember val
  if (tail == ltemp) // Reset tail
    tail = curr;
  delete ltemp;       // Reclaim space
       cnt--;
  return it;
}
```

System new and delete are slow.

与赋值，运算，比较等常规语句比较，new， delete这些动态空间操作语句耗时较多

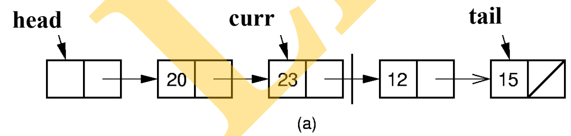
52

07:47

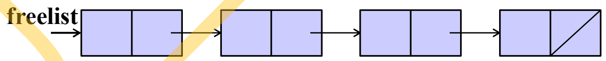
52

4.1.4 freeList 自由链表

System new and delete are slow.



在结点class（Link）中添加一个私有成员 freelist



freeList: speeding new and delete operation.

53

07:47

53

```
// Singly-linked list node class with freelist
template <class Elem> class Link {
private:
  static Link<Elem>* freelist;   // Head
public:
  Elem element;   // Value for this node
  Link* next;     // Point to next node
  Link(const Elem& elemval, Link* nextval =NULL)
  {
     element = elemval;  next = nextval;
  }
  Link(Link* nextval =NULL)
  {  next=nextval; }
```

54

07:47

54

```
void* operator new( )   // Overload new
{
 if (freelist == NULL) return ::new Link;
 Link<Elem>* temp = freelist; // Reuse
 freelist = freelist->next;
 return temp;        // Return the link
}
void operator delete(void* ptr) // Overload delete
{
 ((Link<Elem>*)ptr)->next = freelist;
 freelist = (Link<Elem>*)ptr;
}
};
template <class Elem>
Link<Elem>*  Link<Elem>:: freelist = NULL;
```

55

07:47

55

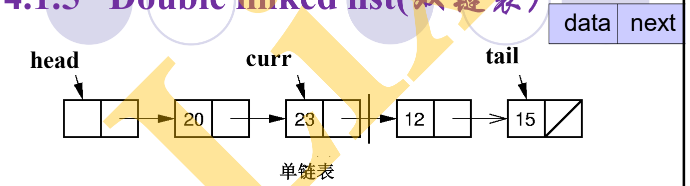
FreeList 应用说明

- 用 上述p56-57带freelist的结点类 替换 LList.h中的结点类Link（前面学过的P29），其余内容不变
- 在插入和删除操作频繁的LList应用中，用带free List的结点类 可以提高速度。
- 但是在链表生存期会多占用一部分内存空间，占用空间的大小是动态的，跟new，delete操作的次数有关

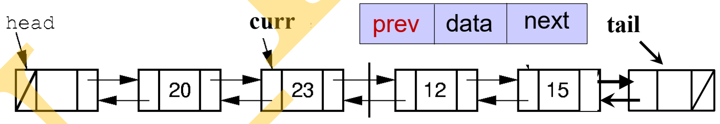
07:47 56

56

4.1.5 Double linked list(双链表)



单链表



双链表

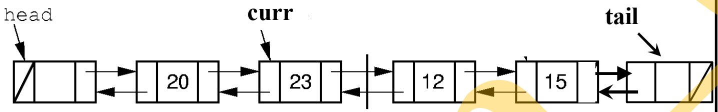
Simplify Prev operation:

1. Add a `prev` pointer in `Link node`.
2. Add a node without data in the LList as tail node

07:47 57

57

Doubly Linked List prev



双链表

```
curr = curr->prev;
```

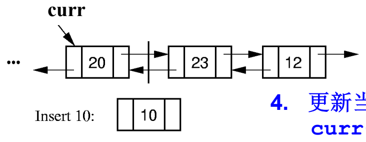
O(1)

07:47 58

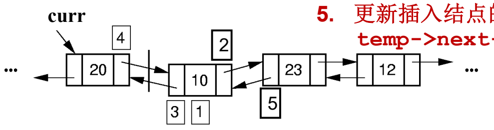
58

Doubly Linked List Insert

1. 申请一个新结点
   `Link<Elem>* temp = new Link<Elem>(10);`
2. 给temp的后向指针赋值
   `temp->next=curr->next;`
3. 给temp的前向指针赋值
   `temp->prev=curr;`
4. 更新当前指针的后向指针
   `curr->next=temp;`
5. 更新插入结点的后继结点的前向指针
   `temp->next->prev = temp;`



(a)



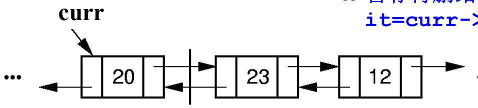
(b)

07:47 59

59

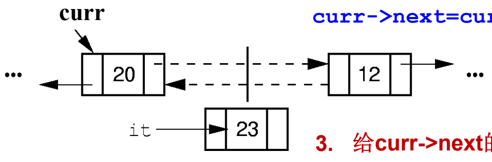
Doubly Linked Remove

1. 暂存待删结点数据域的数据
`it=curr->next->element`



(a)

2. 给当前指针curr的后向指针赋值
`curr->next=curr->next->next;`



3. 给curr->next的前向指针赋值
`curr->next->prev=curr;`

(b)

60

07:47

60

```
// Doubly-linked list Link node
template <class Elem> class Link {
public:
  Elem element;   // Value for this node
  Link *next;     // Pointer to next node
  Link *prev;     // Pointer to previous node
  Link(const Elem& e, Link* prevp =NULL,
                      Link* nextp =NULL)
    { element=e;  prev=prevp;  next=nextp; }
  Link(Link* prevp =NULL, Link* nextp =NULL)
    { prev = prevp;  next = nextp; }
};
```

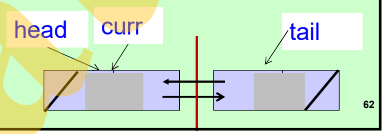
61

07:47

61

Doubly Linked List Class (1)

```
// Linked list implementation
template <class Elem> class DLList
{
private:
  Link<Elem>* head; // Point to list header
  Link<Elem>* tail; // Pointer to last Elem
  Link<Elem>* curr; // Last element on left
  int cnt;          // Size of List
  void init() {     // Intialization routine
    tail = new Link<Elem>;
    curr = head = new Link<Elem>(NULL,tail);
    tail->prev=head;
    cnt = 0;
  }
```



62

07:47

62

Doubly Linked List Class (2)

```
void removeall() {
  while(head != NULL) {
      curr = head;
      head = head->next;
      delete curr;
    }
  }
public:
  DLList( )
    { init(); }
  ~DLList() { removeall(); }  // Destructor
  void clear() { removeall(); init(); }
```

63

07:47

63

Doubly Linked List Class (3)

```
void moveToStart() { curr = head; }

void moveToEnd() { curr = tail->prev }

void next() {
 // Don't move curr if right empty
 if (curr->next != tail)  curr = curr->next; }

int Length() const  { return cnt; }

Elem getValue() const {
  Assert(curr->next != tail, "No value");
  return curr->next->element;
}
```

64

07:47

64

Doubly Linked List Class (4)

```
// Insert at front of right partition
void insert(const Elem& item) {
  Link<Elem>* temp = new Link<Elem>(item,
  curr, curr->next);
  curr->next = temp;
  temp->next->prev = temp;
  cnt++;
}
// Append Elem to end of the list
Void append(const Elem& item) {
  tail->prev = tail->prev->next =
    new Link<Elem>(item,tail->prev, tail);
  cnt++;
}
```

65

07:47

65

Doubly Linked List Class (5)

```
// Remove, return first Elem in right part
Elem remove( ) {
  if (curr->next == tail) return Null;
  Elem it = curr->next->element;
  Link<Elem>* ltemp = curr->next;
  ltemp->next->prev = curr;
  curr->next = ltemp->next; // Remove
  delete ltemp;     // Release space
  cnt--;
  return it;
}
// Move curr one step left;
Void prev()
  { if (curr != head)  curr = curr->prev;}
```

O(1)

66

07:47

66

Doubly Linked List Class (6)

```
void moveToPos(int pos) {
   Assert ((pos >= 0) && (pos <= cnt), "Pos
out of range");
  curr = head;
  for(int i=0;i<pos;i++)
      curr=curr->next;
 }
int currPos() const {
  Link<Elem> *temp=head;
  int i;
  for(i=0;curr!=temp;i++)
    temp=temp->next;
  return i;
 }
};
```

思考：
是否有方法可改进 `currPos()` 和 `moveToPos()` 的计算复杂度？

67

07:47

67

Double linked list 应用说明

- 用Double linked list 可以提高 prev操作的速度，但是要增加space开销。
- 可结合free List 来提高Double linked list的 new和delete的速度，有兴趣同学可参阅见教材p117，figure4.13代码

07:47 68

68

本章作业二

- 4.5 (a)
- 4.7
- 4.11 (b), (d)
- 4.12 (a)

07:47 69

69

根据应用选择合适的数据结构（1）

A1：假设要你设计一个学生管理系统，学生信息已拟用线性表描述。已知学生最多为1000人，存储每个学生的信息需要10字节，存一个指针需要4字节，

Q1：若系统中实际学生数大部分情况下为600以下，偶尔为800以上，那么你选顺序表还是链表？ Why ？

Q2：若系统中实际学生数大部分情况下为800以上，偶尔为600以下，那么你选顺序表还是链表？ Why?

07:47 70

70

根据应用选择合适的数据结构（2）

A2：假设要设计一个学生管理系统，学生信息拟用线性表描述。

Q1：若这个系统要频繁进行插入和删除操作，偶尔需要向前移动和移动到某个具体位置操作，那么你选顺序表还是链表？ Why?

Q2：若这个系统要频繁进行向前移动和移动到某个具体位置操作，偶尔需要插入和删除操作，那么你选顺序表还是链表？ Why?

07:47 71

71