6.3 其它链表实现

variation: keeping current position

- disadvantage of simply Linked List
  - accesses an entry of the list begins by tracing through the list from its start until the desired position is reached.
- improvement
  - remember the last-used position in the list
- attention
  - it will not speed up every application using lists.
  - mutable data members :used to record the last-used position. (can be changed, even by constant methods)

variation: keeping current position

- Enlarged definition

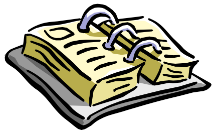
```
template <class List_entry>
class List{
  public:
        //add specifications for the methods of the list ADT
        //all methods in previous definition
 protected:
        int count;
        mutable int current_position;
        Node<List_entry> *head;
        mutable Node<List_entry> *current;
        void set_postion(int position) const;
};
```

- All the new class members have protected visibility, so, from the perspective of a client, the class looks exactly like the earlier implementation.
- The current position is now a member of the **class** List, so there is no longer a need for set_position to return a pointer; instead, the function simply resets the pointer current directly within the List.

variation: keeping current position

- realization of new set_position method

```
template <class List_entry>
void List<List_entry>::set_postion(int position) const{
    if (position<current_position){  //从头开始
      current_position=0;
      current=head;
    }
    //从current开始，进行后移
    for (;current_position!=position; current_position++)
      current=current->next;
};
```



realization of insert function

```
template <class List_entry>
    Error_code List<List_entry>::insert(int position,const List_entry& x){
        if (position<0||position>count)   return range_error;
        Node <List_entry> *new_node,*previous,*following;
        if (position>0){
         set_position(position-1);
                    //寻找插入点前一个结点位置
         following=current->next;
        }else   following=head; //定位插入点的后继位置
        new_node=new Node<List_entry>(x,following);
                   //生成一个新结点并插在following之前
        if (new_node==NULL) return overflow;//判断是否溢出
        if (position==0)  head=new_node;
                 //在0号位置插入时，将head指向新插入结点
        else        current->next=new_node;
                 //其他位置插入时，将新结点连接在previous之后
        count++;//计数变化
       current=current->next;
       current_position++;
        return success;
    }
```

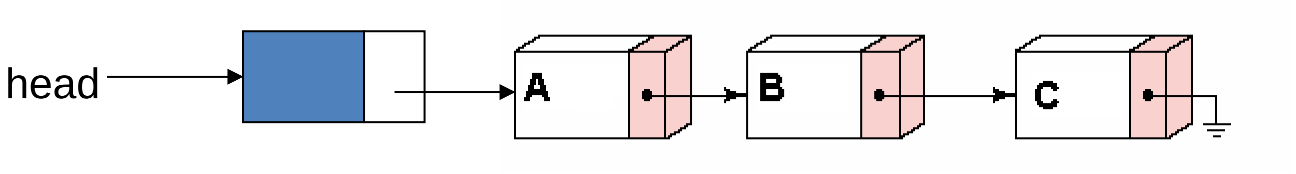
❑For repeated references to the same position, neither the body of the **if** statement nor the body of the **for** statement will be executed, and hence the function will take almost no time.(重复访问某个结点时)

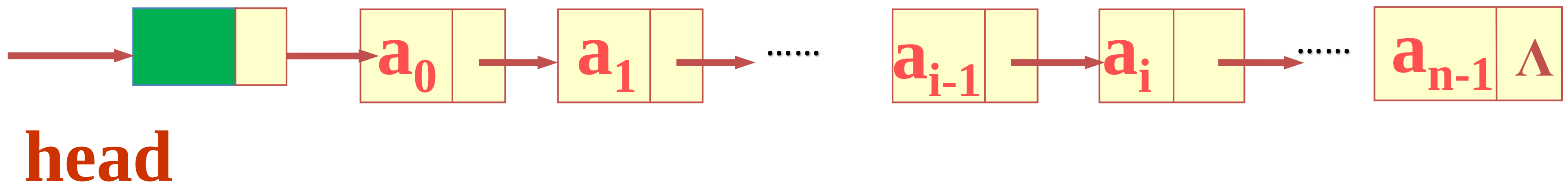
❑If we move forward only one position, the body of the **for** statement will be executed only once, so again the function will be very fast.（向后访问时）

❑If it is necessary to move backwards through the List, then the function operates in almost the same way as the version of set position used in the previous implementation.（向前访问时）

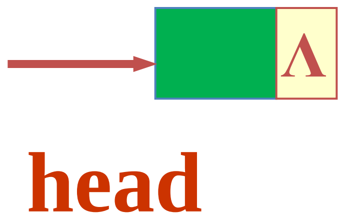
Supplement :Add a dummy node

```
template <class List_entry>
class List{
  public:
    List(const List<List_entry>& copy);          //拷贝构造函数
    ~List();           //析构函数
    void operator=(const List<List_entry>& copy);
               //赋值运算符号的重载
    //声明ADT中定义的insert、remove等
  protected:
    int count;          //结点数量
    Node <List_entry> *head;          //单链表的头指针
    Node <List_entry> * set_position(int position) const;
               //获得第position个结点的位置——返回指针
};
```





a_0
a_1
……
a_{i-1}
a_i
……
a_{n-1}
Λ
head



Λ
head

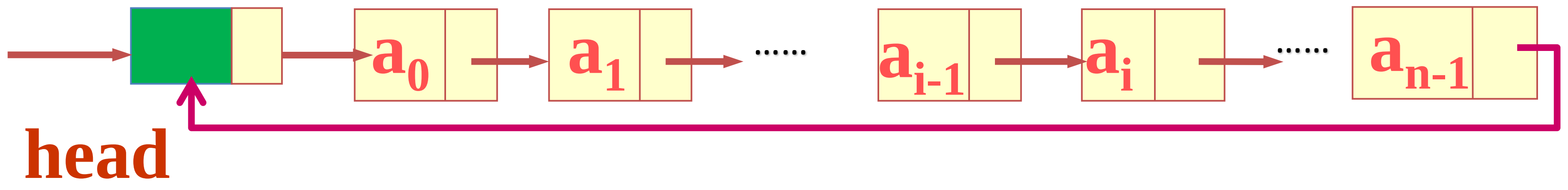
Supplement :Add a dummy node

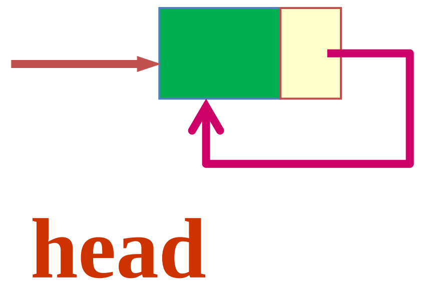
- set_position(to be corrected)

```
template <class List_entry>
Node<List_entry>* List<List_entry>::set_position(int position
const
{
     Node <List_entry> *q=head;
     for (int i=-1;i<position;i++) q=q->next;
     return q;
}
```

Supplement :Add a dummy node

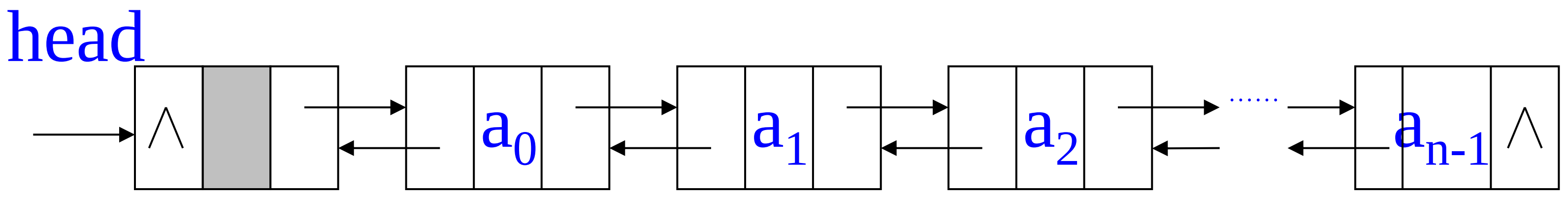
```
    if (position<0||position>count)   return range_error;
    Node <List_entry> *new_node,*previous,*following;
     previous=set_position(position-1);
             //寻找插入点前一个结点位置
     following=previous->next;
     new_node=new Node<List_entry>(x,following);//生成一个
    新结点并插在following之前
    if (new_node==NULL) return overflow;//判断是否溢出

    previous->next=new_node; //其他位置插入时，将新结点连
    接在previous之后
    count++;        //计数变化
    return success;
}
```

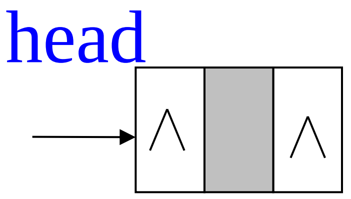

a_0
a_1
……
a_{i-1}
a_i
……
a_{n-1}
head


head

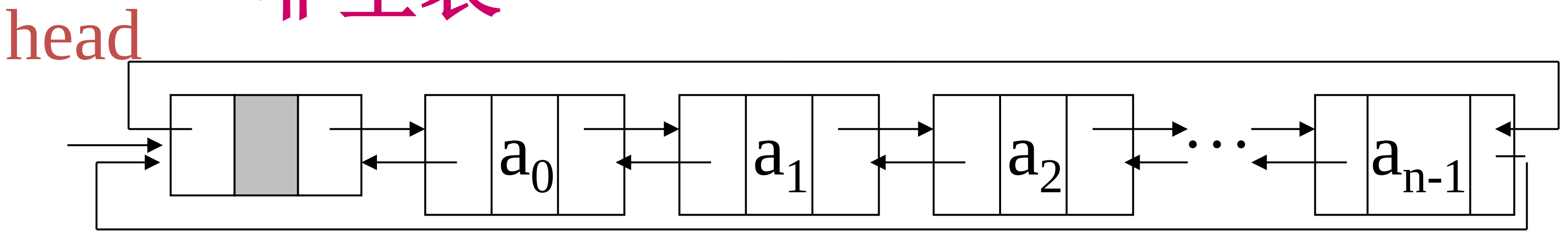
Doubly linked Lists

- characteristic of simply linked List
  - can access entry in forward order (by the pointer "next")
  - accessing entry in backward order is difficult.
- Doubly Linked List

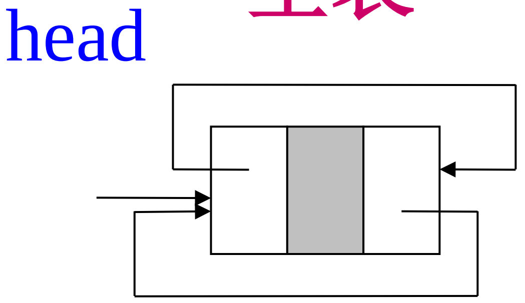




非空表



空表



判空条件

head.next==head

Doubly linked Lists

- Doubly Linked List
  - Node definition

```
template <class Node_entry>
struct Node{
    Node_entry entry;//数据域
    Node<Node_entry> *next;
    Node<Node_entry> *back; //两个指针域分别指向后继和前驱
    Node();
    Node(Node_entry,Node<Node_entry>* link_back=NULL,
                             Node<Node_entry>* link_next=NULL);
};
```

Doubly linked Lists

- Doubly Linked List
  - definition of doubly-linked list class:

```
template <class List_entry>
class List{
  public:
        //Add specification for methods of the list ADT
        //constructors and deconstructor
  protected:
        int count;
        mutable int current_position;
        mutable Node<List_entry>* current;
        void set_position(int position) const;
//注意：没有了head};
```

important characteristic:

current=current->next->back=current->back->next;

Doubly linked Lists

- Doubly Linked List
  - realization of some member functions
    - set_position （设置位置）

```
template <class List_entry>
void List<List_entry>::set_position(int position) const{
   if (current_position<=position)
      for (;current_position!=position;current_position++)
                current=current->next;
      //利用循环进行后移
   else
      for (;current_position!=position;current_position--)
                current=current->back;
      //利用循环进行前移
}
```

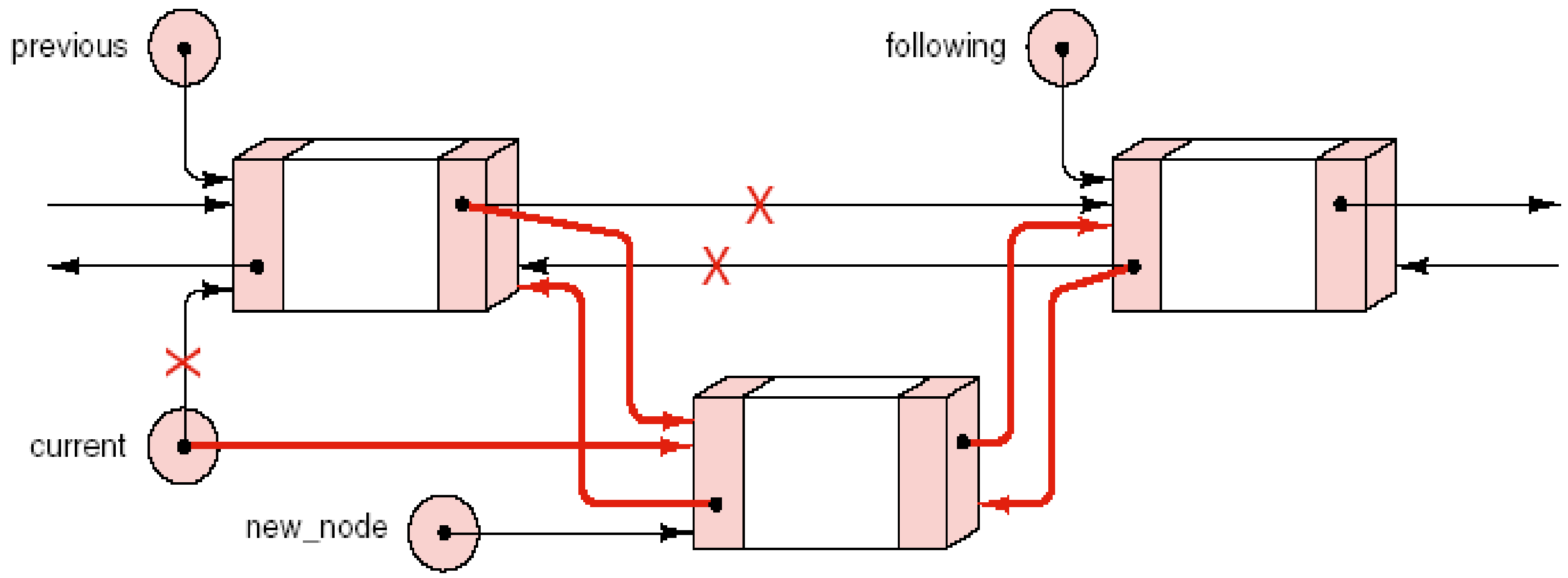
We can move either direction through the List while keeping only one pointer, current, into the List.(只有current 一个指针）

We do not need pointers to the head or the tail of the List,since they can be found by tracing back or forth from any given node.（不需要头指针或尾指针）

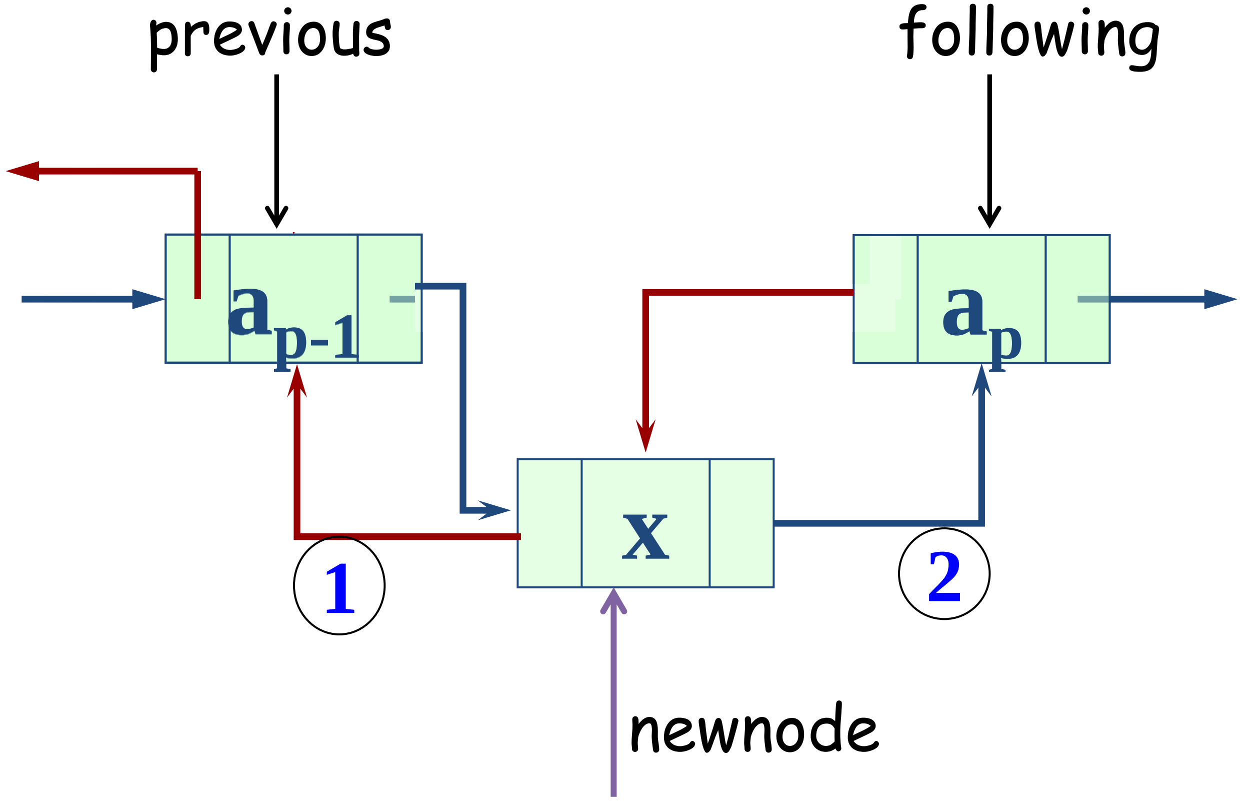
To find any position in the doubly linked list, we first decide whether to move forward or backward from the current position,and then we do a partial traversal of the list until we reach the desired position.（定位之前，先确定是应该往前还是往后，然后再作相应的移动）

Doubly linked Lists

- Doubly Linked List
  - realization of some member functions:
    - Insert（插入结点）



双向链表



```
  new_node=new
  Node<List_entry>(x,previous,following);
previous->next=new_node;
```

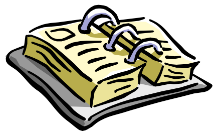
if (following!=NULL) following->back=new_node;

Doubly linked Lists

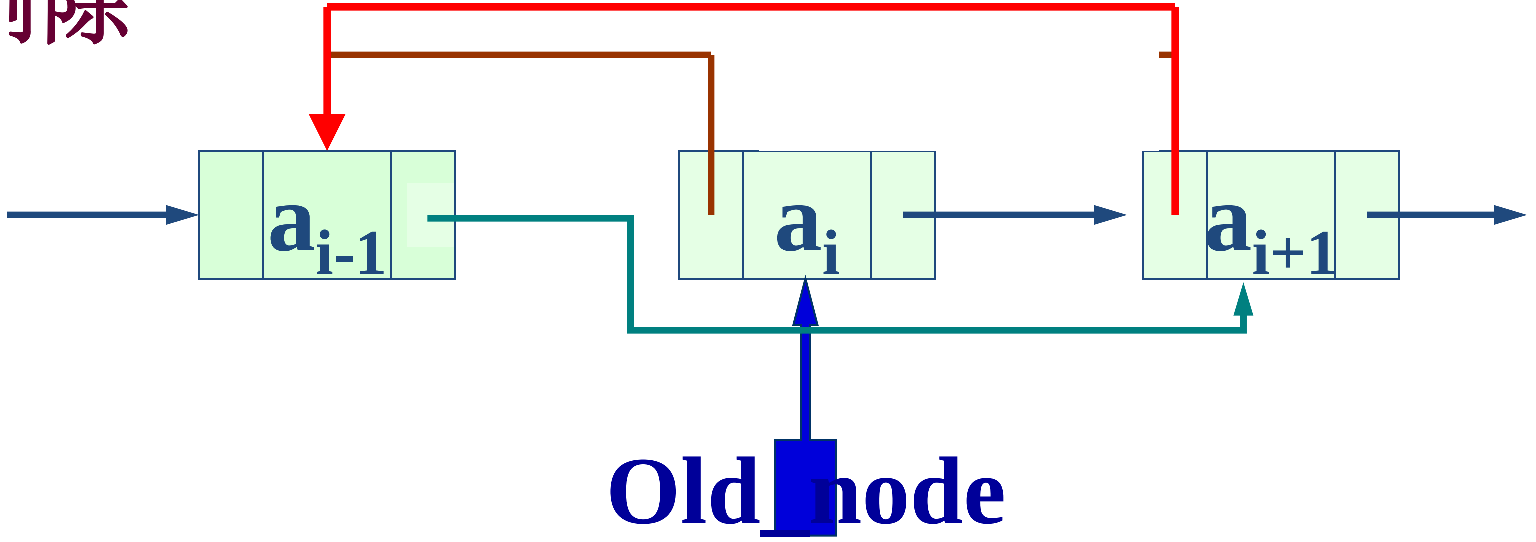
```
template <class List_entry>
 Error_code List<List_entry>::insert(int position, const List_entry& x){
        if (position<0||position>count) return range_error;
        Node<List_entry> *new_node,*following,*preceding;
        //为插入做好准备工作，包括following和preceding
        if (position==0){
                if (count==0) following=NULL;
                else{
                        set_position(0);
                        following=current;
                }
                preceding=NULL;
        }
```

Doubly linked Lists

```
   else{
           set_position(position-1);
           preceding=current;
           following=preceding->next;
   }
   //新增结点，并进行插入
   new_node=new Node<List_entry>(x,preceding,following);
   if (new_node==NULL) return overflow;
   if (preceding!=NULL) preceding->next=new_node;
   if (following!=NULL) following->back=new_node;
   //插入需要修改两个方向上的指针域
   current=new_node;
   current_position=position;
   count++;
   return success;
}
```



删除



```
template <class List_entry>
Error_code List<List_entry> :: remove(int position, List_entry &x)
/* Post: If 0<position < n; where n is the number of entries in the
List, the function succeeds:
The entry in position is removed from the List, and the entries in
all later positions have their position numbers decreased by 1.
The parameter x records a copy of the entry formerly in position.
Otherwise the function fails with a diagnostic error code. */
{
Node<List_entry> *old_node, *neighbor;
if (count == 0) return fail;
if (position < 0 || position >= count) return range_error;
set_position(position);
old_node = current;
if (neighbor = current->back)
          neighbor->next = current->next;
if (neighbor = current->next) {
         neighbor->back = current->back;
         current = neighbor;
         }
else {
         current = current->back;
         current_position--;
         }
x = old_node->entry;
delete old_node;
count--;
return success;
}
```

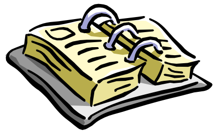
A contiguous list:
Overflow,
Must determine the maximum amount of the list,
Insert will cause moving .
Random access

A linked list:
Don't worry about overflow
Dynamic structure
Insert only need change the links.

The links also take space. (When the node is large,…)
Not to suited to random access.

Comparison of Implementations

- Contiguous storage is generally preferable
  - when the size of the list is known when the program is written;
  - when the entries are individually very small;
  - when few insertions or deletions need to be made except at the end of the list;
  - when random access is important
- Linked storage prove superior
  - when the size of the list is not known in advance;
  - when the entries are large;
  - when flexibility is needed in inserting, deleting and rearranging the entries



To choose among linked list implementations, consider:

❑Which of the operations will actually be performed on the list and which of these are the most important?(需要哪些操作，那个最重要？）

❑Is there locality of reference? That is, if one entry is accessed,is it likely that it will next be accessed again?（引用是否有连续性）

❑Are the entries processed in sequential order? If so, then it may be worthwhile to maintain the last-used position as part of the list structure.（元素是否按顺序访问，如果是，则值得保留一个最近使用的位置。）

❑Is it necessary to move both directions through the list? If so,then doubly linked lists may prove advantageous.（是否需要向两个方向操作？如果是，则用双向链表比较好。）