6.2 单链表实现

simply Linked Implementation

- Declarations

  Node template:

```
template <class Node_entry>
struct Node{
    Node_entry entry;//数据域
    Node<Node_entry> *next;//指针域
    Node();
    Node(Node_entry, Node<Node_entry>* link=NULL);
    //此处提供了两个构造函数，彼此间是函数的重载关系，函数的具体实现与4.1.3中的方法基本一样。
};
```

对于一个非空线性表L=(a_0,a_1,a_2,...a_i...a_{n-1}）
可以表示为这样的存储结构：

空表时，head指针为NULL。

这种链表结构我们称为单链表。意思是每个结点只有一个向后的单方向的指针。

在单链表中，整个线性表可由head指针唯一确定，即可以通过head指针寻访到任一个结点并进行操作。另外为了方便地获得线性表的长度，增加了一个count值记录表长。

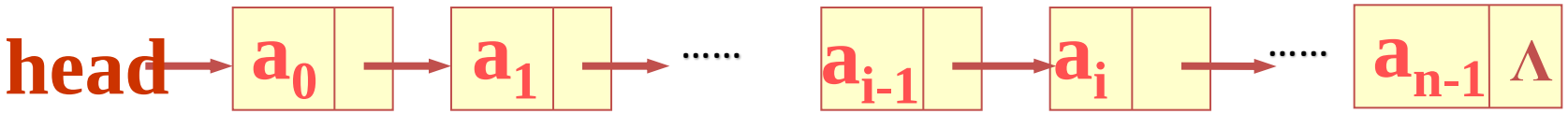
这是单链表类的定义框架。除了包含有count和head指针这两个protected属性的数据成员的的定义之外，
还有ADT定义中的10个方法以及为了保证链式结构安全性而配备的safeguards方法：析构函数、拷贝构造函数和赋值重载运算。

simply Linked Implementation

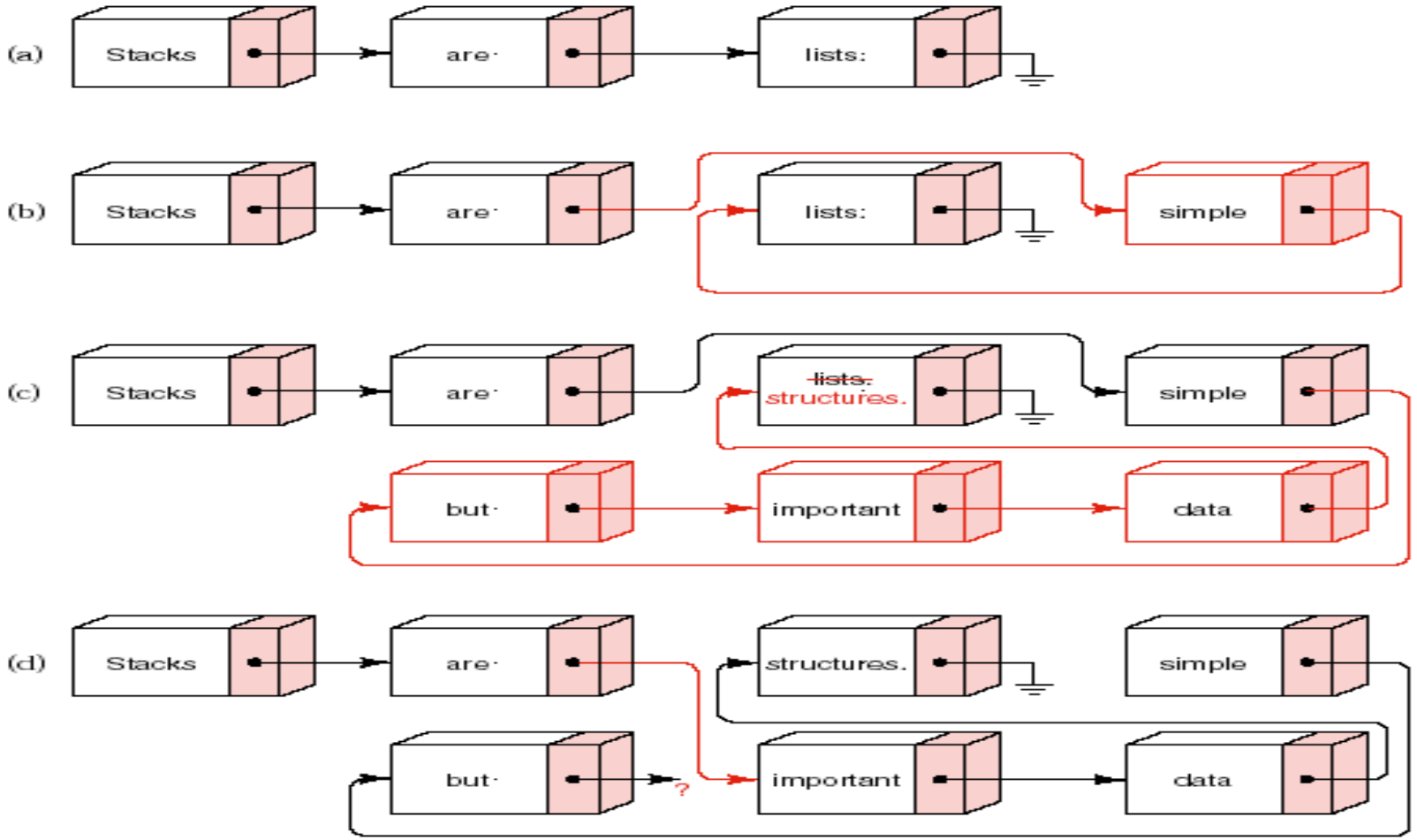
- List template

```
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
};
```

```
List();  //构造函数
int size() const;
bool full() const;
bool empty() const;
void clear();
void traverse(void (*visit)(List_entry& ));
Error_code retrive(int position,List_entry& x) const;
Error_code replace(int position, const List_entry& x);
Error_code remove(int position, List_entry& x);
Error_code insert(int position, const List_entry& x);
```


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

simply Linked Implementation

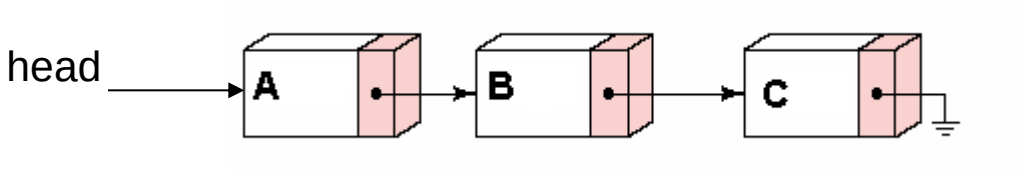


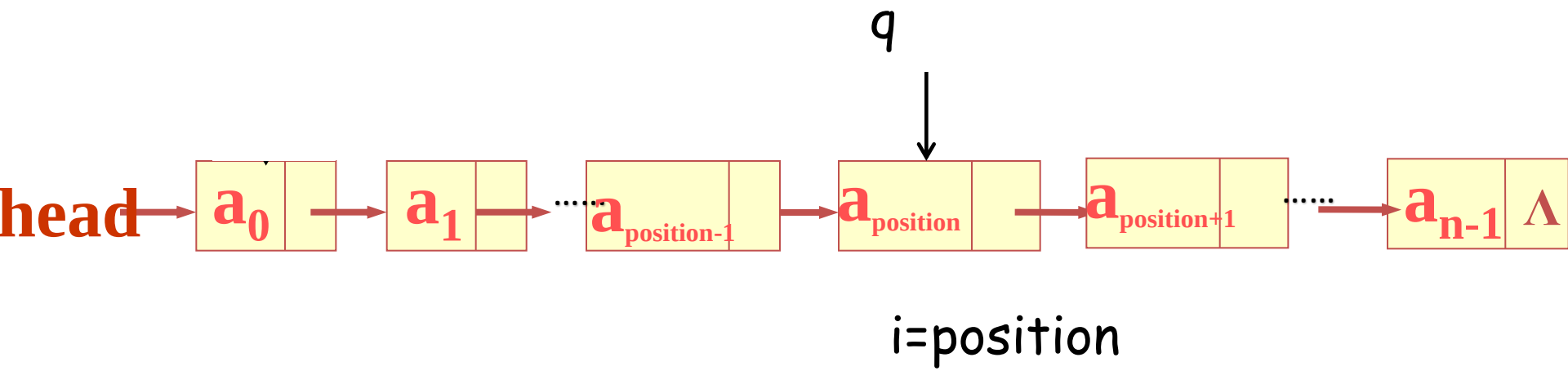
simply Linked Implementation

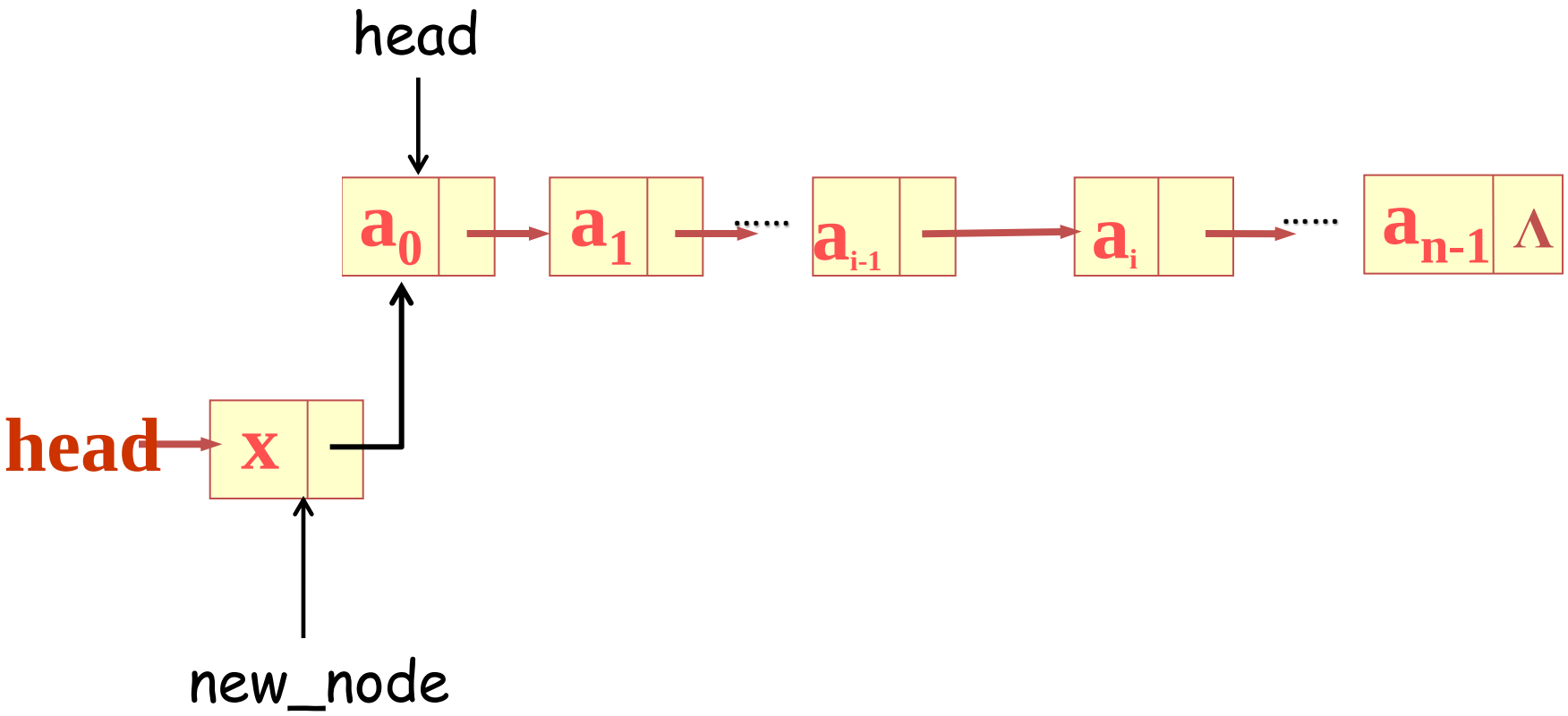
- Some member functions
  - set_position

```
template <class List_entry>
Node<List_entry>* List<List_entry>::set_position(int position) const{
        Node <List_entry> *q=head;
        for (int i=0;i<position;i++)
                q=q->next;
        return q;
}
```

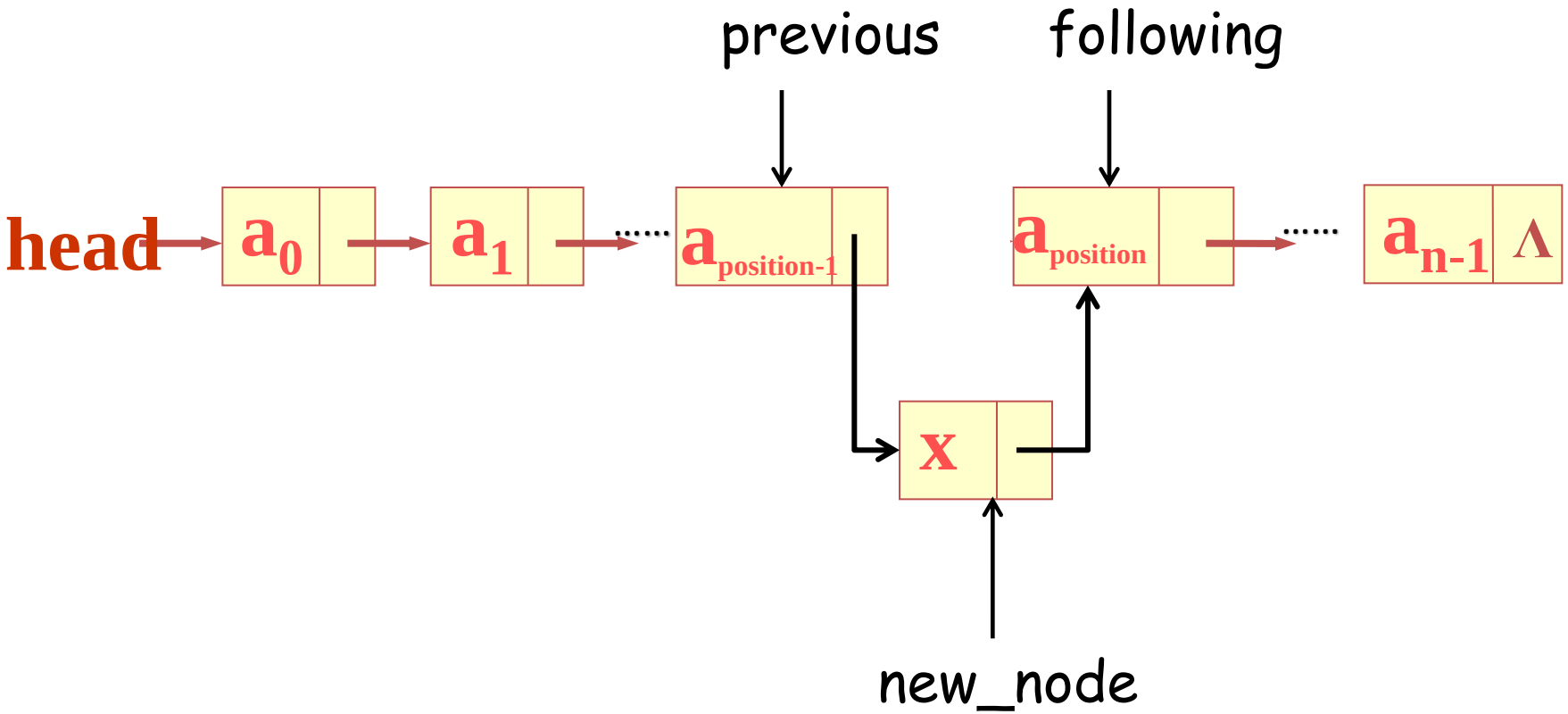
T(n)=O(n)




q
head
a_0
a_1
.....
$a_{position-1}$
$a_{position}$
$a_{position+1}$
.....
a_{n-1}
Λ
i=position



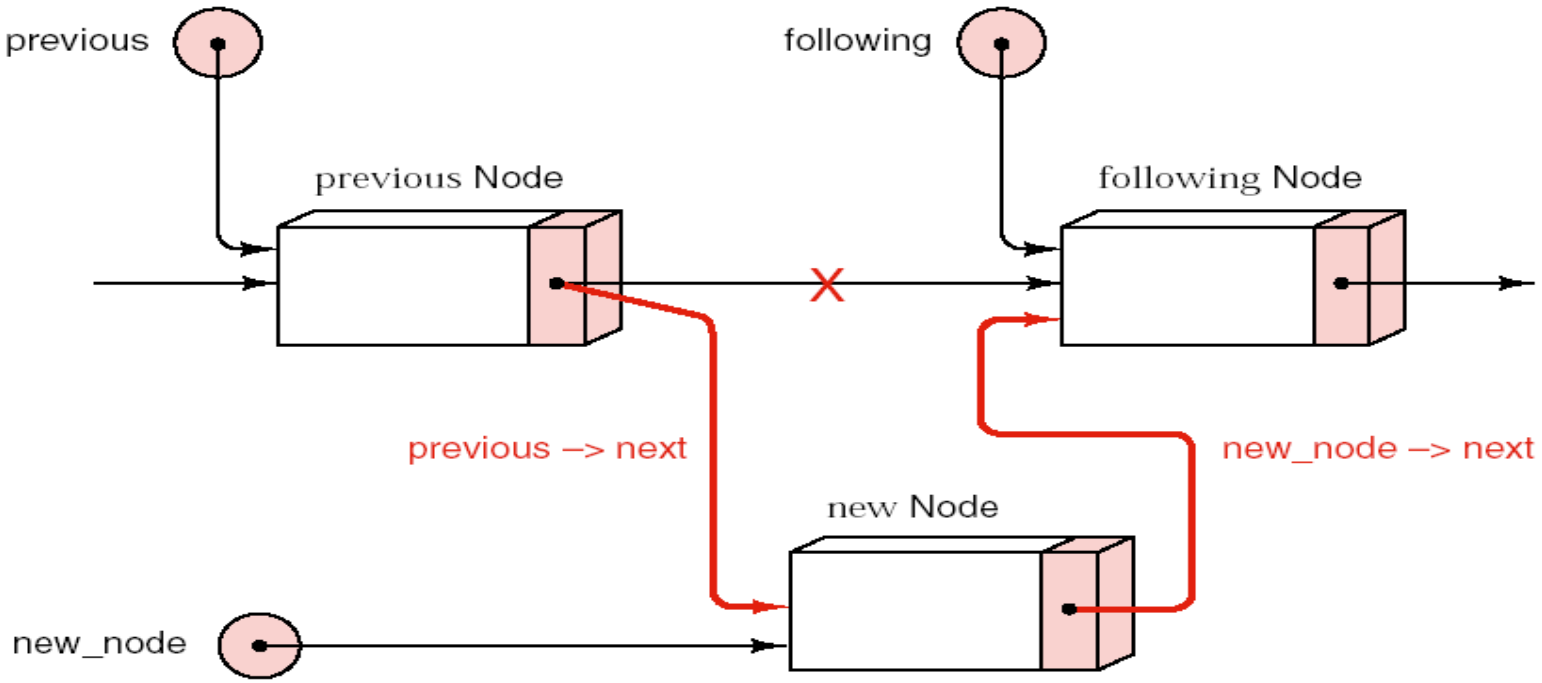

```
new_node=new Node<List_entry>(x,head);

head=new_node;
```


previous
following
head
a_0
a_1
.....
$a_{position-1}$
$a_{position}$
.....
a_{n-1}
Λ
x
new_node

simply Linked Implementation

- insert
  - insert process

```
new_node=new Node<List_entry>(x,following);
if (new_node==NULL) return overflow; //判断是否溢出
previous->next=new_node;
```

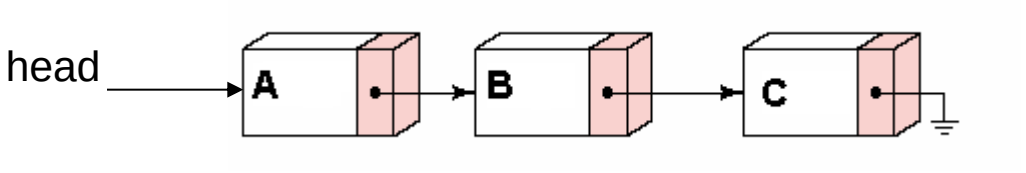


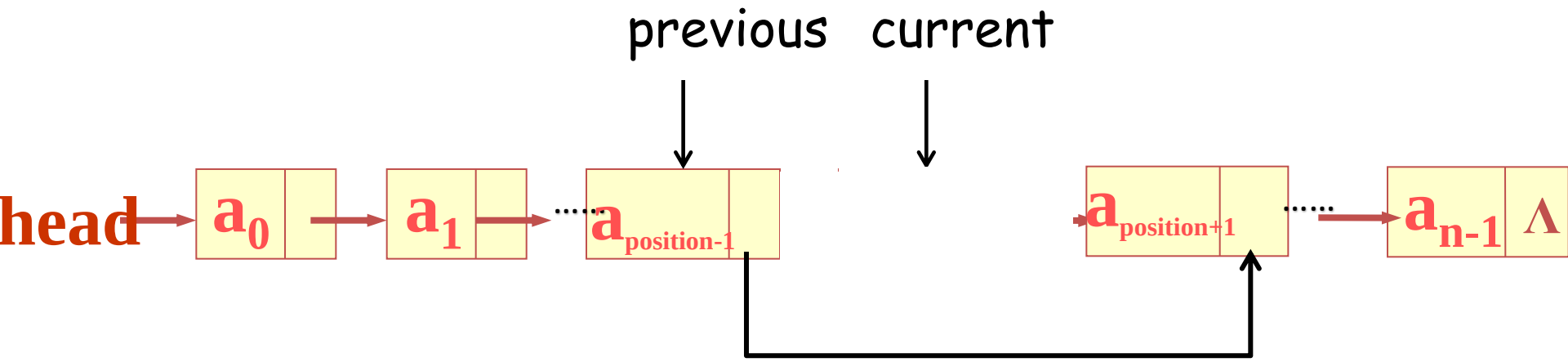
```
template <class List_entry>
Error_code List<List_entry>::insert(int position,const
List_entry& x){



return success;
          }
```

```
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){
        previous=set_position(position-1);
                    //寻找插入点前一个结点位置
        following=previous->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          previous->next=new_node;
          //其他位置插入时，将新结点连接在previous之后
    count++;    //计数器变化
    return success;
}
```

时间效率分析：　　T(n)=O(n)

```
template <class List_entry>
Error_code List<List_entry>::remove(int position,const
List_entry& x){



}
```






```
previous->next=current->next;
delete current;
```

```
template <class List_entry>
Error_code List<List_entry> :: remove(int position, List_entry &x)
{
Node<List_entry> *prior, *current;
if (count == 0) return fail;
if (position < 0 || position >= count) return range_over;
if (position > 0) {
previous = set_position(position – 1);
current = previous->next;
previous ->next = current->next;
}
else {
current = head;
head = head->next;
}
x = current->entry;
delete current;
count--;
return success;
}
```