Chapter 6 LIST

- 可直接引用C++头文件

```
#include <list>
```

- 访问链表中任意元素所需的最坏时间复杂度为O(n) （取决于迭代器移到该元素前需要经过多少元素
  - 对比：vector和array可通过下标随机访问位置 时间复杂度为常数（因为元素连续）
  - 链表中不可进行下标运算（因为无法做到随机访问）
- 链表的插入删除操作的时间复杂度都为常数
- 链表内的元素有序
- 重要：使用迭代器

Here are some of the methods in the list class, templated on t:

push_front (**const** t& x), push_back (**const** t& x)

pop_front( ), pop_back( )

empty( ), size( ), begin( ), end( )

insert (iterator position, **const** t& x)

erase (iterator position)

erase (iterator first, iterator last) **deletes all items in the range from** first **(inclusive) to** last **(exclusive)**

一些新操作

- splice
  - void splice (iterator position, list<t>& x);
  - 将x的内容插入到iterator指向位置前面且插入后x为空
- operator<
  - 最坏时间复杂度：O(nlogn)
- 逆序输出操作

```
list<double> roots;

for(int j = 0; j < 20; j++)\{
roots.push_back(sqrt(j));

//Print sqrt(19), sqrt(18),...
list<double>::iterator itr;
for(itr = --roots.end(); itr != --roots.begin(); itr--)
    cout << *itr << endl;
```

※最基础的list实现及应用

```
list<string>words;

    list<string>::iterator itr;

    string word;

    for (int i = 0; i < 5; i++)
    {
            cin >> word;
            words.push_back (word);
    } // for
    words.pop_front();
```

```
    words.pop_back();
    for (itr = words.begin(); itr != words.end(); itr++)
            cout << *itr << endl;
```

双向链表

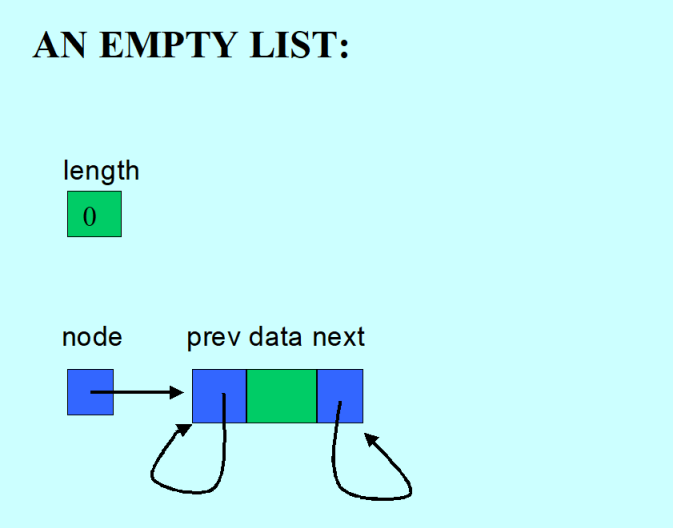
```
// class list
struct list_node
{
    list_node* next;
  list_node* prev;
  T data;    // holds one item
}; // list_node

protected:
    unsigned length;
    list_node* node;
    iterator(list_node* x): node(x){} // node字段被初始化成x，即将x赋值给iterator类的node字段（注意不是list类的node
```

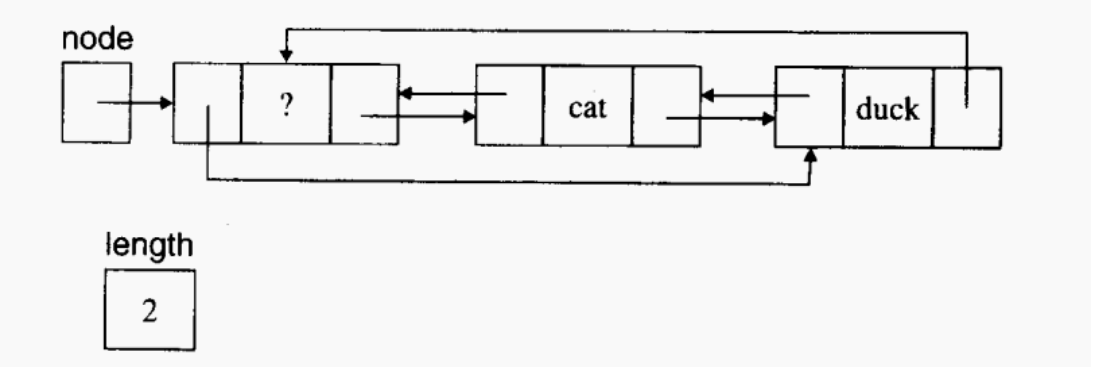
- header node
  - 空链表
    - data为空
    - prev和next指针都指向自己

      (*node).next = node;

      (*node).prev = node;



- 非空链表
  - next指向链表第一项
  - prev指向链表最后一项
  - 链表最后一项的next指向头节点

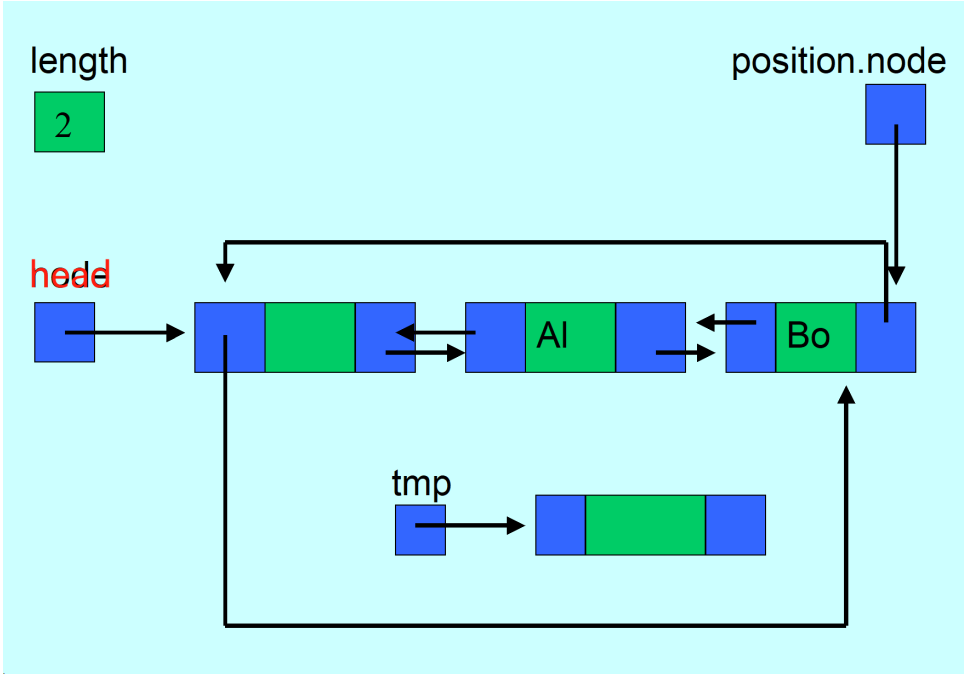


- 头指针指向哑节点（也为头节点）list_node 不用于存放真正元素 只用于表示起始
- 空链表 pre和next指向自己 使得指针无需指向NULL
- 通过头节点实现循环 头结点的next指向第一个节点 头节点的pre指向最后一个节点

```
Iterator begin(){
    return Iterator(head -> next);
}

Iterator end(){
```

```
    retrn Iterator(head);
}
```



插入操作

- **最坏时间复杂度为常量**
- 新插入的节点位于迭代器position所在的list_node的前面
- 返回值为位于新插入节点上的迭代器

```
iterator insert (iterator position, const T& x)
{
        // step1：分配一个新节点
        list_node* tmp = get_node();
        // step2：将x赋值给新节点的data字段
        (*tmp).data = x;
    // 又相当于↓
        construct(value_allocator.address((*tmp).data), x);
        // step3：赋值next指针
    (*tmp).next = position.node;
        // step4：赋值prev指针
```

```
    (*tmp).prev = (*position.node).prev;
        // step5：插入位置node的前一节点的next指针指向tmp
    (* ((*position.node).prev)).next = tmp;
        // step6：插入位置node的prev指针指向prev
    (*position.node).prev = tmp;
        // step7：增加链表长度
    ++length;
        // step8：返回tmp
    return tmp;
}
//以上代码对于空链表也适用
```

说明：双向链表指针和迭代器指针

```
class list{
    protected:
        struct list_node {
        ...
        }
        list_node* head; //书上定义为list_node* node 与迭代器指针名称相同 虽然在使用时不影响 但为了避免混淆 将此处双向链表的头节点命名为head更合适
    public:
        class Iterator{
            protected:
                list_node* node
            ...
}
```

应用：行编译器

1. 第一行为line 0
2. 总有一行为当前行current line
3. 每一个命令都以$开始

- $Insert：在当前行的后面插入文本 注意：输入$Insert只为进入插入状态，当输入非命令字符语句时才算真正调用插入命令。
- $Delete k m：删除从k行开始到m行结束的部分；若当前行在删除范围内，k-1行成为当前行
- $Line m：指定任意合法的行号为当前行
- $Done：打印

链表用于存储字符串，每一节点为一行

当前行的迭代器指向当前行在链表对应节点的迭代器

当前的行line number

pause用于检查当前是否为命令 是命令则传给command_check

command_check用于对应当前为什么命令

两个空格之间的为第一个参数

插入在前面还是后面？

first是迭代器吗