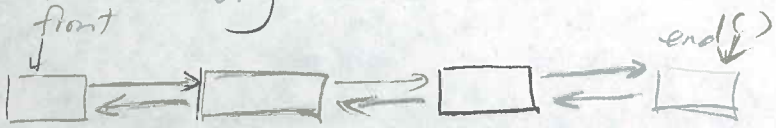
6-1 List Container

Vector container
- size can be changed
- contiguous block of memory
- use index to direct access
- operation at the end of the vector
- no very efficient if add/erase in the middle

List container
- size can be changed
- random memory location



- front proceeds to end
- use iterator to access the elements

List API — see print out

List ex. ex1.

```
List<int>    intList;
(a) List<double>  realList(8);
(b) List<time24>  timeList(6, time24(8,30));
    string strArr[] = {"array", "vector", "List"};
(c) List<string>  strList(strArr, strArr+3);
```

(a) | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

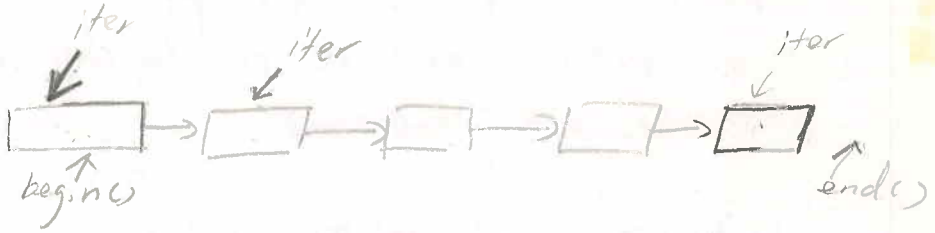
(b) | 8:30 | 8:30 | 8:30 | 8:30 | 8:30 | 8:30 |

(c) | array | vector | List |

```
intList.push_front(3);
intList.push_front(20);
intList.push_back(10);
intList.push_back(25);

int value = intList.front();
    intList.front() = 15;
    intList.back() = 20;

    intList.pop_front();
    intList.pop_back();
    intList.pop_back();
```

Ex.

```
template <typename T>
list<T>::iterator  find_last_of (list<T> & alist,
                                 const T& value)
{
    list<T>::iterator  iter = alist.end();

    do
        iter--;
    while (iter != alist.end() && (*iter != value));
    return iter;
}
```



Note: List iterators move through the list in a circular fashion.

- When an iterator is at end(), ++ moves the iterator to the first list element, at location begin().
- When an iterator is at begin(), -- moves the iterator to end().

Ex.

```
int    arr[] = {1, 3, 3, 4, 5};
int    arrSize = sizeof(arr)/sizeof(int);
list<int>  intList(arr, arr+arrSize);
list<int>::iterator iter;
```

```
iter = find_last_of(intList, e);
if (iter != intlist.end())
{   cout << *iter << " ";
    iter++;
    cout << *iter << endl;
}

iter = find_last_of(intList, 7);
if (iter != intList.end())
    cout << "7 is in the list";
else
    cout << "7 is not in the list";
```

constant iterator

- same as iterator except *iterator can't be on the left hand side of assignment statement.

```
List<T>::const_iterator citer;    // constant iterator
List<T>::iterator       iter;     // iterator | non constant
```

Rule for use constant or non-constant iterator

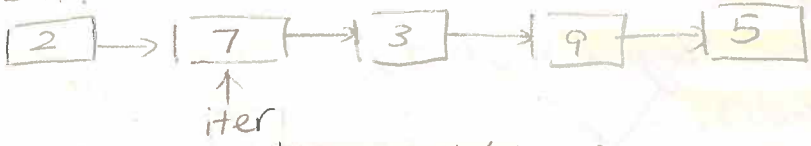
- Use a constant iterator to access & scan a constant list.
- Use a nonconstant iterator to access & scan a non-constant list.

```
template <typename T>
void writelist(const List<T>& alist, const string& sep=" ")
{   List<T>::const_iterator citer;
    for (citer = alist.begin(); citer != alist.end(); citer++)
        cout << *citer << sep;
    cout << endl;
}
```

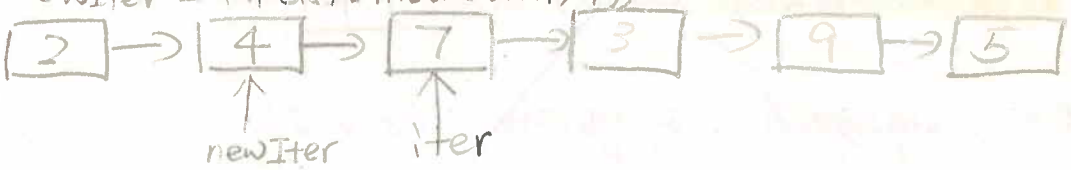
6-3 General List insert and erase operations

iterator insert(iterator pos, const T& value)

intList:



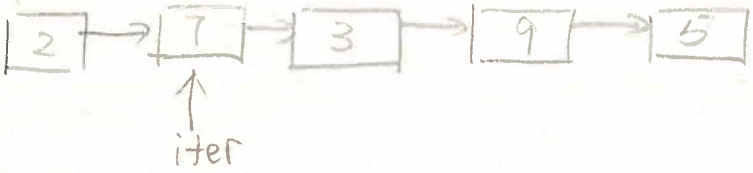
newIter = intList.insert(iter, 4);



void erase(iterator pos);

void erase(iterator first, iterator last);

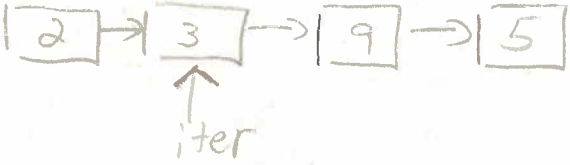
intList:



intList.erase(iter);

iter → 7 // invalidate

intList.erase(iter++);



intList.erase(intList.begin(), intList.end());

empty List.

Ex 6-4

1.
```
template <typename T>
void doubleData (list<T>& aList)
{
    list<T>::iterator p=aList.begin();
    while (p!=aList.end())
    {
        aList.insert(p,*p);
        p++;
    }
}
```

2.
```
template <typename T>
void eraseSmallValues (list<T>& aList, const T& target)
{
    list<T>::iterator iter = aList.begin();
    while (iter != aList.end())
        if (*iter < target)
            aList.erase(iter++);
        else
            iter++;
```

3.
```
intList.erase(intList.begin(), intList.end());
```

Ex

Insert / maintain an ordered list

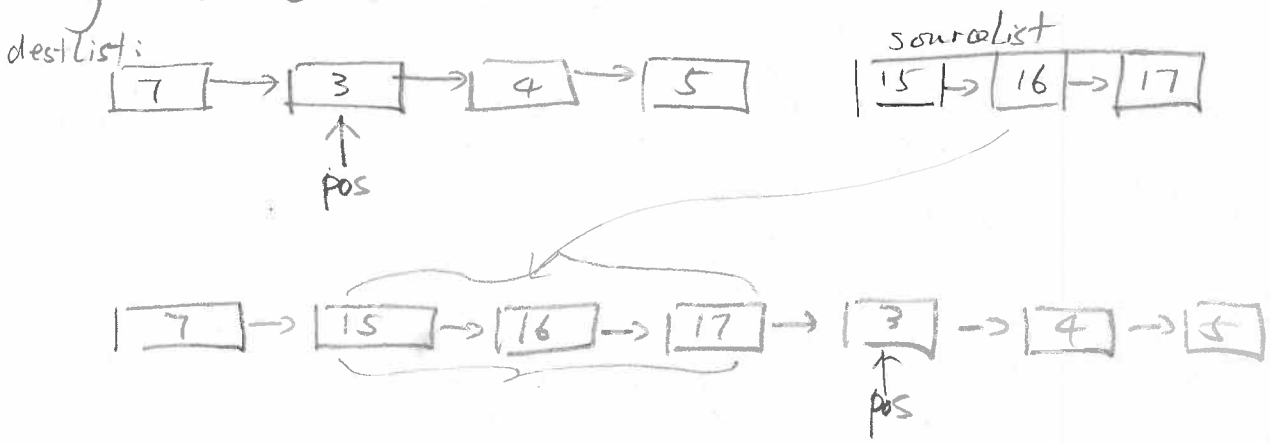
```
template <typename T>
void insertOrder ( list<T>& orderedList, const T& item)
{
        list<T>::iterator curr = orderedList.begin(),
                          stop = orderedList.end();
        while ((curr != stop) && (*curr < item))
                curr++;
        orderedList.insert (curr, item);
}
```

Ex.

Removing Duplicates

```
template <typename T>
void removeDuplicates (list<T>& aList)
{
        T currValue;
        list<T>::iterator curr, p;
        curr = aList.begin();
        while (curr != aList.end())
        {
                currValue = *curr;
                p = curr;
                p++;
                while(p != aList.end())
                  if (*p == currValue)
                        aList.erase(p++);
                  else
                        p++;
        }       curr++;
}
```

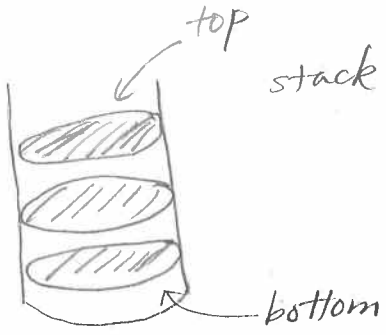
Splicing two Lists




```
template <typename T>
void splice (list<T>& dest, list<T>::iterator pos,
             const list<T>& source)
{       list<T>::const_iterator sourceIter;
        sourceIter = source.begin();

        while (sourceIter != source.end())
        {       dest.insert(pos, *sourceIter);
                sourceIter++;
        }
}
```

chapter 7 stacks

stack : a sequence of items that are accessible at only one end of the sequence (LIFO)



stack class API

class stack constructor <stack>

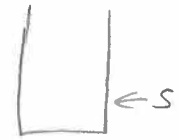
```
stack();
```

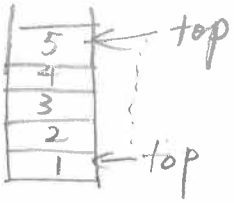
class stack operator <stack>

```
bool empty() const;
void pop();
void push(const T& item);
int size() const;
T& top();
const T& top() const;
```

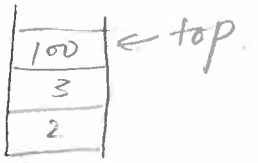
Examples

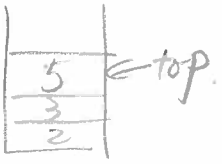
1.
```
#include <stack>

stack<int> s;                    //
int i;
for (i=1; i<=5; i++)
        s.push(i);

cout << s.size() << endl;
while (!s.empty())
{
        cout << s.top() << " ";
        s.pop();
}
cout << endl;
```





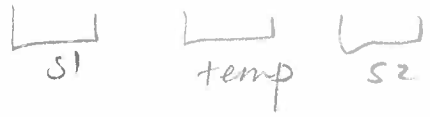
2.
```
stack<int> s;

s.push(2);
s.push(3);
s.push(100);

cout << s.top() << endl;

s.top() = 5;

cout << s.top() << endl;
```

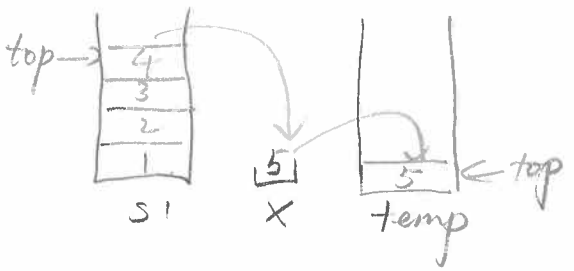


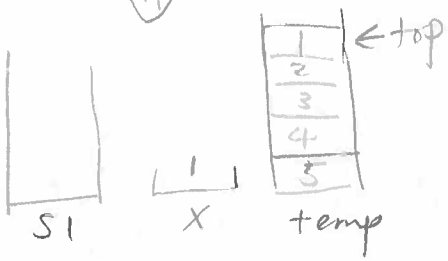


3.

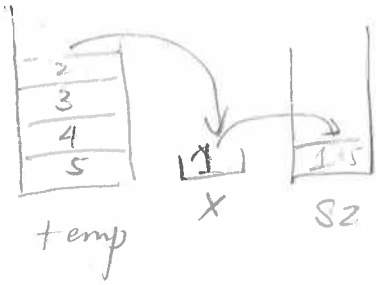
```
stack<int>  s1, temp, s2;
int x;

for (int i=0; i<5; i++)
    s1.push(i);

while (!s1.empty())
{
    x = s1.top();
    s1.pop();
    temp.push(x);
}
```








```
while (!temp.empty())
{
    x = temp.top();
    temp.pop();
    s2.push(x);
}
```



* temp stack is used to manipulate the stack.

d_stack.h

```
template <typename T>
class miniStack          //
{
    public:
        miniStack();
        void push(const T& item);
        void pop();
        T& top();
        const T& top() const;
        bool empty() const;
        int size() const;
    private:
        vector<T> stackVector;
};

template <typename T>
miniStack::miniStack()
{
    stackVector.resize(0);
}
template <typename T>
void miniStack::push(const T& item)
{
    stackVector.push_back(item);

}
template <typename T>
void miniStack::pop()
{
    if (empty())
        throw underflowError("stack empty");
    stackVector.pop_back();
}
```