



# 게임 자료구조와 알고리즘

## -CHAPTER17-

SOULSEEK



# 목차

---

**1. deque** – 시퀀스 컨테이너

**2. list** – 시퀀스 컨테이너



# DEQUE

# 1.DEQUE

## 특징

- 배열 기반의 컨테이너이다.
- 임의의 접근 반복자를 지원한다.
- **vector**와 다르게 원소가 메모리 블록 한군데가 아닌 여러 블록으로 나뉘어 저장된다.
- 앞쪽과 뒤쪽으로 모두 추가, 삭제 할 수 있다.
- **vector**보다 조금 효율적이다.

## 인터페이스

템플릿 형식

**Template<typename T, typename Allocator = allocator<T>>**

**class deque**

생성자	
<b>Deque dq</b>	<b>Dq</b> 는 빈 컨테이너이다.
<b>Deque dq(n)</b>	<b>Dq</b> 는 기본값으로 초기화된 <b>n</b> 개의 원소를 갖는다.
<b>Deque dq(n, x)</b>	<b>Dq</b> 는 <b>x</b> 값으로 초기화된 <b>n</b> 개의 원소를 갖는다.
<b>Deque dq(dq2)</b>	<b>Dq</b> 는 <b>dq2</b> 컨테이너의 복사본이다(복사 생성자 호출)
<b>Deque dq(b, e)</b>	<b>Dq</b> 는 반복자 구간 <b>[b, e)</b> 로 초기화된 원소를 갖는다.

# 1.DEQUE

## 연산자

<b>dq1 == dq2</b>	<b>dq1</b> 과 <b>dq2</b> 의 모든 원소가 같은가?(bool형식)
<b>dq1 != dq2</b>	<b>dq1</b> 과 <b>dq2</b> 의 모든 원소 중 하나라도 다른 원소가 있는가?(bool형식)
<b>dq1 &lt; dq2</b>	문자열 비교처럼 <b>dq2</b> 가 <b>dq1</b> 보다 큰가?(bool형식)
<b>dq1 &lt;= dq2</b>	문자열 비교처럼 <b>dq2</b> 가 <b>dq1</b> 보다 크거나 같은가?(bool형식)
<b>dq1 &gt; dq2</b>	문자열 비교처럼 <b>dq1</b> 이 <b>dq2</b> 보다 큰가?(bool형식)
<b>dq1 &gt;= dq2</b>	문자열 비교처럼 <b>dq1</b> 이 <b>dq2</b> 보다 크거나 같은가?(bool형식)
<b>dq[i]</b>	<b>sq</b> 의 <b>i</b> 번째 원소를 참조한다( <b>const</b> , 비 <b>const</b> 버전이 있으며 범위 점검이 없음)

# 1.DEQUE

## 멤버 형식

<b>allocator_type</b>	메모리 관리자 형식
<b>const_iterator</b>	<b>const</b> 반복자 형식
<b>const_pointer</b>	<b>const value_type*</b> 형식
<b>const_reference</b>	<b>const value_type&amp;</b> 형식
<b>const_reverse_iterator</b>	<b>const</b> 역 반복자 형식
<b>difference_type</b>	두 반복자 차이의 형식
<b>iterator</b>	반복자 형식
<b>pointer</b>	<b>value_type*</b> 형식
<b>reference</b>	<b>value_type&amp;</b> 형식
<b>reverse_iterator</b>	역 반복자 형식
<b>size_type</b>	첨자( <b>index</b> )나 원소의 개수 등의 형식
<b>value_type</b>	원소의 형식

# 1.DEQUE

## 멤버 함수

<b>dq.assign</b>	<b>dq</b> 에 <b>x</b> 값으로 <b>n</b> 개의 원소를 할당한다.
<b>dq.assign(b, e)</b>	<b>dq</b> 를 반복자 구간 <b>[b, e)</b> 로 할당한다.
<b>dq.at(i)</b>	<b>dq</b> 의 <b>i</b> 번째 원소를 참조한다( <b>const</b> , 비 <b>const</b> 버전이 있으며 범위 점검 포함)
<b>dq.back()</b>	<b>dq</b> 의 마지막 원소를 참조한다.( <b>const</b> , 비 <b>const</b> 버전이 있음)
<b>P=dq.begin()</b>	<b>p</b> 는 <b>dq</b> 의 첫 원소를 가리키는 반복자다( <b>const</b> , 비 <b>const</b> 버전이 있음)
<b>dq.clear()</b>	<b>dq</b> 의 모든 원소를 제거한다.
<b>dq.empty()</b>	<b>dq</b> 가 비었는지 조사한다.
<b>p=dq.end()</b>	<b>p</b> 는 <b>dq</b> 의 끝을 표시하는 반복자다( <b>const</b> , 비 <b>const</b> 버전이 있음)
<b>q=dq.erase(p)</b>	<b>p</b> 가 가리키는 원소를 제거한다. <b>q</b> 는 다음 원소를 가리킨다.
<b>q=dq.erase(b, e)</b>	반복자 구간 <b>[b, e)</b> 의 모든 원소를 제거한다. <b>q</b> 는 다음원소를 가리킨다.
<b>dq.front()</b>	<b>dq</b> 의 첫 번째 원소를 참조한다.( <b>const</b> , 비 <b>const</b> 버전이 있음)
<b>q=dq.insert(p, x)</b>	<b>p</b> 가 가리키는 위치에 <b>x</b> 값을 삽입한다. <b>q</b> 는 삽입한 원소를 가리키는 반복자
<b>dq.insert(p, n, x)</b>	<b>p</b> 가 가리키는 위치에 <b>n</b> 개의 <b>x</b> 값을 삽입한다.
<b>dq.insert(p, b, e)</b>	<b>p</b> 가 가리키는 위치에 반복자 구간 <b>[b, e)</b> 의 원소를 삽입한다.

# 1.DEQUE

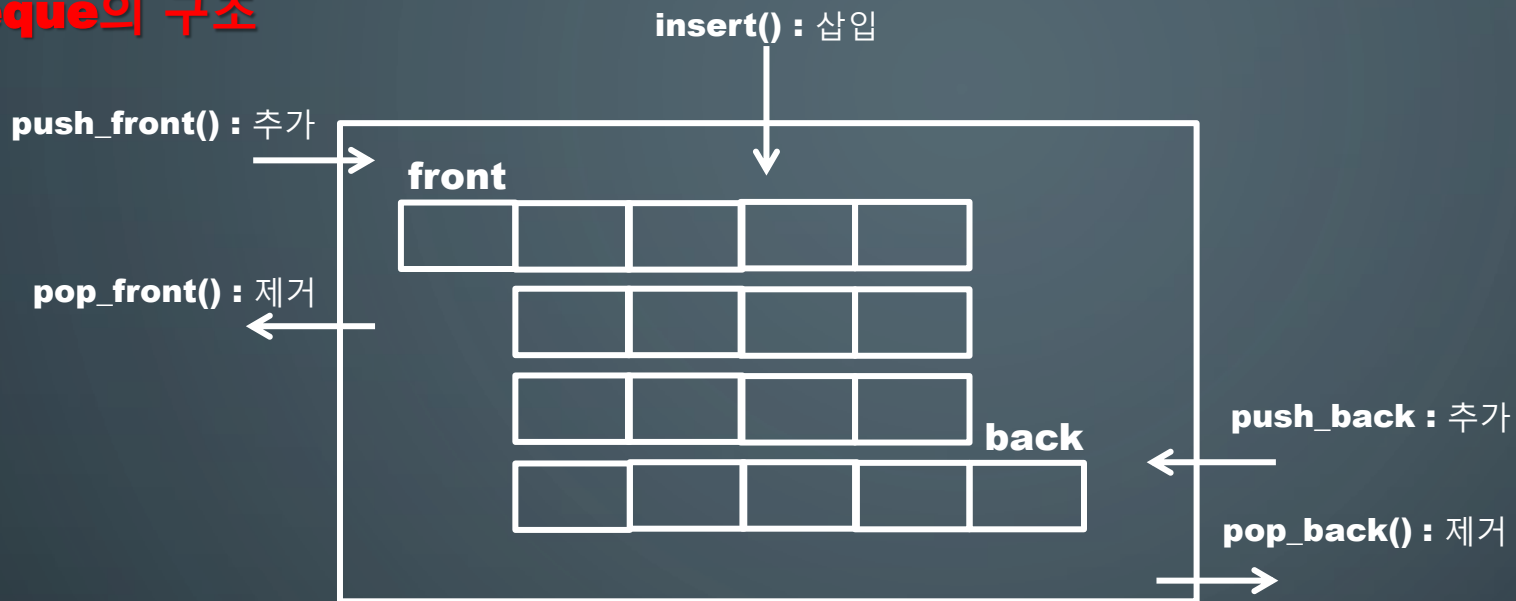
## 멤버 함수

<b>x=dq.max_size()</b>	<b>X</b> 는 <b>dq</b> 가 담을 수 있는 최대 원소의 개수(메모리의 크기)다.
<b>dq.pop_back()</b>	<b>dq</b> 는 마지막 원소를 제거한다.
<b>dq.pop_front()</b>	<b>dq</b> 의 첫 원소를 제거한다.
<b>dq.push_back(x)</b>	<b>dq</b> 의 끝에 <b>x</b> 를 추가한다.
<b>dq.push_front(x)</b>	<b>dq</b> 의 앞쪽에 <b>x</b> 를 추가한다.
<b>p=dq.rbegin()</b>	<b>p</b> 는 <b>dq</b> 의 역 순차열의 첫 원소를 가리키는 반복자다.
<b>p=dq.rend()</b>	<b>p</b> 는 <b>dq</b> 의 역 순차의 끝을 표시하는 반복자다.
<b>dq.rsize(n)</b>	<b>dq</b> 의 크기를 <b>n</b> 으로 변경하고 확장되는 공간의 값을 기본값으로 초기화한다.
<b>dq.rsize(n, x)</b>	<b>dq</b> 의 크기를 <b>n</b> 으로 변경하고 확장되는 공간의 값을 <b>x</b> 값으로 초기화한다.
<b>dq.size()</b>	<b>dq</b> 원소의 개수다.
<b>dq.swap(dq2)</b>	<b>dq</b> 와 <b>dq2</b> 를 <b>swap</b> 한다.



# 1.DEQUE

## deque의 구조



```
void main()
{
    deque<int> dq;

    for (deque<int>::size_type i = 0; i < 10; ++i)
        dq.push_back((i + 1) * 10);

    for (deque<int>::size_type i = 0; i < dq.size(); ++i)
        cout << dq[i] << ' ';
    cout << endl;
}
```

# 1.DEQUE

size : 0

size : 1



push\_back(10)  
메모리 할당

size : 2



push\_back(20)

size : 3



push\_back(30)

size : 4



push\_back(40)

size : 5



push\_back(50)

메모리 할당

# 1.DEQUE

## vector와 deque의 비교

```
void main()
```

```
{
```

```
    vector<int> v(4, 100); //100으로 초기화한 4개의 원소를 갖는 컨테이너 v
```

```
    deque<int> dq(4, 100); //100으로 초기화한 4개의 원소를 갖는 컨테이너 dq
```

```
    v.push_back(200); // v에 200 추가
```

```
    dq.push_back(200); // dq에 200 추가
```

```
    for (vector<int>::size_type i = 0; i < v.size(); ++i)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
    for (deque<int>::size_type i = 0; i < dq.size(); ++i)
```

```
        cout << dq[i] << " ";
```

```
    cout << endl;
```

```
}
```

**vector**은 할당이 넘어서면 원소들을 재할당한 뒤 복사를 하는데 **deque**는 새로 추가 할당만 하고 새로운 원소만 추가한다.

# 1.DEQUEUE

## push\_front를 사용하는 예제..

```
void main()
{
    deque<int> dq;

    dq.push_back(10);
    dq.push_back(20);
    dq.push_back(30);
    dq.push_back(40);
    dq.push_back(50);

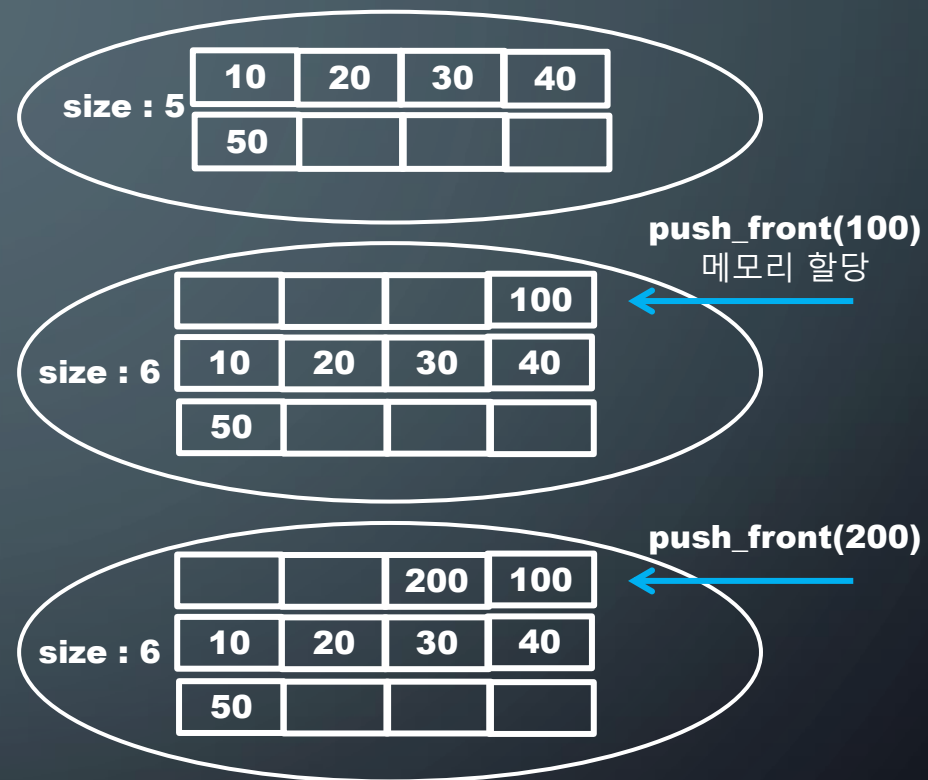
    for (deque<int>::size_type i = 0; i < dq.size(); ++i)
        cout << dq[i] << " ";

    cout << endl;

    dq.push_front(100); // 앞쪽에 추가합니다.
    dq.push_front(200);

    for (deque<int>::size_type i = 0; i < dq.size(); ++i)
        cout << dq[i] << " ";

    cout << endl;
}
```



# 1.DEQUE

## deque의 반복자 사용 예제..

```
void main()
{
    deque<int> dq;

    dq.push_back( 10 );
    dq.push_back( 20 );
    dq.push_back( 30 );
    dq.push_back( 40 );
    dq.push_back( 50 );

    deque<int>::iterator iter;
    for(iter = dq.begin(); iter != dq.end() ; ++iter)
        cout << *iter << " ";

    cout << endl;

    iter = dq.begin()+2; //반복자에 +2합니다.
    cout << *iter << endl;

    iter += 2; //반복자에 +2합니다.
    cout << *iter << endl;

    iter -= 3; //반복자에 -3합니다.
    cout << *iter << endl;
}
```

# 1.DEQUE

## Deque의 insert() 사용 예제..

```
void main( )
{
    deque<int> dq;

    for(int i = 0 ; i < 10 ; i++)
        dq.push_back( (i+1)*10 );

    deque<int>::iterator iter;
    deque<int>::iterator iter2;

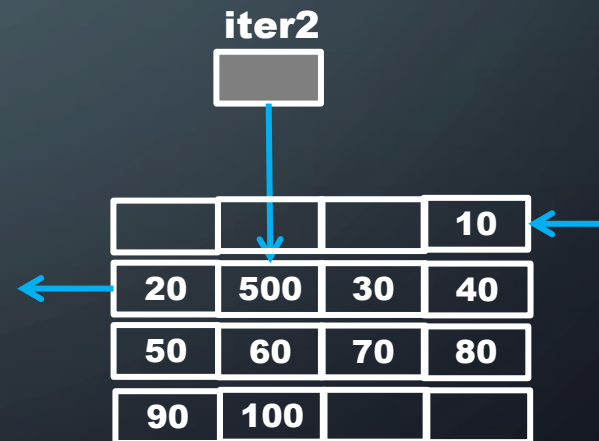
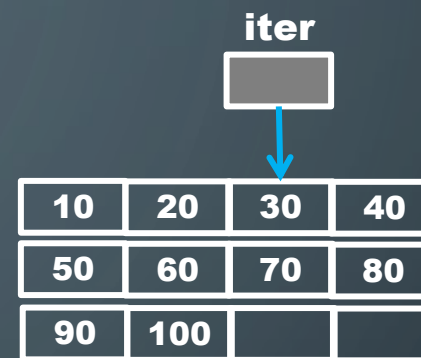
    for(iter = dq.begin(); iter != dq.end(); ++iter)
        cout << *iter << ' ';

    cout << endl;

    iter = dq.begin()+2;
    iter2 = dq.insert( iter , 500);
    cout << *iter2 << endl;

    for(iter = dq.begin(); iter != dq.end(); ++iter)
        cout << *iter << ' ';

    cout << endl;
}
```



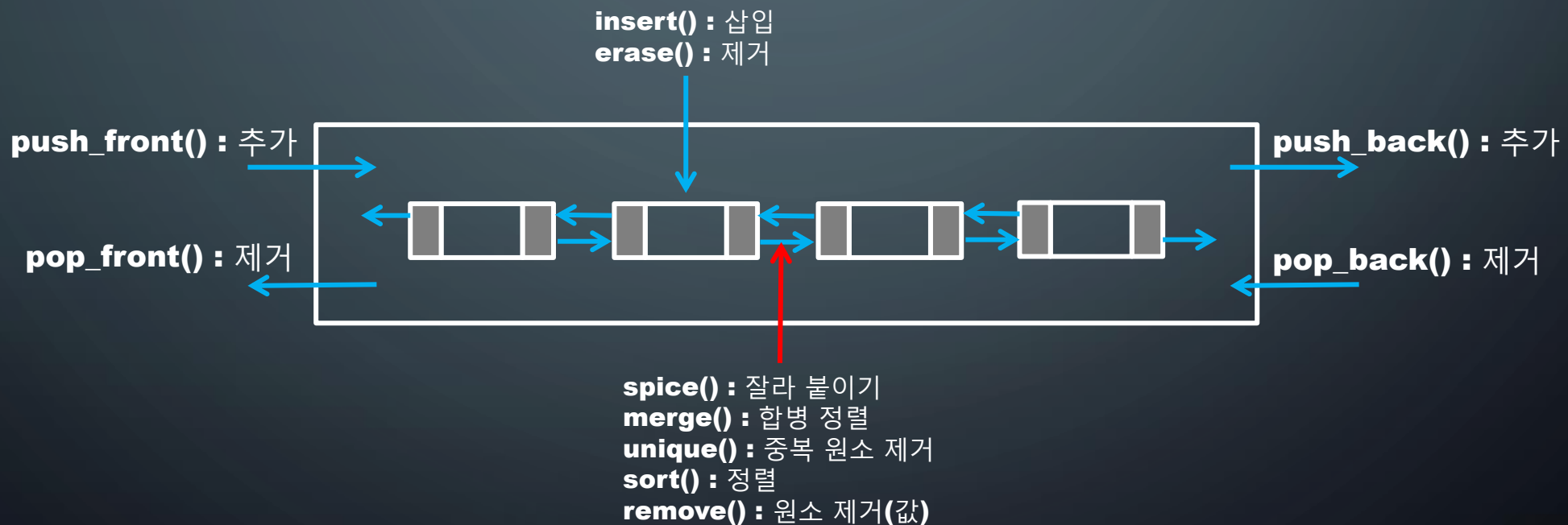
The image features a dark blue background with a subtle radial gradient. In the four corners, there are decorative white line art elements resembling electronic circuit traces or a stylized city skyline. These elements consist of thin lines and small circles, creating a modern, technological aesthetic.

# LIST

# 1.LIST

## 특징

- 노드 기반의 컨테이너
- 이중 연결 리스트 기반
- 인덱스 접근 연산자를 지원하지 않는다
- 다른 **list**와 결합할 때 좋은 컨테이너
- 순차열 중간에 삽입, 제거가 빈번하게 발생하며 원소의 상대적인 순서가 중요하면 **list** 컨테이너를 사용





# 1.LIST

## 인터페이스

### 템플릿 형식

Template<typename T, typename Allocator = allocator<T>>

class list

#### 생성자

<b>list lt</b>	<b>lt</b> 는 빈 컨테이너
<b>list lt(n)</b>	<b>lt</b> 는 기본값으로 초기화된 <b>n</b> 개의 원소를 갖는다.
<b>list lt(n, x)</b>	<b>lt</b> 는 <b>x</b> 값으로 초기화된 <b>n</b> 개의 원소를 갖는다.
<b>list lt(lt2)</b>	<b>lt</b> 는 <b>lt2</b> 컨테이너의 복사본이다(복사 생성자)
<b>list lt(b, e)</b>	<b>lt</b> 는 반복자 구간( <b>b, e</b> )로 초기화된 원소를 갖는다.

#### 연산자

<b>Lt1 == lt2</b>	<b>Lt1</b> 과 <b>lt2</b> 의 모든 원소가 같은가?(bool형식)
<b>Lt1 != lt2</b>	<b>Lt1</b> 과 <b>lt2</b> 의 모든 원소 중 하나라도 다른 원소가 있는가?(bool형식)
<b>Lt1 &lt; lt2</b>	문자열 비교처럼 <b>lt2</b> 가 <b>lt1</b> 보다 큰가?(bool형식)
<b>Lt1 &lt;= lt2</b>	문자열 비교처럼 <b>lt2</b> 가 <b>lt1</b> 보다 크거나 같은가?(bool형식)
<b>Lt1 &gt; lt2</b>	문자열 비교처럼 <b>lt1</b> 이 <b>lt2</b> 보다 큰가?(bool형식)
<b>Lt1 &gt;= v2</b>	문자열 비교처럼 <b>lt1</b> 이 <b>lt1</b> 보다 크거나 같은가?(bool형식)

# 1.LIST

멤버 형식	
<b>allocator_type</b>	메모리 관리자 형식
<b>const_iterator</b>	<b>const</b> 반복자 형식
<b>const_pointer</b>	<b>const value_type*</b> 형식
<b>const_reference</b>	<b>const value_type&amp;</b> 형식
<b>const_reverse_iterator</b>	<b>const</b> 역 반복자 형식
<b>difference_type</b>	두 반복자 차이의 형식
<b>iterator</b>	반복자 형식
<b>pointer</b>	<b>value_type*</b> 형식
<b>reference</b>	<b>value_type&amp;</b> 형식
<b>reverse_iterator</b>	역 반복자 형식
<b>size_type</b>	첨자( <b>index</b> )나 원소의 개수 등의 형식
<b>value_type</b>	원소의 형식

# 1.LIST

## 멤버 함수

<b>lt.assign(n, x)</b>	<b>lt</b> 에 <b>x</b> 값으로 <b>n</b> 개의 원소를 할당한다.
<b>lt.assign(b, e)</b>	<b>lt</b> 를 반복자 구간[ <b>b, e</b> )로 할당한다.
<b>lt.back()</b>	<b>lt</b> 의 마지막 원소를 참조한다.( <b>const</b> , 비 <b>const</b> 버전 있음)
<b>p=lt.begin()</b>	<b>p</b> 는 <b>lt</b> 의 첫 원소를 가리키는 반복자다( <b>const</b> , 비 <b>const</b> 버전이 있음)
<b>lt.clear()</b>	<b>lt</b> 의 모든 원소를 제거한다.
<b>lt.empty()</b>	<b>lt</b> 가 비었는지 조사한다.
<b>p=lt.end()</b>	<b>p</b> 는 <b>lt</b> 의 끝을 표식하는 반복자다( <b>const</b> , 비 <b>const</b> 버전이 있음)
<b>q=lt.erase(p)</b>	<b>p</b> 가 가리키는 원소를 제거한다. <b>q</b> 는 다음 원소를 가리킨다.
<b>q=lt.erase(b,e)</b>	반복자 구간[ <b>b, e</b> )의 모든 원소를 제거한다. <b>q</b> 는 다음 원소다.
<b>lt.front()</b>	<b>lt</b> 의 첫 번째 원소를 참조한다( <b>const</b> , 비 <b>const</b> 버전이 있음)
<b>q=lt.insert(p, x)</b>	<b>p</b> 가 가리키는 위치에 <b>x</b> 값을 삽입한다. <b>q</b> 는 삽입한 원소를 가리키는 반복자다.
<b>lt.insert(p, n, x)</b>	<b>p</b> 가 가리키는 위치에 <b>n</b> 개의 <b>x</b> 값을 삽입한다.
<b>lt.insert(p, b, e)</b>	<b>p</b> 가 가리키는 위치에 반복자 구간[ <b>b, e</b> )의 원소를 삽입한다.
<b>x=lt.max_size()</b>	<b>x</b> 는 <b>lt</b> 가 담을 수 있는 최대 원소의 개수다.(메모리의 크기)
<b>lt.merge(lt2)</b>	<b>lt2</b> 를 <b>lt</b> 로 합병 정렬한다.(오름차순 : <b>less</b> )

# 1.LIST

## 멤버 함수

<b>lt.merge(lt2, pred)</b>	<b>lt2</b> 를 <b>lt</b> 로 합병 정렬한다. <b>pred</b> (조건자)를 기준으로 합병( <b>pred</b> 는 이항 조건자)
<b>lt.pop_back()</b>	<b>lt</b> 의 마지막 원소를 제거한다.
<b>lt.pop_front()</b>	<b>lt</b> 의 첫 원소를 제거한다.
<b>lt.push_back(x)</b>	<b>lt</b> 의 끝에 <b>x</b> 를 추가한다.
<b>lt.push_front(x)</b>	<b>lt</b> 의 앞쪽에 <b>x</b> 를 추가한다.
<b>p=lt.rbegin()</b>	<b>p</b> 는 <b>lt</b> 의 역 순차열의 첫 원소를 가리키는 반복자.( <b>const</b> , 비 <b>const</b> 버전 있음)
<b>lt.remove(x)</b>	<b>x</b> 원소를 모두 제거
<b>lt.remove_if(pred)</b>	<b>pred</b> (단항 조건자)가 ‘참’인 모든 원소를 제거한다.
<b>p=lt.rend()</b>	<b>p</b> 는 <b>lt</b> 의 역 순차열의 끝을 표시하는 반복자.( <b>const</b> , 비 <b>const</b> 버전 있음)
<b>lt.rsize()</b>	<b>lt</b> 의 크기를 <b>n</b> 으로 변경하고 확장되는 공간의 값을 기본값으로 초기화
<b>lt.rsize(n, x)</b>	<b>lt</b> 의 크기를 <b>n</b> 으로 변경하고 확장되는 공간의 값을 <b>x</b> 로 초기화
<b>lt.reverse()</b>	<b>lt</b> 순차열을 뒤집는다.
<b>lt.size()</b>	<b>lt</b> 원소의 개수다.
<b>lt.sort()</b>	<b>lt</b> 의 모든 원소를 오름차순( <b>less</b> )으로 정렬한다.
<b>lt.sort(pred)</b>	<b>lt</b> 의 모든 원소를 <b>pred</b> (조건자)를 기준으로 정렬한다.( <b>pred</b> 는 이항 조건자)

# 1.LIST

## 멤버 함수

<b>lt.splice(p, lt2)</b>	<b>P</b> 가 가리키는 위치에 <b>lt2</b> 의 모든 원소를 잘라 붙인다.
<b>lt.splice(p, lt2, q)</b>	<b>P</b> 가 가리키는 위치에 <b>lt2</b> 의 <b>q</b> 가 가리키는 원소를 잘라 붙인다.
<b>lt.splice(p, lt2, b, e)</b>	<b>P</b> 가 가리키는 위치에 <b>lt2</b> 의 순차열 <b>[b, e)</b> 을 잘라 붙인다.
<b>lt.swap(lt2)</b>	<b>Lt</b> 와 <b>lt2</b> 를 <b>swap</b> 한다.
<b>lt.unique()</b>	인접한 원소의 값이 같다면 유일한 원소의 순차열로 만든다.
<b>lt.unique(pred)</b>	인접한 원소가 <b>pred</b> (이랑 조건자)의 기준에 맞다면 유일한 원소의 수타열로 만든다.

# 1.LIST

## list의 push\_back, push\_front, 반복자 예제..

```
void main()
```

```
{
```

```
    list<int> lt;
```

```
    lt.push_back(10);
```

```
    lt.push_back(20);
```

```
    lt.push_back(30);
```

```
    lt.push_back(40);
```

```
    lt.push_back(50);
```

- 양방향 반복자를 제공하기 때문에 ++연산과 \*연산 !=연산으로 list의 모든 원소의 출력이 가능하다
- 순차열 중간에 원소를 삽입, 제거하더라도 상수 시간 복잡도의 수행 성능을 보인다.
- 노드 연결만 다시 하기때문에 배열 기반 컨테이너 보다 좋은 성능을 발휘한다.

```
    list<int>::iterator iter;
```

```
    for (iter = lt.begin(); iter != lt.end(); ++iter)
```

```
        cout << *iter << ' ';
```

```
    cout << endl;
```

```
    lt.push_front(100);
```

```
    lt.push_front(200);
```

```
    for (iter = lt.begin(); iter != lt.end(); ++iter)
```

```
        cout << *iter << ' ';
```

```
    cout << endl;
```

```
}
```

# 1.LIST

## Insert()와 erase() 사용 예제

```
void main()
{
    list<int> lt;

    lt.push_back(10);
    lt.push_back(20);
    lt.push_back(30);
    lt.push_back(40);
    lt.push_back(50);

    list<int>::iterator iter = lt.begin();
    list<int>::iterator iter2;
    ++iter;
    ++iter;

    iter2 = lt.erase(iter); //iter(30)의 원소를 제거합니다.

    for (iter = lt.begin(); iter != lt.end(); ++iter)
        cout << *iter << ' ';
    cout << endl;

    cout << "iter2 : " << *iter2 << endl;

    iter = iter2;
    iter2 = lt.insert(iter, 300); //iter2(40)이 가리키는 위치에 300을 삽입합니다.
    for (iter = lt.begin(); iter != lt.end(); ++iter)
        cout << *iter << ' ';
    cout << endl;

    cout << "iter2 : " << *iter2 << endl;
}
```

# 1.LIST

## List와 vector의 삽입 동작 차이..

```
void main()
```

```
{
```

```
    vector<int> v;  
    list<int> lt;
```

```
    v.push_back(10);  
    v.push_back(20);  
    v.push_back(30);  
    v.push_back(40);  
    v.push_back(50);
```

```
    lt.push_back(10);  
    lt.push_back(20);  
    lt.push_back(30);  
    lt.push_back(40);  
    lt.push_back(50);
```

```
    vector<int>::iterator viter = v.begin();  
    ++viter; // 20원소를 가리킴  
    list<int>::iterator liter = lt.begin();  
    ++liter; // 20원소를 가리킴
```

```
    viter = v.insert(viter, 600); // v의 두 번째 요소로 삽입  
    liter = lt.insert(liter, 600); // lt의 두 번째 요소로 삽입
```

```
    cout << "vector: " << *viter << endl;  
    cout << "list: " << *liter << endl;
```

```
    cout << "vector : ";  
    for (viter = v.begin(); viter != v.end(); ++viter)  
        cout << *viter << ' ';  
    cout << endl;
```

```
    cout << "list : ";  
    for (liter = lt.begin(); liter != lt.end(); ++liter)  
        cout << *liter << ' ';  
    cout << endl;
```

```
}
```



# 1.LIST

## **remove()** 예제..

```
void main( )
{
    list<int> lt;

    lt.push_back(10);
    lt.push_back(20);
    lt.push_back(30);
    lt.push_back(10);
    lt.push_back(40);
    lt.push_back(50);
    lt.push_back(10);
    lt.push_back(10);

    list<int>::iterator iter;
    for(iter = lt.begin(); iter != lt.end(); ++iter)
        cout << *iter << ' ';
    cout << endl;

    lt.remove(10); // 10 원소의 노드를 모두 제거
    for(iter = lt.begin(); iter != lt.end(); ++iter)
        cout << *iter << ' ';
    cout << endl;
}
```

# 1.LIST

## **remove\_if()** 사용 예제..

```
bool Predicate(int n) // 단항 조건자
{
    return 10 <= n && n <= 30;
}
```

```
void main( )
{
    list<int> lt;

    lt.push_back(10);
    lt.push_back(20);
    lt.push_back(30);
    lt.push_back(40);
    lt.push_back(50);
    lt.push_back(10);

    list<int>::iterator iter;
    for(iter = lt.begin(); iter != lt.end(); ++iter)
        cout << *iter << ' ';
    cout << endl;

    lt.remove_if(Predicate); // 조건자가 참인 모든 원소를 제거합니다.
    for(iter = lt.begin(); iter != lt.end(); ++iter)
        cout << *iter << ' ';
    cout << endl;
}
```

# 1.LIST

## splice() 사용 예제..

```
void main( )
```

```
{
```

```
    list<int> lt1;
```

```
    list<int> lt2;
```

```
    lt1.push_back(10);
```

```
    lt1.push_back(20);
```

```
    lt1.push_back(30);
```

```
    lt1.push_back(40);
```

```
    lt1.push_back(50);
```

```
    lt2.push_back(100);
```

```
    lt2.push_back(200);
```

```
    lt2.push_back(300);
```

```
    lt2.push_back(400);
```

```
    lt2.push_back(500);
```

```
    list<int>::iterator iter;    }
```

```
    cout << "lt1: ";
```

```
    for(iter = lt1.begin(); iter != lt1.end(); ++iter)
```

```
        cout << *iter << ' ';
```

```
    cout << endl;
```

```
    cout << "lt2: ";
```

```
    for(iter = lt2.begin(); iter != lt2.end(); ++iter)
```

```
        cout << *iter << ' ';
```

```
    cout << endl << "======" << endl;
```

```
    iter = lt1.begin();
```

```
    ++iter;
```

```
    ++iter; // 30 원소의 위치를 가리킴
```

```
    lt1.splice(iter, lt2); //iter가 가리키는 위치에 lt2의 모든 원소를 잘라 붙임
```

```
    cout << "lt1: ";
```

```
    for(iter = lt1.begin(); iter != lt1.end(); ++iter)
```

```
        cout << *iter << ' ';
```

```
    cout << endl;
```

```
    cout << "lt2: ";
```

```
    for(iter = lt2.begin(); iter != lt2.end(); ++iter)
```

```
        cout << *iter << ' ';
```

```
    cout << endl;
```

리스트에 splice를 통해 한 쪽 리스트에 붙이게 되면 붙이고 난 후의 옮겨진 리스트는 빈 컨테이너가 된다.

# 1.LIST

## splice() 사용 예제..

```
void main( )
```

```
{
```

```
    list<int> lt1;
```

```
    list<int> lt2;
```

```
    lt1.push_back(10);
```

```
    lt1.push_back(20);
```

```
    lt1.push_back(30);
```

```
    lt1.push_back(40);
```

```
    lt1.push_back(50);
```

```
    lt2.push_back(100);
```

```
    lt2.push_back(200);
```

```
    lt2.push_back(300);
```

```
    lt2.push_back(400);
```

```
    lt2.push_back(500);
```

```
    list<int>::iterator iter1;
```

```
    list<int>::iterator iter2;
```

```
    cout << "lt1: ";
```

```
    for(iter1 = lt1.begin(); iter1 != lt1.end(); ++iter1)
```

```
        cout << *iter1 << ' ';
```

```
    cout << endl;
```

```
    cout << "lt2: ";
```

```
    for(iter2 = lt2.begin(); iter2 != lt2.end(); ++iter2)
```

```
        cout << *iter2 << ' ';
```

```
    cout << endl << "=====" << endl;
```

```
    iter1 = lt1.begin();
```

```
    ++iter1;
```

```
    ++iter1; // 30 원소의 위치를 가리킴
```

```
    iter2 = lt2.begin();
```

```
    ++iter2; // 200 원소의 위치를 가리킴
```

```
    //iter1이 가리키는 위치에 iter2가 가리키는 위치의 lt2의 원소를 잘라 붙임
```

```
    lt1.splice(iter1, lt2, iter2);
```

```
    cout << "lt1: ";
```

```
    for(iter1 = lt1.begin(); iter1 != lt1.end(); ++iter1)
```

```
        cout << *iter1 << ' ';
```

```
    cout << endl;
```

```
    cout << "lt2: ";
```

```
    for(iter2 = lt2.begin(); iter2 != lt2.end(); ++iter2)
```

```
        cout << *iter2 << ' ';
```

```
    cout << endl << "=====" << endl;
```

```
    //lt1.end()가 가리키는 위치에 순차열 [lt2.begin(), lt2.end())를 잘라 붙임
```

```
    lt1.splice(lt1.end(), lt2, lt2.begin(), lt2.end());
```

```
    cout << "lt1: ";
```

```
    for(iter1 = lt1.begin(); iter1 != lt1.end(); ++iter1)
```

```
        cout << *iter1 << ' ';
```

```
    cout << endl;
```

```
    cout << "lt2: ";
```

```
    for(iter2 = lt2.begin(); iter2 != lt2.end(); ++iter2)
```

```
        cout << *iter2 << ' ';
```

```
    cout << endl;
```

```
}
```

# 1.LIST

## reverse 사용 예제..

```
void main( )
{
    list<int> lt;

    lt.push_back(10);
    lt.push_back(20);
    lt.push_back(30);
    lt.push_back(40);
    lt.push_back(50);

    list<int>::iterator iter;
    for(iter = lt.begin(); iter != lt.end(); ++iter)
        cout << *iter << ' ';

    cout << endl;

    lt.reverse();
    for(iter = lt.begin(); iter != lt.end(); ++iter)
        cout << *iter << ' ';

    cout << endl;
}
```

# 1.LIST

**unique** 사용 예제..

```
void main( )
{
    list<int> lt;

    lt.push_back(10);
    lt.push_back(10);
    lt.push_back(20);
    lt.push_back(30);
    lt.push_back(30);
    lt.push_back(30);
    lt.push_back(40);
    lt.push_back(50);
    lt.push_back(10);

    list<int>::iterator iter;
    for(iter = lt.begin(); iter != lt.end(); ++iter)
        cout << *iter << ' ';
    cout << endl;

    lt.unique();
    for(iter = lt.begin(); iter != lt.end(); ++iter)
        cout << *iter << ' ';
    cout << endl;
}
```

서로 인접한 원소들과 비교를 한 후 같은 것이 있다면 모두 소거하고 한 개만 남긴다.

# 1.LIST

## sort 사용 예제..

```
void main( )
{
    list<int> lt;

    lt.push_back(20);
    lt.push_back(10);
    lt.push_back(50);
    lt.push_back(30);
    lt.push_back(40);

    list<int>::iterator iter;
    for(iter = lt.begin(); iter != lt.end(); ++iter)
        cout << *iter << ' ';

    cout << endl;

    lt.sort( ); // 오름차순( less, < 연산) 정렬
    for(iter = lt.begin(); iter != lt.end(); ++iter)
        cout << *iter << ' ';

    cout << endl;
}
```

**List**는 임의의 접근자를 쓸 수 없기 때문에 **slt**이 제공하는 **sort** 알고리즘을 사용할 수 없고 스스로 멤버 함수로 가지고 있고 **퀵 정렬**을 이용한 **sort** 함수이다.



# 1.LIST

## sort 사용 예제..

```
struct Greater
```

```
{
```

```
    bool operator () (int left, int right) const
```

```
    {
```

```
        return left > right;
```

```
    }
```

```
};
```

```
void main()
```

```
{
```

```
    list<int> lt;
```

```
    lt.push_back(20);
```

```
    lt.push_back(10);
```

```
    lt.push_back(50);
```

```
    lt.push_back(30);
```

```
    lt.push_back(40);
```

```
    list<int>::iterator iter;
```

```
    for (iter = lt.begin(); iter != lt.end(); ++iter)
```

```
        cout << *iter << ' ';
```

```
    cout << endl;
```

```
    lt.sort(greater<int>()); // 조건자 greater를 사용하여 내림차순 정렬
```

```
    for (iter = lt.begin(); iter != lt.end(); ++iter)
```

```
        cout << *iter << ' ';
```

```
    cout << endl;
```

```
lt.sort(less<int>()); // 조건자 less를 사용하여 다시 오름차순 정렬
```

```
for (iter = lt.begin(); iter != lt.end(); ++iter)
```

```
cout << *iter << ' ';
```

```
cout << endl;
```

```
lt.sort(Greater()); // 사용자 정의 조건자를 사용하여 내림차순 정렬
```

```
for (iter = lt.begin(); iter != lt.end(); ++iter)
```

```
cout << *iter << ' ';
```

```
cout << endl;
```



# 1.LIST

## Merage 사용 예제..

```
void main()
{
    list<int> lt1;
    list<int> lt2;

    lt1.push_back(10);
    lt1.push_back(20);
    lt1.push_back(30);
    lt1.push_back(40);
    lt1.push_back(50);

    lt2.push_back(25);
    lt2.push_back(35);
    lt2.push_back(60);

    list<int>::iterator iter;
    cout << "lt1: ";
    for (iter = lt1.begin(); iter != lt1.end(); ++iter)
        cout << *iter << ' ';
    cout << endl;

    cout << "lt2: ";
    for (iter = lt2.begin(); iter != lt2.end(); ++iter)
        cout << *iter << ' ';
    cout << endl << "=====" << endl;
}
```

**lt1.merge(lt2);** // lt2를 lt1으로 합병 정렬합니다. 정렬 기준은 **less**

```
cout << "lt1: ";
for (iter = lt1.begin(); iter != lt1.end(); ++iter)
    cout << *iter << ' ';
cout << endl;
```

```
cout << "lt2: ";
for (iter = lt2.begin(); iter != lt2.end(); ++iter)
    cout << *iter << ' ';
cout << endl;
```

**Merge**는 서로 정렬된 순서의 기준이 다르다면 정렬을 시켜준 뒤 **merge**를 하게 되면 오류가 발생한다.

# 1.LIST

## merge 사용 예제..

```
void main()
```

```
{
```

```
    list<int> lt1;
```

```
    list<int> lt2;
```

```
    lt1.push_back(50);
```

```
    lt1.push_back(40);
```

```
    lt1.push_back(30);
```

```
    lt1.push_back(20);
```

```
    lt1.push_back(10);
```

```
    //lt1과 lt2의 정렬 방식이 다르므로 오류
```

```
    //lt2.push_back(25);
```

```
    //lt2.push_back(35);
```

```
    //lt2.push_back(60);
```

```
}
```

```
    // lt1과 lt2는 정렬 방식이 같다.
```

```
    // greater 조건자( > 연산 ) 정렬 기준을 사용함
```

```
    lt2.push_back(60);
```

```
    lt2.push_back(35);
```

```
    lt2.push_back(25);
```

```
    list<int>::iterator iter;
```

```
    cout << "lt1: ";
```

```
    for (iter = lt1.begin(); iter != lt1.end(); ++iter)
```

```
        cout << *iter << ' ';
```

```
    cout << endl;
```

```
    cout << "lt2: ";
```

```
    for (iter = lt2.begin(); iter != lt2.end(); ++iter)
```

```
        cout << *iter << ' ';
```

```
    cout << endl << "=====" << endl;
```

```
    // lt2를 lt1으로 합병 정렬합니다.
```

```
    // 두 list의 정렬 기준이 > 연산인 greater라는 것을 predicate로 지정합니다.
```

```
    lt1.merge(lt2, greater<int>());
```

```
    cout << "lt1: ";
```

```
    for (iter = lt1.begin(); iter != lt1.end(); ++iter)
```

```
        cout << *iter << ' ';
```

```
    cout << endl;
```

```
    cout << "lt2: ";
```

```
    for (iter = lt2.begin(); iter != lt2.end(); ++iter)
```

```
        cout << *iter << ' ';
```

```
    cout << endl;
```