

본 PSet 은 저의 강의 경험과 학생들의 의견 및 강의에서 수집된 자료를 토대로 작성되었습니다. 본 PSet 에 문제가 있거나, 질문 혹은 의견이 있다면, 언제든지 알려 주시면 감사하겠습니다. 강의 개선에 많은 도움이 되겠습니다. idebtor@gmail.com

Lab listdbl: a doubly-linked list

내용

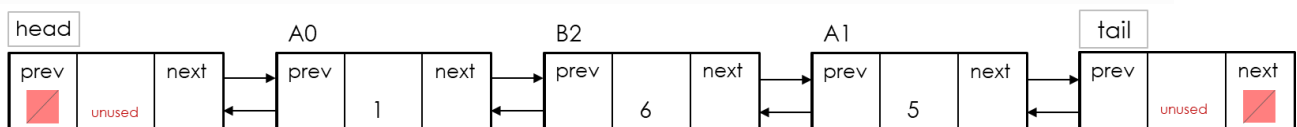
Warming-up	1
주요 작업 1: begin() and end()	2
주요 작업 2: erase() and insert()	3
Step 1: find(), more() & less()	4
Step 2: push commands	5
도우미 함수: rand_extended() in librand.a or rand.lib	5
Step 3: pop commands – pop_all()*	6
Step 4: show()*, half()	7
과제 제출	8
제출 파일 목록	9
마감 기한 & 배점	9

이 PSet 은 두 개의 센티널 노드로 이중 연결 리스트를 구현하는 것으로 구성됩니다. 주어진 프로그램 listdbl.cpp 를 완료하십시오. 다음 파일들이 제공됩니다.

1. listdbl.h - 수정 금지
2. driver.cpp - 수정 금지
3. listdbl.cpp - 뼈대 코드, 대부분의 코드를 이 파일에서 작성합니다
4. listdbl.exe, listdbl - 참고용 실행 파일, 버그가 있을 수 있습니다

Warming-up

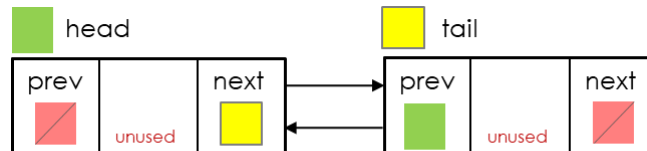
이 PSet 의 목표는 아래와 같이 두 개의 센티널 노드로 이중 연결된 리스트를 구현하는 것입니다:



이런 부가적인 노드를 **센티널 노드**라고 하며, 앞에 있는 노드는 **헤드** 노드, 끝에 있는 노드는 **테일** 노드라고 합니다. 이중 연결 리스트가 초기화되면 헤드 노드와 테일 노드가 생성됩니다. 센티널 노드를 사용하면 리스트가 비어 있을 때 또는 리스트의 헤드나 테일 부분에 노드를 삽입할 때와 같이 특수한 경우를 위한

코드가 따로 필요하지 않게 되어 insert, push/pop front & back 작업이 간소화됩니다. **이렇게 하면 코딩이 굉장히 간단해집니다.**

예를 들어, 헤드 노드를 사용하지 않는 경우, 첫 번째 노드를 제거하는 것이 특수한 경우입니다. 첫 번째 노드를 제거하면 리스트의 링크를 재설정해야 하고, 일반적인 제거 알고리즘은 노드를 제거하기 전에 제거할 노드의 이전 노드에 접근해야 합니다. 하지만 헤드 노드가 없으면 첫 번째 노드의 이전 노드는 존재하지 않으므로 이런 경우는 특수한 경우로 취급합니다. 다음 그림과 같이 헤드와 테일 노드만으로 빈 노드를 구성할 수 있습니다.



An **empty** doubly-linked list with sentinel nodes

Node 와 **List** 라는 다음 두 데이터 구조를 사용하여 두 개의 센티널 노드뿐 만 아니라 이중으로 연결된 노드를 유지할 수 있습니다.

```
struct Node {
    int    data;
    Node*  prev;
    Node*  next;
};

struct List {
    Node* head; //sentinel
    Node* tail; //sentinel
    int   size; //size of list, optional

    List() {
        head = new Node{};    tail = new Node{};
        head->next = tail;    tail->prev = head;
        size = 0;
    }
    ~List() {}
};

using pNode = Node*;
using pList = List*;
```

주요 작업 1: begin() and end()

The function **begin()** returns the first node that the head node points.

begin() 함수는 헤드 노드가 가리키는 첫 번째 노드를 반환합니다.

```
// Returns the first node which List::head points to in the container.
pNode begin(pList p) {
    return p->head->next;
}
```

end() 함수는 리스트의 지나간 -마지막- 노드를 참조하는 테일 노드를 반환합니다.

지나간 -마지막- 노드는 마지막 노드 뒤에 오는, **센티널로만 사용되는** 센티널 노드입니다. 다음 노드를 가리키지 않으므로 역참조할 수 없습니다. 작업을 반복하는 동안 사용할 방법으로 인해, 이 노드가 가리키는

노드를 포함하지 않습니다. 이 함수는 리스트의 모든 노드를 포함하는 범위를 지정하기 위해 주로 `List::begin` 과 함께 사용됩니다. STL 을 모방하는 것이라고 볼 수 있습니다. 컨테이너가 비어 있으면 이 함수는 `List::begin` 과 동일한 값을 반환합니다.

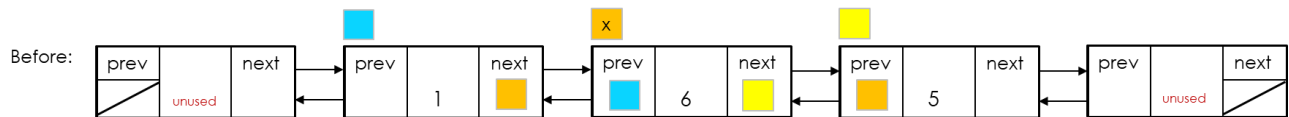
```
pNode end(pList p) {
    return p->tail;          // not tail->next
}
```

주요 작업 2: erase() and insert()

`erase()` 함수는 지정된 단일 노드 `x` 를 리스트에서 제거합니다. 이렇게 하면 컨테이너의 크기를 제거할 노드 한 개만큼 효과적으로 줄일 수 있습니다. 리스트의 전면, 후면 또는 중간 등 위치에 관계없이 노드를 효율적으로 삽입 및 제거할 수 있도록 특별하게 설계되었습니다.

노드 `x` 를 제거한다고 가정해봅시다. `erase()` 함수에는 아래와 같이 `x` 인수 하나만 존재합니다.

```
void erase(pNode x);
```



노드 `x` 제거 후 리스트는 다음과 같습니다.



노드 `x` (주황색)가 제거되었으므로 이제 노드(파란색)와 노드(노란색)를 연결해야 합니다. 즉, 첫 번째 오렌지(blue→next 또는 `x→prev→next`)는 노란색(`x→next`)으로 설정하고, 두 번째 오렌지(yellow→prev 또는 `x->next->prev`)는 `x→prev` 로 설정해야 합니다.

```
void erase(pNode x) {
    x->prev->next = x->next;
    x->next->prev = x->prev;
    delete x;
}
```

이 `erase()` 함수는 인수에 `p` 리스트를 추가하여 아래와 같이 일부 에러가 발생하는 경우를 처리할 수 있습니다.

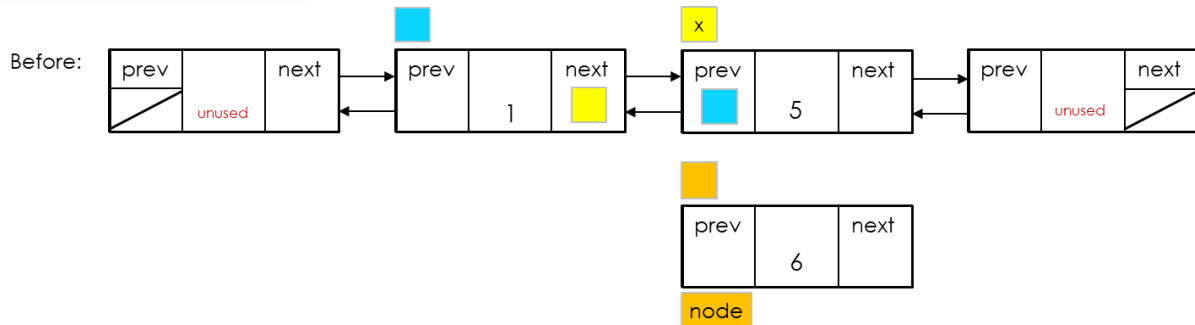
```
void erase(pList p, pNode x) { // checks if x is neither tail nor head
    if (x == p->tail || x == p->head || x == nullptr) return;
    x->prev->next = x->next;
    x->next->prev = x->prev;
    delete x;
}
```

`insert()` 함수는 지정된 위치 `x` 에서 노드 앞에 값을 가진 새 노드를 삽입하는 용기를 확장합니다. 이렇게 하면 리스트 크기가 효과적으로 1 씩 늘어납니다. 예를 들어 `begin(p)`가 삽입 위치로 지정된 경우 새 노드가 리스트의 첫 번째 노드가 됩니다.

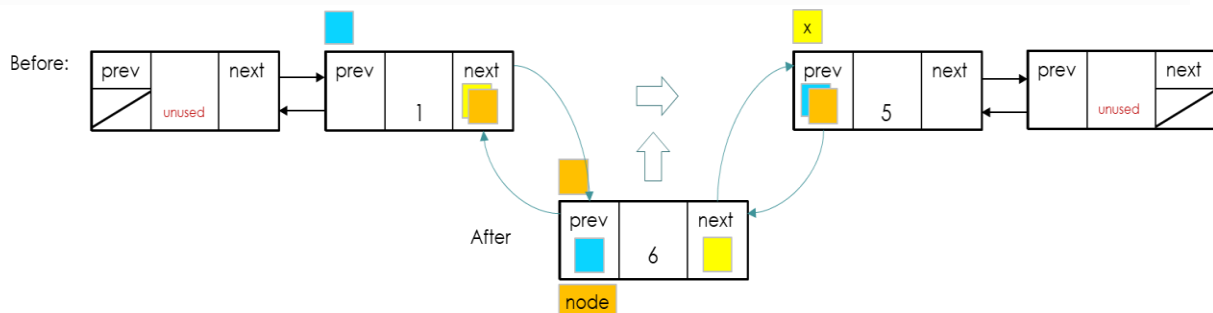
값이 있는 노드 x 를 삽입한다고 가정합니다. insert() 함수에는 아래와 같이 두 개의 인수가 있습니다.

```
void insert(pNode x, int value)
```

아래 그림과 같이 새 노드(B2)를 만들어 노드 x (A1)에 삽입하고자 합니다. 즉, 노드 A0 과 노드 A1 사이에 새 노드 B2 를 삽입합니다.



값이 6 인 노드(주황색)를 하나 만들고 아래와 같이 파란색 노드와 노란색 노드 사이에 삽입합니다:



새로운 포인터로 네 곳을 설정합니다:

실제 코드에서 다음과 같이 나타낼 수 있습니다.

- (1) 주황(node)의 prev: node→prev
- (2) 주황(node)의 next: node→next
- (3) 노랑: x or $x \rightarrow \text{prev} \rightarrow \text{next}$
- (4) 파랑: $x \rightarrow \text{prev}$

(1) 과 (2) 링크의 경우, 다음 그림과 같이 초기화하여 노드를 초기화하는 동안 새 노드에 두 개의 링크를 설정합니다.

```
void insert(pNode x, int value){
    pNode node = new Node(val, x->prev, x);
    x->prev->next = node;
    x->prev = node;
}
```

Step 1: find(), more() & less()

find()는 리스트에서 제일 먼저 발견된, x 와 데이터 항목이 동일한 노드를 반환하고, 동일한 노드를 찾을 수 없는 경우 테일 노드를 반환합니다. 뼈대 코드는 다음과 같이 주어집니다.

```
pNode find(pList p, int value) {
    pNode curr = begin(p);
    for (; curr != end(p); curr = curr->next)
        if (curr->data == value) return curr;
    return curr;
}
```

for 문과 if 문을 사용하지 않고, 하나의 while() loop 만 사용하도록 find()를 재작성하세요. while(.....) 안에서 두 가지 조건을 추가하는 방법을 생각해 보세요.

find() 코드를 복사하여 리스트에서 제일 먼저 발견된, 데이터 항목이 x 보다 크거나 작은 노드를 반환하는 **more()** 및 **less()**를 구현하도록 수정하세요. 찾을 수 없는 경우, **테일** 노드를 반환합니다.

```
pNode more(pList p, int x) {
    cout << "your code here\n";
}
```

```
pNode less(pList p, int x) {
    cout << "your code here\n";
}
```

Step 2: push commands

아래 함수들은 push 커맨드와 연관된 몇 가지 기본 push 함수입니다:

- push()
- push_back()
- push_backN(int N)
- push_backN(int N, int value)
- push_front()

push() 함수는 어떤 값을 가진 새 노드를 x 값을 가진 노드가 있는 위치에 삽입합니다. 새 노드는 사실상 x 값을 가진 노드 앞에 삽입됩니다. x의 위치를 찾을 수 없으면 값을 리스트에 push **하지 않고**, 입력은 무시합니다.

```
void push(pList p, int value, int x)
```

힌트: find()와 insert()를 사용하면 2~3 줄만으로 구현할 수 있습니다.

push_backN() 함수는 리스트 끝에 N 개의 새 노드를 추가합니다. 값이 주어지지 **않으면** 임의로 생성된 숫자가 $[0 \cdots (N + \text{size}(p))]$ 의 범위 안에서 push 됩니다. 값이 주어진다면 동일한 값을 N 번만 삽입하면 됩니다. push_back()의 시간 복잡도는 $O(n)$ 이므로 N 번 호출할 수 있습니다.

```
void push_backN(pList p, int N)
void push_backN(pList p, int N, int value)
```

자세한 내용은 제공된 뼈대 코드 또는 헤더 파일을 참조하세요.

도우미 함수: rand_extended() in librand.a or rand.lib

rand()를 사용해서 난수를 생성하는 것에 대한 참고 사항입니다.

일반적으로 RAND_MAX 가 32767 로 정의되어 있어서 rand()를 사용하여 난수를 생성하면 우리가 원하는 것보다 훨씬 작은 수가 생성됩니다. 도우미 함수 rand_extended()를 활용해보세요.

Nowic/lib/librand.a 또는 rand.lib 에 제공된 rand_extended()를 사용하세요.

```
// returns an extended random number of which the range is from 0
// to (RAND_MAX + 1)^2 - 1. // We do this since rand() returns too
// small range [0..RAND_MAX) where RAND_MAX is usually defined as
// 32767 in cstdlib. Refer to the following link for details
// https://stackoverflow.com/questions/9775313/extend-rand-max-range

unsigned long rand_extended() {
    if (RAND_MAX > 32767) return rand();
    return rand() * RAND_MAX + rand();
}
```

Step 3: pop commands - pop_all()*

아래 함수들은 pop 커맨드와 연관된 몇 가지 기본 pop 함수입니다:

- pop()
- pop_front()
- pop_back()
- pop_backN()
- pop_all()

pop() 함수는 값을 가지고 있는 첫 번째 노드를 리스트에서 제거하고, 노드를 찾을 수 없는 경우 아무 작업도 수행하지 않습니다. 위치별로 노드를 지우는 멤버 함수 **List::erase** 와는 달리, 이 함수는 값을 기준으로 노드를 제거합니다. **pop()**과 달리 **pop_all()**은 주어진 값을 가진 모든 노드를 제거합니다. (힌트: find()와 erase()를 사용해보세요.)

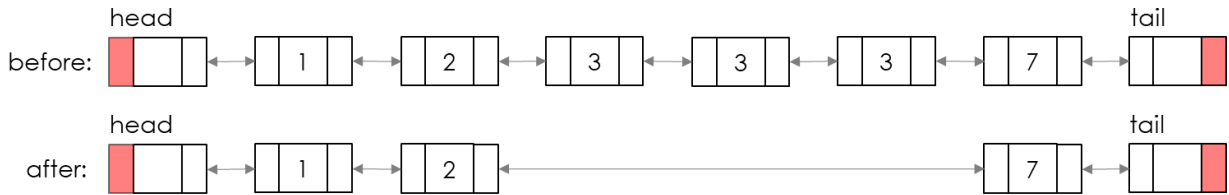
```
void pop(pList p, int value);
```

pop_all()은 동일한 값이 주어진 모든 노드를 리스트에서 제거합니다. 아래에 있는 코드는 제대로 작동하지만, 리스트를 여러 번 거쳐갑니다.

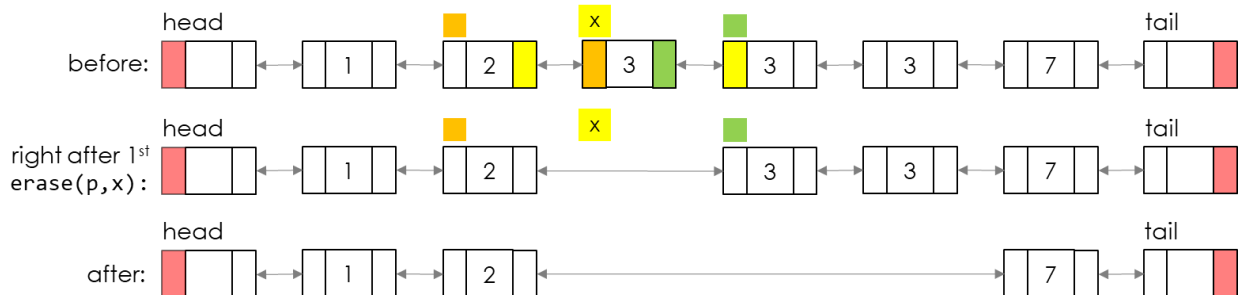
```
#if 1
// remove all occurrences of nodes with the value given in the list.
void pop_all(pList p, int value) {
    while (find(p, value) != end(p)) {
        pop(p, value);
    }
} // version.1
#else
// remove all occurrences of nodes with the value given in the list.
void pop_all(pList p, int value) {
    for (pNode c = begin(p); c != end(p); c = c->next)
        if (c->data == value) erase(p, c);
} // version.2 fast but buggy
#endif
```

위치별로 노드를 지우는 **erase()**와 달리 이 함수는 값별로 노드를 제거합니다. **pop_all()**과 달리 **pop()**은 제일 먼저 발견한, 주어진 값과 동일한 값을 가진 노드를 제거합니다.

버그가 있는 **pop_all()**의 뼈대 코드는 위와 같이 제공됩니다. 다음 그림은 코드의 잠재적인 문제를 설명합니다.



예를 들어, 이 코드는 3 을 성공적으로 제거합니다. 만약 리스트에서 연속해서 3 이 발생하는 경우 간혹 성공적으로 제거하지 못할 수도 있습니다. 또한, **erase()** 직후에 출력문을 추가하여 "3"을 제거한 후 어떤 내용이 출력되는지 확인하세요. 버그를 수정하기 위해 "for 문에서 **x** 를 지운 후 **x** 가 바로 그다음에 위치한 (값이 3 인) 노드를 가리킬 수 있을까?"에 대한 답을 생각해 보세요.



자세한 내용은 제공된 뼈대 코드 또는 헤더 파일을 참조하세요.

Step 4: show()*, half()

뼈대 코드의 **show()** 함수는 모든 노드를 출력합니다. **show()** 함수가 **show_all** 및 **show_n** 두 인수와 함께 작동하도록 함수를 완성하세요. **show_all** 이 false 인 경우 가운데 노드도 또한 출력합니다.

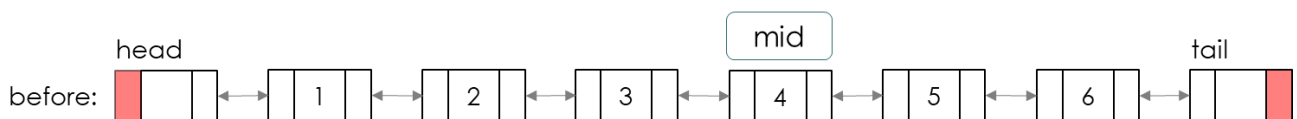
```
Command[q to quit]: s
Enter b:bubble, i:insertion, s:selection, q:quicksort: s
cpu: 0 sec

> 0 > 2 > 2 > 4 > 5 > 5 > 6 > 7 > 9 > 9
... 44 ...
> 89 > 89 > 90 > 91 > 94 > 95 > 96 > 97 > 98 > 99

Doubly Linked List(nodes:100, show:HEAD/TAIL,10)
f - push front    0(1)    p - pop front    0(1)
b - push back     0(1)    y - pop back     0(1)
```

중간 노드를 출력하려면 **half()**라는 함수를 사용하여 리스트의 중점(midpoint)을 찾아야 합니다. 이 함수는 **show()**에서 사용됩니다. "**show [HEAD/TAIL]**" 옵션을 설정하고 메인 메뉴에 리스트를 출력하면 이 기능이 작동 중임을 알 수 있습니다. **show()**는 위와 같이 1~100 개의 노드 중 중점에 있는 **"51"**을 출력합니다.

- 노드 수가 짝수인 경우, 두 번째 절반의 첫 번째 노드를 반환합니다. 노드가 10 개인 경우, 6 번째 노드를 반환합니다..
- 리스트에 5 개(홀수)의 노드가 있으면, 세 번째 노드(또는 중간 노드)를 반환합니다.



half() 함수를 구현하는 방법이 몇 가지 있습니다. 방법 1 과 방법 2 를 모두 구현해야 합니다. 수업 시간에 배웠듯이 방법 2 가 더 빠릅니다.

구현 방법 1:

리스트에 있는 노드 수를 센 다음 중간 지점까지 스캔한 뒤, 해당 위치의 마지막 링크를 끊어냅니다.

```
#if 1
pNode half(pList p) {
    int N = size(p);
    // go through the list
    // break at the halfway point
    // return the current pointer
}
#endif
```

구현 방법 2:

리스트에서 토끼와 거북이를 아래로 내려 보내는 방법입니다. 거북이는 속도 1 로 움직이고 토끼는 속도 2 로 움직입니다. 토끼가 끝에 도착하면, 거북이는 중간 지점에 와있을 것입니다.

```
#if 0
pNode half(pList p) {
    pNode rabbit = begin(p);
    pNode turtle = begin(p);
    while (rabbit != end(p)) {
        rabbit = rabbit->next->next;
        turtle = turtle->next;
    }
    return turtle;
} // buggy on purpose
#endif
```

과제 제출

- 소스 파일 상단에 아래와 같이 아너 코드 문장을 적고 서명하세요.

On my honor, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.

서명: _____ 분반: _____ 학번: _____

- 제출하기 전에 코드가 제대로 컴파일이 되고 실행되는지 확인하세요. 제출 직전에 급하게 코드를 수정한 후 코드가 제대로 컴파일이 될 거라고 짐작하지 않는 게 좋습니다. “거의” 작동하는 코드도 틀린 것입니다.
- 과제가 컴파일 및 실행된다면, 마감 기한 전까지 과제의 일부만 완성했더라도 제출하기 바랍니다. 컴파일 및 실행되지 않는다면 제출하지 마세요. 마감 시간 이후 24 시간 이내 제출하면, 만점에서 25% 감점하고 채점합니다. 그 이상 늦은 것은 채점하지 않으며, 0 점 처리합니다.
- 제출 후, 마감 기한 전까지 수정 및 재제출이 가능합니다. 파일 하나만 수정하더라도 해당 파일과 관련된 파일들을 모두 재제출해야 합니다. 재제출 횟수는 제한 없습니다. 마감 기한 전에 **가장 마지막으로** 제출된 파일을 채점할 것입니다.

제출 파일 목록

- 다음 파일들을 piazza **lab8** 폴더에 제출하세요.
 - listdbl.cpp

마감 기한 & 배점

- 마감 기한: 11:55 pm