6 장. 리스트

Jinseog Kim

컴퓨터 공학과
동국대학교 WISE 캠퍼스

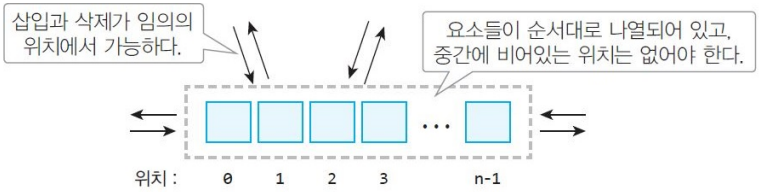
학습 내용

1. 리스트란?
2. 배열을 이용한 리스트 (Array List)
3. 단순 연결 구조의 리스트 (Singly linked list)
4. 이중 연결 구조의 리스트 (Doubly linked list)
5. 리스트의 응용: 맛집 웨이팅 프로그램

dongguk UNIVERSITY WISE

리스트란?

- 리스트 (list), 선형리스트 (linear list)
  - **순서를 가진** 항목들의 모임, 항목의 중복 허용
  - **집합 (set)**: 항목간의 순서와 중복이 없음
- 리스트의 예
  - 핸드폰의 문자 메시지 리스트
  - 다항식의 각 항들
- 리스트의 구조

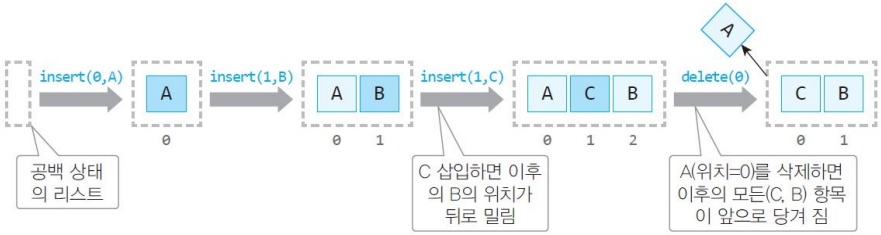


리스트의 추상 자료형

- 리스트의 추상 자료형

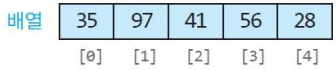
  1. 데이터: 순서를 가진 항목들의 모임
  2. 연산
     - `insert(pos, e)`: pos위치에 새로운 요소 e를 삽입
     - `delete(pos)`: pos위치의 요소를 삭제 반환
     - `is_empty()`: 리스트가 비어 있는지 검사
     - `is_full()`: 리스트가 가득 차 있는지 검사
     - `get_entry(pos)`: pos위치의 요소를 반환

- 리스트의 일련의 연산

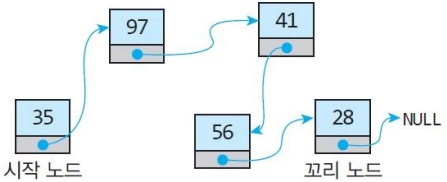


리스트의 구현 방법

1. Array 이용: ArrayList
2. Linked structure 이용: Linked List
   1. 단순연결 (Singly linked list)
   2. 이중연결 (Doubly linked list)



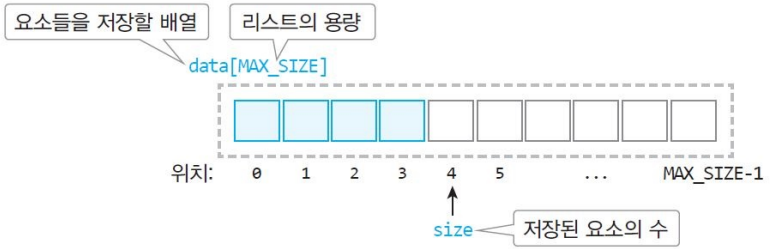
(a) 배열 구조의 리스트



(b) 연결된 구조의 리스트

ArrayList 의 구조와 연산

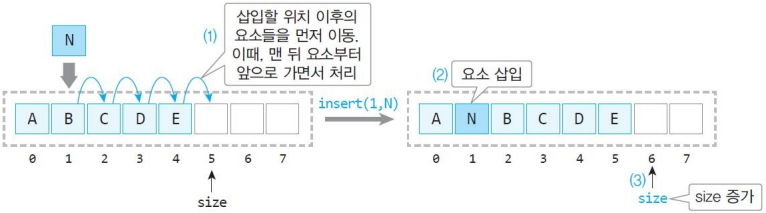
- 배열을 이용한 리스트의 구조



- 공백 상태와 포화 상태를 검사하는 is_empty(), is_full()

ArrayList 에서 삽입 연산

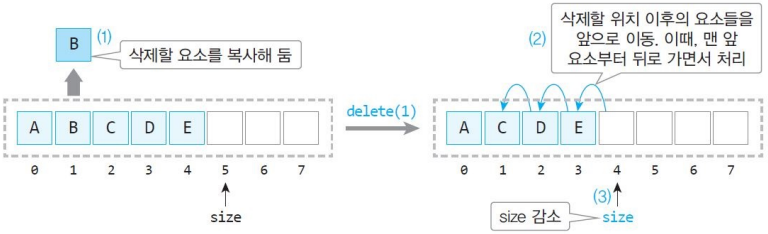
- pos 에 새로운 요소 e 를 삽입하는 insert(pos, e) 연산




```
1  insert(pos, e)
2    if is_full():
3      error "overflow"
4    else :
5      for i = size - 1 to pos :   // (1)
6        data[i+1] <- data[i]
7      data[pos] <- e              // (2)
8      size <- size + 1            // (3)
```

ArrayList 에서 삭제 연산

- pos 위치의 요소를 꺼내서 반환하는 delete(pos) 연산




```
1  delete(pos)
2    if is_empty():
3      error "Underflow"
4    else :
5      e <- data[pos]                  // (1)
6      for i = pos + 1 to size - 1 :   // (2)
7        data[i-1] <- data[i]
8      size <- size + 1                // (3)
9      return e
```

ArrayList 에서 참조 연산

- pos 위치의 요소를 참조하는 get_entry(pos) 연산

```
1  get_entry(pos)
2    if is_empty():
3      error "Underflow"
4    else :
5      return data[pos]
```

ArrayList 의 구현

- 리스트의 데이터

```
1 // Element 와 MAX_SIZE 는 미리 정의되어 있어야 함
2 Element data[MAX_SIZE]; // 요소의 배열
3 int size = 0;           // 요소의 수
```

- 리스트의 연산 (초기화, 상태검사)

```
1 void init_list() { size = 0; }
2 int is_empty() { return size == 0; }
3 int is_full() { return size == MAX_SIZE; }
```

ArrayList 의 구현

- 삽입 연산

```
1  void insert(int pos, Element e){
2      if (is_full())
3          error("Overflow Error!");
4
5      if (pos < 0 || pos > size)
6          error("Invalid Position Error!");
7
8      for (int i = size - 1; i >= pos; i--)
9          data[i + 1] = data[i];
10     data[pos] = e;
11     size += 1;
12 }
```

ArrayList 의 구현

- 삭제 연산

```
1  Element delete(int pos){
2      if (is_empty())
3          error("Underflow Error!");
4
5      if (pos < 0 || pos >= size)
6          error("Invalid Position Error!");
7
8      Element e = data[pos];
9      for (int i = pos + 1; i < size; i++)
10         data[i - 1] = data[i];
11     size -= 1;
12     return e;
13 }
```

ArrayList 의 구현

- 테스트 프로그램 (ArrayList.c)

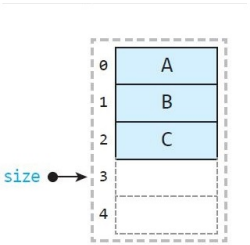
```
1  void main(){
2      init_list();        // [ ]
3      insert(0, 10);      // [10]
4      insert(0, 20);      // [20, 10]
5      insert(1, 30);      // [20, 30, 10]
6      insert(size, 40);   // [20, 30, 10, 40]
7      insert(2, 50);      // [20, 30, 50, 10, 40]
8      print_list("(삽입 x5)");
9
10     delete(2);          // [20, 30, 10, 40]
11     delete(size - 1);   // [20, 30, 10]
12     delete(0);          // [30, 10]
13     print_list("(삭제 x3)");
14 }
```

Lab: ArrayList 에 연산 추가하기

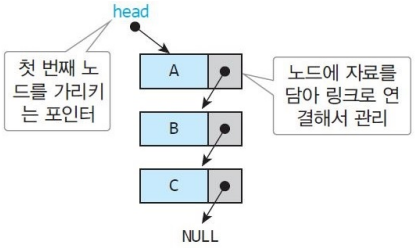
1. **append(e)**: 리스트 맨 뒤에 요소 e 를 추가
2. **pop()**: 리스트 맨 뒤의 요소를 꺼내서 반환
3. **replace(pos, e)**: pos 위치의 요소를 e 로 수정
4. **find(e)**: 리스트에 요소 e 가 있으면 그 요소의 위치 (인덱스) 를, 없으면 –1 을 반환

```
1  find(e)
2    if is_empty() : error "Underflow"
3    else if is_full() : error "Overflow"
4    else :
5      idx <- -1
6      for i = 0 to (size - 1) :
7        if data[i] == e :
8          idx <- i
9          break
10     return idx
```

단순 연결 구조의 리스트 (Singly Linked-List)



(a) 배열 구조의 리스트



(b) 단순 연결 구조의 리스트

- 노드 구조체와 헤드 포인터

```
1  typedef struct Node {   // 자기참조 구조체
2      Element data;       // 데이터 필드
3      struct Node* link;  // 링크 필드
4  } Node;
5  Node * head = NULL;    // head pointer
```

연결 리스트의 연산

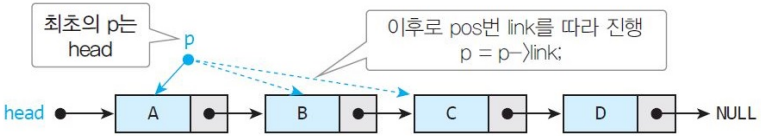
- 리스트의 초기화

```
1 init()
2     head <- NULL
```

- 리스트의 상태 검사

```
1 is_empty()
2     return head <- NULL
```

get_node, get_entry: 특정 위치의 노드 탐색



- pos 위치에 있는 노드 탐색

```
1  get_node(pos)
2      p <- head
3      for i = 0 to pos-1 :
4          if p == NULL: return NULL
5          else: p <- p.link
6      return p
```

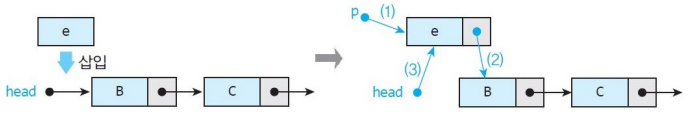
- pos 위치에 있는 데이터 탐색

```
1  get_entry(pos)
2      p = get_node(pos)
3      if p == NULL:
4          error "Invalid Position Error!"
5      else: return p.data
```

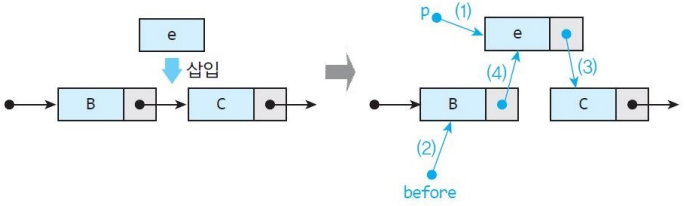
insert: 정해진 위치에 요소 삽입

- 맨앞과 중간에 삽입할 때를 구분: 헤드포인터를 사용하기 때문

1. 맨 앞에 삽입 (pos=0)



2. 중간에 삽입 (pos > 0)

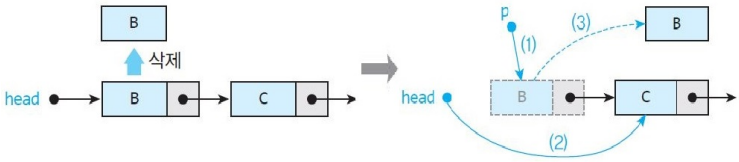



```
1  insert(pos, e)
2    // 삽입할 노드 생성
3    p = alloc_node(e) // (1)
4
5    if pos == 0:
6      p.link <- head  // 1(2)
7      head <- p       // 1(3)
8    else:
9      before = get_node(pos-1)// 2(2)
10     p.link <- before.link   // 2(3)
11     before.link = p         // 2(4)
```

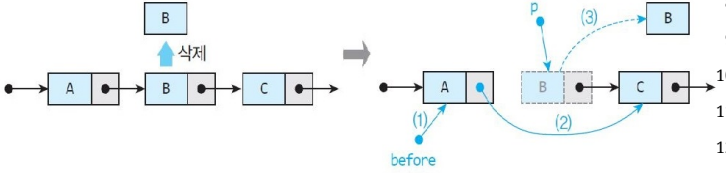
- 2(2) 에서 이전 노드 탐색 시간: O(n)
- 이중연결리스트: 이전 노드를 직접 탐색 가능 O(n) → O(1)

delete: 특정 위치의 요소를 꺼내서 반환

- 맨 앞 노드 삭제 (pos=0)



- before 다음 노드 삭제 (pos>0)



- 이중연결리스트: before 노드를 직접 탐색, O(n) → O(1)

```
1  delete(pos)
2    if is_empty():
3      error "Underflow Error!"
4    else:
5      // 삭제할 노드
6      p <- get_node(pos)
7      if p == head:
8        head <- p.link
9      else:
10       // 이전 노드
11       before <- get_node(pos-1)
12       before.link <- p.link
13   return free_node(p)
```

단순 연결 리스트의 구현: data

```
1  // 리스트의 데이터
2  // Element 는 미리 정의되어 있어야 하고
3  // MAX_SIZE 는 필요 없음
4  typedef struct Node {   // 자기참조 구조체
5      Element data;       // 데이터 필드 (스택 요소)
6      struct Node* link;  // 링크 필드
7  } Node;
8
9  Node* head = NULL;
```

단순 연결 리스트의 구현: insert

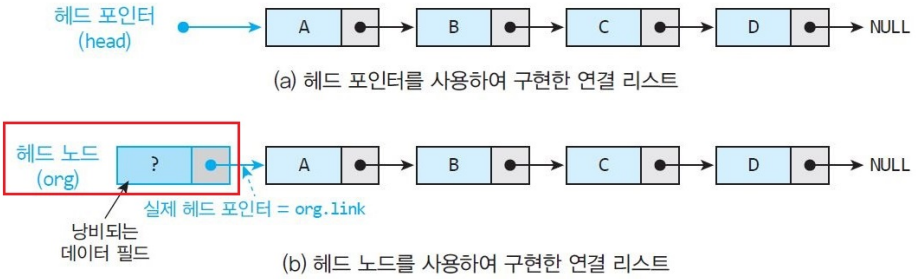
```
1  // 리스트의 삽입 연산
2  void insert(int pos, Element e)
3  {
4      Node* p = alloc_node(e); // 삽입할 노드 생성 및 초기화
5      if (pos == 0) {
6          p->link = head;
7          head = p;
8      }
9      else {
10         Node* before = get_node(pos - 1);
11         if (before == NULL)
12             error("Invalid Position Error!");
13         p->link = before->link;
14         before->link = p;
15     }
16 }
```

단순 연결 리스트의 구현: 삭제 연산

```
1  Element delete(int pos)
2  {
3      if (is_empty())
4          error("Underflow Error!");
5      Node* p = get_node(pos);            // 삭제할 노드
6      if (p == NULL)
7          error("Invalid Position Error!");
8      Node* before = get_node(pos - 1);   // 이전 노드
9      if (before == NULL)                 // 맨 앞 노드 삭제
10         head = head->link;
11     else
12         before->link = p->link;
13     return free_node(p);
14 }
```

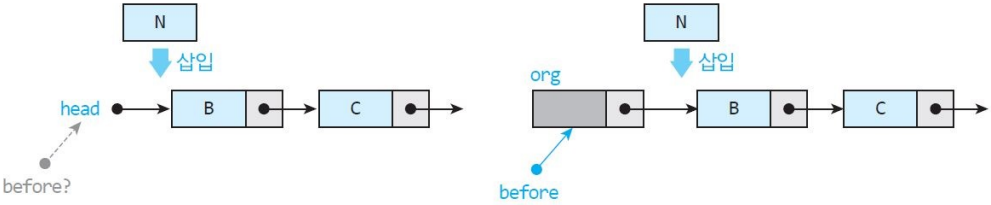
- 헤드 노드 (head node)
  - 시작 노드를 가리키는 헤드 포인터 대신에 노드 **구조체 자체를 사용**하는 것

```
Node org; // head node, head pointer 는 org.link
```



헤드 노드를 사용할 때 장점

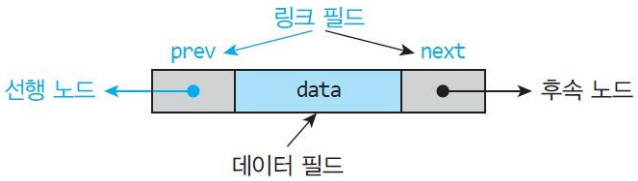
- 헤드 노드를 사용하면 모든 노드가 선행 노드를 갖게 됨



- 헤드노드를 사용할 때 이로운 점
  - 헤드 포인터를 사용하면 삽입/삭제에서, **이전 노드 탐색시 구분**하여 코딩해야 함
  - 헤드 노드를 사용하면 이전 노드 탐색을 간결하게 표현할 수 있음

이중 연결 구조의 리스트

- 이중 연결을 위한 노드 구조

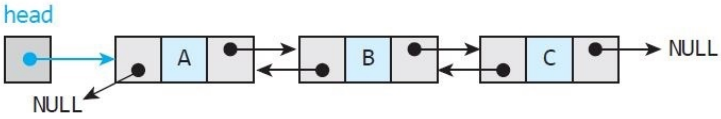



```c
typedef struct DNode {
  Element data;
  struct DNode * prev; //선행노드
  struct DNode * next; //후속노드 (Node 의 link)
} DNode;
```

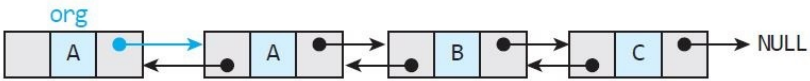
```
p == p->next->prev == p->prev->next
```

이중 연결 리스트의 구조

- 헤드 포인터와 헤드 노드 비교



(a) 이중 연결 리스트(헤드 포인터 head 사용)



(b) 이중 연결 리스트(헤드 노드 org 사용)

이중 연결 리스트의 연산

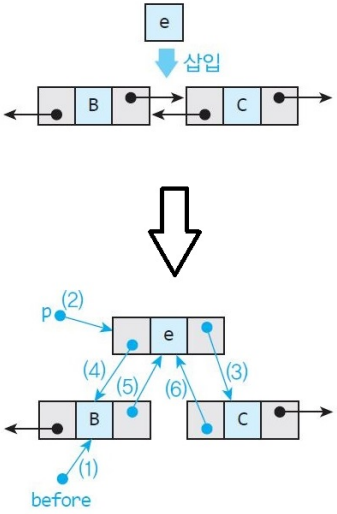
- pos 위치의 노드를 참조하는 get_node(pos)

```
1  get_node(pos)
2      p <- org 의 주소     // -1 인 위치 (헤드 노드 주소) 에서 출발
3      for i <- 0 to pos:
4          if p == NULL:
5              return NULL
6          else: p <- p.next // link 대신에 next
7      return p
```

- 주요 변경 사항 (단순연결 리스트와 비교)
  - Node → DNode
  - 후속 노드의 링크 이름: link → next
  - 선행 노드의 링크 이름: prev
  - head → org.next

이중 연결 리스트의 연산

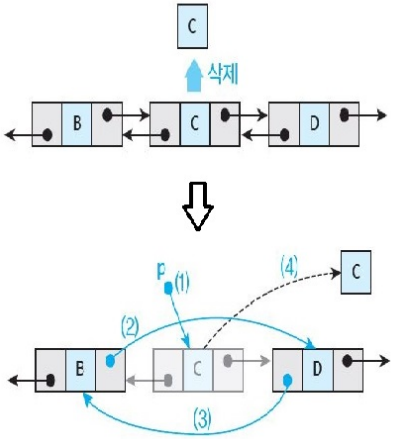
- pos 위치에 요소 e 를 삽입하는 insert(pos,e) 연산




```
1  insert(pos, e)
2    before <- get_node(pos-1) // (1)
3    if before == NULL :
4      error "Invalid Position Error"
5    else :
6      p <- alloc_dnode(e)    // (2)
7      p.next <- before.next  // (3)
8      p.prev <- before       // (4)
9      before.next <- p       // (5)
10     if p.next != NULL :
11       p.next->prev <- p    // (6)
```

이중 연결 리스트의 연산

- pos 위치의 요소를 꺼내서 반환하는 delete() 연산




```
1  delete(pos)
2    p <- get_node(pos)          // (1)
3    if p == NULL :
4      error "Invalid Position Error"
5    p.prev.next <- p.next       // (2)
6    else :
7      if p.next != NULL :
8        p.next.prev <- p.prev // (3)
9      return free_node(p)       // (4)
```

이중 연결 리스트의 구현

- DblLinkedList.h

```c
// 이중연결을 위한 노드 구조체
typedef struct DNode {
    Element data; // 데이터
    struct DNode* prev; // 선행 노드
    struct DNode* next; // 후속 노드
} DNode;

DNode org; // 헤드노드
```

```c
DNode* alloc_dnode(Element e){
    DNode* p = (DNode*)malloc(sizeof(DNode));
    p->data = e; // 데이터 초기화
    p->prev = NULL; // 선행노드 링크 초기화
    p->next = NULL; // 후속노드 링크 초기화
    return p;
}
Element free_dnode(DNode* p){
    Element e = p->data;
    free(p);
    return e;
}
```

이중 연결 리스트의 구현

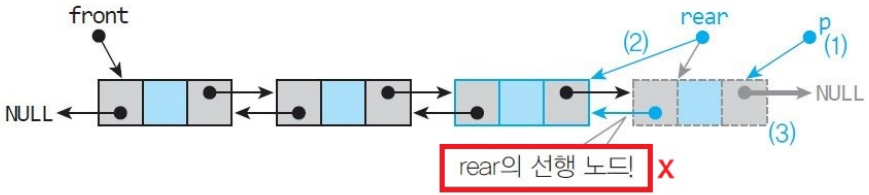
- 삽입/삭제 연산

```
1  // 삽입 연산
2  void insert(int pos, Element e)
3  {
4      DNode* before = get_node(pos - 1);
5      if (before == NULL)
6          error("Invalid Position Error!");
7      DNode* p = alloc_dnode(e);
8      p->next = before->next;
9      p->prev = before;
10     before->next = p;
11     if (p->next != NULL)
12         p->next->prev = p;
13 }
```

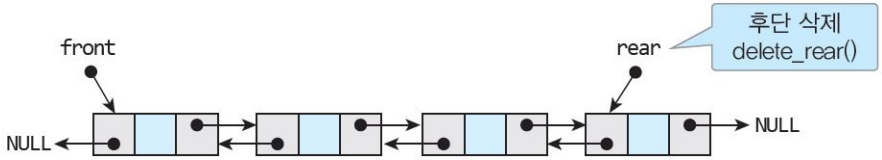
```
1  // 삭제 연산
2  Element delete(int pos)
3  {
4      DNode* p = get_node(pos);
5      if (pos < 0 || p == NULL)
6          error("Invalid Position Error!");
7      p->prev->next = p->next;
8      if (p->next != NULL)
9          p->next->prev = p->prev;
10     return free_dnode(p);
11 }
```

Lab: 이중 연결 구조의 덱 연산 구현하기

- 단일 연결 구조에서 후단 삭제
  - 현 노드의 후행 노드 정보만 알고 있고 **선행노드를 알 수 없음**, 첫 노드 (front) 에서 탐색해야 함



- **이중 연결 구조**에서는 선행노드 정보를 알 수 있음



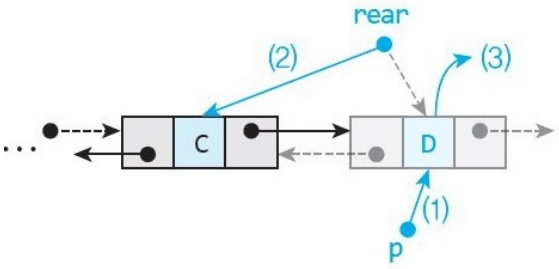
Lab: 이중 연결 구조의 덱 연산 구현하기

- 후단 삭제 (delete_rear())

(a) 노드가 하나인 경우의 후단 삭제

(b) 노드가 둘 이상인 경우의 후단 삭제

Lab: 이중 연결 구조의 덱 연산 구현하기



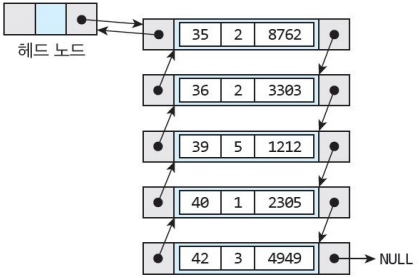

```
1  // 후단 삭제 연산
2  Element delete_rear()
3  {
4      if (is_empty())
5      error("Underflow Error!");
6
7      DNode* p = rear;         // (1)
8      if (front == rear)  // 요소가 하나
9          front = rear = NULL;
10     else { // 요소가 둘 이상
11         rear = rear->prev;  // (2)
12         rear->next = NULL;
13     }
14     return free_dnode(p);   // (3)
15 }
```

리스트의 응용: 맛집 웨이팅 프로그램

- 웨이팅 리스트와 이중 연결 리스트를 이용한 표현

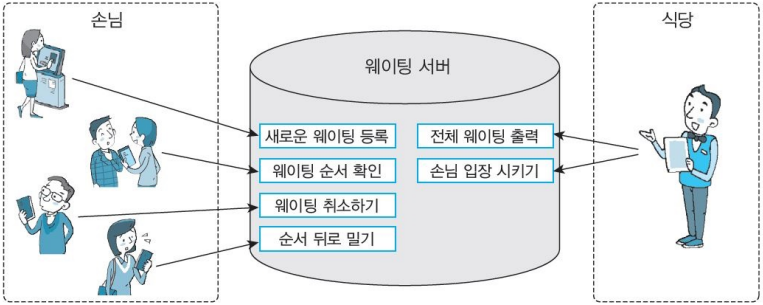
대기 번호	인원	전화번호
35	2	8762
36	2	3303
39	5	1212
40	1	2305
42	3	4949

(a) 웨이팅 리스트의 예



(b) 웨이팅 리스트의 내부 표현 예

웨이팅 프로그램의 기능



1. 새로운 웨이팅 등록: reserve(nperson, info)
2. 웨이팅 순서 확인: find(wid)
3. 웨이팅 취소: calcel(wid)
4. 웨이팅 순서 한 칸 뒤로 밀기: delay(wid)
5. 전체 웨이팅 출력: print()
6. 손님 입장하기: service()

- Data

```
1  struct Waiting {    // 맛집 웨이팅 큐를 위한 구조체
2      int id;         // 대기번호
3      int nperson;    // 인원
4      char info[32];  // 전화번호
5  };
6  typedef struct Waiting Element; // Element 의 자료형 정의
7
8  #include "DblLinkedList.h"      // 이중 연결 리스트 사용
```

1. 새로운 웨이팅 등록을 위한 reserve(nperson, info) 연산

```
1  void reserve(int nperson, const char info[]){
2      static int id = 0;
3      Element e;
4      e.id = ++id;                // 대기번호 자동 부여
5      e.nperson = nperson;        // 인원
6      strcpy_s(e.info, 32, info); // 전화번호
7      insert(size(), e);          // 리스트의 맨 뒤에 추가
8      printf("< 등록 > 번호 %d: 인원 %d 명 %s\n", e.id, e.nperson, e.info);
9  }
```

② 웨이팅 순서 확인을 위한 find(wid) 연산

```
1  void find(int wid)
2  {
3      int nTeam = 0, nPeople = 0; // 앞의 팀 수와 인원 계산을 위한 변수
4      for (int pos = 0; pos < size(); pos++) {
5          Element e = get_entry(pos);
6          if (e.id == wid) {
7              printf("< 확인 > 번호 %d 번 앞 대기팀: %d 팀 %d 명\n", wid, nTeam, nPeople);
8              return;
9          }
10         nTeam += 1;              // 앞의 팀
11         nPeople += e.nperson;    // 앞의 대기 인원
12     }
13 }
```

③ 웨이팅 취소를 위한 calcel(wid) 연산

```
1  void cancel(int wid)
2  {
3      for (int pos = 0; pos < size(); pos++) {
4          Element e = get_entry(pos);
5          if (e.id == wid) {
6              delete(pos);
7              printf("< 취소 > %d 번 웨이팅이 취소되었습니다.\n", wid);
8              return;
9          }
10     }
11 }
```

④ 웨이팅 순서 한 칸 뒤로 밀기를 위한 delay(id) 연산

```
1  void delay(int wid)
2  {
3      for (int pos = 0; pos < size() - 1; pos++) {
4          Element e = get_entry(pos);
5          if (e.id == wid) {
6              delete(pos);
7              insert(pos + 1, e);
8              printf("< 연기 > %d 번 웨이팅이 한 칸 연기되었습니다.\n", wid);
9              return;
10         }
11     }
12 }
```

⑤ 전체 웨이팅 리스트 출력

```
1  void print(){
2      printf("< 출력 >\n");
3      for (int pos = 0; pos < size(); pos++) {
4          Element e = get_entry(pos);
5          printf(" 번호 %2d: %d 명 %s\n", e.id, e.nperson, e.info);
6      }
7      printf("\n");
8  }
```

⑥ 손님 입장시키기

```
1  void service(){
2      Element e = delete(0);
3      printf("< 입장 > %d 번 손님 입장 (%d 명, %s)\n", e.id, e.nperson, e.info);
4  }
```

테스트 프로그램

```
1  void main(){
2      init_list();     print();
3      reserve(2, "010-xxxx-8762");
4      reserve(2, "010-xxxx-3303");
5      reserve(5, "010-xxxx-1212");
6      reserve(1, "010-xxxx-2305");     print();
7
8      service();   print();
9
10     reserve(3, "010-xxxx-4949");
11     reserve(4, "010-xxxx-7345");
12     print();
13
14     find(4);
15     delay(3);
16     delay(3);     print();
17     cancel(5);     print();
18
19     destroy_list();
20 }
```

연습문제

- 연습문제 6.1 ~ 6.15 (P238 ~ P241)