SLL
마지막주소 Null
CLL
마지막주소 처음 가리키게
DLL
앞 data 뒤

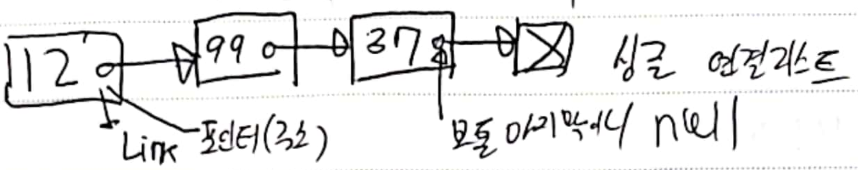
6. 연결된 구조

- SLL 싱글
- CLL 원형 — 3가지가 있음
- DLL 이중 → 이건 많이 쓰음 → 대신 복잡함

스택, 리스트, 큐, 덱 응용함
↘ 저 3가지로 구현가능

연결된구조

◦ Linked List, 연결 리스트 라고함
◦ 각 노드는 데이터와 포인터를 가지고, 한 줄로 연결되어 있는 방식의 자료구조
◦ 연결된 구조는 떨어진 공간에 존재하는 데이터를 화살표로 연결해서 관리하는 구조
◦ 파이썬은 리스트 타입이 연결된 구조의 기능을 모두 지원



특징

배열이랑 비교함 → 검색 어려움
1. 용량이 고정되지 않음
→ 필요한 것만 필요할 때 사용 → 효율적
연결리스트? → X 파이썬
→ 컴퓨터 메모리가 남아 있다면 계속 자료를 넣을 수 있다.
편리함 → 2. 중간에 자료를 삽입, 삭제하는 것이 용이
연결 →. 리스트와 같은 배열 구조에는 많은 항목의 이동 필요
배열 → 연결된 구조에서는 링크만 수정하면 되므로 시간복잡도 O(1)
3. N번째 항목에 접근하는데 시간복잡도 O(n)
단점임 → 연결 정보를 찾는 시간이 필요하므로 접근속도가 느림

차례대로 가야됨

→ 삽입/삭제 갱신에 유용, 메모리에 효율
검색 기능은 별로 (배열이 좋음)

노드

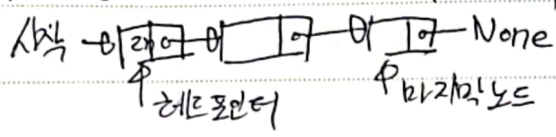
• 데이터 저장단위 (데이터 값, 포인트)로 구성

□ => | data | list |

헤더 — 데이터필드, 링크필드

시작 / 꼬리 노드, 헤드포인터

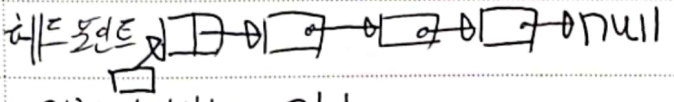
→ 각 노드안에서, 다음이나 이전의 노드와의 연결 정보를 가지고 있는 공간



종류

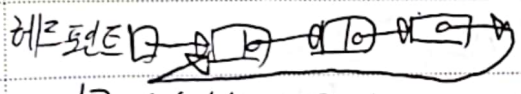
SLL, CLL 은 변형없음

- 단순 연결리스트 SLL
  - → 하나의 방향으로만 연결되어 있는 구조
  - → 링크하나이며, 이 링크는 다음 노드의 주소를 갖고있음
  - → 마지막 노드의 링크는 None 값을 가지고있음

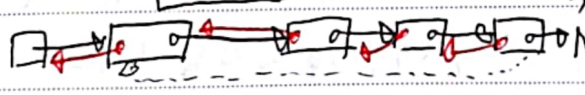


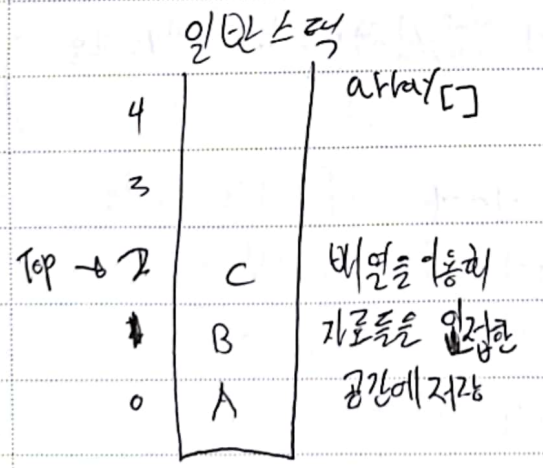
- 원형 연결리스트 CLL
  - 단순 연결리스트와 동일한 노드구조
  - 마지막 링크값이 첫번째 노드를 가리킴

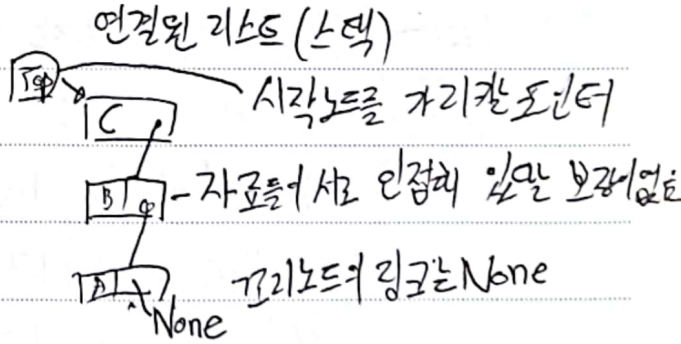
중간이 바뀌면상관X 하지만 맨처음이 바뀌면 복잡(마지막도함)



- 이중 연결리스트 DLL + 원형으로하게되면... 이중원형 연결리스트
  - → 하나의 노드는 이전노드, 다음노드 모두 알고있음
  - → 두개의 링크로 각각 선행노드, 후속노드를 가리킴
  - → 선행 ← Prev | Data | Next → 후속







삽입연산 (연결스택)

→ 연결리스트에서는 삽입 메서드를 직접 삽입할 수 없음

→ 데이터를 넣은 노드를 생성 후 추가 해야함

1. 입력 데이터 E를 이용해 새로운 노드 n을 생성함 : n=Node(E)
2. n의 링크가 시작 노드를 가리키도록함 : n.link=top
3. top이 n을 가리키도록 함 : top=n

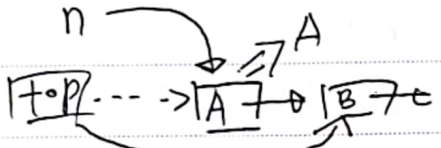
삭제연산 과정

→ 상단 항목을 꺼내서 반환하는 연산

→ top이 가리키는 노드를 꺼내고 데이터필드만을 반환

→ 반환은 노드가 아닌 스택항목을 반환

→ 메모리에게 신경쓸 필요없음

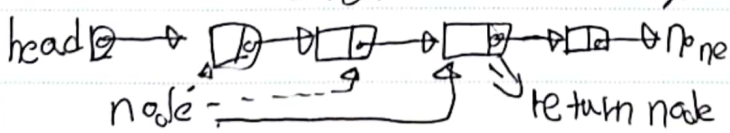


(리스트가 공백상태라면 None을 반환 ← 연결리스트의 이야기)

1. 변수 n이 시작 노드를 가리키도록 함 n=top
2. top이 다음노드를 가리키도록 함 top=n.link
3. n이 가리키는 노드의 데이터를 반환함 return n.data

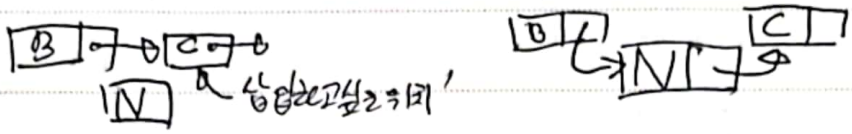
연결리스트 — (스택과 달리 리스트는 항목의 삽입, 삭제가 어느위치에서도 가능
스택보다 복잡한구조)

1) head가 가리키는 목표을 node가 가리키게함 node=self.head
2) 위치(pos)가 0보다크고 node가 None이 아닐때까지 pos>0 and node!=None
3) 노드가 가리키는 다음목표을 노드라고 하고위치(pos)를 1씩감소
4) 노드가 가리킬 목표가 while문을 만족 못하면 반환



삽입연산

→ Pos 위치에서 새로운 노드를 삽입하려면 그 노드의 선행노드를 알아야 함

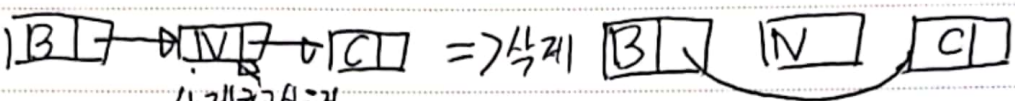


1) 노드N이 노드C를 가리키게 함

2) 노드B가 노드N을 가리키게 함

삭제연산

→ 삭제연산도 마찬가지로 before 노드를 알아야 삭제가능



1) before의 link가 삭제할 노드의 다음노드를 가리키도록 함

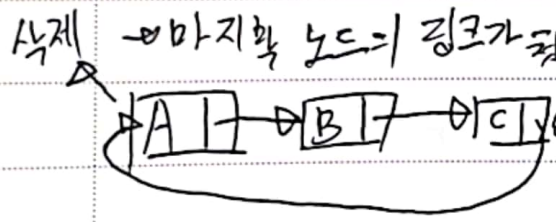
연결된 큐

→ 연결된 구조로 구현한 큐 = 연결된큐

→ 크기가 제한되지 않고 필요한 메모리만 사용가능

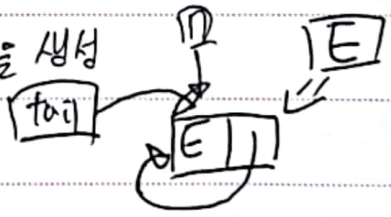
→ 코드가 더 복잡하고 링크필드 때문에 메모리공간을 더 사용

→ 마지막 노드의 링크가 첫 노드를 가리킴



삽입연산

→ 입력데이터 E를 이용해 새로운 노드 n을 생성



→ n의 링크가 자신을 가리키도록 함

→ tail이 n을 가리키도록 함

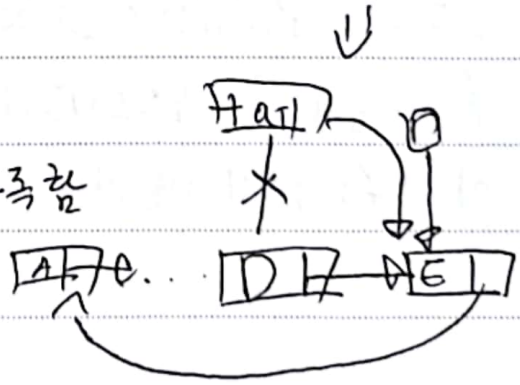
2번째

→ n의 링크가 front를 가리키도록 함

tail이 n을 가리키도록 함

링크n을

tail이 n을 가리킴

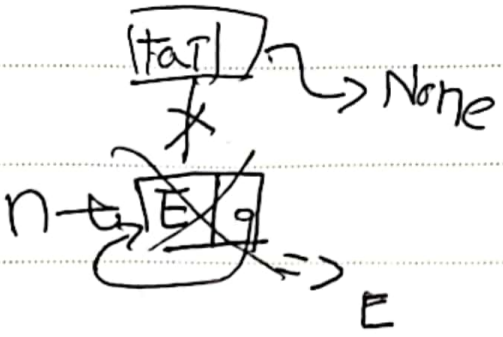


삭제연산

n이 전단노드(front)를 가리키도록함

tail이 None을 가리키도록 함

n이 가리키는 노드의 데이터를 반환함



Case 2

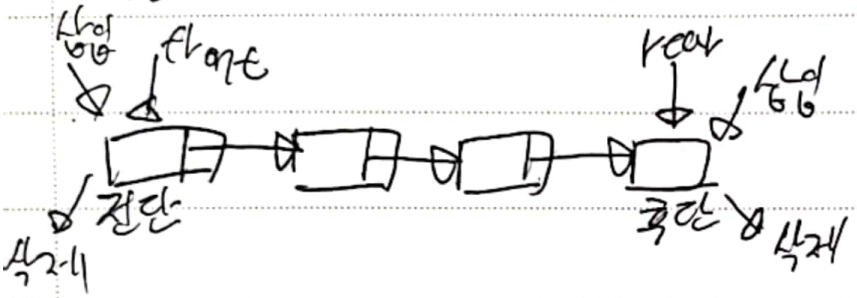
n이 전단노드(front)를 가리키도록 함

tail의 링크가 front의 링크를 가리키도록함

n이 가리키는 노드의 데이터를 반환할

연결된 덱

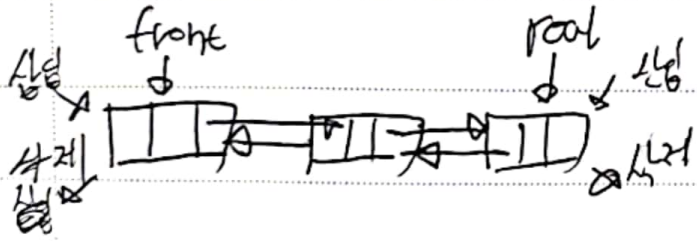
덱을 단일 연결 리스트로 표현가능



이중 연결 리스트로 구현한 덱

노드마다 그쪽의 링크를 관리해야 함으로복잡

but, 삽입/삭제 연산들을 가장 효율적으로 구현가능

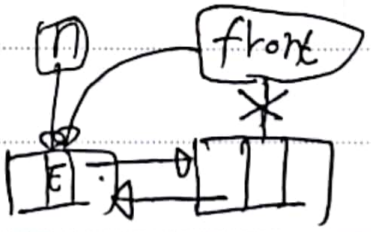


삽입연산

노드생성 및 Prev, next 초기화

front가 선형 노드로 n을 가리킴

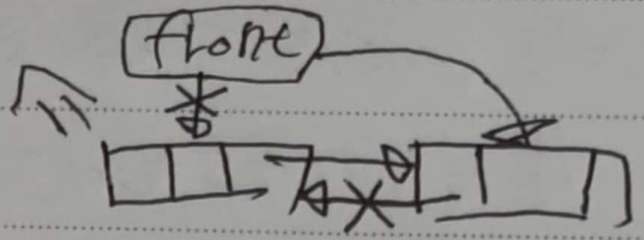
이제 front가 n을 가리킴



후단 삭제 연산

삭제할 노드(front)의 데이터 보고

front를 다음으로 옮김



front의 이전 노드를 None: front.prev = None

데이터 반환