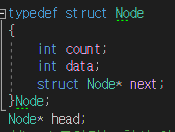
리스트에 대해서

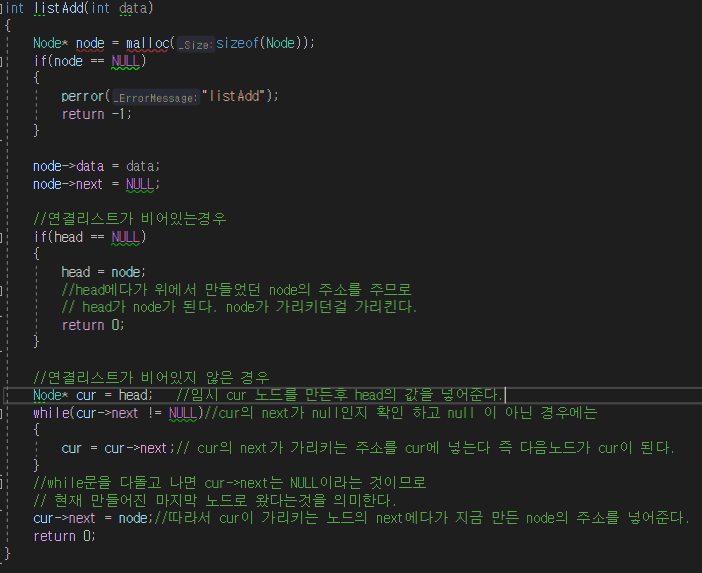
리스트는 노드라는 것이 있어서 노드끼리 연결이 되어있어서 다리를 놓아서 연결되는 느낌이 있다. 리스트에 대해 알아보자.

일단 노드부터 설계를 한다.

옆의 코드인 노드 구조체이다. 각각의 노드에는 당연히 있어야할 데이터와 다음 노드를 가리키는 next 포인터가 있고 총 몇 개가 있는지 알기 위한 count가 있다.

그리고 head 포인터를 뒀는데 head포인터는 stack에 만들어지고 여기에서 할당을 하며 첫노드가 만들어지는 것이다.

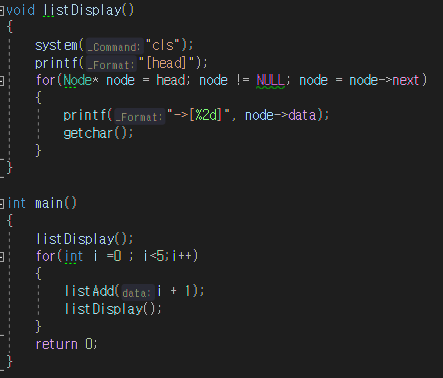
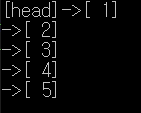
head포인터는 항상 처음 노드만을 가리키게 만든다. 이것을 수정하려면 그 이전의 노드들에게 갈 수가 없기 때문이다. 따라서 current node를 가리키는 포인터를 만든다.



step 1. 데이터를 마지막에 추가하는 listAdd함수를구현해본다.

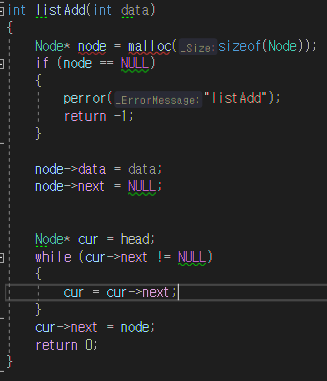
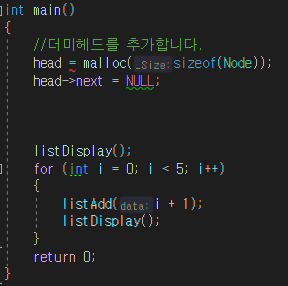
먼저 Node\*를 만들어서 동적할당을 한다. 동적할당에 실패를 할시 에러를 낸다. 노드가 만들어졌으면 그 노드에 데이터를 넣고 이 노드가 마지막 노드가 될 테니 next는 NULL을 가리키게 한다. 그리고 만약 연결리스트가 비어있는 경우에는 head에다가 노드의 주소를 줘서 head가 방금만든 node포인터가 되도록 한다. 즉 node포인터가 가리키던걸 가리킨다.

그리고 연결리스트가 비어있지 않는 경우는 그마지막 노드까지 가야한다. 그 마지막 노드를 가리키기 위한 노드 cur을 만든다. head에서 시작해서 next가 NULL일때까지 이동을 하면 결국엔 마지막 노드까지 오게 된다. 그렇게 되면 이제 할일은 별로없다. cur->next에 아까만든 node포인터를 넣어줘서 node를 가리키게 하면 된다.

step 2.이전 자료구존느 삽입 알고리즘이 이원화되어 있다는 단점이 있다. 이를 해결 하기위해 더미 헤드를 삽입한다. 이원화라는 이유는 listAdd에서 하나의 주제가 두개의 식으로 나온다는 것이다. 위의 코드에서는 연결리스트가 있을때와 없을때로 구분해서 코드를 넣은것처럼 말이다.

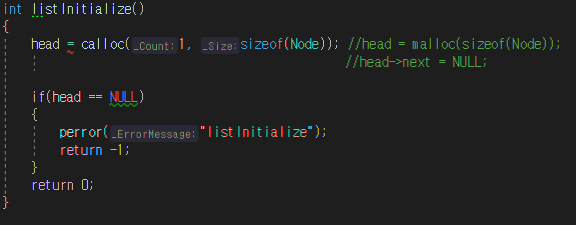
따라서 처음에 아무노드가 없을때도 head가 가리키는 아무의미 없는 노드인 더미헤드를 만들어서 그것을 가리키도록 한다. 이렇게 하면 노드가 있을때랑 똑 같은 동작을 하게 해서 하나의 식으로 완성시키게 된다.

노드를 만들고 데이터와 next를 설정하는것에 있어서는 위의 코드와 동일하다. 하지만 연결리스트가 있는지 없는지는 확인하지 않고 cur을 만들어서 head의 값을 넣는다. 그리고는 NULL이 있을때까지 반복해서 마지막노드까지 이동한다. 

이미 임시로 더미헤드를 만들어놨으므로 listAdd를 수행하는데는 아무지장이없다.

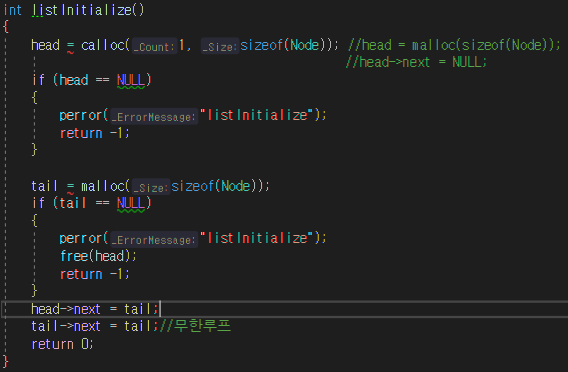
이제 메인함수에서 동적할당을 하게하지않고 미리 초기의 설정을 만들기 위해서 함수를 만든다.

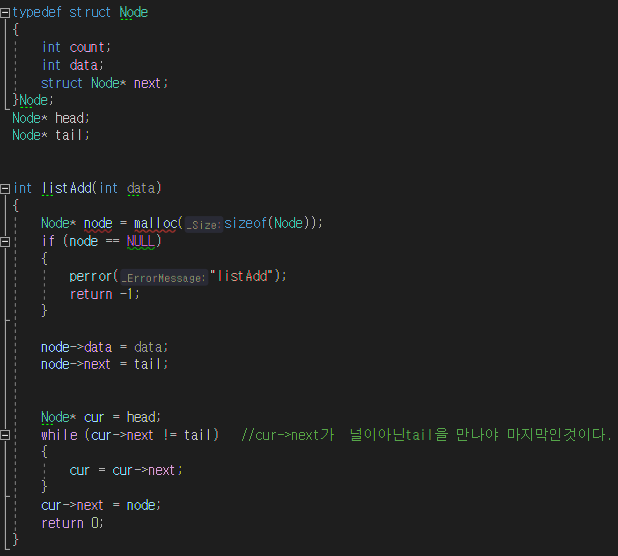
listInitailize()함수이다.



step 3. 이전에는 자료구조의 초기화를 사용자가 해야한다는 단점이 있다. 이를 해결하기 위해 위와 같은 자료구조를 초기화하는 함수를 제공한다.

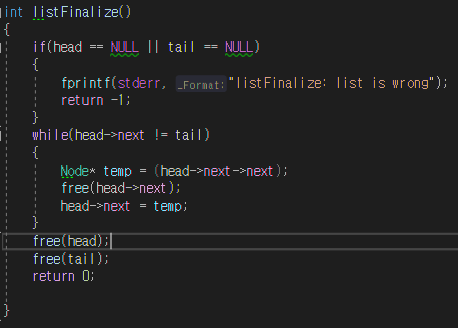
step 4. 이전 자료구조는 마지막 노드를 식별하기 위해 널을 사용하고 있다. 이는 잠재적으로 널 참조가 발생할 위험이 있다. 이를 해결하기 위해 더미 테일을 도입안다. 멀티스레드에서는 매우 효과적이다. 왜냐하면 프로세스가 죽지 않기 때문에 디버깅을 하기 좋기때문이다.

head 뿐만 아니라 tail도 동적할당을 해준다. 그리고 나서 초기화에서는 아무 리스트가 아직 추가되지않았으므로 head->next는 tail을 가리키도록 하고 tail->next 도 tail을 가리키게 한다. tail에서 잘못가도 무한루프를 돌며 시스템이 멈추진 않을 것이다.



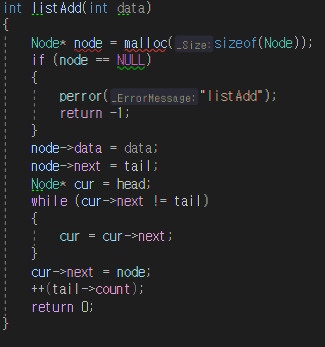
노드가 추가가 되면 노드가 가리키는 다음 노드는 tail이어야 한다. 그리고 cur->next로 끝까지 이동하면 cur->next가 NULL이 아닌 tail이어야 한다. 이렇게 해야 양쪽에 더미노드들이 위치하고있고 NULL값을 참조할 일도 없다.

step 5. 이전 자료구조는 내부적으로 사용되는 더미 헤드와 더미 테일을 삭제하지 않으므로 메모리 누수 발생한다는 문제가 있다. 이를 해결하기 위해 마무리 함수를 도입한다.

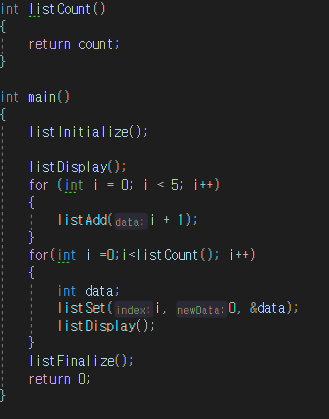
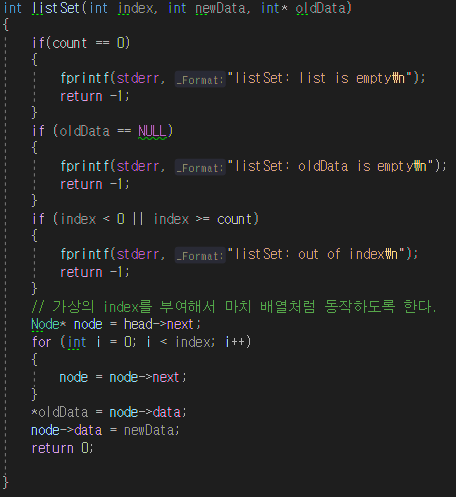
마무리함수에서는 동적할당되었던 모든 애들을 해제해주는역할을 한다.

먼저 연결리스트로 head가 가리키는 첫번째부터 차례로 free를 해준뒤 마지막에 head와 tail을 free해준다.

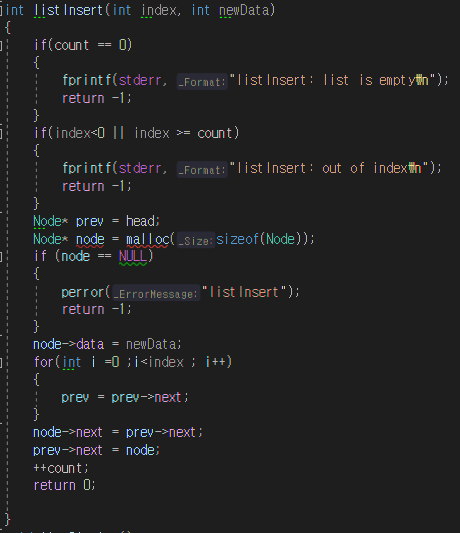
step 6. 리스트에는 배열처럼 index의 개념이 없다. 따라서 배열처럼 index를 가지고 사용하는효과를 내기 위해서 tail에 count를 도입한다. listAdd를 할때마다 tail->count에다가 하나씩 더해주는 것이다.

물론 초기화함수에서는 tail->count = 0으로 세팅을 해놔야 이것을 사용할 수 있을것이다.

이렇게 count를 만들어놓으면 이제 원하는 index자리에다가 새로운 데이터를 넣어보자. 배열편에서 그랬듯이 oldData는 사용자가 알아서 처리하도록 한다.

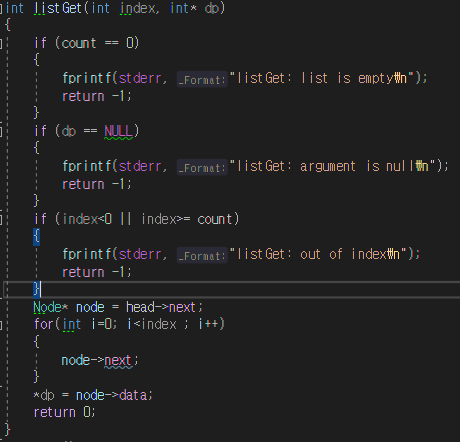
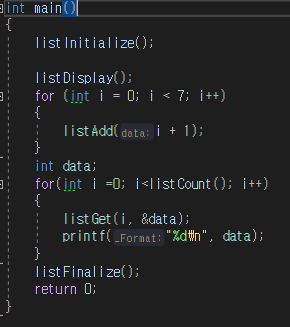


step 6. 이제 인덱스처럼 사용도 가능해 졌으니 마지막에 새로운 노드만 추가하는 것이아닌, 내가 원하는 자리에도 노드를 추가할 수 있도록 해보자. listInsert함수를 만든다.

count를 노드안에 설정해놨었는데 그냥 밖에다가 놓았다. 전역변수로 설정을 했기에 아무때서나 count를 이용할 수 있다.

먼저 count가 0 인지를 확인하고 index의 범위가 유효한지 체크한 후 prev라는 노드포인터 하나를 만들어서 head의 값을 넣고, node라는 노드포인터를 생성해서 data에 newData를 넣는다. prev를 이용해서 원하는 index 전까지 이동하고 node->next = prev->next를 해서 원래 index자리에 있던 노드를 next로 가리키게 한다. 그 후 prev->next = node 를 하여 index-1번째 노드가 새로만든 node를 가리키게 한다. 이렇게 하면 자연스럽게 새로만든 node가 그 사이에 끼어 들어가게 되는것이다. 당연히 빼먹지 말고 하나가 추가됬으므로 count++을 해준다.

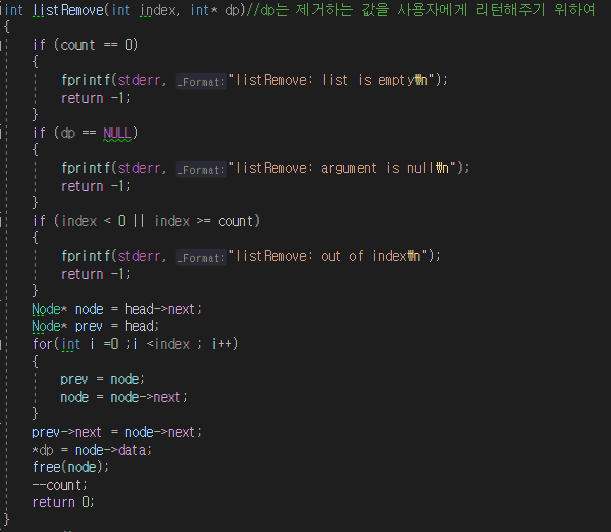
step 7. 이제 리스트에서 원하는 index의 값을 가져온다. 읽는다는 뜻이다.



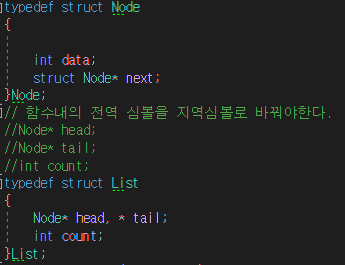
listGet함수를 사용하게 되면 값을 읽어야하므로 메인함수 즉 사용자 측에서 읽어온 값을 저장할임의의 data를 만들고 그 주소를 인자로 보내면 listGet에서는 포인터 dp로 받고 listGet에서 노드포인터를 만들어서 index를 가리키면 dp포인터를 역참조해서 node포인터가 가리키는 data를

넣는다. 이렇게 되면 임시의 메인에서 만든 data에 값이 들어오게 되고 그것을 printf()를 통해 읽으면 된다.

step 8. insert처럼 넣는함수도 만들었으니 이번에는 원하는 곳의 노드를 제거하는 함수도 만들어보자.

db는 역시 위에서 get처럼 사용자에게 제거하는 값을 처리하라고 리턴해주기 위해 사용한다. Node\* node, prev를 만들고 node = head->next, prev->head 의 값을 넣어준다. for문을 통하여 node는 index의 노드를 가리키게 되고 prev는 그 바로 이전의 노드를 가리킨다. prev->next 가 node->next를 가리키게 하여 가운데 index노드를 건너뛰게 하고 dp를 통해 node->data를 넣어준뒤 index번째의 노드를 해제한다. 물론 카운트도 1을 빼준다. 이렇게 되면 index번째의 노드만 쏙빼서 해제하게 되는것이다.

step 9. 이제 거의 함수는 정의가 되었다. 하지만 문제는 이 코드에서는 전역 심볼을 사용한다는 것이다. 함수내의 전역 심볼을 지역심볼로 바꿔야한다.

어차피 head,tail,count는 뗄래야 뗄 수 없는 사이이므로 List라는 구조체로 묶는다.

대신에 이제 이 세개의 심볼을 사용하는곳에는 List도 인수로 넣어줘야한다.

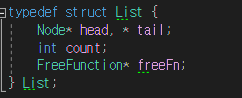
그리고 그냥 head가 아닌 list->head 이런식으로 참조를 해야한다.

step 10. 이전 자료구조는 정수만 저장할 수 있다는 단점이 있다. 이제 모든 타입을 저장하기 위해 void\*를 도입한다. 이것은 배열 편과 동일하므로 생략하겠다. void\*로 모든 타입을 가리키게 하여 어떤값이든 받을 수 있게 하고, display즉 출력함수에서는 toString함수를 이용하여 void\*를 그 타입으로 변환하여 그 크기만큼을 읽을 수 있게 만들어준다.

step 11. 이전 자료구조는 연결 리스트 안에 저장된 데이터의 정보를 알 수 없다. 때문에 내부적으로 동적 할당된 자원이 있을 경우, 메모리 누수가 발생할 수 있다. 이를 해결하기 위해 저장된 데이터를 사용자에게 전달하여 사용자가 자원을 해제하도록 코드를 변경한다.

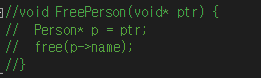
자원해제를 위한 콜백 함수타입을 선언한다.

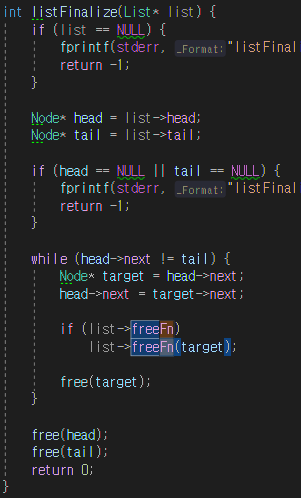


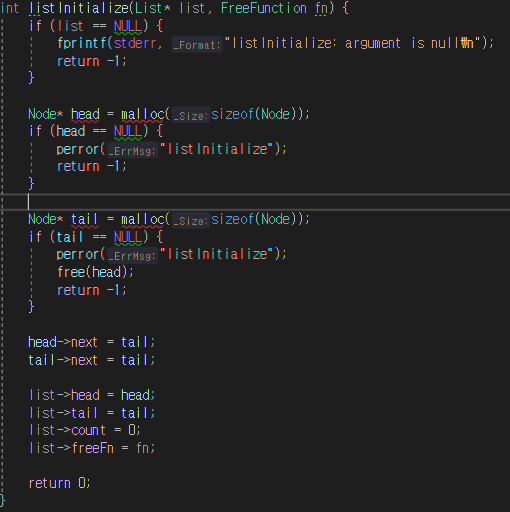


List구조체에 freeFn이라는 이름으로 함수를 가리키는 포인터를 지정한다.









초기화함수에서 list안에 있던 freeFn을 NULL로(fn)으로 설정한다.

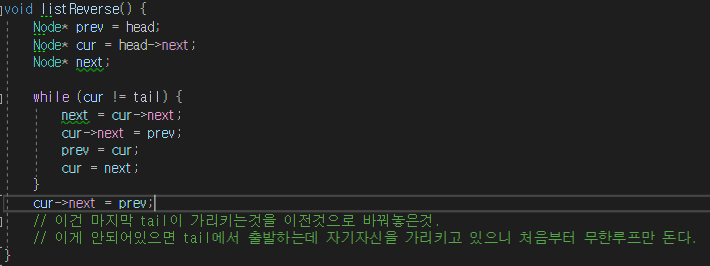
만약 사용자가 만든 구조체에서 그 구조체내부에서 또 동적할당을 한다면 해제를 이함수로 해주는 것이다. 따라서 해제를 해줘야할 때 freePerson이라는 함수를 만들어서 할당해제를 하는것이다.

하지만 만약 사용자가 만든 구조체가 동적할당을 하지 않는다면 전혀 걱정할 필요가 없으며 이 메소드를 사용할 일도 없다.

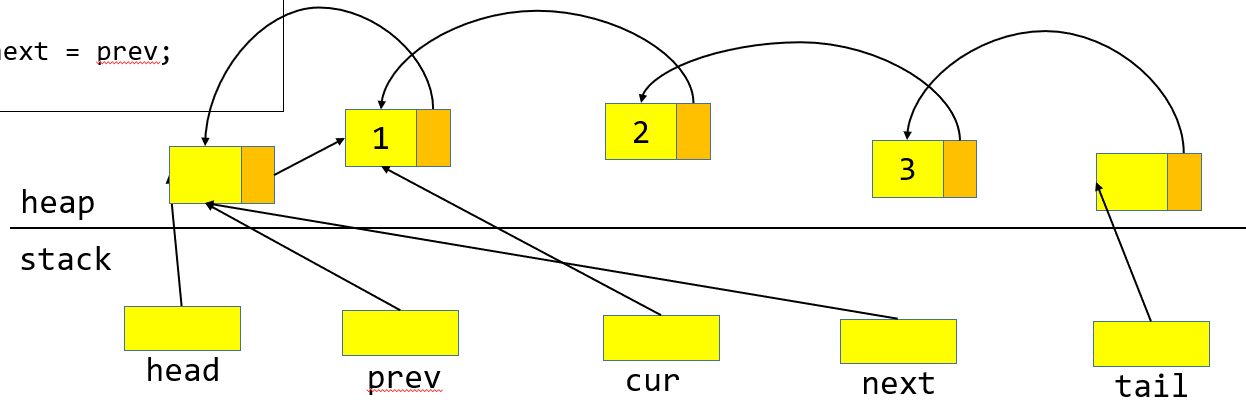
이렇게 단일 연결 리스트는 끝이 난다. 물론 모듈화를 해줘야하는 것은 당연하다 은닉을 위해.

그럼 단일 연결 리스트는 문제가 무엇인가. 역방향탐색이 어렵다는 단점이 있다. 따라서 이것을 해결하기 위해 자료구조의 포인터를 다음이 아닌 이전 노드를 가리키도록 변경한다.

listReverse함수를 구현해본다.

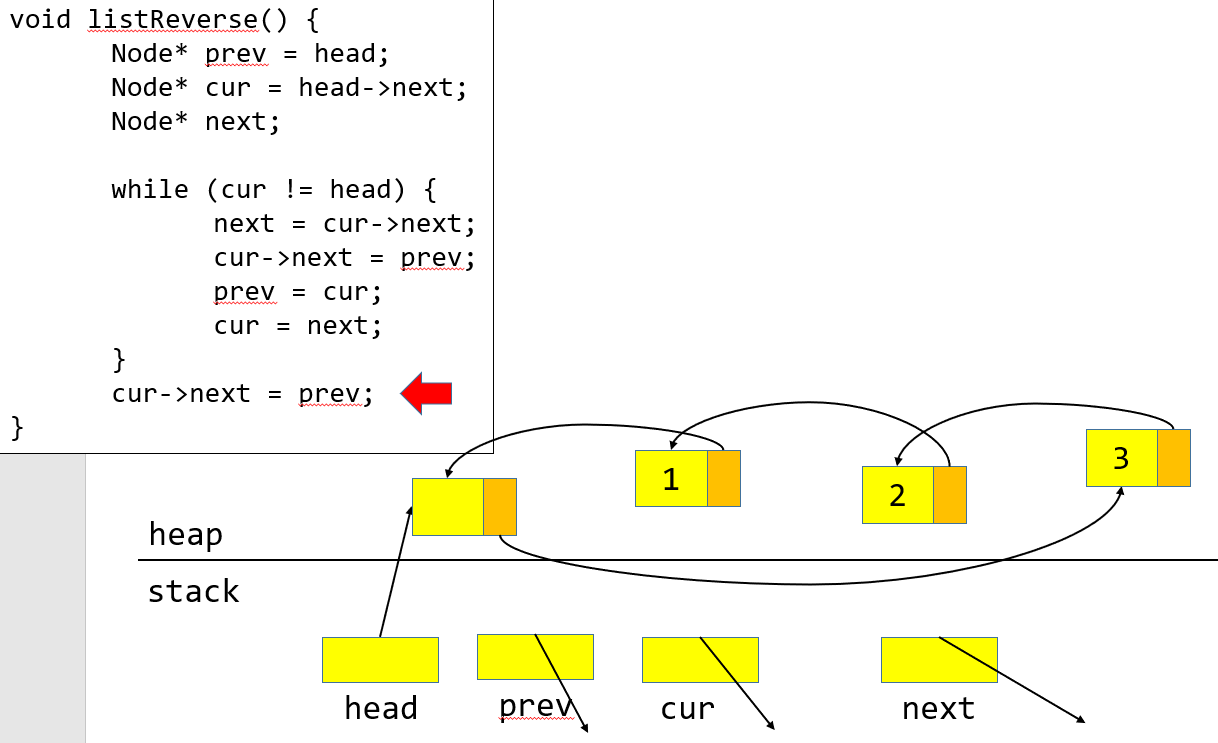


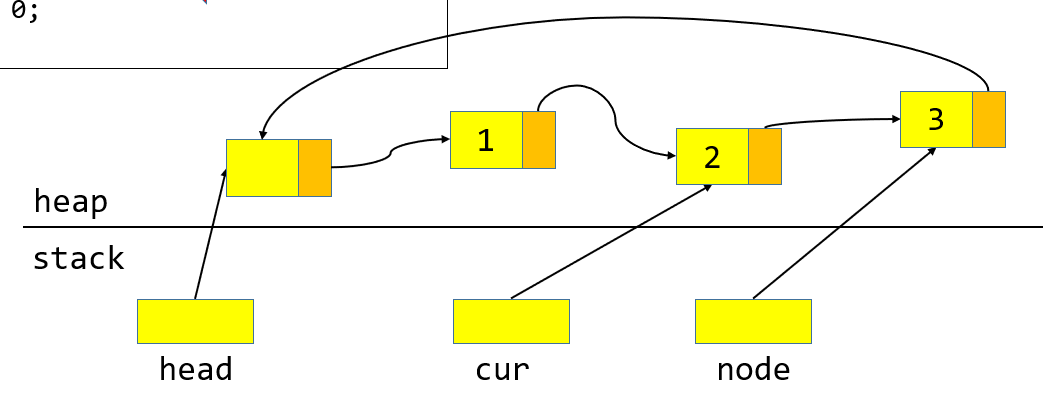
제일 먼저 next로 cur->next의 값을 넣어놓는다. 그런다음에 cur->next 에 prev즉 이전의 노드를 가리키게 하고 prev에는 cur의 값을 넣어서 한칸 이동을 한다. 그런다음 cur 에 next에 prev값을 넣어서 마지막노드까지 전거의 노드를 가리키게 설정한다. 이것은 마지막tail이 가리키는 것을 이전것으로 바꿔놓기위함이다. 이게 안되어있으면tail에서 출발하는데 자기자신을 가리키고 있으니 처음부터 무한루프가 돌게 될것이기 때문이다.



이전 자료구조는 자원 해제가 정상적으로 되지않는다는 문제가 있고 또 다른 문제는 역방향을 순순방향으 변경했을 때, 역시나 정상적으로 동작하지 않는다는 문제가 있다. 이를 해결하기 위해 플래그와 같은 기법을 사용할 수도 있지만 이는 자료구조의 사용을 매우 지저분하게 만든다는 단점이 있다. 이를 해결하기 위해 간단하게 더미 테일을 삭제한다.

이것이 원형연결리스트라는 것이고 이렇게 하면 순방향 역방향이 하나의 식으로 된다.





listReverse식을 따라가보면 이해할수 있다.