안녕하세요? 여러분, 안녕하세요. 먼저 이 강의를 직접 진행하지 못해 사과드리고 싶습니다. 글쎄요, 언제 무슨 일이 일어날지 알 수 없잖아요. 다시 한번 죄송합니다. 하지만 이 강의를 계속 진행해 보겠습니다. 지난번에 우리는 연결 리스트에서 1번째를 추가하는 방법을 배웠고, 그 과정이 매우 쉽고, 아주 단순하고, 계산적으로도 가볍다는 것을 배웠습니다. 그리고 여기에는 포인터 재배치나 스티칭이 많이 포함되는데, 제가 부르고 싶은 대로, 상당히 직관적이고, 시간 복잡도가 매우 높았습니다. 여기에는 어떤 종류의 루프도 포함되지 않았고, 여기의 모든 문장은 비교, 할당, 산술 연산 때문에 모두 간단한 문장입니다. 이제 연결 리스트의 첫 번째 요소나 첫 번째 노드를 제거하는 것을 시도해 보겠습니다. 그리고 add first.st의 정반대가 될 겁니다. 왼쪽에 보이는 코드는 remove first, st의 주요 메서드의 일종이며, 제거한 요소를 반환해야 한다는 것을 알 수 있습니다. 모든 종류의 파괴적 업데이트의 공통적인 특성이니까요. 무언가를 제거할 때마다 또는 무언가를 덮어쓸 때마다. 그런 다음 이전 값을 호출자에게 반환해야 합니다. 그래서 여기서 우리는 1번째 포인터, 즉 레이크 리스트의 헤드 포인터를 캐싱합니다. 따라서 F는 이제 바로 1번째 노드, 즉 제거하려는 대상 노드를 가리키고 있습니다. 그리고 F가 null이면 전체 연결 리스트가 비어 있다는 것을 의미합니다. 맞죠? 따라서 1번째 포인터가 null이면 가리킬 것이 아무것도 없고, 이는 이 연결 리스트에 아무것도 없다는 것을 분명히 나타냅니다. 알았습니다. 이제 빈 연결 리스트에서 무언가를 제거하거나 빈 컨테이너를 비우려고 하면, 그러면 작동하지 않을 것입니다. 좋아요, 이건 일종의 오류 사례입니다. 일반적으로 오류가 발생했음을 나타내기 위해 null을 반환하려고 하죠, 맞죠? 아주 흔한 일이죠. 하지만 여기서는 null을 반환하는 대신 예외를 발생시키고 있습니다. 알겠어요? 그리고 이건 제가 앞서 설명하지 않았기 때문에 매우 비용이 많이 드는 일처럼 보입니다. 하지만 예외를 발생시키면 많은 계산 주기가 소모됩니다. 그리고 사용자 경험 측면에서도 매우 나쁩니다. 기본적으로 현재 실행 흐름을 망가뜨리기 때문입니다. 좋아요, 사용자가 경쟁 중간에 갑자기 쫓겨나고 시스템이 제대로 작동하지 않을 수 있습니다. 따라서 예외를 발생시키는 것은 매우 신중하게 해야 하는 일입니다. 그런데도 빈 목록에서 무언가를 제거하려고 한다고 해서 예외가 발생하는 것입니다. 맞죠? 그럼 왜 null을 반환하는 대신 이런 일이 일어날까요? 여러분이 생각해 보시고, 아직 할 수 있다면요. 답을 생각해 보시고, 내일 리뷰 세션에서 질문해 보세요. 하지만 어쨌든, 그게 remove first, st의 공식 동작이고, F가 null이 아니면, 우리는 F, 즉 1번째 포인터를 전달하여 unlink 1st 헬퍼 메서드를 호출할 것입니다. 좋아요. 그럼 unlink 1st는 오른쪽에 있는 이 녀석입니다. 먼저, st는 1번째 노드의 항목을 캐시할 것입니다. 그래서 element는 이제 item을 가리키고, 1번째 노드의 다음 포인터도 가리킵니다. 좋아요?

그리고 나서 우리는 그 두 구성 요소를 명시적으로 null로 설정할 것입니다. 알다시피, f dot item은 null f dot입니다. Next는 null이고, 그 단계는 동작 측면에서 실제로 필요하지 않습니다. 하지만 우리는 가비지 콜렉터를 도와 이러한 객체가 가능한 한 빨리 회수되도록 하려고 합니다. 그리고 나서 1번째 포인터가 next를 가리키도록 설정할 것입니다. 이는 올바른 일입니다. 왜냐하면 1번째 노드를 제거하는 경우 그 다음에 오는 노드가 새로운 1번째 노드여야 하기 때문입니다. 맞죠? 그래서 1번째 포인터가 다음 노드를 가리키도록 하는 것입니다. 그리고 이 마지막 부분은 약간 혼란스러울 수 있습니다. 하지만 그렇지 않습니다. 매우 합리적인 일입니다. 따라서 next가 null이면. 좋아요, 그럼 next는 이 사람입니다. 따라서 f dot, next는 null입니다. 즉, 단일 요소만 포함된 리스트에서 무언가를 제거하려고 한다는 뜻입니다. 맞죠? 왜냐하면 첫 번째 next 포인터가 null이면 next가 없고, 첫 번째만 있기 때문입니다. 따라서 이 목록에는 단일 항목만 포함되어 있고 방금 제거했습니다. 알겠어요? 이제 빈 목록이 생성되어야 합니다. 연결 목록의 경우 빈 목록을 나타내려면 마지막 포인터도 null로 설정해야 합니다. 이것이 if 부분 내부에서 발생하는 일입니다. 이제 else 부분은 몇 가지 다른 요소가 남아 있음을 의미합니다. 따라서 이제 첫 번째 항목을 제거한 후에도

element, 여전히 비어 있지 않은 리스트가 있습니다. 그리고 그 경우 next dot previous를 null로 설정합니다. 기억하세요, 이것은 이중 연결 리스트이고 모든 노드의 previous 포인터는 그 앞에 있는 노드를 가리켜야 하기 때문입니다. 알겠어요? 하지만 이 지점에서 next는 새로운 헤드입니다. 그래서 그 앞에는 아무것도 없어야 합니다. 그래서 previous 포인터를 null로 설정하는 것입니다.
그리고 이것으로 연결 해제된 절차가 끝납니다. 크기를 하나 줄이는 것을 잊지 말고, 방금 제거한 요소를 반환해야 합니다. 알겠어요? 그리고 그것도 remove 1st에서 제거됩니다. 따라서 remove 1st의 전체 시간 복잡도는 add, first, st와 같습니다. 왜냐하면, 그것은 예외 생성자에 대한 생성자 호출을 하기 때문입니다. 하지만 이것 자체는 연결 리스트의 크기와 관련이 없습니다. 알겠어요, 그것은 상수입니다. 연결 리스트와 관련하여, 다른 모든 것은 할당, 비교 및 산술 연산자일 뿐입니다.

그래서 그것들은 모두 간단한 문장이고, 루프가 없습니다. 그래서 이 전체 연산은 상수 시간 안에 수행될 수 있습니다. 그래서 우리는 또한 remove 1st가 매우, 매우 효율적으로 수행될 수 있다는 것을 알고 있습니다. 좋아요, 그럼 다음으로, add last를 살펴볼 것입니다. 이것은 목록의 끝에 새 요소를 추가하는 연산입니다. 이것은 Arraylist에서 배운 append 연산입니다. 그래서 그냥 배열 목록의 일반 ad와 같습니다. 그저, 아시다시피요. 그것은 last에 특별한 이름을 가지고 있습니다. 좋아요. 그래서 add last는, 글쎄요, 사실 그것입니다. add의 메인 메서드를 거쳐야 합니다. Last, 여기서 그 코드를 생략한 것 같습니다. 하지만 실제로 가장 많이 일어나는 일은, 흥미로운 일이 일어나는 메서드는 link last라는 또 다른 도우미 메서드입니다.

따라서 이 링크는 마지막으로 E가 가리키는 요소를 제거하려고 시도합니다.
좋아요. 알았어요. 그래서 링크 또는 죄송하지만 마지막에 추가의 동작은 첫 번째, st를 추가하는 것과 거의 같지만, 이제 이전을 사용해야 합니다. 이제 첫 번째 포인터 대신 다음 포인터와 마지막 포인터를 사용해야 합니다. st 그래서 대략적인 애니메이션 버전은 이렇게 하는 것입니다. 좋아요? 그럼 먼저 st 삽입하려는 새 데이터를 포함하는 새 노드를 구성한 다음 마지막 포인터가 새로 구성된 노드를 가리키도록 합니다. 동시에 이전 마지막 다음 노드는 이제 새 노드를 가리켜야 합니다. 맞죠? 이제 이 사람이 새 마지막이 되기 때문입니다. 그래서 이전, 아시다시피, 이전 마지막은 이제 이 새 마지막보다 앞서야 하고, 이전 포인터가 이렇게 이전 마지막을 가리키도록 하는 것을 잊지 마세요. 그리고 일이 이렇게 진행됩니다. 그리고 이 절차는 이 코드에 요약되어 있습니다. 마지막 L 변수 L은 마지막 포인터를 가리키고, 데이터를 제공하여 새 노드를 구성합니다. 이전 노드, 즉 이전 마지막 노드. 그리고 다음 노드는 없습니다. 아니요, 이게 새로운 마지막이어야 하거든요. 그래서 우리는 그냥 null을 전달하고, 그러면 마지막 포인터가 여기 새 노드를 가리키게 됩니다. 그래서 이게 3번째 줄인데, L이 null이면, 새로운 마지막이 null이라는 걸 알 수 있다는 뜻입니다. 죄송합니다, 선생님. 이전의 마지막은 null이었습니다.

알겠어요? 그리고 그것은 당신이 빈 리스트에 무언가를 추가하려고 한다는 것을 의미하는데, 이 경우 새로 생성된 노드도 첫 번째 노드여야 합니다. 그래서 당신은 first, st를 할당하거나 새로운 노드에서 1번째 포인트를 갖는 것입니다. 그렇지 않고, 만약 당신이 그것을 기존 리스트에 추가했다면, 이전 last의 다음 포인터를 연결하여 새로운 노드를 가리키도록 한 다음 크기를 증가시켜야 합니다. 그리고 우리는 그것을 끝냈습니다. 이 작업은 또한 매우 효율적입니다. 왜냐하면 이전과 마찬가지로 루프가 없고 모든 간단한 문장이 있기 때문입니다. 그리고 이것은 상수 시간 작업입니다. 맞죠? 그러니 first를 추가하고, st를 마지막에 추가하고, first, st를 제거하고. 곧 알게 되겠지만, remove blast는 모두 상수 시간 안에 완료될 수 있고, 저는 그냥 이야기하고, 바로 이야기하겠습니다.

따라서 remove last는 다를 바 없습니다. 글쎄요, remove first, st와 크게 다르지 않아서 반대쪽 거울과 같습니다. remove의 대칭 버전과 같습니다. 이해하신다면, first, st, remove first, st, remove last는 아주 쉬운 일입니다. 좋아요. 그럼 바로 unlink last 헬퍼 메서드로 넘어가겠습니다. 그리고 L은 이제 마지막 노드를 가리킵니다. 좋아요, return@theendl.previous로 갈 요소가 있습니다. 이는 이전 last의 이전 노드입니다. 그리고 가비지로 갑니다. 이러한 항목을 수집하면 이전 노드를 가리키는 마지막 포인터가 생깁니다. 이는 새로운 마지막 노드입니다. 마지막 노드를 제거하기 때문입니다. 그 전 노드는

새로운 마지막 노드가 됩니다. 이제 이전 노드가 null이면 연결 리스트의 유일한 기존 항목을 제거한 것입니다. 그러면 첫 번째 포인터도 null로 설정해야 합니다. 이제 리스트가 비어 있기 때문입니다. 그렇지 않으면 이전 노드의 다음 포인터를 null로 설정합니다. 이전 노드를 제거하면 그 전 노드가 새 노드, 새로운 마지막 노드가 되기 때문입니다. 따라서 다음 노드가 없어야 합니다. 그래서 null로 처리합니다.

크기를 감소시키고 요소를 반환합니다. 알겠어요? 그러니까, 전에 말했듯이, 이건 모두 상수 시간이에요. 같은 이유로 1번째를 제거하는 것과 마찬가지예요. 그리고 이것을 배열 목록과 비교해보세요.
상응하는 버전. 맞죠? 그리고 강의 마지막에 그 비교 차트를 다루겠습니다.
좋습니다. 이제 get 1st와 get last가 있는데, 이는 목록의 첫 번째 항목을 반환하려는 것을 의미합니다. 그리고 목록의 마지막 항목도요. 알겠습니다. 그리고 이것은 정말 정말 쉬운데, 우리는 항상 첫 번째와 마지막 노드에 대한 포인터를 유지하기 때문입니다. 그러니 get first.st 우리는 단순히 첫 번째 항목을 가져옵니다. 그리고 그것이 null이면, 우리는 다시 예외를 throw할 것입니다. 왜 null을 반환하는 대신 예외를 throw할까요? 생각해 보세요. 그렇지 않으면 우리는 그 두 노드의 항목을 반환할 것입니다. 그리고 시간 복잡도는 두 경우 모두 분명히 상수 시간입니다. 그리고 이제 우리는 연결 목록의 더 흥미로운 부분으로 넘어갈 것입니다. 그래서 이것은 전형적인 삽입 방법입니다. 우리는 배열 목록에 같은 방법을 가지고 있습니다. 맞죠? 우리는 배열 목록을 가지고 있습니다.

여기에 같은 것을 삽입하기 위해 인덱스와 객체를 허용하는 것을 추가합니다. 하지만 동작은 분명히 동일할 것입니다. 하지만 구현은 크게 다를 것입니다. 배열 목록과 달리 우리는 그렇지 않기 때문입니다. 연결 목록 클래스에서는 더 이상 임의 액세스가 없습니다. 알겠습니다. 여전히 인덱스를 사용하여 구현, 가져오기 또는 설정할 수 있지만, 그것은 임의 액세스의 환상을 제공할 뿐입니다. 실제로는 처음부터 그 인덱스 지점까지 선형 시간이 소요되는 이동을 해야 합니다. 따라서 임의 액세스가 없습니다. 따라서 삽입도 그렇게 잘 되지 않을 것입니다. 배열 목록과 같지 않을 것입니다.

추가하세요. 하지만 생각해보면 아무것도 아닐 겁니다. 말하자면, 알다시피요. 배열 리스트보다 더 어렵죠. 추가하세요. 항상 말했듯이 포인터를 재배열하는 문제일 뿐입니다. 좋아요. 이 시나리오를 고려해 보세요. 기존 연결 리스트가 있다고 가정해 보겠습니다. 첫 번째, st, 마지막이 있습니다. 그리고 이것은 인덱스, 오른쪽 인덱스 1, 0 인덱스 1입니다. 그리고 인덱스 1에 새 데이터를 삽입하고 싶다고 가정해 보겠습니다. 좋아요, 이것은 이제 새로운 인덱스 1이 되어야 한다는 것을 의미합니다. 다른 모든 것은 인덱스 측면에서 하나씩 이동해야 합니다. 그럼 먼저 무슨 일이 일어나야 할까요. st 글쎄요, 새 노드를 만들고 이전 포인터가 이전 노드를 가리키고 다음 포인터가 대상 인덱스 노드, 즉 현재 대상 인덱스 노드를 가리키도록 합니다.

그 직후에, 오래된 포인터는 분리되어야 하거나, 오래된 연결 또는 오래된 링크는 그렇게 끊어져야 합니다. 알겠어요? 그리고 새로운 포인터도 그렇게 설정하는 것을 기억하세요.
알겠어요? 그러면 전체 삽입 작업이 끝났습니다. 물론 1번째나 마지막에 삽입할 때는 조심해야 합니다. 왜냐하면 경계 케이스이기 때문이고 포인터를 배열하는 면에서 특별히 조심해야 할 수도 있고 코드에서 이에 대해 이야기하겠습니다. 하지만 적어도 지금은 포인터를 할당하는 면에서 이것이 어떻게 동작해야 하는지 이해하셨을 겁니다. 왼쪽에 있는 실제 코드를 살펴보겠습니다.

그게 우리의 주요 진입점이 될 겁니다. 그게 바로 public입니다. Add 메서드입니다. 그리고 인덱스가 크기와 같으면 링크 목록의 맨 끝에 삽입하려는 것입니다. 좋아요, 그럼 인덱스의 마지막이나 링크 목록의 마지막에 추가하는 것과 정확히 같습니다. 이 경우 이미 add last에 사용한 linklast라는 매우 유용한 도우미 메서드가 있습니다. 그래서 크기를 인덱스로 제공하면 동작은 add last와 정확히 동일합니다. 그러나 그렇지 않은 경우입니다. 그리고 그런데, 이 시점에서 인덱스는 음수가 아닐 것입니다. 일반적인 인덱스 검사가 여기서 수행되기 때문입니다. 좋아요, 그렇지 않으면 Link라는 다른 도우미 메서드를 호출할 것입니다. 그리고 두 개의 인수를 받습니다. 하나는 삽입하려는 인수 또는 요소입니다. 두 번째는 바로 그 인덱스에 있는 노드입니다. 좋아요. 그럼 이전의 노드 도우미 메서드를 기억하시죠? 이

1은 처음부터 해당 인덱스 지점까지 이동하여 해당 노드를 반환합니다.
알겠어요? 그러면 주어진 인덱스에서 노드 객체를 검색합니다. signature 이전의 링크를 보면 E, 요소, 후속 노드를 수신하고 있습니다. 후속 노드라는 이름이 붙었는데, 해당 인덱스의 노드가 삽입하려는 노드의 후속 노드가 되려고 하기 때문입니다.
이를 염두에 두고 로컬 변수를 살펴보겠습니다.

첫째, 선행자 또는 예측자가 있습니다. 선행자는 후속자의 이전입니다.
좋아요, 그럼 우리가 가진 후속자나 알다시피, 타겟. 알다시피, 인덱스 위치에 있는 노드, 그 녀석의 이전 포인터가 선행자가 될 겁니다. 그러니까 이 예제 케이스에서 Pred는 이제 이 노드를 가리키고 있죠? 그럼 새로운 노드를 만들 겁니다. 그러니까 이 노드 N은 선행자를 이전 노드로 합니다. 그러니까 이 녀석은 이 녀석의 이전 노드가 되어야 합니다. 당연하죠. 데이터 E를 제공하고 후속자는 제 새로운 녀석의 다음 노드가 될 겁니다. 이 녀석을 이 둘 사이에 삽입하고 싶기 때문에 말이 되는 거죠. A와 B, 알겠어요.
그러니까 이 시점에서 새로운 노드의 이전 및 다음 포인터는 이미 올바르게 설정되었습니다. 알겠어요. 알았어요. 다음 줄의 후속자 이전은 새로운 노드 또는 죄송하지만 후속자가 되려고 합니다. 이전은 새로운 노드를 가리키고 있습니다.

알겠어요? 그럼, successor는 이 대상 위치를 의미합니다. 이 녀석의 이전은 이제 저를 가리켜야 합니다. 제가 그 자리를 차지하려고 하니까요. 알겠어요, 말이 되죠. 이제, 여기서 확인하는 한 가지는 precedence가 null인지 여부입니다. 알겠어요? 그럼 precedence는 무엇일까요? Predecessor는 successor의 이전입니다. 그래서, 이 시나리오에서는 노드 A를 가리킵니다. 하지만 precedence가 null이라는 것은 무슨 뜻일까요? 간단히 말해서 인덱스가 0이라는 뜻입니다.

알겠습니다. 인덱스가 0이라면 헤드 노드, 첫 번째 노드를 검색했을 것이고, 첫 번째 노드에는 이전 포인터가 없습니다. 알겠습니다. 그래서 선행 노드가 null이 되는 것입니다. 또한 새 노드를 0 인덱스 0의 위치에 삽입하여 새 첫 번째 노드로 만들고자 합니다. 그래서 이 경우 새 노드에 첫 번째 지점을 두는 것입니다. 그렇지 않으면 선행 노드의 다음 지점이 새 노드를 가리키도록 합니다. 따라서 이 다음은 이제 유효한 객체, 유효한 포인터입니다. 따라서 이것은 guy입니다. 이것은 이제 새 노드를 가리킵니다. 알겠습니다. 이것이 else 부분에서 일어나는 일입니다. 그래서 스티칭 부분이 완료되었습니다. 이제 크기를 하나 증가시켰고 끝났습니다.
알겠습니다. 이제 사람들에게 이 추가 작업의 시간 복잡도를 분석해 달라고 요청할 때마다 많은 혼란이 있습니다. 알겠습니다. 왜냐하면 저는 항상 add 연산, 이 전체에 대해 묻기 때문입니다. 하지만 사람들은 종종 그 지시가 이 특정 메서드에 적용된다고 착각합니다. 알겠어요? 그래서 저는 개별 도우미 메서드의 시간 복잡도를 분석하라고는 절대 요청하지 않을 겁니다. 알겠어요, 저는 항상 전체 연산의 시간 복잡도를 분석하라고 요청할 겁니다. 알겠어요, 제가 이 광고의 시간 복잡도가 얼마냐고 묻는다면, 왼쪽에 있는 이 녀석을 분석해야 합니다. 알겠어요? 그런데, 이 메서드는 여전히 add 내부에서 호출되기 때문에 이것의 시간 복잡도를 분석해야 합니다. 알겠어요. 그래서 그것도 필요합니다. 하지만 다행히도 우리에게는 매우 간단하죠? 모든 간단한 문장, 할당과 비교뿐입니다. 그래서 이것은 상수 시간 연산입니다. Link last link last는 add last와 같고, 우리는 모두 상수 시간이 걸린다는 것을 알고 있습니다. 그래서 이 함수 호출은 상수 시간입니다. 이것은 비교입니다. 인덱스 검사는 링크 목록 크기에 관계없이 상수 시간으로 수행할 수도 있습니다. 따라서 상수, 상수, 상수 link before도 상수입니다. 그래서 많은 사람들이 삽입이 일정 시간 안에 이루어질 수 있다고 주장하는 거예요.

하지만 여기서 또 다른 메서드 호출을 한다는 것을 기억하세요. 노드 메서드의 동작은 처음부터 시작하여 전체 체인 또는 연결된 구조를 반복하는 것입니다. 따라서 이는 필연적으로 루핑을 포함합니다. 알겠어요? 그리고 노드 메서드는 연결 리스트의 크기에 따라 선형 시간이 걸린다는 것을 우리 모두 알고 있습니다. 따라서 그 메서드 때문에 전체 작업이 선형 시간이 걸립니다. 따라서 이 스티칭 작업이 매우 효율적으로 수행될 수 있지만

상수 시간 내에 해당 위치로 이동하여 스티칭을 수행해야 합니다. 좋아요. 이것이 이 삽입 방법이 선형 시간 작업이 되는 주요 병목 현상입니다. 애니메이션은 이렇습니다. 이제 반대로 해보겠습니다. 주어진 인덱스에서 요소를 제거합니다. 이것이 주요 진입점입니다. 인덱스가 있습니다. 당연히 인덱스를 확인합니다. 잘못된 인덱스가 주어지면 어떤 종류의 예외가 발생하고, 왜 null을 반환하는 대신 예외를 발생시켜야 하는지 이해해야 합니다. 하지만 인덱스가 유효하면 주어진 인덱스에서 노드를 제공하는 Unlink라는 또 다른 도우미 메서드를 호출합니다. 따라서 이전과 동일한 논리를 따릅니다. 먼저 제거할 대상 노드를 찾습니다. 그런 다음 도우미 메서드 unlink를 호출하여 해당 노드에서 실제 제거를 수행합니다. 좋아요, 이제 unlink 작업에 대해 이야기하기 전에 노드를 제거하는 프로세스를 살펴보겠습니다. 매우 간단합니다. 좋아요? 그러니까 먼저 이 두 개의 다음, 죄송하지만 이 두 개의 다음 포인터를 끊어야 합니다. 좋아요, 이 연결을 끊고 그 다음에 저 연결을 끊습니다. 좋아요, 아직 끝나지 않았는데, 그런 종류의 건너뛰기 연결을 설정해야 하거든요. 좋아요? 새로운 노트나 대상 노드가 제거되면, 내 이전은 다음이 되어서는 안 되고 내 다음 이전은 이전이 되어서는 안 됩니다. 좋아요, 이건 꽤 긴 표현이네요. 하지만 이 애니메이션을 보면 정확히 그 의미입니다. 좋아요. 제가 이 스티칭을 수동으로, 시각적으로 하라고 하면, 누구나 제대로 할 수 있을 겁니다. 그리고 그게 바로 일이 이루어지는 방식입니다. 그래서 그 코드 버전은 unlink 메서드에 나와 있습니다.

먼저 대상 목록의 항목을 가져옵니다. 따라서 마지막에 반환되는 항목이 됩니다. 또한 대상 노드의 다음 및 이전 항목을 가져옵니다. 그리고 먼저 확인할 것은 previous가 null인지 여부입니다. Previous가 null이면 원하는 것을 나타냅니다. 헤드 노드를 제거하려고 합니다. 알겠어요? 그래서 첫 번째 포인터 지점을 현재 next에 설정하는 거죠? 알겠어요? 헤드 노드를 제거하려고 한다면 괜찮습니다. 헤드 노트 다음이 새 헤드가 되어야 합니다. 알겠어요? 그렇지 않으면 설정합니다. previous가 next가 됩니다. 여기 이 줄입니다. 따라서 previous는 현재 next의 다음 지점입니다. 알겠어요, 여기 이 줄입니다. 그리고 가비지 수집을 위해 대상 노트의 previous 포인터를 null로 처리합니다. 여기 이 줄입니다. 그리고 다음으로 확인할 것은 next가 null인지 여부입니다. 다음으로 null인 것은 previous가 null인 것의 거울상 케이스입니다. 즉, 마지막 노드를 제거하려고 합니다. 알겠어요? 그리고 만약 그렇다면, 새로운 last가 어디에 추가되어야 하는지 매우 분명합니다. 그래서 새로운 last는 이제 previous를 가리키고 있습니다. 알겠어요. 그렇지 않다면, 다음 previous를 previous로 설정할 것입니다. 그래서 next는 previous가 내 previous를 가리키고 있습니다. 그래서 여기 이 줄입니다. 그리고 우리는 방금 그것을 달성했습니다. 그리고 다시, 우리는 가비지 수집을 위해 대상 노트의 next 포인터를 null로 처리합니다. 그리고 마지막으로, 우리는 이것이 수행되도록 합니다. null로 설정하고 크기를 장식하고 요소를 반환하여 가비지 수집할 항목을 알 수 있습니다. 알겠어요. 그래서 삽입 메서드와 마찬가지로 이번 시간 전의 메서드입니다. 복잡도는 연결 목록의 크기에 따라 선형적입니다. 이 노드가 있기 때문입니다. 노드 도우미 메서드에 대한 호출을 호출합니다. 알겠어요, 다른 모든 것. 이것이 이것들임을 분명히 알 수 있습니다. 이것은 상수 또는 간단한 명령문의 집합입니다. 따라서 이 제거 작업도 선형 시간과 목록의 크기가 필요합니다. 알겠습니다. 그럼 clear 메서드입니다. 이제 가비지 수집에 대해 신경 쓰지 않는다면 매우, 매우 쉽게 할 수 있습니다. 맞죠? 첫 번째와 마지막을 null로 설정하면 됩니다. 상수 시간 내에 할 수 있습니다. 하지만 물론 배열 목록처럼 clear를 수행한다면 가비지 수집을 용이하게 하려면 전체 목록을 처음부터 끝까지 실제로 살펴보고 이동하면서 포인터를 0으로 설정해야 합니다. 하지만 그 방법은 약간 다를 것입니다. 이제 포인터를 이동하면서 방금 살펴본 모든 연결을 끊어야 하기 때문입니다. 알겠습니다. 노드를 이동하면서 항목을 null로 설정하고 이전 항목을 null로 설정해야 합니다.

null 옆에 두고 다음 항목으로 넘어가는 식으로 계속합니다. 알겠습니다. 스스로 알아내도록 하겠습니다. 이를 수행하는 방법은 여러 가지가 있습니다. 따라서 최소한 의사 코드에서 직접 구현해 보는 것이 좋습니다. 연결 목록과 레이 목록의 대응 항목 간의 복잡성, 시간 복잡성을 비교하는 차트가 있습니다. 알겠습니다. 첫 번째 방법은 삽입 방법이며 배열 목록에 대한 것입니다. 여기에는 기본 배열의 큰 청크를 이동하는 것이 포함되므로 목록 크기에 선형 시간이 소요됩니다. 연결 목록도 해당 위치로 이동해야 하므로 선형 시간이 필요합니다. 물론 해당 노드를 찾았으면 해당 위치에서 새 노드를 제거하거나 삽입합니다. 상수 시간 안에 수행할 수 있습니다. 그러나 해당 대상 위치로 이동하는 데 드는 엄청난 복잡성으로 인해 선형 시간이 소요됩니다.

Add는 append 연산입니다. 알겠어요? 배열 리스트에 추가하는 경우, 그것은 상각될 것입니다. 상수 시간, 대부분의 경우 상당히 효율적입니다. 연결 리스트에 추가하는 경우, 그것은 항상 상수 시간 내에 완료될 수 있습니다. 그러므로 이것은 add last와 같습니다. 중간에 있는 무언가를 제거하고 싶다면요. 맞죠? 그러면 연결 리스트도 선형 시간이 필요할 것입니다. 배열 리스트도 마찬가지입니다. 알겠어요, 전체를 덮기 위해 배열의 청크를 이동해야 하기 때문에 remove last는 연결 리스트에서 상수 시간 내에 매우 효율적으로 완료될 수 있습니다. 배열 리스트도 마찬가지입니다. 적어도 배열 리스트의 현재 방식은 용량을 줄이지 않기 때문입니다.

좋아요, 마지막을 제거하는 것은 매우 쉽게 할 수 있습니다. 그러나 배열 목록에서 1번째를 제거하려면 전체 청크를 왼쪽으로 이동해야 합니다. 따라서 배열 목록에서 1번째를 제거하는 데는 선형 시간이 걸립니다. 하지만 연결 목록에서는 상수 시간 안에 할 수 있습니다. 1번째를 연결 목록에 추가합니다. 목록도 상수 시간입니다. 같은 이유로 1번째를 제거하는 것도 상수 시간이지만, 레이 목록에서 1번째를 추가하려면 항상 전체를 오른쪽으로 이동해야 합니다. 따라서 n개의 요소를 복사해야 합니다. 그리고 선형 시간이 걸립니다. 연결 목록에서 특정 인덱스에서 항목을 검색하는 것은 선형 시간이 걸립니다. 다시 말하지만, 해당 위치로 이동해야 하기 때문입니다. 반면 배열 목록에서는 임의 액세스의 이점이 있습니다. 그래서 바로 그 항목을 가져올 수 있습니다. 그리고 이것은 상수 시간 안에 할 수 있습니다. 왜냐하면 우리는 배열에서 작업하고 있고 배열은 상수 시간 세트에서 임의 액세스를 지원하기 때문입니다. 같은 이유로 연결 목록에서는 선형 시간이 걸립니다. 하지만 배열 목록에서도 상수 시간이 걸립니다. 좋아요? 그러니까 우리가 많은 타협을 하는 것 같죠? 더 중요한 방법 중 일부는 배열 리스트에서 더 효율적입니다. 그리고 어떤 경우에는 연결 리스트 구현이 더 효율적입니다. 좋아요, 특히 앞과 뒤, 또는 리스트의 첫 번째와 마지막에서 작업하는 경우 배열 연결 리스트를 사용하는 것이 훨씬 더 효율적입니다. 그리고 언제 유용할지 궁금할 것입니다. 왜 우리는 앞과 뒤에 대해 신경을 써야 할까요? 왜냐하면 중간에 삽입하는 것도 마찬가지로 중요하기 때문입니다. 그리고 다음 주에 시험에서 돌아온 후에 예제 데이터 구조에 대해 알아보겠습니다. 좋아요, 그걸 붙잡아 두세요.

그것이 유용할 수 있는 경우를 생각해 보세요. 하지만 Linkless는 그 나름대로의 용도가 있습니다. 알겠어요? 그리고 이것으로 강의의 목록 부분을 마칩니다. 또한 이야기하고 싶었던 것이 있습니다. 죄송합니다. 반복자에 대해 이야기하겠습니다. 좋아요, 잠깐 이야기해 보겠습니다. 반복자는 2~3회 정도 암송해야 하는 내용이었습니다. 하지만 다른 강의 때문에 참석할 수 없었습니다. 하지만 이제 시간이 충분하니 잠깐 이야기해 보겠습니다. 이것도 중간고사의 일부가 될 겁니다. 좋아요? 반복자는 연결 리스트나 배열 리스트에 실제로 연결되어 있지 않습니다. 사실, 리스트 기반 데이터 구조에 전혀 연결되어 있지 않습니다. 좋아요, 반복자는 생각할 수 있는 모든 종류의 데이터 구조에 적용할 수 있는 훨씬 더 일반적인 메커니즘입니다. 좋아요. 그리고 이름에서 알 수 있듯이 반복자는 데이터 구조의 컬렉션을 반복합니다. 그리고 이것은 리스트 기반 데이터 구조의 경우 대체로 간과되었던 중요한 개념입니다. 리스트 기반 구조에서는 항상 인덱스라는 개념이 있었으니까요, 맞죠? 연결 리스트가 임의 접근을 지원하지 않더라도 우리는 여전히 인덱스를 가지고 있었습니다. 그래서 우리는 그 특정

위치 맞죠? 그리고 항목의 순서에 대한 자연스러운 개념이 있었습니다. 그래서 인덱스가 있었던 것입니다. 하지만 일부 데이터 구조는 집합 데이터 구조처럼 인덱스가 없습니다.

예를 들어, 그것은 단순히 물건의 모음입니다. 좋아요, 몇 주 후에 살펴볼 트리 데이터 구조입니다. 인덱스도 없습니다. 좋아요? 그리고 데이터 구조 내부를 살펴보고 싶다면 인덱스에 대한 개념이 없다면 어떻게 할 수 있을까요? 맞죠? 그래서 반복자가 작용할 때입니다. 반복자는 이 특정 데이터 구조에 시퀀스 개념이 없더라도 순차적으로 데이터 구조 내부를 살펴볼 수 있게 해줍니다. 이것은 목록 기반 데이터 구조의 경우는 아니었습니다. 항상 인덱스 순서대로 미리 설정된 순서대로 항목에 쉽게 액세스할 수 있기 때문입니다. 하지만 새 데이터 구조에 순차 정보가 없더라도 반복자를 사용하여 액세스할 수 있습니다. 좋아요, 데이터 구조 내부를 살펴보면 반복자가 하는 일을 이해하기 위해 반복자가 하는 일이 바로 그것입니다. 반복자 인터페이스부터 시작해야 합니다. 이는 Java 표준 라이브러리 세트의 일부이므로 항상 포함되어 있습니다.

할 수 있습니다. 거의 항상 접근할 수 있습니다. utils 라이브러리 패키지를 임포트하고 remove 부분을 잊어버린다면, 지금은 그것에 대해 걱정하지 않을 것입니다. 하지만 이 두 가지 방법은 다음과 같습니다. 다음은 가장 중요한 것입니다. 가장 중요한 두 가지는 다음과 같습니다.

Next는 Boolean을 반환합니다. 그리고 has next를 호출할 때마다 반복자의 현재 상태를 확인하고 true를 반환합니다. 탐색할 항목이 아직 남아 있는 경우. 좋아요, 반복자를 기억하세요. 반복자를 사용하여 모든 종류의 데이터 구조 내의 모든 항목을 열거할 건가요? 그리고 순차적으로 수행될 겁니다. 한 번에 하나씩 항목을 선택할 겁니다. 좋아요? 그리고 아직 선택하지 않은 항목이 있는 경우 has next가 반환됩니다. True입니다. 문자 그대로 검색할 수 있는 다음 항목이 있습니다. 하지만 컬렉션의 모든 항목을 살펴봤다면 false를 반환합니다. 그리고 반복이 끝났다는 것을 알고 있습니다. 좋아요? 그리고 next라는 이 메서드는 실제로 줄의 다음 항목을 반환하는 메서드입니다. 좋아요. 제거 부분은 걱정할 필요가 없다고 말씀드렸고, 지금은 실제 제거를 수행하는 방법을 이해하면 됩니다. 하지만 저는 그런 종류의 구현으로 여러분을 귀찮게 하려는 것이 아니라는 걸 알죠. 저는 그저 반복자 인터페이스와 그 사용 방법을 소개하려고 할 뿐입니다. 좋아요. 그럼 반복자의 특정 버전을 살펴보죠. 리스트 반복자라고 합니다. 좋아요? 리스트 반복자는 일반적인 반복자 인터페이스를 확장한 인터페이스이고 반복자와 리스트 반복자의 주요 차이점은 양방향 반복을 지원한다는 것입니다. 좋아요? 그럼 has가 있습니다. Next, it has next, 일반적인 것입니다. 하지만 has previous와 previous도 있습니다. 좋아요? 그게 주요 차이점이고, 몇 가지 다른 유틸리티 메서드도 있습니다. 리스트 기반 데이터 구조에서 반복자를 사용하려고 하면 이것이 가장 널리 사용됩니다. 어쨌든, 먼저 반복하는 방법을 보여드리겠습니다. 좋아요, 잠깐 미안한 말을 꺼내보겠습니다. 테스트 사례를 하나 들어보겠습니다. 미안하지만 여기에 뭔가 잘못된 게 있는 듯합니다. 알겠습니다. 다음은 배열 목록을 사용하는 방법을 보여드리고자 하는 예입니다. 미안합니다. 반복자입니다. 좋습니다. 정수일 필요는 없는 배열 목록을 허용하는 메서드가 있다고 가정해 보겠습니다. 하지만 그냥 정수 배열 목록이라고 부르고, 그 안을 살펴보고 싶습니다. 내레이터를 사용하고 싶을 것입니다. 이제 분명히 그런 식으로 for 루프를 사용할 수 있습니다. 그리고 그냥 그런 식으로 평소처럼 출력하면 되죠? 그것은 모든 종류의 배열 목록의 모든 내용을 되뇌는 표준 절차와 비슷합니다. 하지만 반복자 컨텍스트에서 in에서 사용해 보겠습니다. 그리고 여기. 따라서 먼저 배열 목록 유형과 같은 종류의 유형을 받는 반복자의 인스턴스를 선언합니다. 이것은 정수이고, 점 반복자를 호출할 수 있습니다. 좋습니다. 이제 점 반복자는 내부적으로 가지고 있는 어떤 종류의 반복자의 인스턴스를 반환합니다. 그리고 이제 그것을 가지고 있으므로, 그것을 사용하여 배열 목록 안을 살펴볼 수 있습니다. 좋습니다. 그럼 내가 has next를 어떻게 사용하는지 알아차리세요. 알겠어요? 그럼 next가 있죠. 그럼 반복자는 next를 가지고 있죠? 그러니까 빠져나갈 것이 있는 한 계속 실행될 겁니다. 그러니까 탐색하지 않은 항목이 있는 한요. 그럼 while 루프가 실행될 겁니다.

계속해서 반복합니다. 그리고 다음으로, 어떻게 물건을 꺼내는지 살펴보겠습니다. 그냥 simple을 호출하면 됩니다. 그냥 iterator dot next를 그렇게 호출하면 됩니다. 그리고 물건을 인쇄하는 데 사용할 수 있습니다.
그리고 그것은 1번째 루프에서 한 것과 같은 숫자나 데이터의 시퀀스를 출력할 것입니다. 그리고 저는 요점을 보여주기 위해 배열 목록을 사용했습니다. 하지만 배열 목록에만 국한될 필요는 없습니다. Java에서 구현된 다른 데이터 구조를 사용할 수 있습니다. 그리고 여전히 이 반복자 메커니즘에 액세스할 수 있습니다. 그러면 반복자 메서드 내부에서 무슨 일이 일어나는지 살펴보겠습니다. 알겠어요? 그리고 이것을 호출하면 Iter ITR이라는 새 클래스의 인스턴스를 생성합니다. 그리고 그것을 반환합니다. 알겠어요, 훌륭합니다. 그럼 이 반복자 클래스는 어떻게 생겼을까요? 그것은 개인 내부 클래스입니다. 그래서 그것은 개인 내부 클래스이고, 즉 외부에서 이것에 액세스할 수 없지만, 이것의 인스턴스를 생성한 다음 허용되는 외부로 반환할 수 있습니다. 그리고 그것이 허용되는 이유 중 하나는 Itr 클래스가 반복자를 구현하고 있기 때문입니다. 그러므로 그것은 has를 가지고 있습니다. 그 다음에, 그것은 또한 next를 가지고 있습니다. 알겠어요. 그럼 저는 next와 next가 어떻게 구현되는지에 대해서만 이야기하겠습니다. 이것들은 매우 간단한 규칙을 따를 것입니다. 그리고 그것들은 모두 커서라는 개념에 기반을 둡니다. 그래서 이 반복자가 내부를 살펴볼 항목의 순서를 유지하는 방법은 이 커서를 추적하는 것입니다. 그래서 커서는 이 항목, 즉 반환되어야 할 다음 항목을 가리킬 것입니다. 그리고 커서는 기본적으로 인덱스입니다. 이것은 0으로 초기화됩니다. 그리고 Last Ret이라는 또 다른 정보가 있습니다. 그리고 이것은 우리가 반환하는 최신 항목의 인덱스입니다. 그리고 처음에는 아무것도 반환하지 않았기 때문에 음수 1이어야 합니다. 좋아요. 그럼 next가 하는 일은 커서를 배열 목록의 크기와 비교하고, 그것이 크기와 같으면 전체 목록을 실행했다는 것을 의미합니다. 그리고 다음 항목은 인덱스 크기에서 살고 있는 것과 같습니다. 하지만 기억하세요, 크기는 잘못된 인덱스입니다. 좋아요, 우리는 커서가 올바른 범위에 있는지 확인하고 싶고, 커서가 크기와 같지 않은지 살펴봐야 합니다. 그리고 has의 아주 간단한 구현입니다. 그 다음. 나중에 트리를 볼 때 has의 더 복잡한 버전 예를 보여드리겠습니다. 그렇죠, 하지만 꽤 간단합니다. 어쨌든. 다음 메서드는 첫 번째 호출에서 공동 수정을 확인하는 것에 대해 걱정할 필요가 없습니다. 그것은 우리 클래스와 관련이 없습니다. 상수 시간 메서드입니다. 걱정하지 마세요. 저는 그것에 대해 묻지 않을 것입니다. 하지만 가장 먼저 확인해야 할 것은 cursor가 size보다 크거나 같은지 여부입니다.

알겠어요? 그렇다면 예외를 throw할 겁니다. 왜냐하면, 외부 범위로 인덱싱할 방법이 없거든요. 따라서 index가 size보다 크거나 같다는 것은 잘못된 인덱스라는 뜻입니다. 그래서 확인하지 않을 겁니다. 더 이상 진행하지 않을 겁니다.

알겠어요? 그리고 또, 왜 우리는 null을 반환하는 대신 예외를 throw할까요. 생각해 보세요.
나머지는 매우 간단합니다. 좋아요, element. Data는 일반적인 element data입니다. 그리고 우리는 다시 확인할 것입니다. I가 element data dot length보다 크거나 같은지. 그리고 그렇다면 또 다른 예외를 throw합니다. 좋아요, 다소 이상합니다. 이것에 대해 생각해 보면, 그것은 우리가 여기에서 확인한 것과 정확히 같은 종류의 조건입니다. 그럼 왜 다시 확인하는 것일까요? 맞죠? 중복된 작업을 하는 것 같습니다. 하지만 이것은 동시 수정을 고려하고 싶기 때문입니다. 동시 수정은 여러 프로세스 또는 여러 스레드가 동시에 동일한 배열 목록을 수정하려고 할 때 발생합니다. 좋아요, 이 시점에서 기본 배열 목록에 무언가가 발생했을 가능성이 있습니다. 이 녀석을 실행하는 시점입니다. 좋아요, 그래서 우리는 그저 특별히 조심하고 싶습니다. 어머. 죄송합니다. 우리는 그동안 다른 아무것도, 아무도 배열 목록을 건드리지 않도록 특별히 조심하고 싶습니다. 좋아요, 그래서 같은 검사를 두 번 하는 거군요. 하지만 다시 말하지만, 그건 여러분이 걱정할 일이 아니에요. 왜냐하면 그건 수업을 넘어서는 일이니까요. 주제. 좋아요, 하지만 이건 인덱스가 범위를 벗어나지 않도록 특별히 조심하는 조치라는 걸 이해해 주세요. 예외.

좋아요. 하지만 그 부분이 끝나면 나머지는 꽤 간단합니다. 현재 인덱스를 1씩 증가시키고 커서에 할당한 다음 해당 위치에서 요소 데이터를 검색하기만 하면 됩니다.

위치 I에서 마지막 리턴을 I로 설정하거나 확인하는 것을 잊지 마세요.그리고 모든 것이 끝났습니다.다음 메서드가 어떻게 계속해서 줄의 다음 항목을 검색하여 has next가 마침내 false를 반환하는지 볼 수 있습니다.어느 시점에서 커서가 크기를 가리키기 때문입니다.그리고 그것은 반복을 중지하라는 신호입니다.그래서 이것은 같은 코드입니다.여기서 저는 그냥 건너뛰었습니다.II는 약간 번거롭기 때문에 remove 부분을 건너뛰었지만, 아시다시피, 꼭 알아야 한다면, 그것은 remove 메서드를 활용합니다.그리고 이것을 호출합니다.이것은 iter의 하위 클래스인 list iter 클래스입니다.그리고 기억하세요, list, iter는 양방향 반복을 지원하는 list iterator를 구현합니다.좋아요, has previous와 previous는 거의 같습니다.알다시피, 같지는 않지만 has next의 거울 반대입니다.그리고 next.그래서 저는 여러분에게 이것을 설명하는 데 신경 쓰지 않을 것입니다.하지만 그것은 같은 원칙을 따릅니다.기본적으로.좋아요. 다시 한번, 죄송합니다. 슬라이드는 여기서 끝인 것 같습니다. 어떻게 하는지 간단히 보여드리겠습니다. 목록을 사용하고 싶다면 목록을 사용하세요. 반복자. 네, 물론 가능합니다. 사실, 점 목록 반복자를 사용하면 됩니다. 그게 전부입니다. 사용하는 방법은 보여드리지 않았습니다. 아시죠, 점 이전입니다. 하지만 이전인가요? 잠깐만요. 아, 죄송합니다. 이 목록을 반복자로 했어야 했습니다. 맞죠? 그러면 점 이전이어야 합니다. 좋아요, 그렇게요. 이전 요소를 보고 싶다면. 하지만 사용하는 방법은 일반 반복자와 다르지 않습니다. 인스턴스화하거나 선언하고 목록 반복자의 인스턴스로 지정하기만 하면 됩니다. 좋아요. 알겠습니다. 강의는 여기까지입니다. 생각보다 훨씬 빨리 진행된 것 같습니다. 강의에서 제가 한 모든 질문을 생각해 보세요. 그리고 질문은 내일이나 수요일 낭송에 가져오세요. 목요일 중간고사에 대한 리뷰 세션이거든요. 좋아요, 다시 한번, 오늘 수업을 놓친 것에 대해 진심으로 사과드립니다. 여러분께 큰 영향을 미치지 않았으면 좋겠습니다. 하지만 최대한 보충하려고 노력하겠습니다. 그리고 놓친 사무실 시간도 언젠가 보충하겠습니다. 발표하겠습니다. 좋아요. 모두 감사합니다. 수요일에 뵙겠습니다.