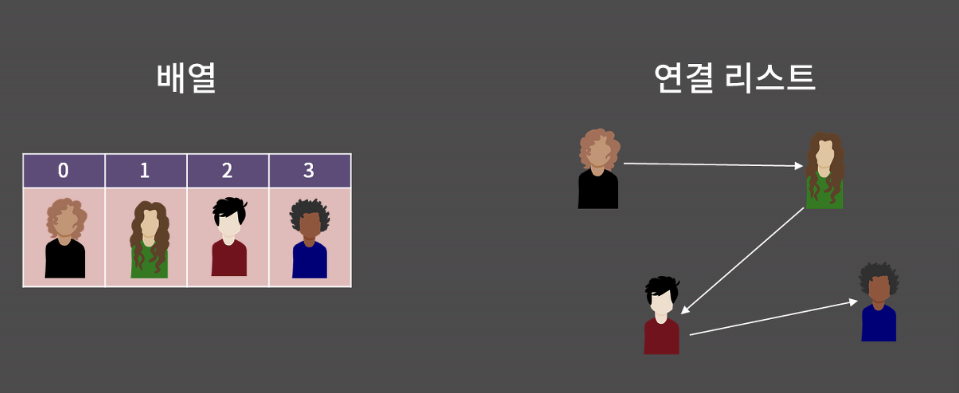
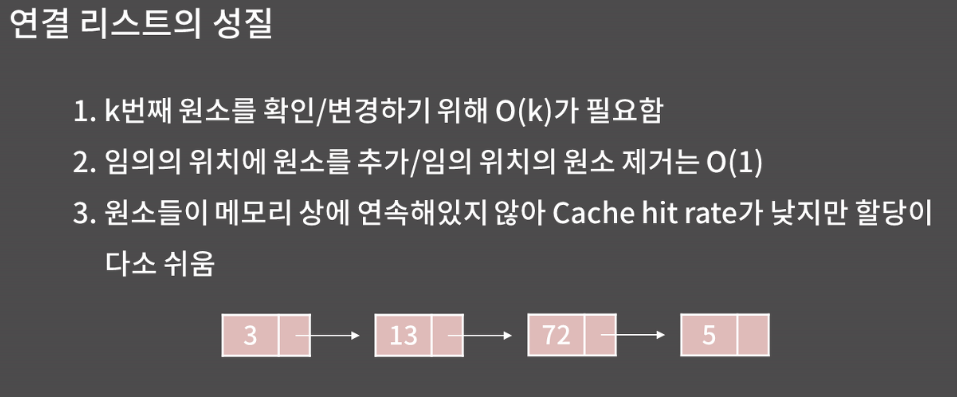
연결리스트

1. 정의와 성질

연결 리스트가 무엇인가 하면, 원소들을 저장할 때 그 다음 원소가 있는 위치를 포함시키는 방식으로 저장하는 자료구조입니다.

연결리스트의 성질

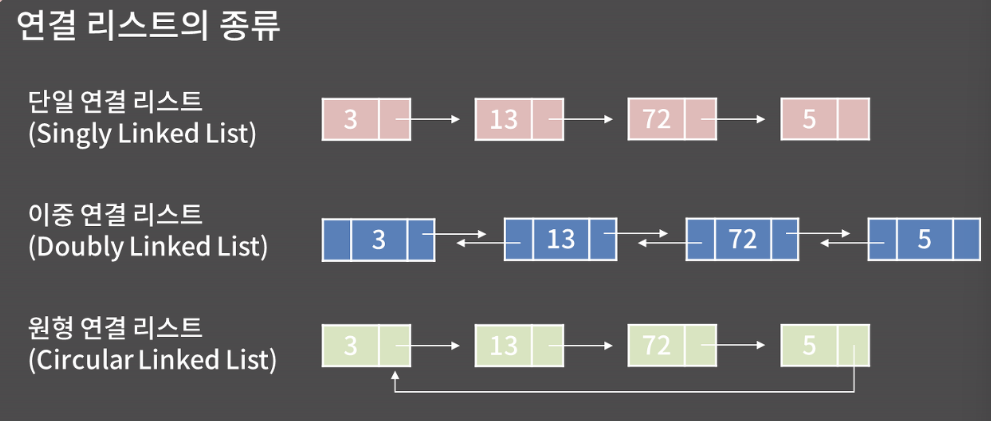


첫 번째로 연결 리스트에서는 k번째 원소를 찾기 위해 O(k)의 시간이 필요합니다.

뭔가 좀 의아할 수 있는데, 연결 리스트의 구조 상 어쩔 수가 없습니다. 지금 이 그림은 3, 13, 72, 5를 저장하는 연결 리스트입니다. 여기서 3번째 원소인 72를 찾고 싶으면 3을 거쳐서 13을 가고, 13을 거쳐서 72를 가야하기 때문에 그렇습니다. 이렇게 가지 않고서는 72가 대체 어디 있는지 알 방법이 없습니다. 배열과 다르게 공간에 원소들이 연속해서 위치하고 있지 않기 때문입니다.

두 번째로 연결 리스트에서는 임의의 위치에 원소를 추가하거나 임의 위치의 원소 제거가 O(1)입니다.

세 번째로 메모리 상에 데이터들이 연속해있지 않으니까 Cache hit rate가 낮지만 할당이 쉽습니다.



첫 번째는 단일 연결 리스트입니다. 단일 연결 리스트는 각 원소가 자신의 다음 원소의 주소를 들고 있는 연결 리스트입니다.

두 번째는 이중 연결 리스트입니다. 이중 연결 리스트에서는 각 원소가 자신의 이전 원소와 다음 원소의 주소를 둘 다 들고 있습니다. 단일 연결 리스트에서는 주어진 원소의 이전 원소가 무엇인지를 알 수 없는데 이중 연결 리스트에서는 알 수 있습니다. 다만 원소가 가지고 있어야 하는 정보가 1개 더 추가되니 메모리를 더 쓴다는 단점이 있습니다. 참고로 STL에 연결 리스트가 있는데, 이 컨테이너의 이름은 list이고 구조는 이중 연결 리스트입니다.

세 번째는 원형 연결 리스트입니다. 원형 연결 리스트에서는 끝이 처음과 연결되어 있습니다. 그림의 예시는 단일 연결 리스트로 표현했지만 각 원소가 이전과 다음 원소의 주소를 모두 들고 있어도 상관이 없습니다.

텍스트, 점수판이(가) 표시된 사진

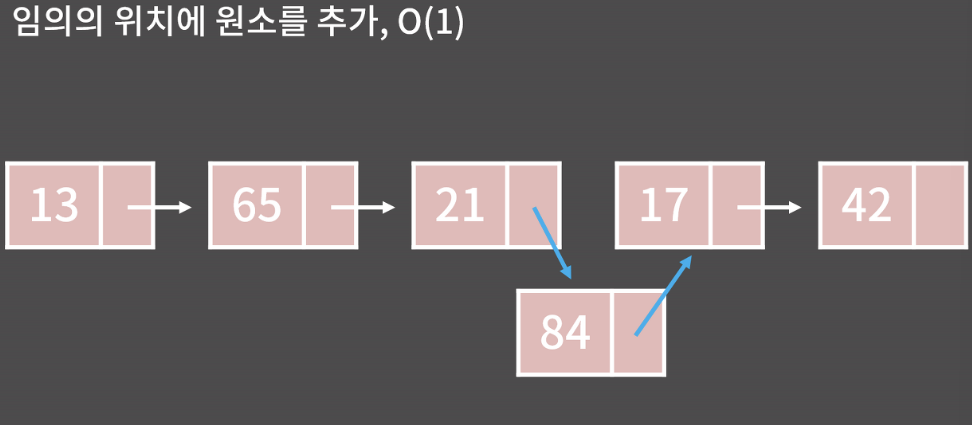
자동 생성된 설명

배열과 연결 리스트는 메모리 상에 원소를 놓는 방법은 다르다고 해도 어찌됐든 원소들 사이의 선후 관계가 일대일로 정의가 됩니다. 즉, 원소들 사이에서 첫 번째 원소, 두 번째 원소, … 이런 개념이 존재하는 것입니다.

그래서 배열과 연결 리스트는 선형 자료구조라고 불립니다. 까마득한 후에 배우게 될 트리, 그래프, 해쉬 등은 비선형 자료구조의 대표적인 예시입니다.

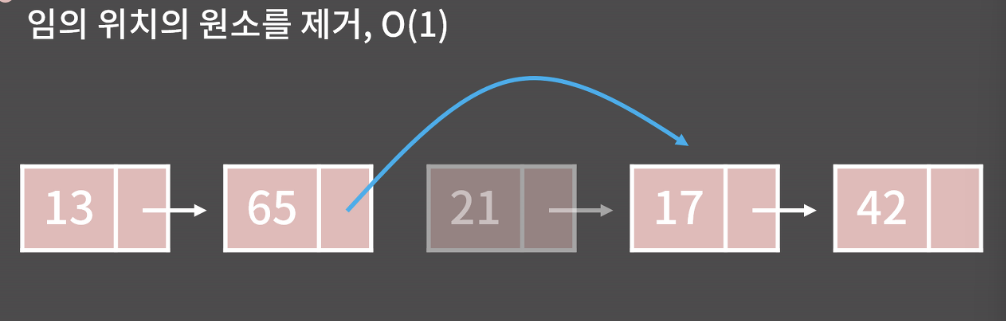
임의 위치에 원소를 추가하거나 제거하는건 배열의 경우 O(N), 연결 리스트의 경우 O(1)입니다. 그런데 엄밀히 말해서 연결 리스트에서도 3번째 원소 뒤에 20이라는 원소를 추가하고 싶다고 하면, 일단 3번째 원소까지는 찾아간 뒤에야 O(1)에 추가가 가능한거라 상황이 조금 다르긴 한데, 이건 구현 파트에서 자세하게 다루겠습니다.

메모리 상의 배치는 배열의 경우 연속이고 연결 리스트의 경우 불연속입니다. 마지막으로 추가적으로 필요한 공간, 즉 overhead를 생각해보면 배열은 데이터만 딱딱 저장하면 될 뿐 딱히 추가적으로 필요한 공간이 없습니다. 굳이 따지면 길이 정보를 저장할 int 1개가 필요할 수 있지만 이건 너무 미미하니 신경을 쓸 필요가 없을 정도입니다. 그런데 연결 리스트에서는 각 원소가 다음 원소, 혹은 이전과 다음 원소의 주소값을 가지고 있어야 합니다. 그래서 32비트 컴퓨터면 주소값이 32비트(=4바이트) 단위이니 4N 바이트가 추가로 필요하고, 64비트 컴퓨터라면 주소값이 64비트(=8바이트) 단위이니 8N 바이트가 추가로 필요하게 됩니다. 즉 N에 비례하는 만큼의 메모리를 추가로 쓰게 됩니다.



임의의 위치에 원소를 추가하는 연산인데, 이건 O(1)에 가능합니다. 왜냐하면, 예를 들어 21 뒤에 84를 추가하고 싶다고 할 때, 배열처럼 그 뒤의 원소들을 전부 옮기는 작업을 할 필요가 없고 그냥 21과 84에서 다음 원소의 주소만 변경을 해주면 되기 때문에 그렇습니다.

단, 착각하면 안되는게 있는데, 추가하고 싶은 위치의 주소를 알고 있을 경우에만 O(1)입니다. 만약 21의 주소를 준 것이 아니라 그냥 84라는 원소를 세 번째 원소 뒤에 추가하라는 명령의 경우에는, 세 번째 원소까지 찾아가야 하는 시간이 추가로 걸려서 O(1)이라고 말할 수 없습니다.



임의 위치의 원소를 제거하는 연산이고 이것 또한 O(1)에 가능합니다. 한 번 21을 지워보겠습니다. 21을 지우려고 하면 딱 1개만 하면 되는데, 그냥 65에다가 "너의 다음 원소가 21이 아닌 17이다"라고만 알려주면 끝입니다. 물론 그런 다음에 21이 들어있는 원소는 메모리 누수를 막기 위해 메모리에서 없애줄 필요가 있습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

지금 연결 리스트를 다루고는 있는데, 배열이 너무 익숙하다보니 도대체 임의의 위치에 있는 원소를 확인/변경하는게 O(N)인 이런 자료구조를 어디에 써먹을 수 있나 싶겠지만 연결 리스트가 쓰이는 대표적인 상황이 바로 메모장과 같은 텍스트 에디터입니다. 물론 실제 텍스트 에디터는 매 순간순간마다 화면에 결과를 출력해야 하니 아마도 그냥 배열로 구현을 할 것 같습니다.

커서를 옮기고 글자를 지우는 것과 같은 연산들이 다양하게 주어진 후 최종 결과를 출력하라는 문제라고 한다면 우리는 커서가 가리키는 위치에 글자를 추가하거나 글자를 지우는 명령을 계속 수행해야 합니다. 이런 경우에 배열은 임의의 위치에 글자를 추가하는 연산이 비효율적인데 연결 리스트에서는 O(1)에 처리할 수 있어서 연결 리스트로 구현할 경우에 효율적일 수 있습니다.

임의의 위치에서 원소를 추가하거나 임의 위치의 원소를 제거하는 연산을 많이 해야 할 경우에는 연결 리스트의 사용을 고려해보면 좋습니다.

1. 기능과 구현

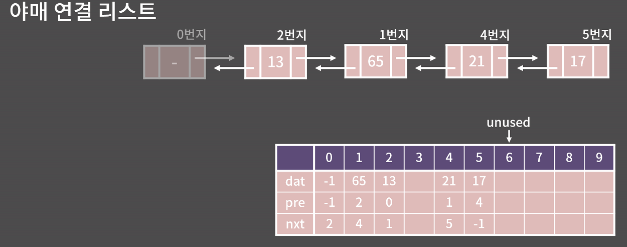
코딩테스트때 대부분 STL씀, but STL못쓰게 할 때 쓸 수 있는 야매 연결리스트 제작

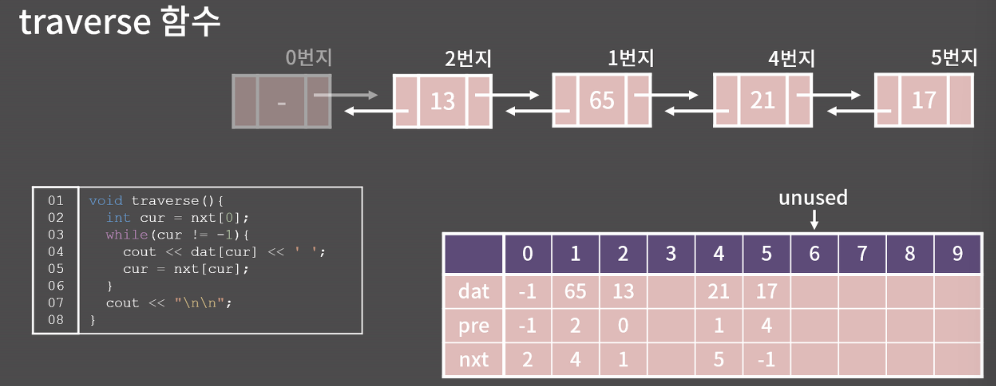
텍스트이(가) 표시된 사진

자동 생성된 설명

이 야매 연결 리스트는 원소를 배열로 관리하고 pre와 nxt에 이전/다음 원소의 포인터 대신 배열 상의 인덱스를 저장하는 방식으로 구현한 연결 리스트입니다. 메모리 누수의 문제 때문에 실무에서는 절대 쓸 수 없는 방식이지만 코딩테스트에서는 구현 난이도가 일반적인 연결 리스트보다 낮고 시간복잡도도 동일하기 때문에 애용하면 됩니다.

dat[i]는 i번지 원소의 값, pre[i]는 i번지 원소에 대해 이전 원소의 인덱스, nxt[i]는 다음 원소의 인덱스입니다. pre나 nxt의 값이 -1이면 해당 원소의 이전/다음 원소가 존재하지 않는다는 의미입니다. unused는 현재 사용되지 않는 인덱스, 즉 새로운 원소가 들어갈 수 있는 인덱스이고 원소가 추가된 이후에는 1씩 증가될 것입니다. 그리고 특별히 0번지는 연결 리스트의 시작 원소로 고정되어 있습니다. 달리 말하면 0번지는 값이 들어가지 않고 단지 시작점을 나타내기 위한 dummy node입니다.





텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

일단 STL list에서 push\_back, pop\_back, push\_front, pop\_front는 모두 O(1)이고, 우리가 이전 구현에서 '번지'라는 개념을 사용했듯 여기서는 iterator가 주소 역할을 합니다. 03번째 줄에서 list::iterator라는 type을 치기가 귀찮으면 C++11 이상일 때 auto t = L.begin()으로 써도 됩니다.

 erase는 제거한 다음 원소의 위치를 반환합니다. 마지막으로 순회를 할 때에는 C++11 이상이라면 12번째 줄과 같이 편하게 할 수 있지만, C++11 미만이라면 14, 15번째 줄과 같이 아주 불편하게 할 수 밖에 없습니다.

야매 연결 리스트에서는 0번지, 즉 제일 앞의 원소를 더미 노드로 사용하지만 STL list에서는 제일 뒤의 원소가 더미 노드입니다. 그래서 L.begin()은 제일 앞에 있는 원소를 가리키지만 L.end()는 제일 뒤에 있는 원소의 한 칸 뒤를 가리킨다는 점에 주의하셔야 합니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

정답: 동일한 노드가 나올 때까지 계속 다음 노드로 가면 됨

공간복잡도: O(1), 시간복잡도 O(N)

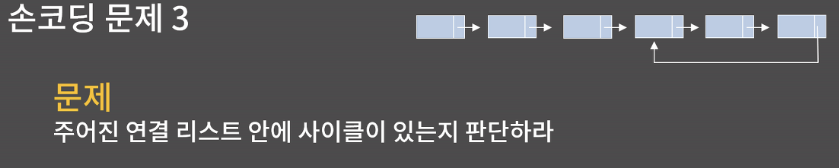


일단 두 시작점 각각에 대해 끝까지 진행시켜서 각각의 길이를 구한다.

그 후 다시 시작점으로 돌아와서 더 긴 쪽을 둘의 차이만큼 앞으로 먼저 이동시켜 놓고

두 시작점이 만날 때까지 두 시작점을 동시에 한 칸씩 전진시키면 됨.

공간복잡도: O(1), 시간복잡도: O(A+B)



Floyd’s cycle-finding algorithm(플로이드 주기 찾기 알고리즘), 공간복잡도: O(1), 시간복잡도: O(N)

한 칸씩 가는 커서와 두 칸씩 가는 커서를 동일한 시작점에서 출발시키면

사이클이 있는 경우 두 커서가 반드시 만나고

사이클이 없는 경우 두 커서가 만나지 못하고 연결리스트 끝에 도달합니다.