실전 알고리즘 0x02강

배열과 연결 리스트

BaaaaaaaaaaaaaaaarkingDog

목차

0x00 배열(Array) – 정의

0	1	2	3	4	5	6	7	8	9
2	4	6	13	-2	1	1	0	x	x

- 배열은 프로그래밍을 배우면서 처음으로 접하게 되는 간단한 자료구조입니다.
- 배열은 메모리 상에 원소를 연속하게 배치한 자료구조입니다. 다른 자료구조와 달리 원소를 저장하는 것 이외에 추가적으로 소모되는 메모리의 양(=오버헤드)이 거의 없는 것이 장점입니다. 또 메모리 상에 원소가 연속해서 있어야 한다는 성질로 인해 cache hit rate가 높다는 장점이 있으나 할당에 제약이 걸린다는 단점도 있습니다.
- C언어의 배열에서는 길이를 제공해주지 않지만 배열을 가지고 작업할 때 따로 변수를 하나 두어 배열의 길이를 저장해둘 수 있습니다. 그러므로 배열의 길이를 알고 있다고 가정하고 여러 가지 연산에 대한 시간복잡도를 알아보겠습니다.

0x00 배열(Array) – 연산의 시간복잡도

임의의 위치에 있는 원소를 확인/변경

0	1	2	3	4	5	6	7	8	9
2	4	6	13	-2	1	1	0	x	x

↓

2	4	6	13	-2	1	6	0	x	x

- 원소가 연속하게 배치되어 있으므로 임의의 위치의 원소를 O(1)에 확인/변경 가능

원소를 끝에 추가

0	1	2	3	4	5	6	7	8	9
2	4	6	13	-2	1	1	0	x	x

↓

2	4	6	13	-2	1	1	0	16	x

- 배열의 길이를 알고 있으니 마지막 위치에 원소를 두기만 하면 되므로 O(1)에 추가 가능

0x00 배열(Array) – 연산의 시간복잡도

마지막 원소를 제거

0	1	2	3	4	5	6	7	8	9
2	4	6	13	-2	1	1	0	x	x

↓

2	4	6	13	-2	1	1	x	x	x

- 배열의 길이를 알고 있으니 마지막 원소를 O(1)에 제거 가능

임의의 위치에 원소를 추가

0	1	2	3	4	5	6	7	8	9
2	4	6	13	-2	1	1	0	x	x

↓

2	4	6	13	-2	13	1	1	0	x

- 임의의 위치에 원소를 추가하고 나면 그 뒤의 원소들을 전부 한 칸씩 밀어야 하므로 O(N)에 추가 가능

0x00 배열(Array) – 연산의 시간복잡도

임의의 위치에 원소를 제거

0	1	2	3	4	5	6	7	8	9
2	4	6	13	-2	1	1	0	x	x

↓

2	4	13	-2	1	1	0	x	x	x

- 임의의 위치에 원소를 제거하고 나면 그 뒤의 원소들을 전부 한 칸씩 앞으로 밀어야 하므로 $O(N)$에 추가 가능

- 임의의 위치에 있는 원소를 확인/변경 = O(1)
- 원소를 끝에 추가 = O(1)
- 마지막 원소를 제거 = O(1)
- 임의의 위치에 원소를 추가/제거 = $O(N)$

0x00 배열(Array) – 사용 팁

- 배열 전체를 특정 값으로 초기화 시켜주고 싶을 때
- 1. `memset` 함수를 이용 : 실수할 여지가 많기 때문에 비추천하는 방식

```
memset(a,0,sizeof a);
```

- 2. `for`문을 이용 : 코드가 조금 길지만 실수할 여지가 별로 없기 때문에 무난함.

```
int a[21][21];
for(int i = 0; i < 21; i++)
  for(int j = 0; j < 21; j++)
    a[i][j] = 4;
```

- 3. `fill` 함수를 이용 : 실수할 여지가 별로 없고 코드도 짧으니 익숙해질수만 있다면 가장 추천하는 방식.

```
int a[20][30];
for(int i = 0; i < 20; i++)
  fill(a[i],a[i]+30,0);
```

0x00 배열(Array) – 사용 팁

- 이외에도 STL에 최댓/최솟값을 찾아주는 함수, 정렬을 수행해주는 함수, 특정 값인 원소들을 전부 다른 값으로 바꿔주는 함수 등이 존재하나 지금 살펴보지는 않고 필요할 때 알려드리는 것으로 하겠습니다.
- 개인적으로 궁금하면 공식 레퍼런스 사이트의 algorithm 헤더 부분을 확인하면 됩니다.(https://en.cppreference.com/w/cpp/algorithm/)

0x00 배열(Array) – 쓰임새

- 배열은 인덱스에 해당하는 원소를 빠르게 접근해야 할 때, 그리고 창고와 같이 데이터를 자주 바꾸거나 확인하는 일 없이 쌓아두고 싶을 때 유용하게 활용할 수 있습니다. 만약 데이터의 삽입/삭제가 빈번한 상황이면 배열이 비효율적입니다.
- 문제에서 일단 입력값을 저장해놓고 시작하는 일이 많기 때문에 거의 대부분의 문제에서 데이터를 쌓아두기 위한 목적으로 배열이 쓰입니다. 이 용도 말고 인덱스에 해당하는 원소를 빠르게 접근하는 목적으로 배열을 사용하면 효율적인 문제를 소개해드리겠습니다.

0x00 배열(Array) – 예제 1

- BOJ 10808번 : 알파벳 갯수(icpc.me/10808)
- 일단 S를 입력을 받고 난 뒤, S에 ‘a’, ‘b’, ‘c’, … ‘z’ 가 몇 개 들어있는지는 어떻게 알 수 있을까요? ‘a’, ‘b’, ‘c’, … ‘z’에 대해 S를 한 바퀴 돌면서 갯수를 세면 되겠네요.
- 정답 코드 : http://boj.kr/0af231bfc9cf41f990fbcd6a05c95c98

```
for(char c = 'a'; c <= 'z'; c++){
  int cnt = 0;
  for(auto s : S){ // s에는 S[0], S[1], ..., S[S.size()-1]이 차례로 들어감
    cnt += (s == c); // s와 c가 같으면 1, 다르면 0
  }
  cout << cnt << ' ';
}
```

0x00 배열(Array) – 예제 1

- 그런데 같은 문자열을 26번씩이나 반복해서 보는 방법 말고 더 효율적인 방법은 없을까요?
- 사람이 이 문제를 푼다면 어떤식으로 해결을 할까요?
- 이런 느낌으로 풀지 않았을까요?

S = a b c a z

0x00 배열(Array) – 예제 1

- 그런데 같은 문자열을 26번씩이나 반복해서 보는 방법 말고 더 효율적인 방법은 없을까요?
- 사람이 이 문제를 푼다면 어떤식으로 해결을 할까요?
- 이런 느낌으로 풀지 않았을까요?

S = a b c a z

0x00 배열(Array) – 예제 1

- 그런데 같은 문자열을 26번씩이나 반복해서 보는 방법 말고 더 효율적인 방법은 없을까요?
- 사람이 이 문제를 푼다면 어떤식으로 해결을 할까요?
- 이런 느낌으로 풀지 않았을까요?

S = a b c a z

0x00 배열(Array) – 예제 1

- 그런데 같은 문자열을 26번씩이나 반복해서 보는 방법 말고 더 효율적인 방법은 없을까요?
- 사람이 이 문제를 푼다면 어떤식으로 해결을 할까요?
- 이런 느낌으로 풀지 않았을까요?

S = a b c a z

0x00 배열(Array) – 예제 1

- 그런데 같은 문자열을 26번씩이나 반복해서 보는 방법 말고 더 효율적인 방법은 없을까요?
- 사람이 이 문제를 푼다면 어떤식으로 해결을 할까요?
- 이런 느낌으로 풀지 않았을까요?

S = a b c a z

0x00 배열(Array) – 예제 1

- 그런데 같은 문자열을 26번씩이나 반복해서 보는 방법 말고 더 효율적인 방법은 없을까요?
- 사람이 이 문제를 푼다면 어떤식으로 해결을 할까요?
- 이런 느낌으로 풀지 않았을까요?

S = a b c a z

0x00 배열(Array) – 예제 1

- 이 방식을 코드로 옮겨내기 위해서는 각 문자의 등장 횟수를 저장할 26칸 짜리 배열이 필요합니다.(정답 코드 상의 `freq` 배열)
- 정답 코드 : http://boj.kr/21949a15d69e40518f2fad093c8cf83e

0x01 STL Vector

- STL에 있는 vector 자료구조는 배열과 거의 동일한 기능을 수행하는 자료구조로, 배열과 마찬가지로 원소가 메모리에 연속하게 저장되어 있기 때문에 배열과 마찬가지로 인덱스로 원소에 O(1)에 접근할 수 있습니다.
- 배열과는 다르게 크기를 자유자재로 늘이거나 줄일 수 있습니다.
- 추후에 그래프의 인접 리스트(Adjacency List)를 다루기 전 까지는 굳이 배열을 쓰지 말고 Vector를 써야하는 상황이 잘 나오지 않아 당장은 몰라도 되지만 다른 사람의 코드를 읽을 때 Vector를 어려워할 것 같아 간략하게 소개하고 갑니다.

0x01 STL Vector – 제공되는 기능들

- 백문이 불여일견, 직접 결과를 확인해보세요.
- 레퍼런스 사이트 : http://www.cplusplus.com/reference/vector/vector/

```
vector<int> v1(3,5); // {5,5,5};
cout << v1.size() << '\n'; // 3
v1.push_back(7); // {5,5,5,7};

vector<int> v2(2); // {0,0}
v2.insert(v2.begin()+1,3); // {0,3,0}

vector<int> v3 = {1,2,3,4}; // {1,2,3,4}
v3.erase(v3.begin()+2); // {1,2,4}

vector<int> v4; // {}
v4 = v3; // {1,2,4}
cout << v4[0] << ' ' << v4[1] << ' ' << v4[2] << '\n'; // 1 2 4
v4.pop_back(); // {1, 2}
v4.clear(); // {}
```

0x01 STL Vector – 주의사항

- `size()` 메소드가 `unsigned int` 값을 반환합니다.
- 아래의 코드는 어디가 잘못되었고 어떻게 수정해야 할까요?

```
vector<int> v1 = {1,2,3};
for(int i = 0; i <= v1.size()-1; i++)
  cout << v1[i] << ' ';
```

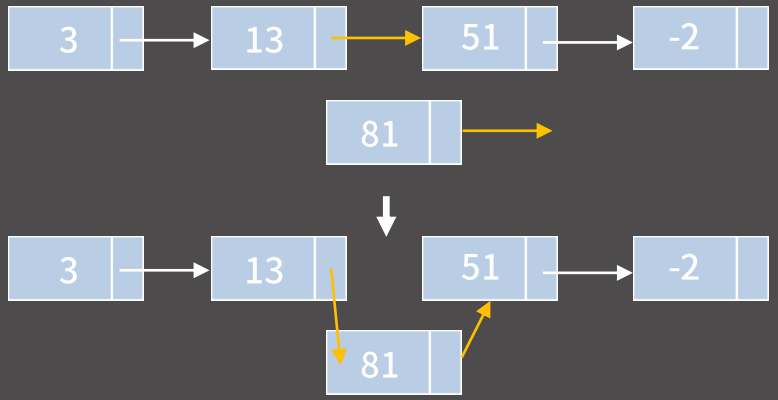
- 답 : v1의 `size()`가 0일 때 `i`가 0부터 2^{32}-1 까지 돌게 됩니다.(unsigned int와 int의 연산 결과는 unsigned int로 형변환이 일어나므로)
- 이를 막기 위해 `(int) v1.size()-1` 혹은 `i < v1.size()`로 수정해야 합니다.
- `insert()`, `erase()` 메소드를 중간에 있는 원소에 대해 사용했을 경우 배열과 마찬가지로 $O(N)$의 시간을 필요로 합니다.

0x02 연결 리스트(Linked List) – 정의

- 연결 리스트는 마치 레고 혹은 직렬연결과 같은 형태로, 각 원소가 자신의 다음 원소(혹은 이전 원소와 다음 원소)의 위치까지 가지고 있는 자료구조입니다.
- 원소들은 꼭 메모리 상에 불연속적으로 위치하고 있어도 무방합니다.
- 각 원소가 자신의 다음 원소의 위치만 가지고 있는 연결 리스트를 Singly Linked List, 자신의 이전/다음 원소의 위치를 모두 가지고 있는 연결 리스트를 Doubly Linked List, 마지막 원소가 처음 원소의 위치를 가지고 있는 연결 리스트를 Circular Linked List 라고 부릅니다.

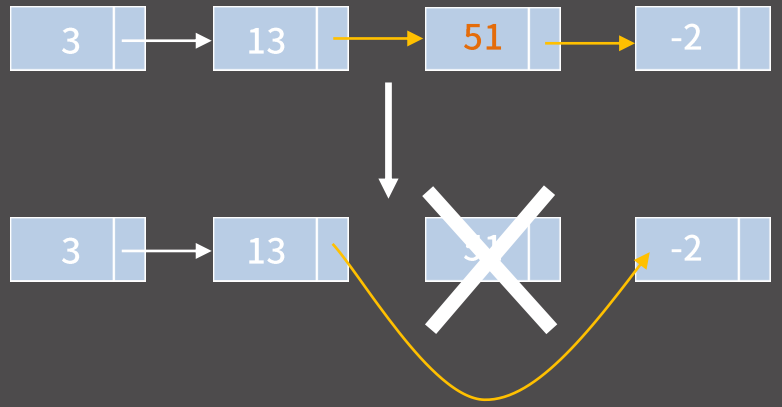
0x02 연결 리스트(Linked List) – 연산의 시간복잡도

임의의 위치에 원소를 추가



- 임의의 위치에 원소를 추가하기 위해 Singly의 경우 2개, Doubly의 경우 4개의 가리키는 값을 바꾸면 되므로 위치에 도달하고 나면 O(1) 에 추가 가능

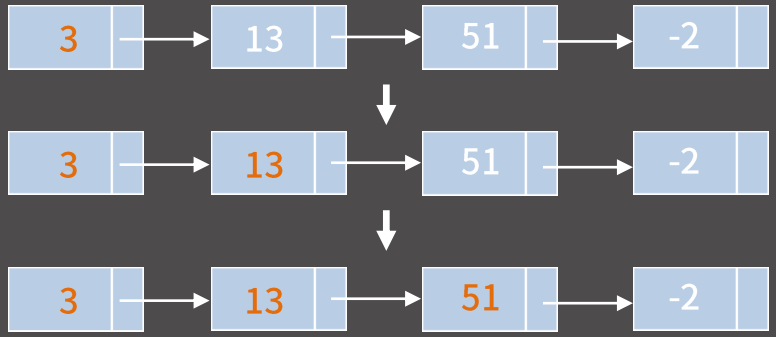
임의 위치의 원소를 제거



- 임의의 위치에 원소를 제거하기 위해 Singly의 경우 2개, Doubly의 경우 4개의 가리키는 값을 삭제/변경하면 되므로 위치에 도달하고 나면 O(1) 에 추가 가능

0x02 연결 리스트(Linked List) – 연산의 시간복잡도

임의의 위치에 있는 원소 값 확인/변경



(3번째에 있는 원소 값 확인)

- 해당 위치까지 가기 위해 이전의 모든 원소를 방문하면서 가야 하므로 $O(N)$ 원소 값 확인/변경 가능

- 임의의 위치에 원소를 추가 = O(1)
- 임의 위치의 원소를 제거 = O(1)
- 임의의 위치에 있는 원소 값 확인/변경 = $O(N)$

0x02 연결 리스트(Linked List) – 사용 팁

- Linked List의 구현은 보통 C의 구조체 + 동적할당을 이용하고, 면접에서 단골 질문문제이기 때문에 따로 공부를 하셔야 합니다. 그러나 이 구현은 긴박한 코딩테스트에서 쓰기엔 별로 좋지 않습니다.
- STL List는 Doubly Linked List 구조를 가지고 있기 때문에 Linked List가 필요하면 직접 구현할 필요 없이 STL List를 가져다 쓰기만 하면 됩니다. STL을 쓸 수 없는 코딩테스트라면, 구조체 + 동적할당 대신 제가 조금 뒤에 알려드릴 야매 Linked List를 이용하시는 것을 추천드립니다.

0x02 연결 리스트(Linked List) – 사용 팁

- Linked List는 딱히 응용해서 낼 만한 여지가 없습니다. Linked List가 코딩테스트에서 나오면 보통 시뮬레이션 느낌의 문제로, 보자마자 “이 문제는 Linked List로 푸는 문제구나” 하고 알아차릴 수 있을 것입니다.
- 만약 Linked List처럼 보이는 문제라고 하더라도, N이 작아 크게 시간복잡도에 연연하지 않고 짜도 상관없는 상황이면 굳이 Linked List를 활용할 필요 없이 그냥 Vector로 구현하는게 마음 편합니다.

0x02 연결 리스트(Linked List) – STL List

- Vector와 마찬가지로 예시를 통해 이해해봅시다.
- 레퍼런스 사이트 : http://www.cplusplus.com/reference/list/list/

```
list<int> L = {1,2}; // 1 -> 2
// iterator를 포인터라고 생각하면 편함. 엄밀한 개념은 독자에게 맡깁니다.
// auto t = L.begin() 이라고 쓰는게 더 간편. auto는 C++11 이상부터 지원
list<int>::iterator t = L.begin(); // t는 1을 가리키는 중
L.push_front(10); // {10,1,2}
cout << *t << '\n'; // t가 가리키는 값 = 1을 출력
L.push_back(5); // {10,1,2,5}
L.insert(t, 6); // {10,6,1,2,5}
t++; // t를 1칸 앞으로 전진, 현재 t가 가리키는 값은 2
t = L.erase(t); // t가 가리키는 값을 제거, 그 다음 원소인 5의 위치를 반환.
                // 리스트는 {10,6,1,5}, t가 가리키는 값은 5
t = L.end(); // 리스트의 마지막 원소보다 한 칸 뒤를 가리킴. 즉 t가 가리키는 곳은 5보다 한칸 뒤인 곳.
for(auto i : L) cout << i << ' '; // {10,6,1,5};
```

0x02 연결 리스트(Linked List) – 야매 Linked List

- STL을 쓸 수 있으면 그냥 STL list를 쓰면 됩니다. 그런데 만약 그렇지 않다면, 포인터가 필요없는 야매 Linked List를 사용합시다.
- 야매 Linked List는 원소를 배열로 관리하고, prev와 next에 이전/다음 원소의 포인터 대신 배열 상의 인덱스를 저장하는 방식으로 구현한 Linked List입니다.
- 메모리 누수의 문제 때문에 실무에서는 절대 쓸 수 없는 방식이지만 코딩테스트에서는 구현 난이도가 일반적인 Linked List보다 쉽고 시간복잡도 또한 동일하기 때문에 애용합시다.

0x02 연결 리스트(Linked List) – 야매 Linked List

변수 설명

```
const int MX = 1000005;
int dat[MX];
int pre[MX];
int nxt[MX];
int unused = 1;

int main(void) {
  fill(pre, pre+MX, -1);
  fill(nxt, nxt+MX, -1);


}
```

- `dat[i]`는 i번지 원소의 값, `pre[i]`는 i번지 원소에 대해 이전 원소의 인덱스, `nxt[i]`는 다음 원소의 인덱스입니다. -1은 해당 노드의 이전/다음 원소가 존재하지 않는다는 의미입니다.
- `unused`는 현재 사용되지 않는 인덱스를 의미하며 새로운 원소가 추가될 때 마다 1씩 증가합니다.
- 특별히 0번지는 Linked List의 시작 원소로 고정되어 있으며, 0번지의 원소는 값이 들어가지 않는 dummy node입니다.
- 길이 정보가 필요하다면 따로 `len` 변수를 두어도 무방합니다.

0x02 연결 리스트(Linked List) – 야매 Linked List

예시

0번지는 dummy note,
1/2/3/5번지에 데이터가
들어있습니다.

unused

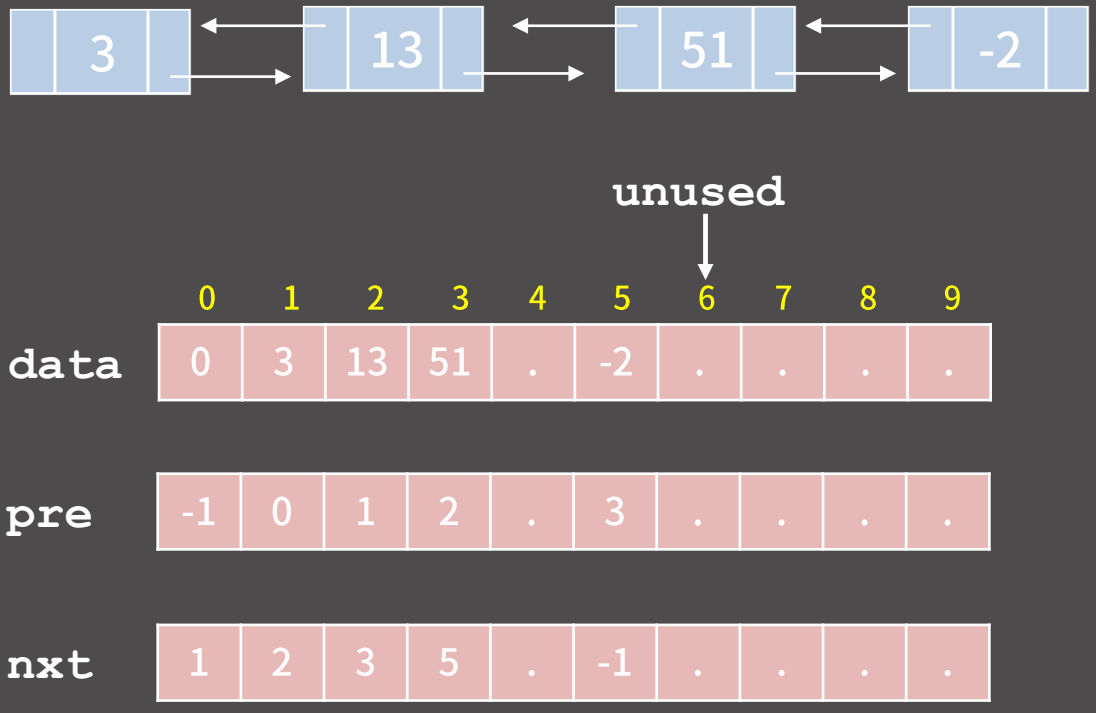
	0	1	2	3	4	5	6	7	8	9
data	0	3	13	51	.	-2	.	.	.	.
pre	-1	0	1	2	.	3	.	.	.	.
nxt	1	2	3	5	.	-1	.	.	.	.

0x02 연결 리스트(Linked List) – 야매 Linked List

삽입

3번지 원소(=51)의 오른쪽에 20을 새로 삽입하고 싶은 상황. 편의상 3번지를 “삽입할 위치” 라고 부르겠습니다.

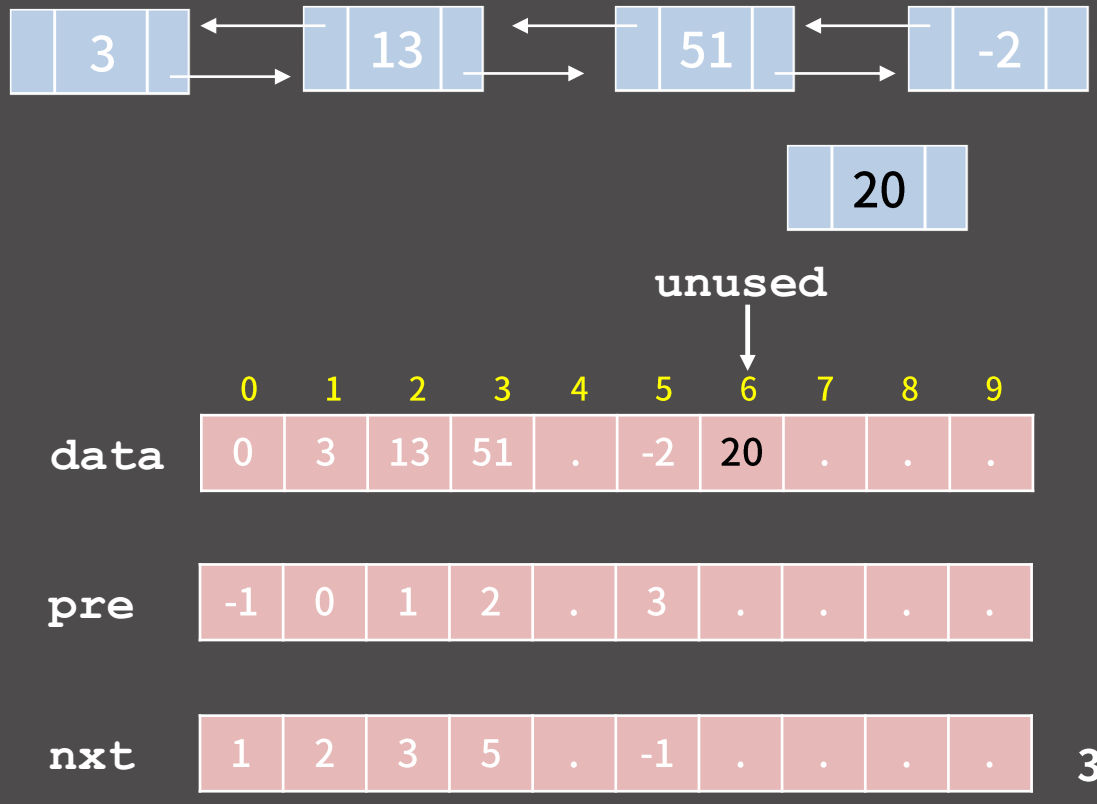
step by step으로 이해해봅시다.



0x02 연결 리스트(Linked List) – 야매 Linked List

삽입

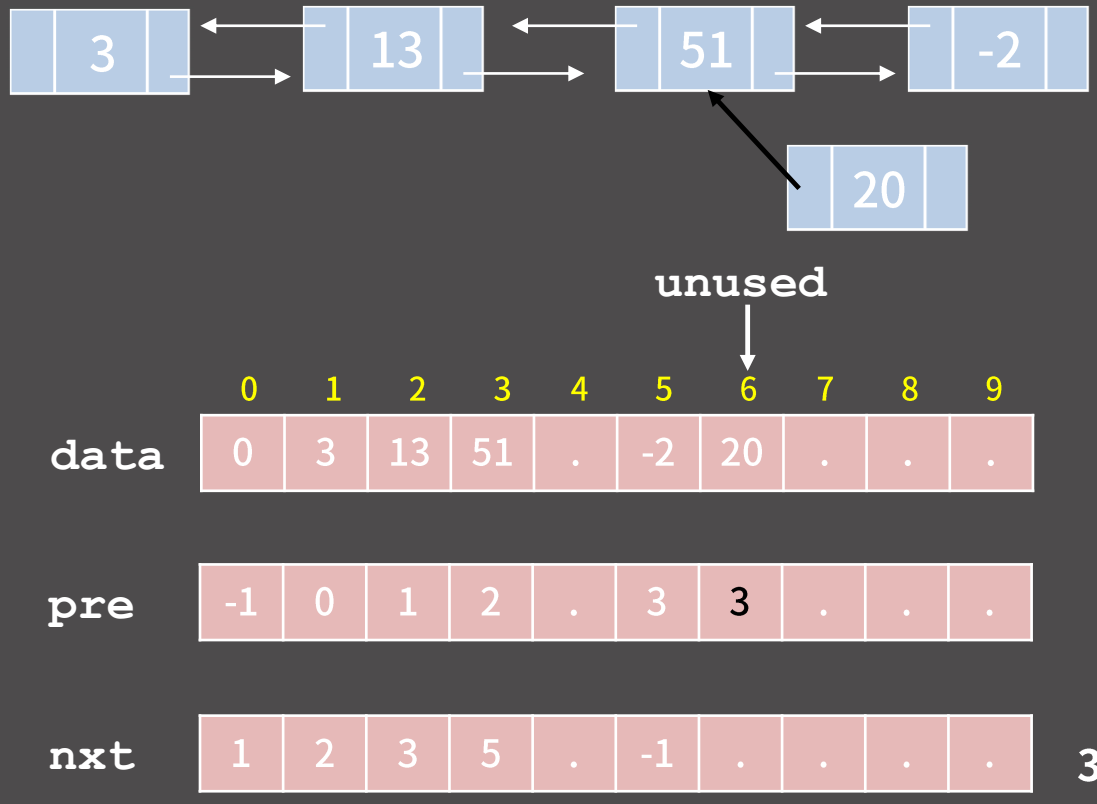
1. 새로운 원소를 만들어냅시다.
아직 `pre`와 `nxt`는 정하지 않았습니다.



0x02 연결 리스트(Linked List) – 야매 Linked List

삽입

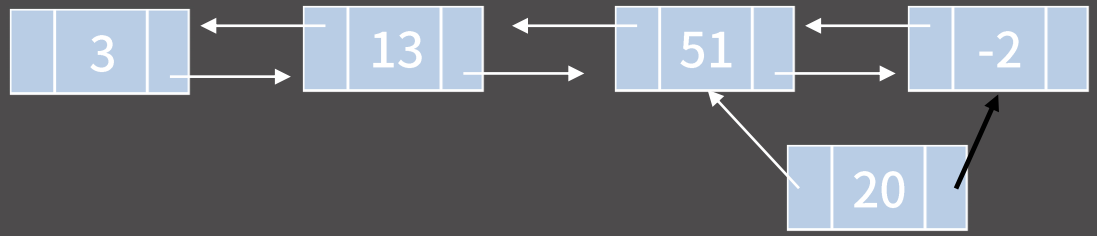
2. 새 원소의 pre의 값에 삽입할 위치를 대입합니다.



0x02 연결 리스트(Linked List) – 야매 Linked List

삽입

3. 새 원소의 `nxt` 값에 삽입할 위치의 `nxt` 값을 대입합니다.



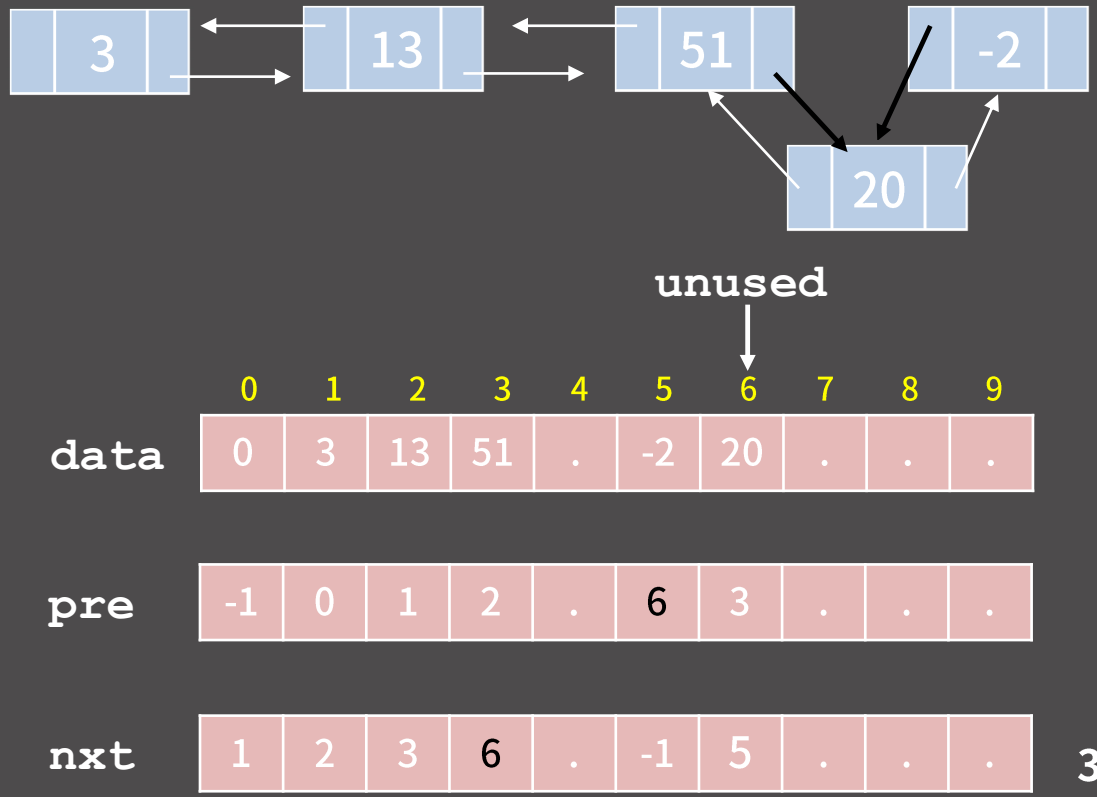
unused

	0	1	2	3	4	5	6	7	8	9
data	0	3	13	51	.	-2	20	.	.	.
pre	-1	0	1	2	.	3	3	.	.	.
nxt	1	2	3	5	.	-1	5	.	.	.

0x02 연결 리스트(Linked List) – 야매 Linked List

삽입

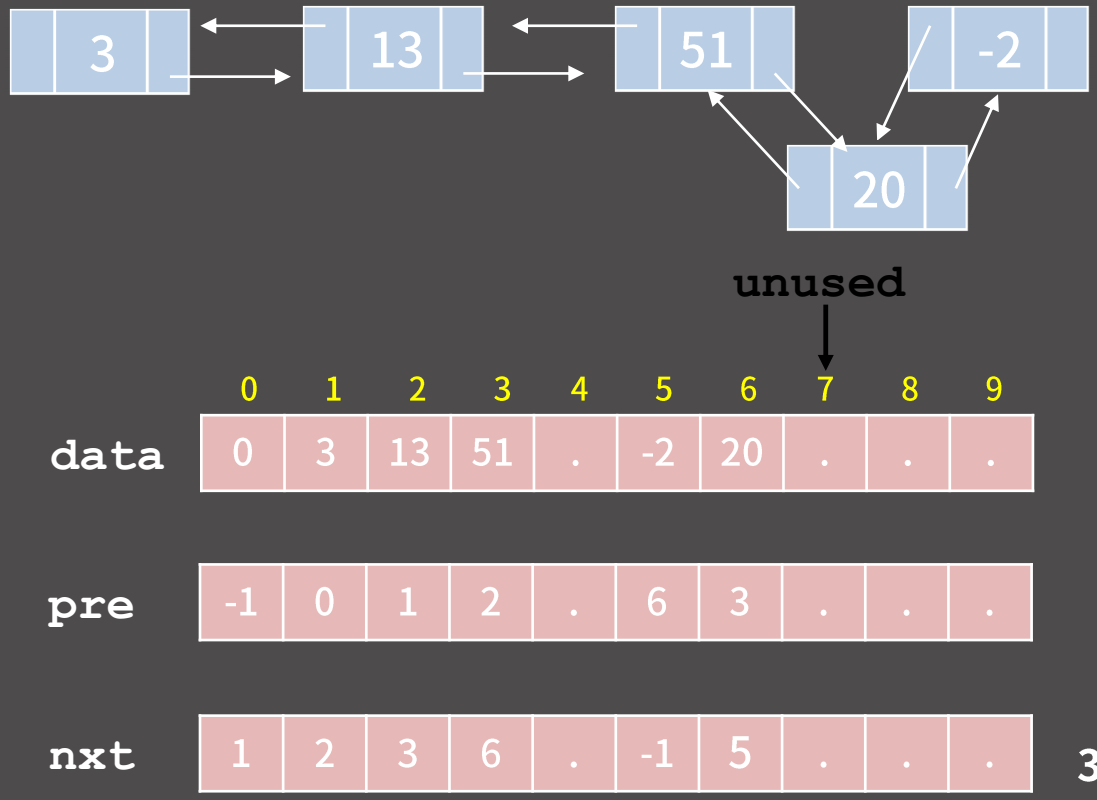
4. 삽입할 위치의 `nxt` 값과 삽입할 위치의 다음 원소의 `pre` 값을 새 원소로 바꿉니다.



0x02 연결 리스트(Linked List) – 야매 Linked List

삽입

5. unused 를 1 증가시킵니다.



0x02 연결 리스트(Linked List) – 야매 Linked List

삽입

편의에 따라 추가된 위치를 반환해도 되고, 그냥 void 함수로 만들어도 되고, 추가된 위치의 이전 위치를 반환해도 됩니다.

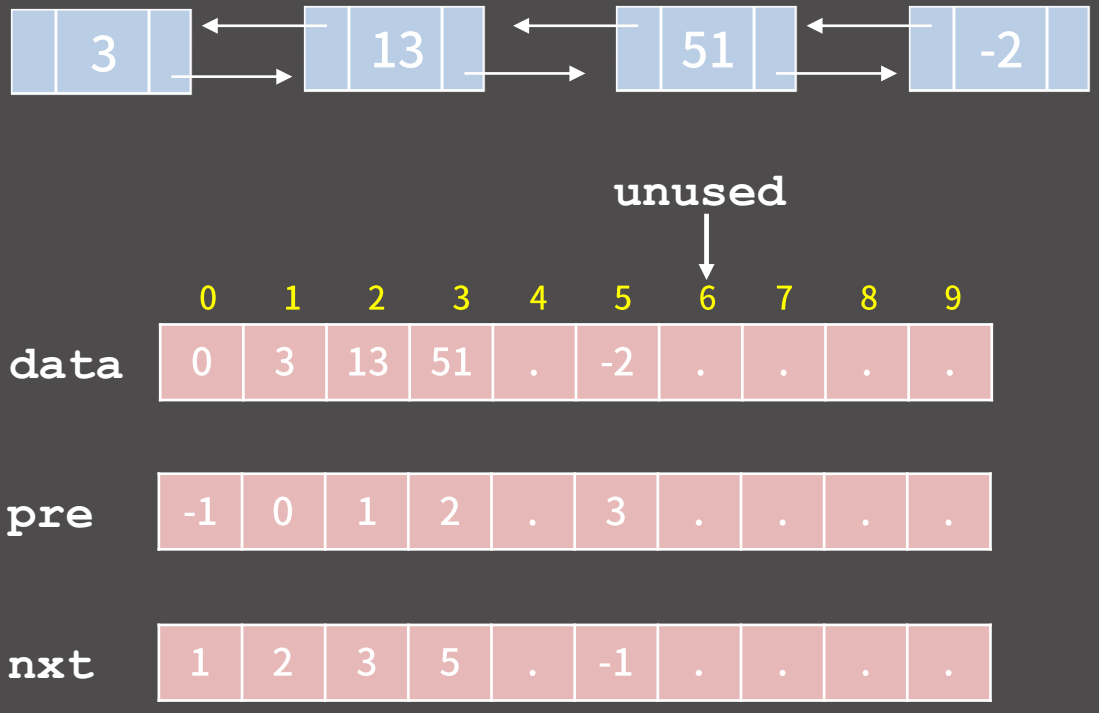
```
// 추가된 위치를 반환
int insert(int idx, char val){
  dat[unused] = val; // step 1
  pre[unused] = idx; // step 2
  nxt[unused] = nxt[idx]; // step 3
  if(nxt[idx] != -1) pre[nxt[idx]] = unused; // step 4
  nxt[idx] = unused; // step 4
  unused++; // step 5
  return nxt[idx];
}
```

0x02 연결 리스트(Linked List) – 야매 Linked List

삭제

3번지 원소(=51)를 지우고 싶은 상황. 편의상 3번지를 "삭제할 위치" ,
삭제할 위치의 이전 원소 (=13)의 위치를 "이전 위치",
삭제할 위치의 다음 원소 (= -2)의 위치를 "다음 위치" 라고 부르겠습니다.

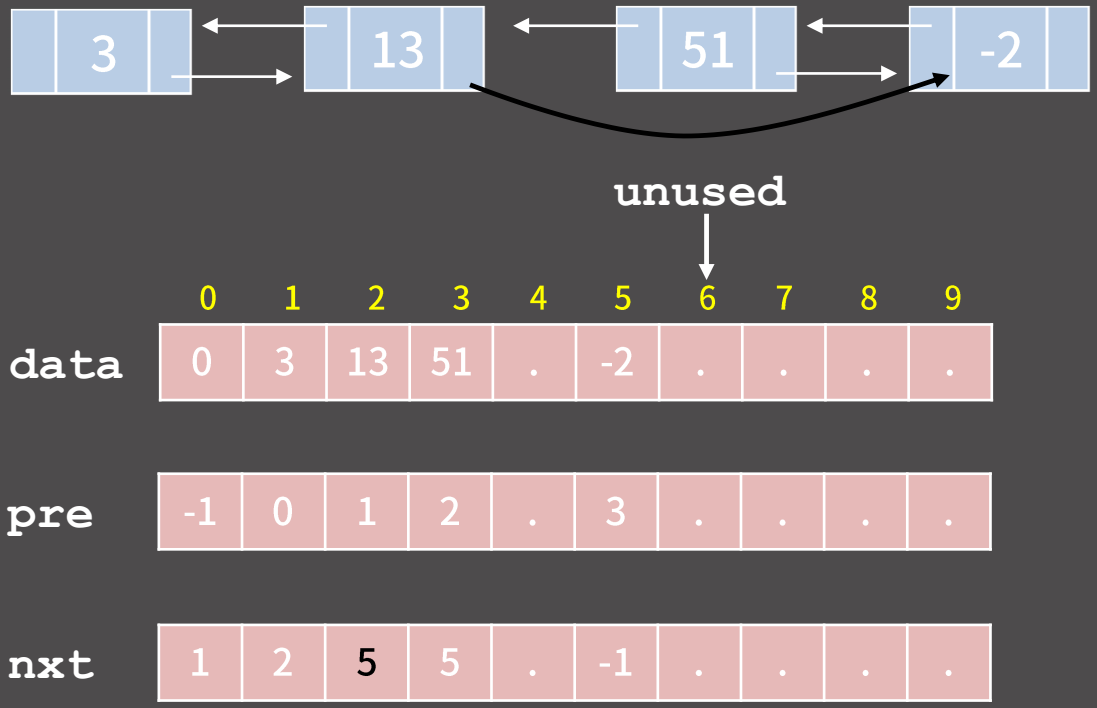
step by step으로 이해해봅시다.



0x02 연결 리스트(Linked List) – 야매 Linked List

삭제

1. 이전 위치의 `nxt`를 삭제할 위치의 `nxt`로 바꿉니다.

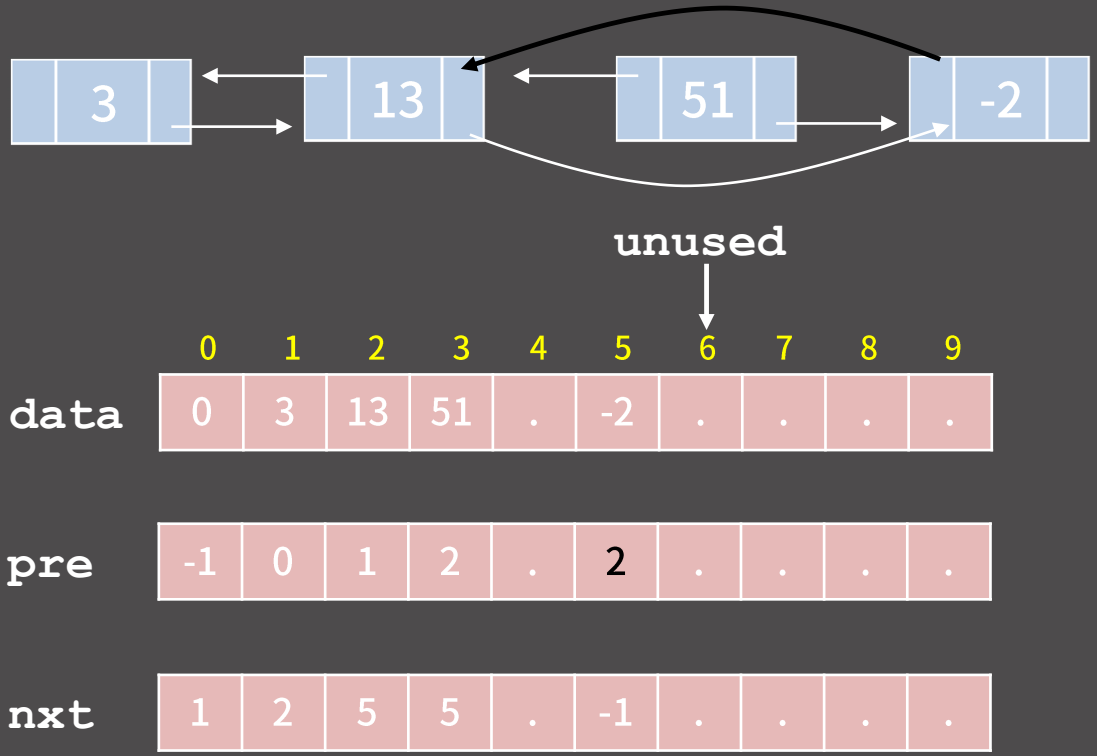


0x02 연결 리스트(Linked List) – 야매 Linked List

삭제

2. 다음 위치의 `pre`를 삭제할 위치의 `pre`로 바꿉니다.

삭제할 위치의 `data`, `pre`, `nxt` 값은 앞으로 참조될 일이 영영 없으니 정리해줄 필요가 없습니다.



0x02 연결 리스트(Linked List) – 야매 Linked List

삭제

- 삽입과 마찬가지로 제거된 원소의 이전 원소를 반환해도 되고, 다음 원소를 반환해도 되고, 반환하지 않아도 됩니다.
- 삭제 후에 잘못된 인덱스의 값을 참조하지 않도록 조심해야합니다.

```
// 제거된 원소의 이전 원소를 반환
int erase(int idx){
  nxt[pre[idx]] = nxt[idx]; // step 1
  if(nxt[idx] != -1) pre[nxt[idx]] = pre[idx]; // step 2
  return pre[idx];
}
```

0x02 연결 리스트(Linked List) – 야매 Linked List

순회

1. dummy node의 `nxt`에서 시작해 -1을 만날 때까지 계속 `nxt`로 이동하면 됩니다.


```
void traversal(){
  int cur = nxt[0];
  while(cur != -1){
    cout << dat[cur];
    cur = nxt[cur];
  }
}
```

0x02 연결 리스트(Linked List) – 예제 1

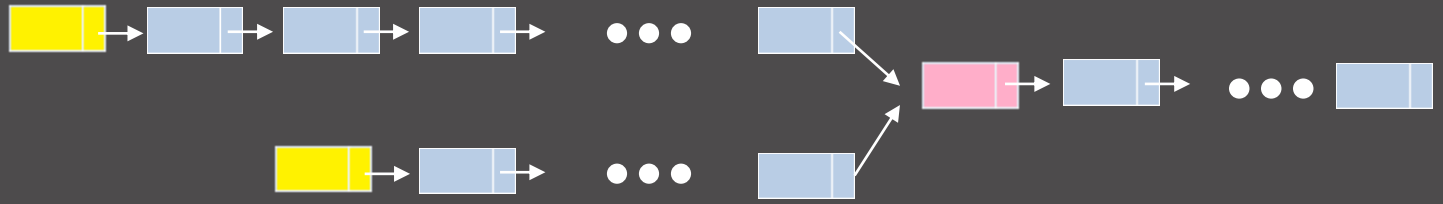
- BOJ 1406번 : 에디터(icpc.me/1406)
- 문장 중간에서의 삭제와 삽입이 빈번하므로 연결 리스트로 풀어야 하는 문제입니다. 문제에서 요구하는 그대로 풀기만 하면 됩니다.
- 정답코드 1(STL List) : http://boj.kr/c1930cc41ba6488a937ce59aff50ea4a
- 정답코드 2(야매 Linked List) : http://boj.kr/a253ae7947f14764ab2e7489ee378a97

0x03 문제 소개

문제 번호	알고리즘 분류	발상 난이도	구현 난이도
2577	Array	1/10	2/10
1475	Array	2/10	2/10
1158	Array	2/10	3/10
5397	Linked List	2/10	3/10

0x03 문제 소개

- 코딩테스트에 나올 수는 없지만 면접 질문이나 손코딩에서 물어볼 수 있는 문제 2가지를 소개해드립니다.
- Q1. Circular Linked List 내의 임의의 노드 하나가 주어졌을 때 해당 List의 길이를 효율적으로 구하는 방법?
- Q2. 중간에 만나는 두 Linked List의 시작점이 주어졌을 때 만나는 지점을 구하는 방법?



0x03 문제 소개

- 코딩테스트에 나올 수는 없지만 면접 질문이나 손코딩에서 물어볼 수 있는 문제 2가지를 소개해드립니다.
- A1. 동일한 노드가 나올 때 까지 계속 가면 됨. 공간복잡도 O(1), 시간복잡도 O(N)

- A2. 일단 두 시작점 각각에 대해 끝까지 쭉 진행시켜서 각각의 길이를 구함. 그 후 다시 두 시작점으로 돌아와서 더 긴 쪽을 둘의 차이만큼 앞으로 먼저 이동시켜놓고 두 시작점이 만날 때 까지 두 시작점을 한 칸씩 전진시키면 됨. 공간복잡도 O(1), 시간복잡도 O($A+B$)

강의 정리

- 배열과 연결 리스트의 정의, 제공되는 연산, 쓰임새를 배웠습니다.
- STL Vector와 List를 간단하게 짚고 넘어갔습니다.
- 야매 Linked List를 배웠습니다.