

연결 리스트 - Linked List

연결 리스트 - Linked List

목차

- 리스트와 연결리스트
- 배열의 한계
- 동적배열과 연결리스트
 - 메모리 참조, 삽입, 삭제
 - 메모리 효율성
 - 캐시 효율성
- 정리 및 결론
- 추가 : C++ STL(Standard Template Library)

연결 리스트 - Linked List

리스트와 연결리스트

- (선형)리스트

- 스택, 큐 등과 같이 데이터를 저장하는 방법, 즉 “자료구조”
- 선형 자료구조란 데이터가 순서 혹은 위치를 가지고 연속적으로 이어져 있는 자료구조



자료구조

- 연결리스트

- 배열, 동적배열과 같이 자료구조를 “구현”하는 프로그래밍 기법
- 스택, 큐 등 선형 자료구조를 구현하기 위해 사용



구현 방식

연결 리스트 - Linked List

배열의 한계

- 배열 → 크기가 고정
- 크기를 동적으로 변경할 수 있는 방법 : 동적배열, 연결리스트, 등

연결 리스트 - Linked List

배열의 한계

- 배열 → 크기가 고정
- 크기를 동적으로 변경할 수 있는 방법 : 동적배열, 연결리스트, 등

in c++

Vector vs List

연결 리스트 - Linked List

동적배열과 연결리스트

	배열	연결리스트
구조		
장점	동적 크기	
	데이터 참조 빠름	중간의 데이터 삽입/삭제 빠름
	메모리 사용 효율적	
	캐시 효율이 좋음	
단점	중간의 데이터 삽입/삭제 느림	데이터 참조 느림
		추가적인 메모리 필요
		캐시 효율이 안 좋음

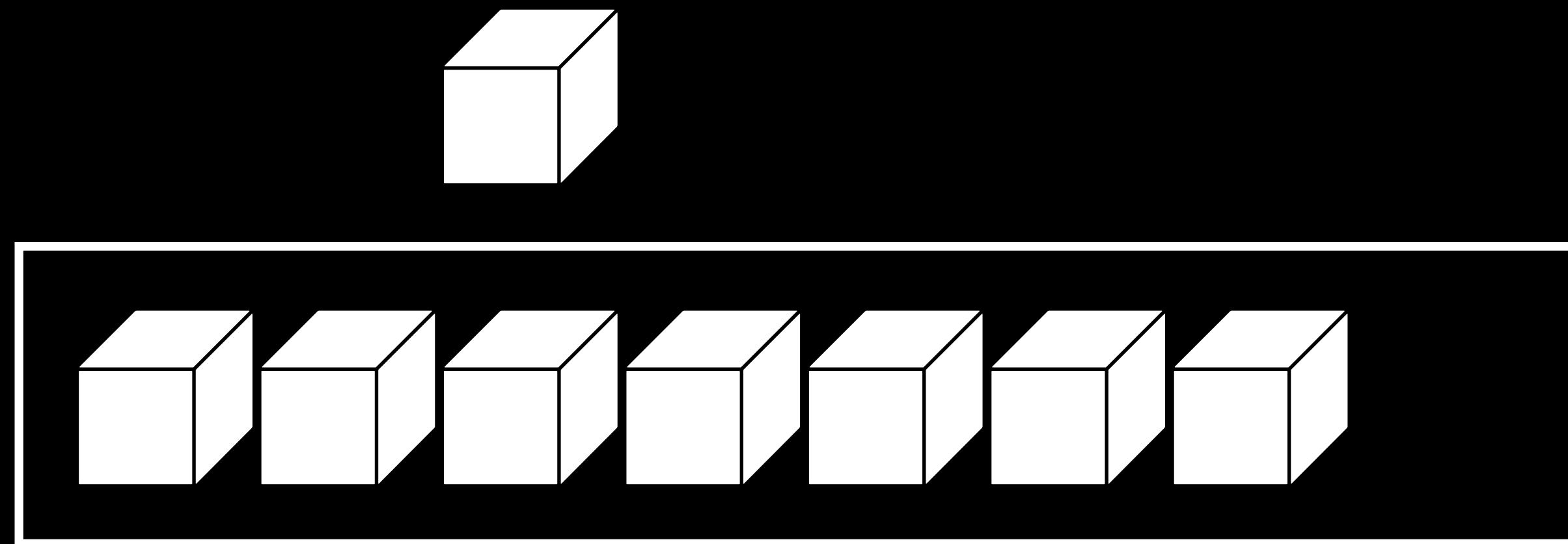
연결 리스트 - Linked List

동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제

	배열	연결리스트
구조		
장점	동적 크기	
	데이터 참조 빠름	<div>중간의 데이터 삽입/삭제 빠름</div>
	메모리 사용 효율적	
	캐시 효율이 좋음	
단점	<div>중간의 데이터 삽입/삭제 느림</div>	데이터 참조 느림
		추가적인 메모리 필요
		캐시 효율이 안 좋음

연결 리스트 - Linked List

동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제

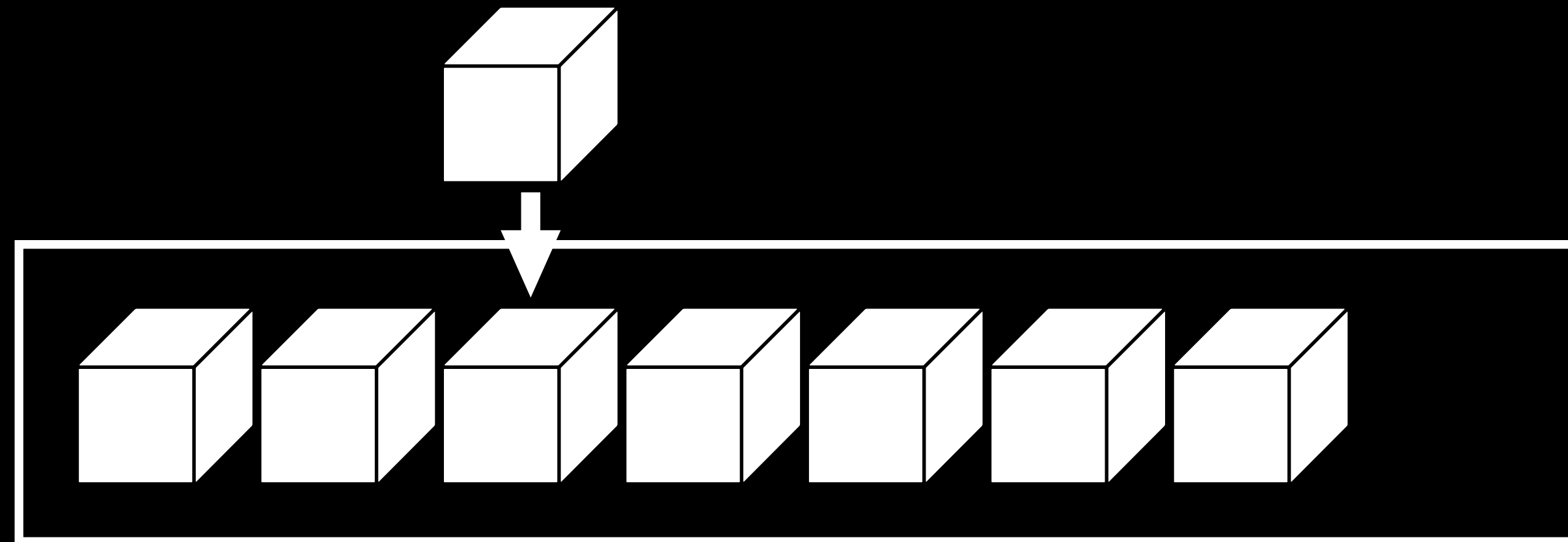


참조 : 0

삽입 : 0

연결 리스트 - Linked List

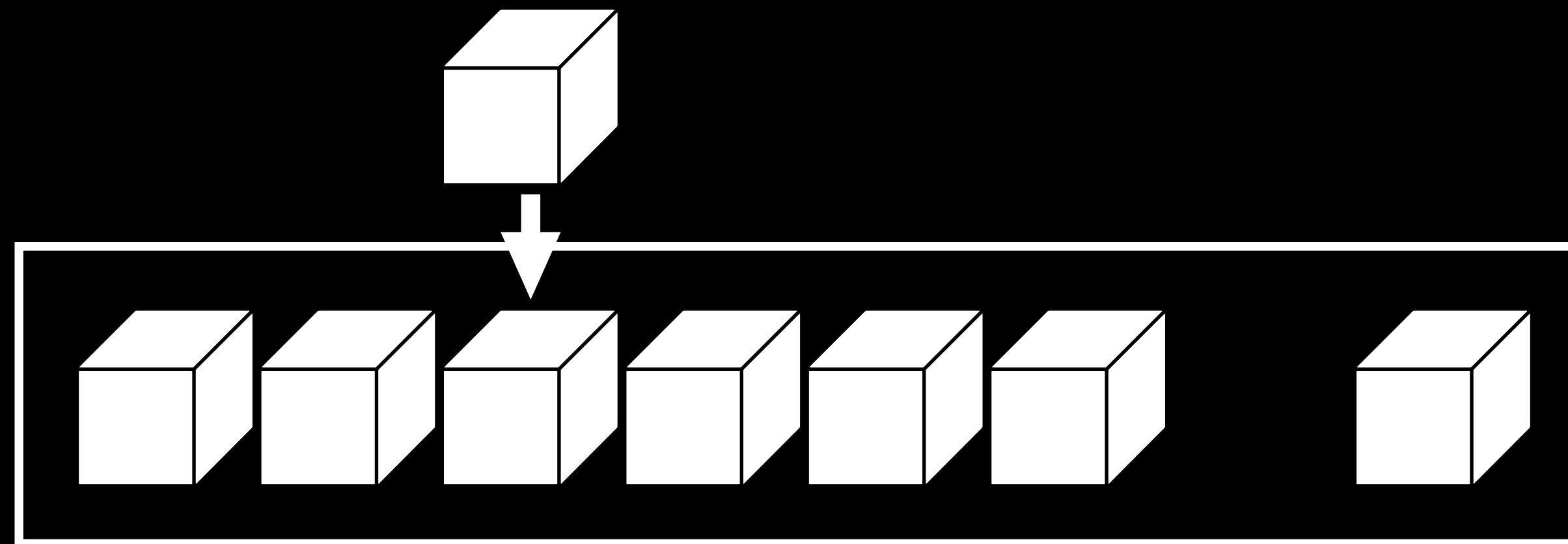
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삽입 : 0

연결 리스트 - Linked List

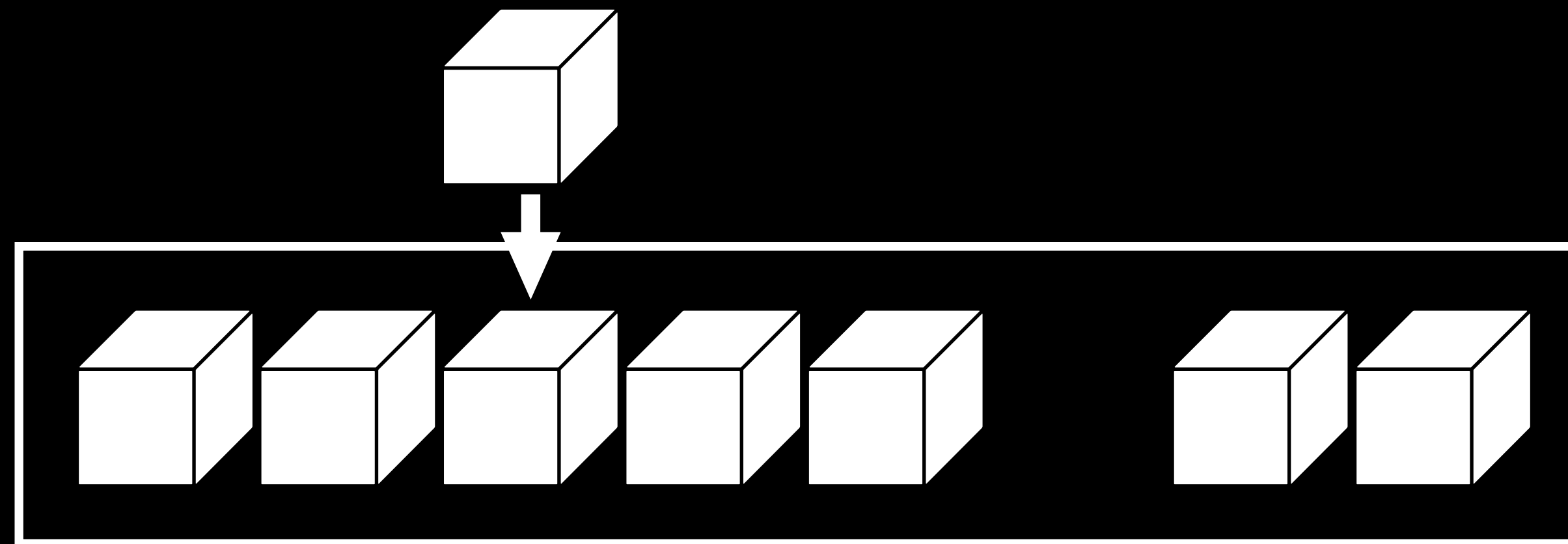
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삽입 : 1

연결 리스트 - Linked List

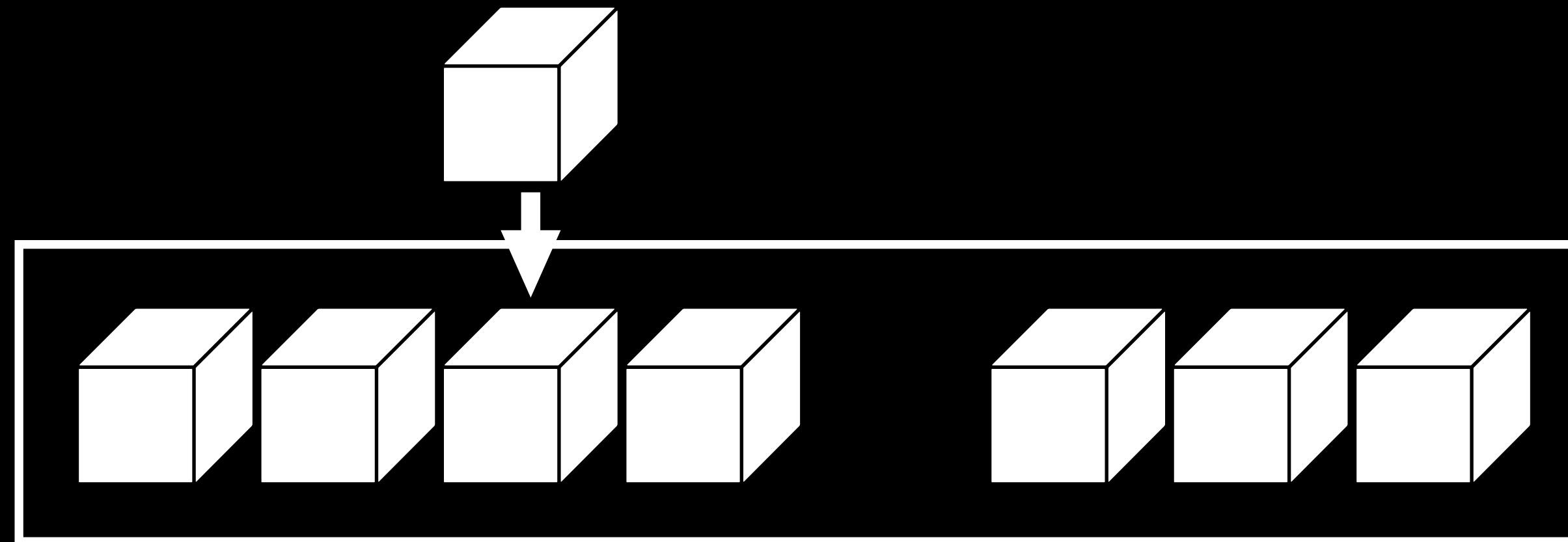
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삽입 : 2

연결 리스트 - Linked List

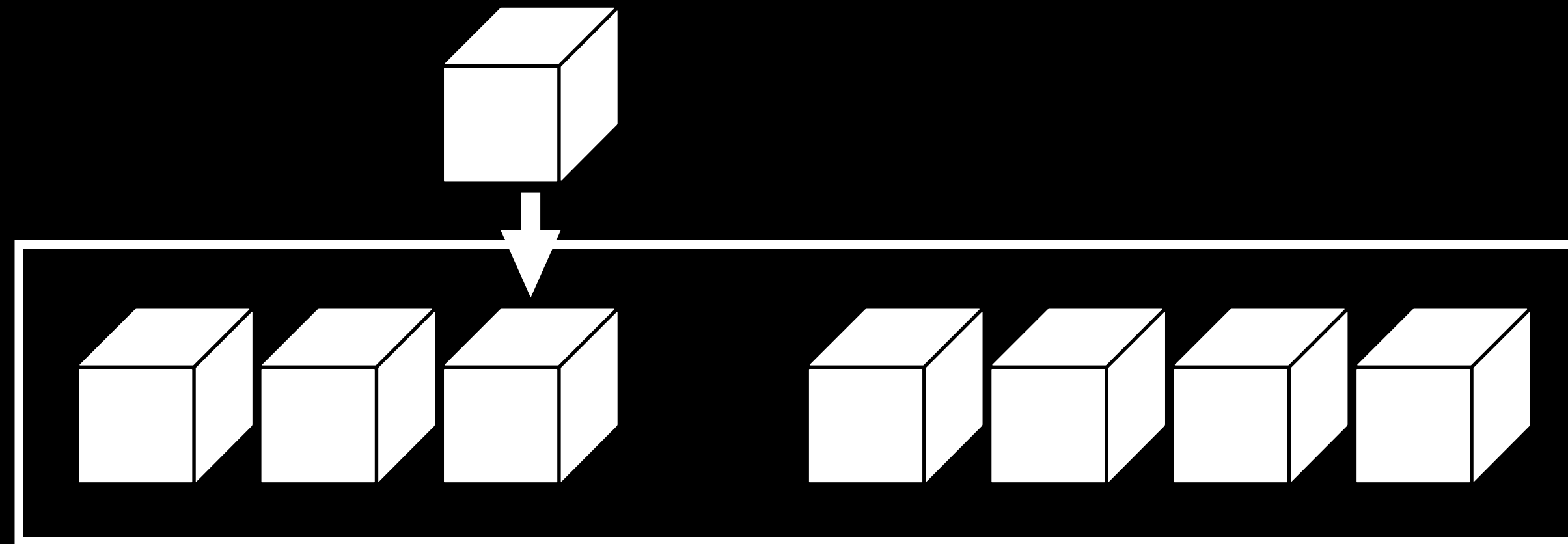
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삽입 : 3

연결 리스트 - Linked List

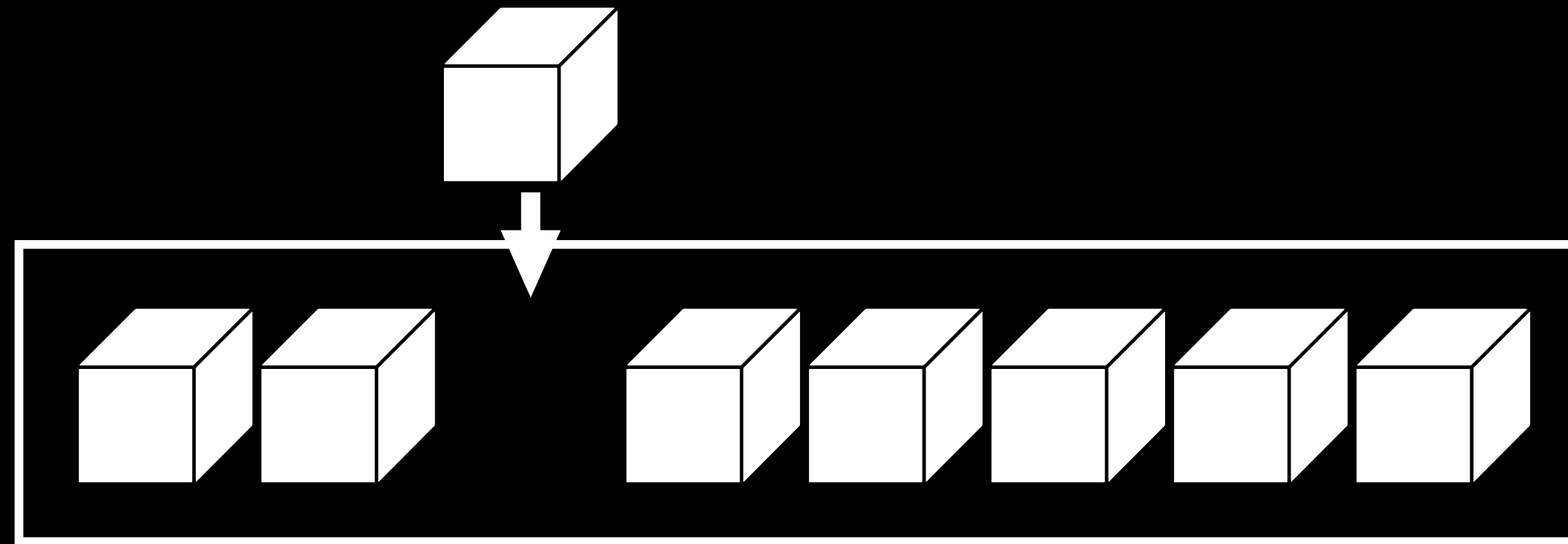
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삽입 : 4

연결 리스트 - Linked List

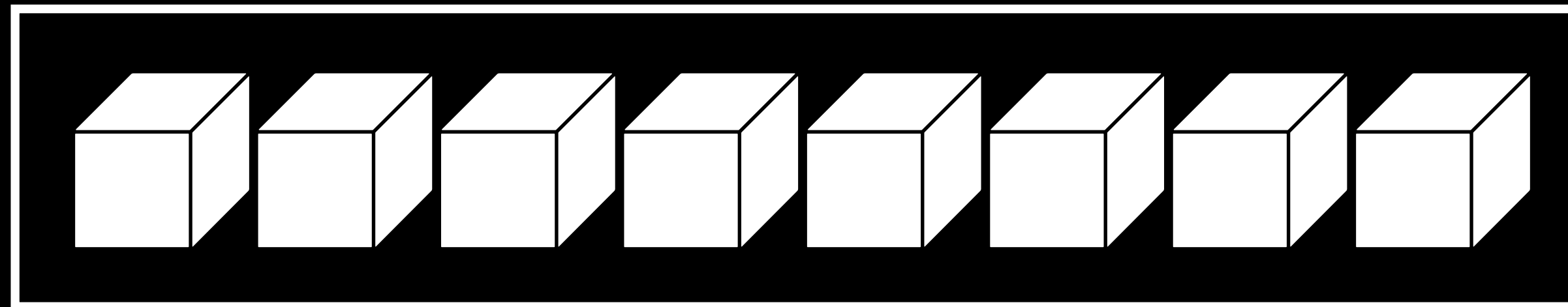
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삽입 : 5

연결 리스트 - Linked List

동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



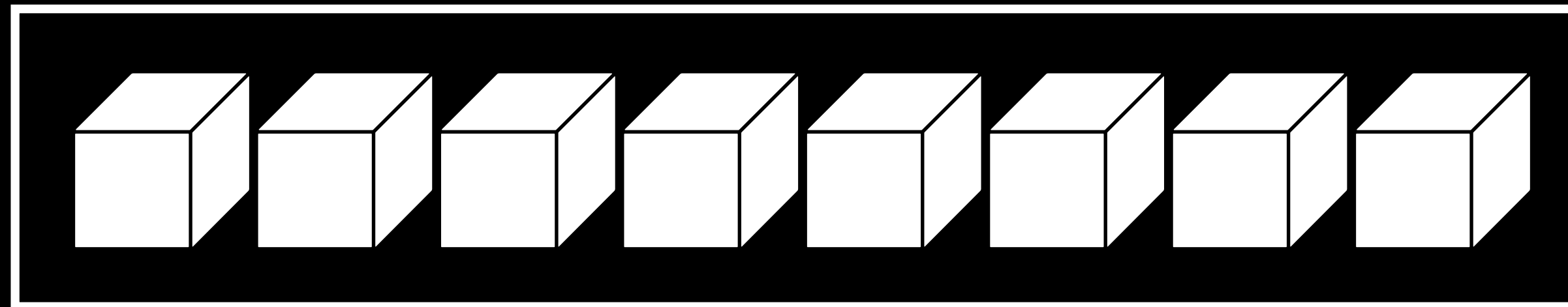
참조 : 1
삽입 : 6

참조연산 시간복잡도 : $O(1)$

삭제연산 시간복잡도 : $O(N)$

연결 리스트 - Linked List

동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제

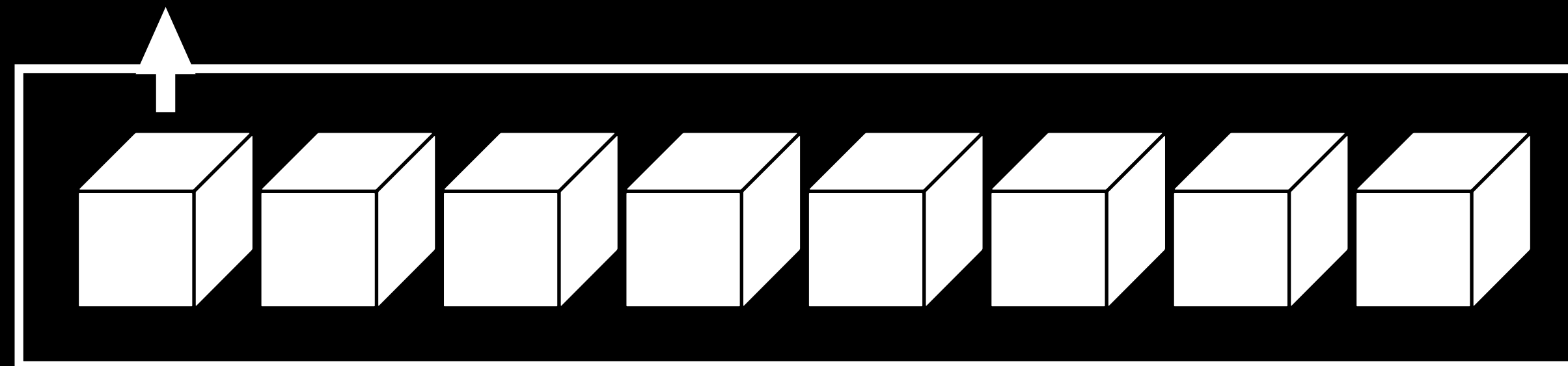


참조 : 0

삭제 : 0

연결 리스트 - Linked List

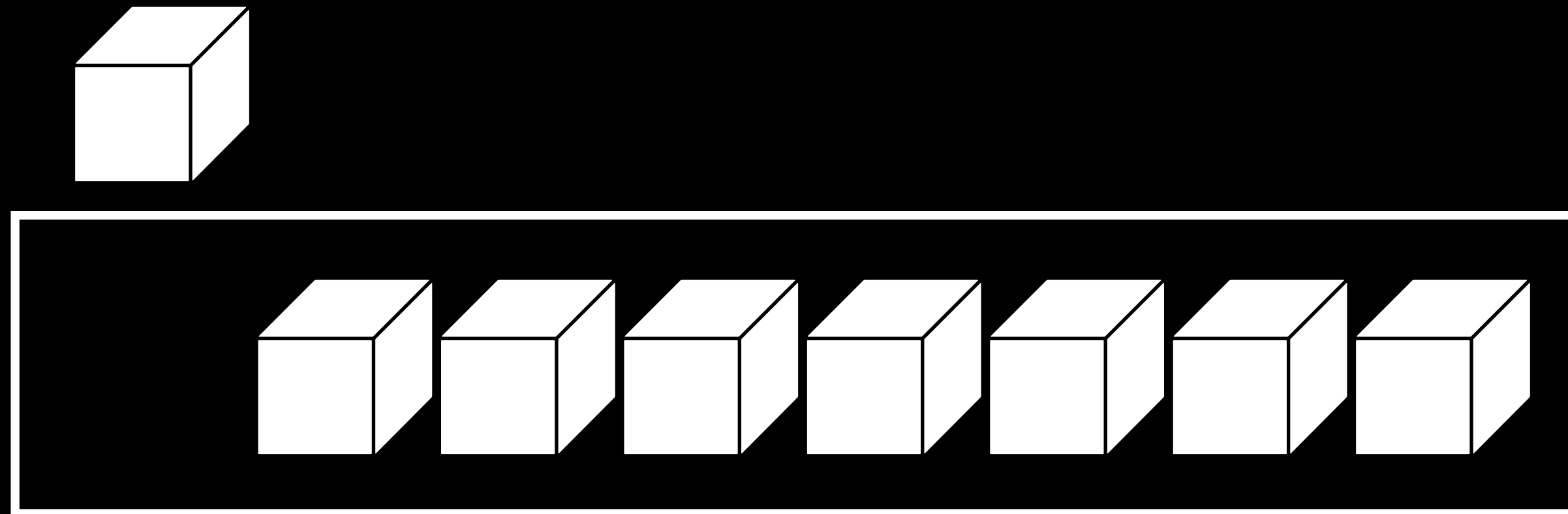
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삭제 : 0

연결 리스트 - Linked List

동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제

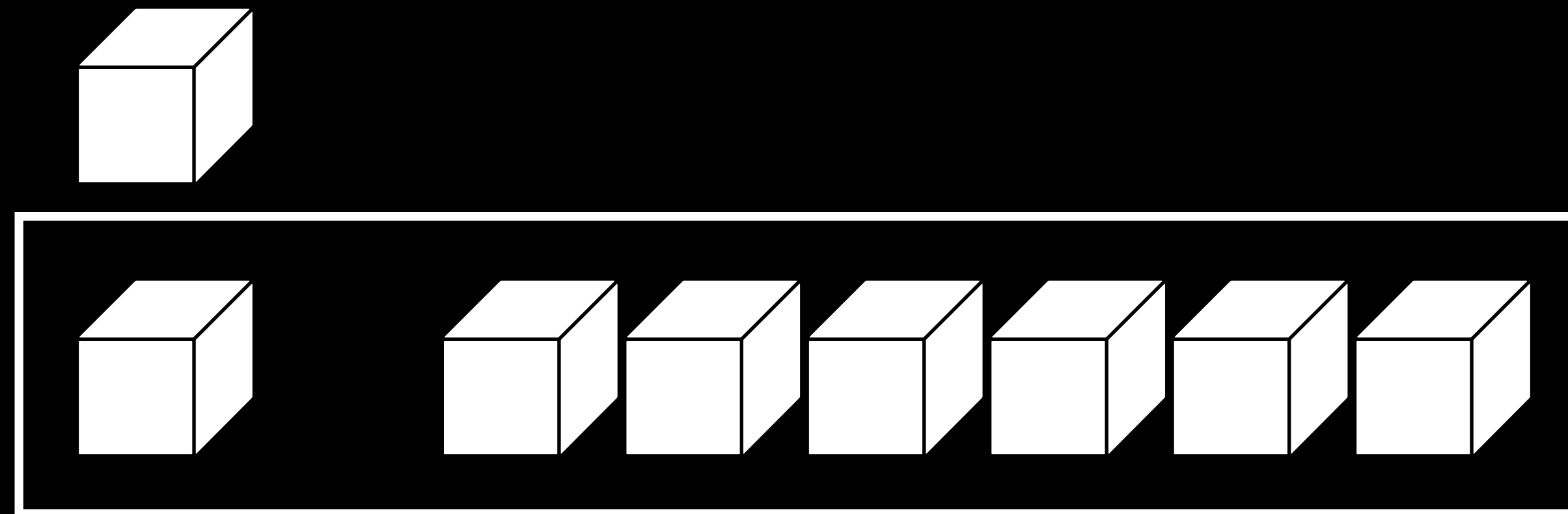


참조 : 1

삭제 : 1

연결 리스트 - Linked List

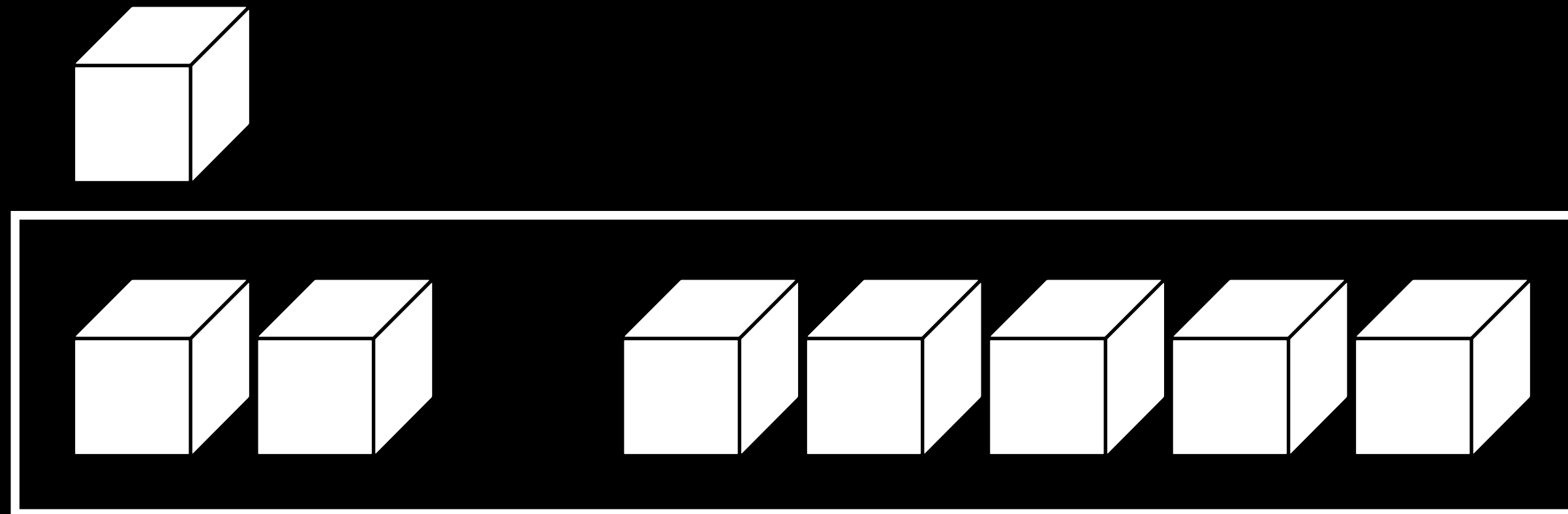
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삭제 : 2

연결 리스트 - Linked List

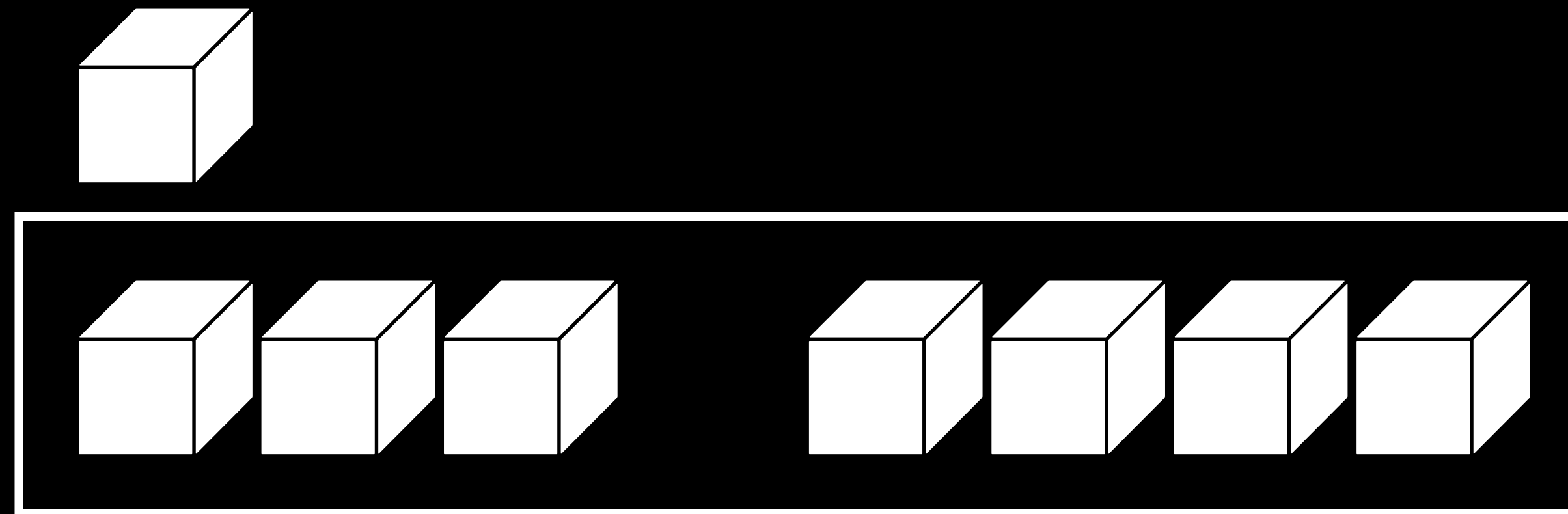
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삭제 : 3

연결 리스트 - Linked List

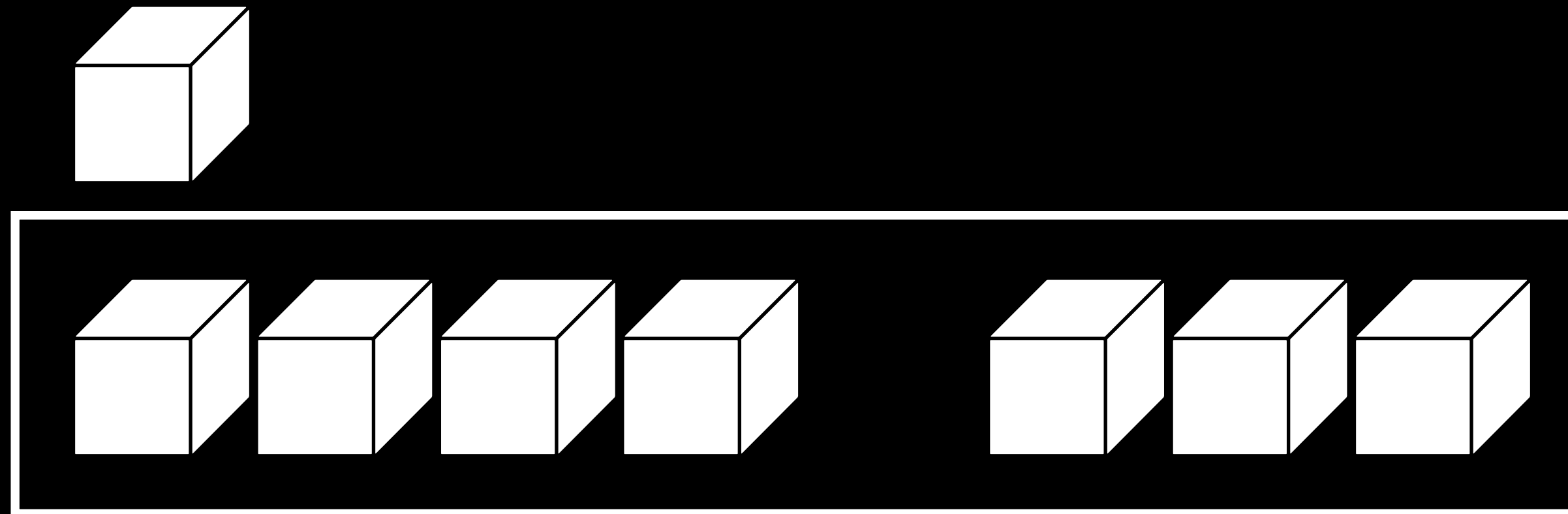
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삭제 : 4

연결 리스트 - Linked List

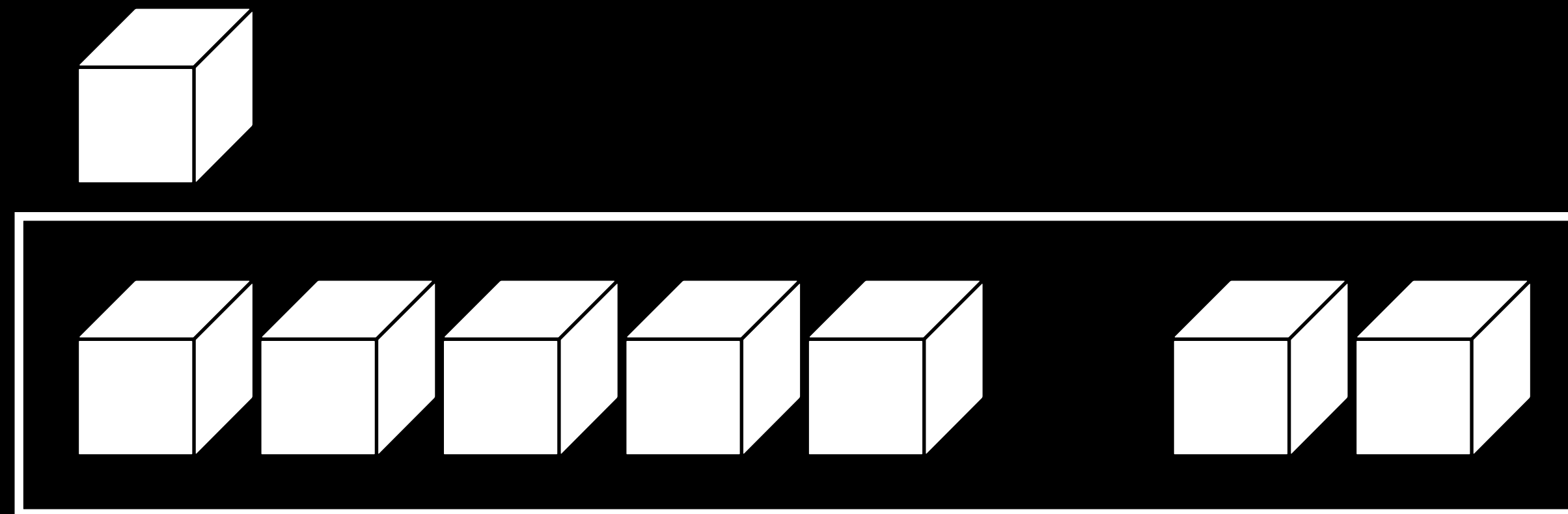
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삭제 : 5

연결 리스트 - Linked List

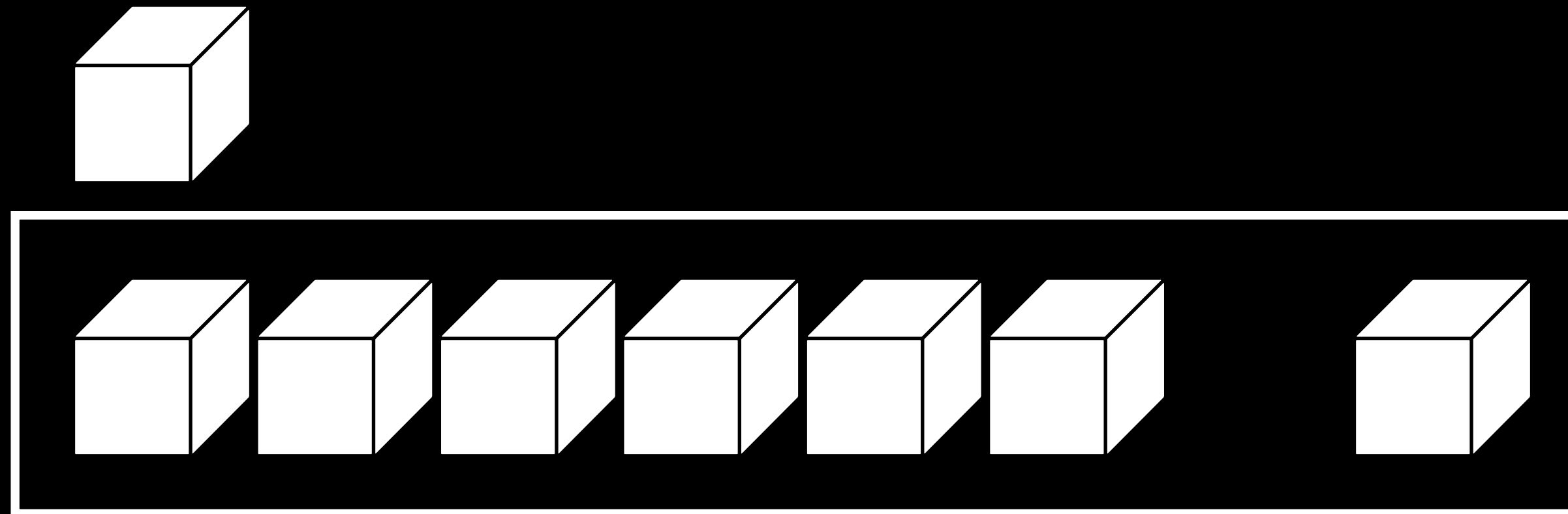
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삭제 : 6

연결 리스트 - Linked List

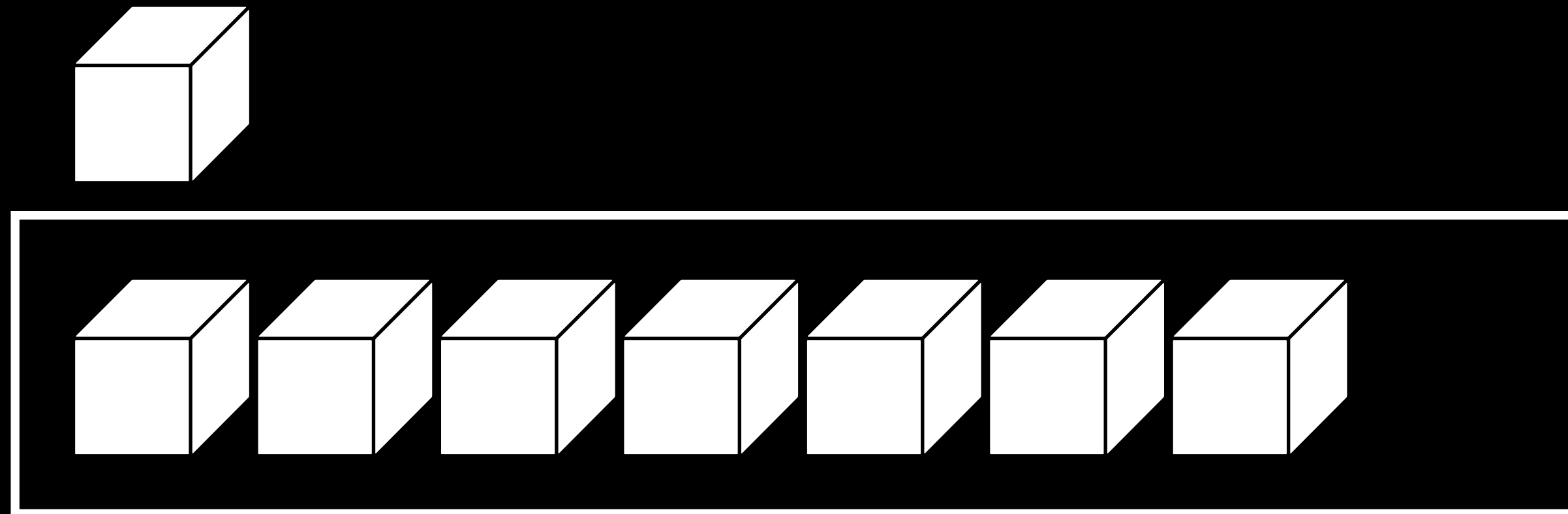
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조 : 1
삭제 : 7

연결 리스트 - Linked List

동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



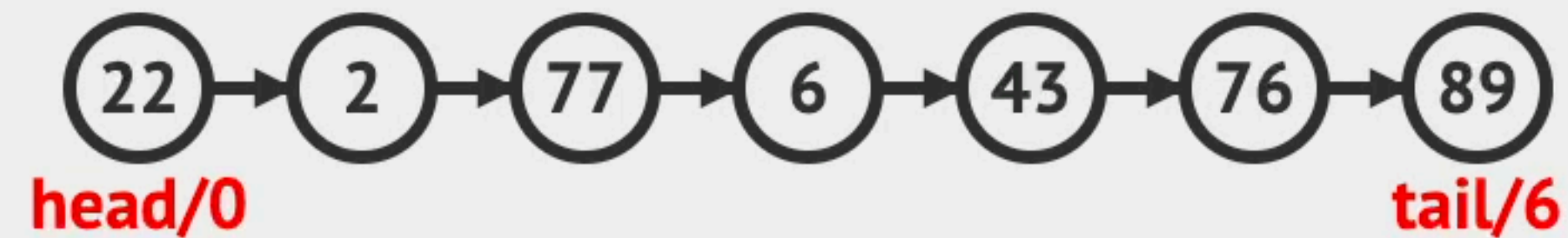
참조 : 1
삭제 : 8

참조연산 시간복잡도 : $O(1)$

삭제연산 시간복잡도 : $O(N)$

연결 리스트 - Linked List

동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제

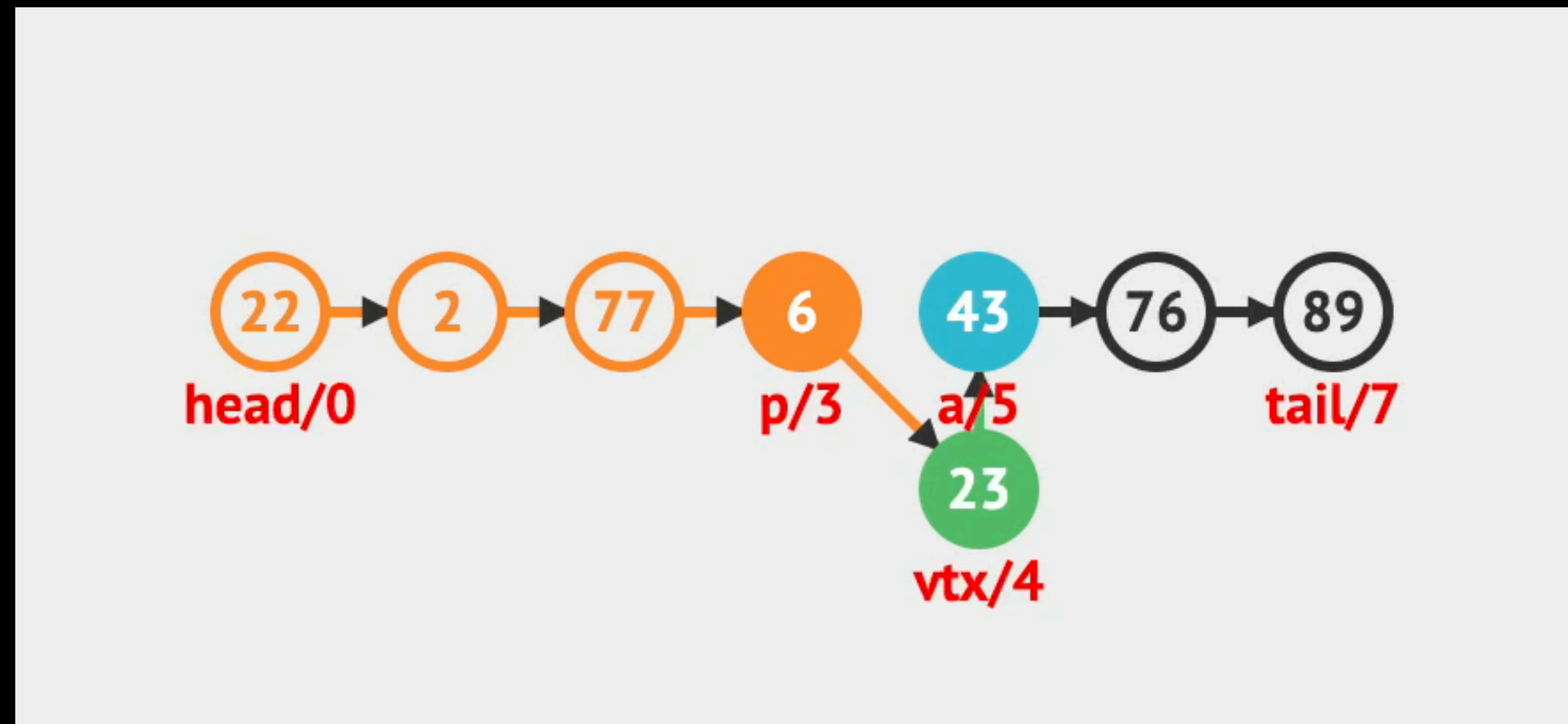


참조연산 시간복잡도 : $O(N)$

삭제연산 시간복잡도 : $O(1)$

연결 리스트 - Linked List

동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제



참조연산 시간복잡도 : $O(N)$

삭제연산 시간복잡도 : $O(1)$

연결 리스트 - Linked List

동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제

	배열	연결리스트
참조 연산	O(1)	O(N)
맨 뒤에 원소 삽입/삭제	O(1)	O(1)
중간에 원소 삽입/삭제	O(N)	O(1)
맨 앞에 원소 삽입/삭제	O(N)	O(1)

연결 리스트 - Linked List

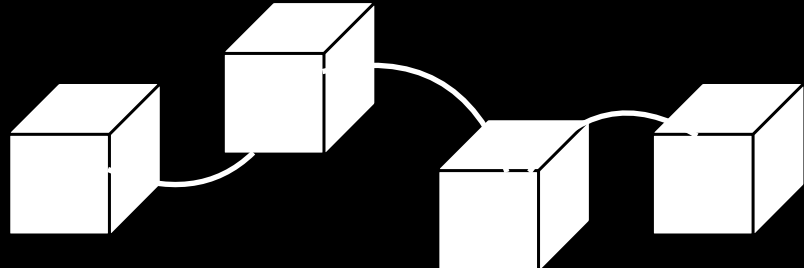
동적배열과 연결리스트 - 데이터 참조, 삽입, 삭제

- 문제 : 유효하지 않은 이메일 형식 데이터 삭제
- 입력값 : 이메일 데이터의 개수 N

	배열	연결리스트
데이터 순회	$O(N)$	$O(N)$
데이터 검사	$O(1)$	$O(1)$
데이터 삭제	$O(N)$	$O(1)$
결과	$O(N^2)$	$O(N)$

연결 리스트 - Linked List

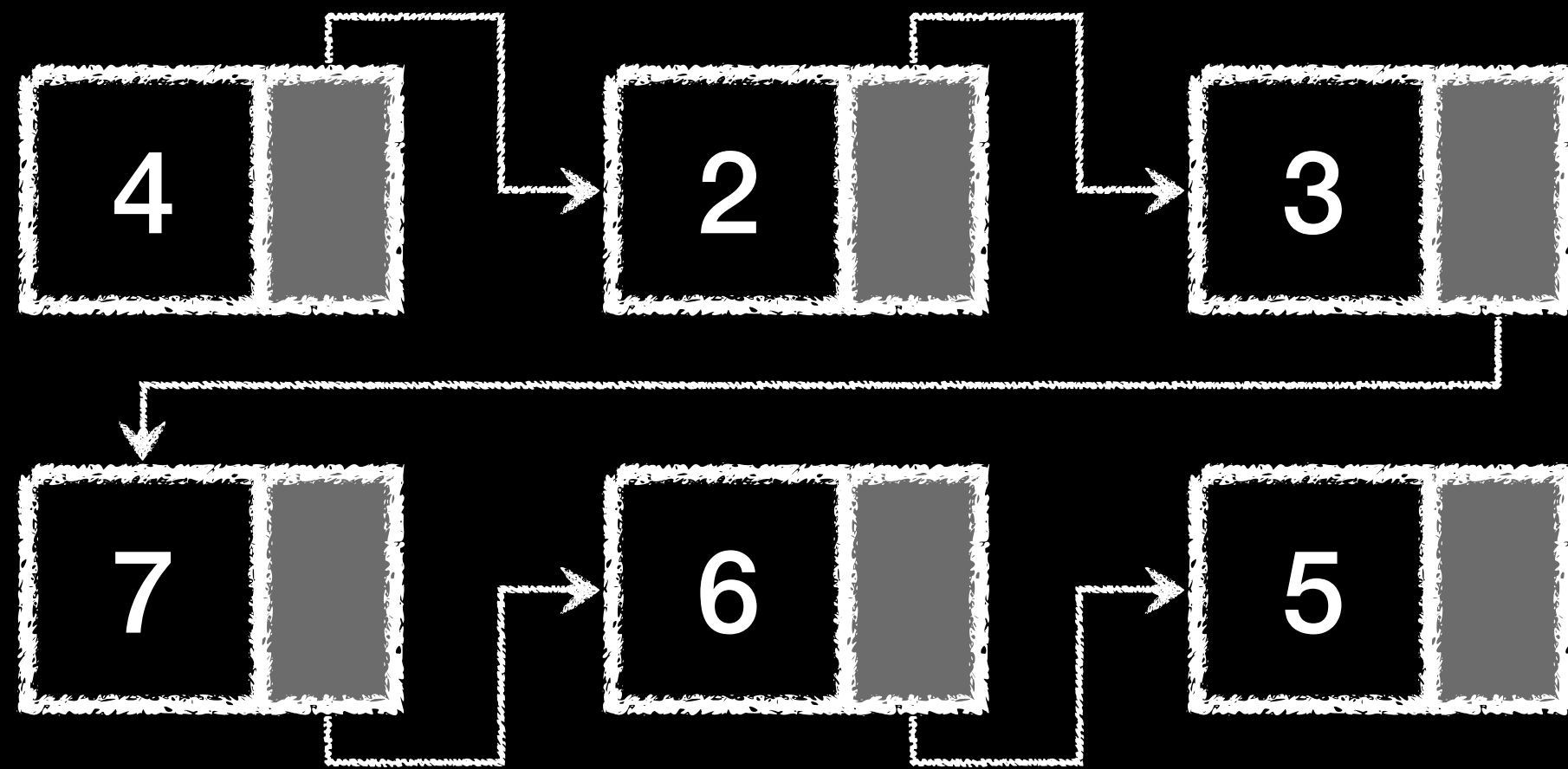
동적배열과 연결리스트 - 메모리 효율성

	배열	연결리스트
구조		
장점	동적 크기	
	데이터 참조 빠름	중간의 데이터 삽입/삭제 빠름
	메모리 사용 효율적	
	캐시 효율이 좋음	
단점	중간의 데이터 삽입/삭제 느림	데이터 참조 느림
		추가적인 메모리 필요
		캐시 효율이 안 좋음

연결 리스트 - Linked List

동적배열과 연결리스트 - 메모리 효율성

- 연결리스트 구조체 (예시)



```
struct Linked_List
{
    int number;
    char name[20];
    // ...
    struct Linked_List *next;
};
```

연결 리스트 - Linked List

동적배열과 연결리스트 - 메모리 효율성

- 데이터 크기가 작은 경우
- 입력 데이터의 개수 : 100개

```
struct Linked_List
{
    int number;    // 4 byte
    struct Linked_List *next; // 8 byte
};
```

배열 : $4 * 100 = 400$ byte

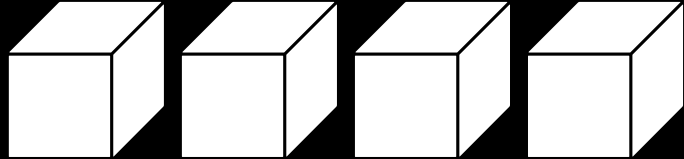
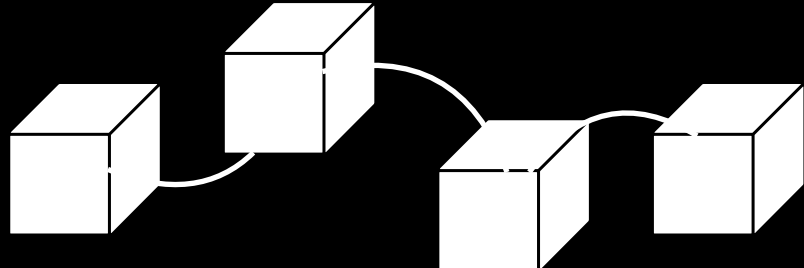
연결리스트 : $(4 + 8) * 100 = 1200$ byte

→ 실제로는 노드 하나에 16 바이트가 되어서 최종적으로 1600byte가 된다.

약 3배차이

연결 리스트 - Linked List

동적배열과 연결리스트 - Cache Hit Rate

	배열	연결리스트
구조		
장점	동적 크기	
	데이터 참조 빠름	중간의 데이터 삽입/삭제 빠름
	메모리 사용 효율적	
	캐시 효율이 좋음	
단점	중간의 데이터 삽입/삭제 느림	데이터 참조 느림
		추가적인 메모리 필요
		캐시 효율이 안 좋음

연결 리스트 - Linked List

동적배열과 연결리스트 - Cache Hit Rate

- CPU는 메모리(RAM)에 접근하기 전에 캐시 메모리에 데이터가 있는지 확인한다.
- Cache Hit Rate이란 요청한 데이터를 캐시 메모리에서 찾을 확률
- 캐시 메모리는 지역성(Locality) 개념에 근거하여 데이터를 저장한다.
 - 시간 지역성(Time Locality) : **최근에** 참조한 데이터는 다시 참조될 확률이 높다.
 - 공간 지역성(Space Locality) : 참조된 데이터와 **인접한** 데이터는 참조될 확률이 높다.

연결 리스트 - Linked List

동적배열과 연결리스트 - Cache Hit Rate

- CPU가 메모리가 접근하기 전에 캐시 데이터가 있는지 확인한다.
- Cache Hit Rate이란 요청한 데이터를 캐시 메모리에서 찾을 확률
- 캐시 메모리는 지역성(Locality) 개념에 근거하여 데이터를 저장한다.
 - 시간 지역성(Time Locality) : 최근에 참조한 데이터는 다시 참조될 확률이 높다.
 - 공간 지역성(Space Locality) : 참조된 데이터와 인접한 데이터는 참조될 확률이 높다.

캐시 메모리에는 최근에 참조된 데이터와

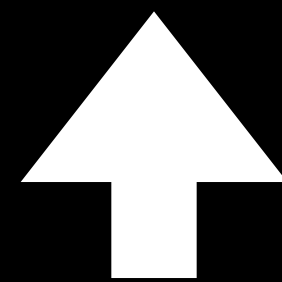
참조된 데이터와 인접한 데이터를 가지고 온다.

연결 리스트 - Linked List

동적배열과 연결리스트 - Cache Hit Rate

물리적으로 연속적인 메모리에 데이터를 저장하는 배열과 달리

연결리스트는 비연속적인 메모리에 저장하기 때문에 캐시 효율성이 떨어진다.



캐시 메모리에는 최근에 참조된 데이터와

참조된 데이터와 인접한 데이터를 가지고 온다.

연결 리스트 - Linked List

동적배열과 연결리스트 - Cache Hit Rate

물리적으로 연속적인 메모리에 데이터를 저장하는 배열과 달리

연결리스트는 비연속적인 메모리에 저장하기 때문에 캐시 효율성이 떨어진다.

따라서 리스트의 요소를 순회할 때 물리적으로 연속적인 배열에 비해

메모리 최적화가 안되는 연결 리스트가 더 느리다.

연결 리스트 - Linked List

정리 및 결론

	배열	연결리스트
구조		
장점	동적 크기	
	데이터 참조 빠름	중간의 데이터 삽입/삭제 빠름
	메모리 사용 효율적	
	캐시 효율이 좋음	
단점	중간의 데이터 삽입/삭제 느림	데이터 참조 느림
		추가적인 메모리 필요
		캐시 효율이 안 좋음

- 반복 중에 삽입/삭제 연산을 해야한다면 **연결리스트!**
- 그 외에는 **동적배열**을 활용!

연결 리스트 - Linked List

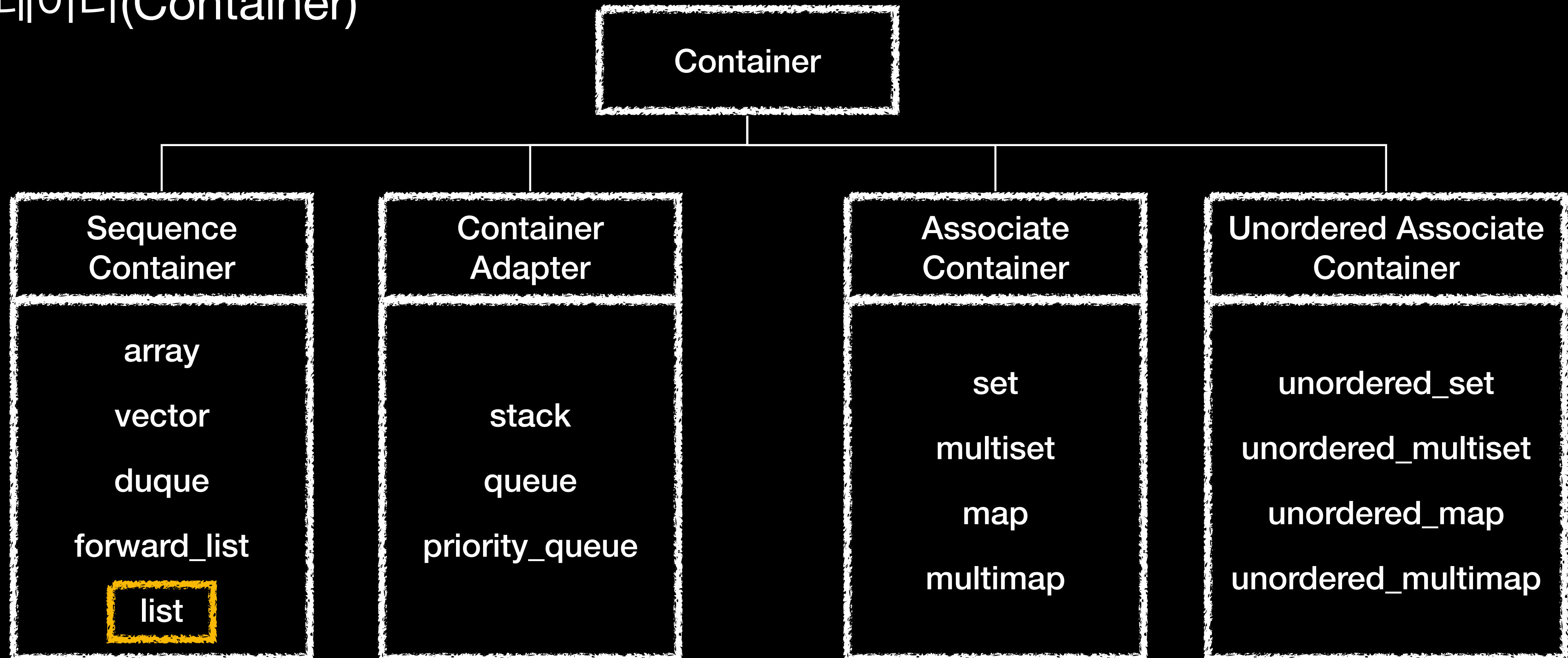
추가 : C++ STL(Standard Template Library)

- STL : Standard Template Library (표준 템플릿 라이브러리)
- 다양한 자료구조, 알고리즘 등을 템플릿을 통해 추상화한 라이브러리
- STL의 구성요소
 - 컨테이너(Container) : vector, list, queue 등
 - 반복자(Iterator) : iterator 등
 - 알고리즘(Algorithm) : sort, find 등

연결 리스트 - Linked List

추가 : C++ STL(Standard Template Library)

- 컨테이너(Container)



연결 리스트 - Linked List

추가 : C++ STL(Standard Template Library)

Container	Insertion	Access	Erase	Find	Persistent Iterators
vector / string	Back: $O(1)$ or $O(n)$ Other: $O(n)$	$O(1)$	Back: $O(1)$ Other: $O(n)$	Sorted: $O(\log n)$ Other: $O(n)$	No
deque	Back/Front: $O(1)$ Other: $O(n)$	$O(1)$	Back/Front: $O(1)$ Other: $O(n)$	Sorted: $O(\log n)$ Other: $O(n)$	Pointers only
list / forward_list	Back/Front: $O(1)$ With iterator: $O(1)$ Index: $O(n)$	Back/Front: $O(1)$ With iterator: $O(1)$ Index: $O(n)$	Back/Front: $O(1)$ With iterator: $O(1)$ Index: $O(n)$	$O(n)$	Yes
set / map	$O(\log n)$	-	$O(\log n)$	$O(\log n)$	Yes
unordered_set / unordered_map	$O(1)$ or $O(n)$	$O(1)$ or $O(n)$	$O(1)$ or $O(n)$	$O(1)$ or $O(n)$	Pointers only
priority_queue	$O(\log n)$	$O(1)$	$O(\log n)$	-	-

연결 리스트 - Linked List

추가 : C++ STL(Standard Template Library)

- 반복자(Iterator) : 컨테이너에 원소에 접근할 수 있는 **포인터**와 같은 객체
- 컨테이너는 반복자를 사용하여 원소에 접근할 수 있다.
- 반복자는 컨테이너에 iterator 멤버 타입으로 정의되어 있다.

```
for (auto iter = list.begin(); iter != list.end(); iter++) {  
    cout << *iter << endl;  
}
```

```
for (auto &iter: list) {  
    cout << iter << endl;  
}
```

```
for (int &iter: list) {  
    cout << iter << endl;  
}
```

* Range-based for loop (범위 기반 for 반복문)는 C++11 부터 사용할 수 있다.

연결 리스트 - Linked List

추가 : C++ STL(Standard Template Library)

- 알고리즘(Algorithm) : 반복자를 통해 컨테이너에서 가져온 원소를 조작

```
int main() {  
    list<int> list;  
  
    list.push_front(x: 1);  
    list.push_front(x: 2);  
    list.push_front(x: 3);  
    list.push_front(x: 4);  
  
    for (int &iter: list)  
        cout << iter << endl;  
  
    list.sort();  
  
    for (int &iter: list)  
        cout << iter << endl;  
  
    return (0);  
}
```

```
int main() {  
    vector<int> list;  
  
    list.push_back(4);  
    list.push_back(3);  
    list.push_back(2);  
    list.push_back(1);  
  
    for (int &iter: list)  
        cout << iter << endl;  
  
    sort(first: list.begin(), last: list.end());  
  
    for (int &iter: list)  
        cout << iter << endl;  
  
    return (0);  
}
```

4
3
2
1
1
2
3
4

끝