UE4
Garbage Collection

쿠재아이
김재경

기존 메모리 관리의 문제점 (C++)

- 메모리 누수

```
void main()
{
    Object* obj = new Object();

    // obj를 사용 중...

    // 아차! obj를 delete하는 걸 깜빡했다!
}
```

기존 메모리 관리의 문제점 (C++)

- 유효하지 않은 포인터 접근

```
void main()
{
    Object* obj = new Object();

    // obj를 사용 중...

    delete obj;

    // 다른 작업 중...

    obj->Something(); // 이런! 메모리에서 해제된 객체에 접근했다!
}
```

기존 메모리 관리의 문제점 (C++)

- 이중 해제

```
void main()
{
    Object* obj = new Object();

    // obj를 사용 중...

    delete obj;

    // 다른 작업 중...

    delete obj; // 헉! 해제된 객체를 또 해제했다!
}
```

하지 말아야 하지만… 누군가 꼭 한다

메모리 관리를 자동으로 할 수 없을까?

그래서 나온 것이 Garbage Collection

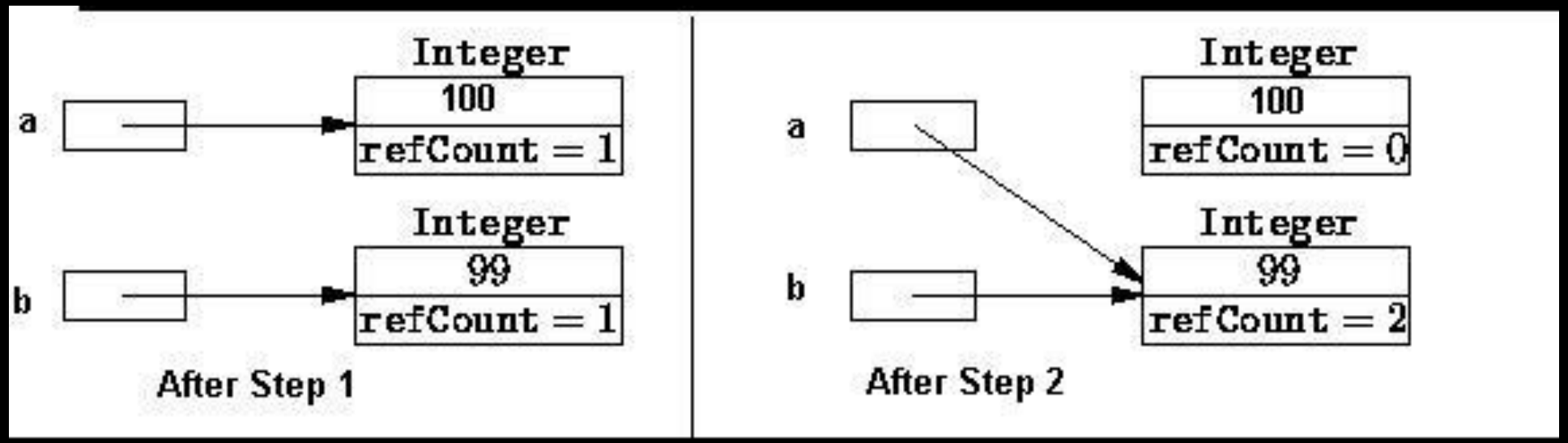
이것만 있으면 메모리를 자동으로 관리할 수 있다

Garbage Collection이 어떻게 구현되는지 알아봅시다

1. Reference Counting

Reference Counting

- 객체마다 Reference Count를 둬서 관리한다



Reference Counting

- 장점

- 프로그래머가 객체의 해제 시점을 어느 정도 예측할 수 있다

- 객체가 사용된 직후에 메모리를 해제하므로, 메모리 해제 시점에 해당 객체는 캐시에 저장되어 있을 확률이 높다. 따라서 메모리 해제가 빠르게 이루어진다

Reference Counting

- 단점

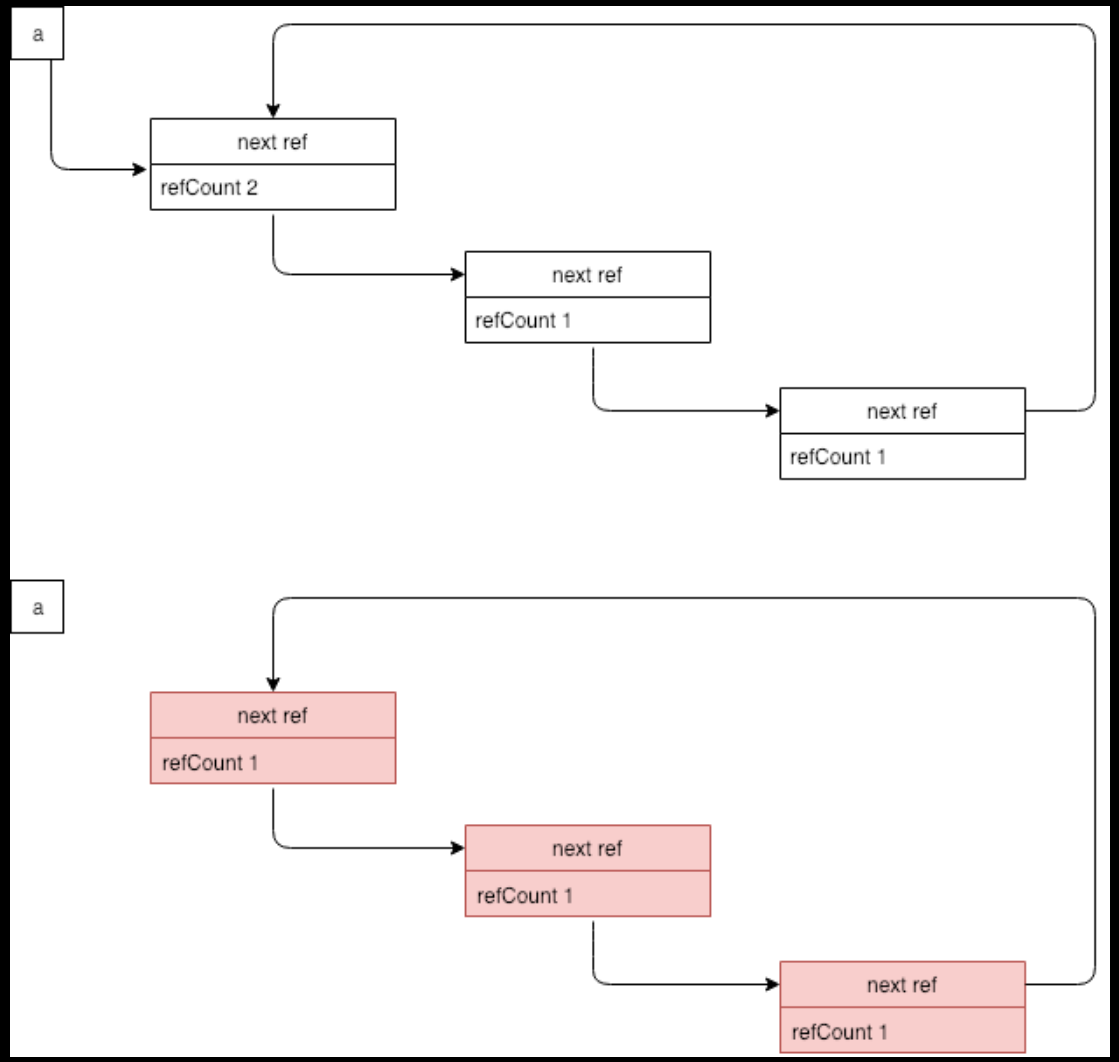
- 멀티스레드 환경에서는, 스레드간에 공유하는 객체의 참조 횟수 계산을 위해 원자적 명령을 사용하거나 락을 걸어야 한다. 이 문제를 회피하기 위해 스레드 단위 지역 변수로 참조 횟수를 따로 관리하면서, 스레드의 참조 횟수가 0이 될 때만 전역 참조 횟수를 확인하는 방식을 사용할 수 있다. 리눅스 커널에서 이 방식을 사용한다

- 참조 횟수가 0이 될 때, 해당 객체가 가리키는 다른 객체들 또한 동시에 0으로 만드는 작업이 일어난다. 이 과정은 경우에 따라 많은 시간이 걸릴 수도 있기 때문에 실시간 시스템에는 적합하지 않을 수 있다

Reference Counting

- 단점

- 순환 참조가 일어날 수 있다

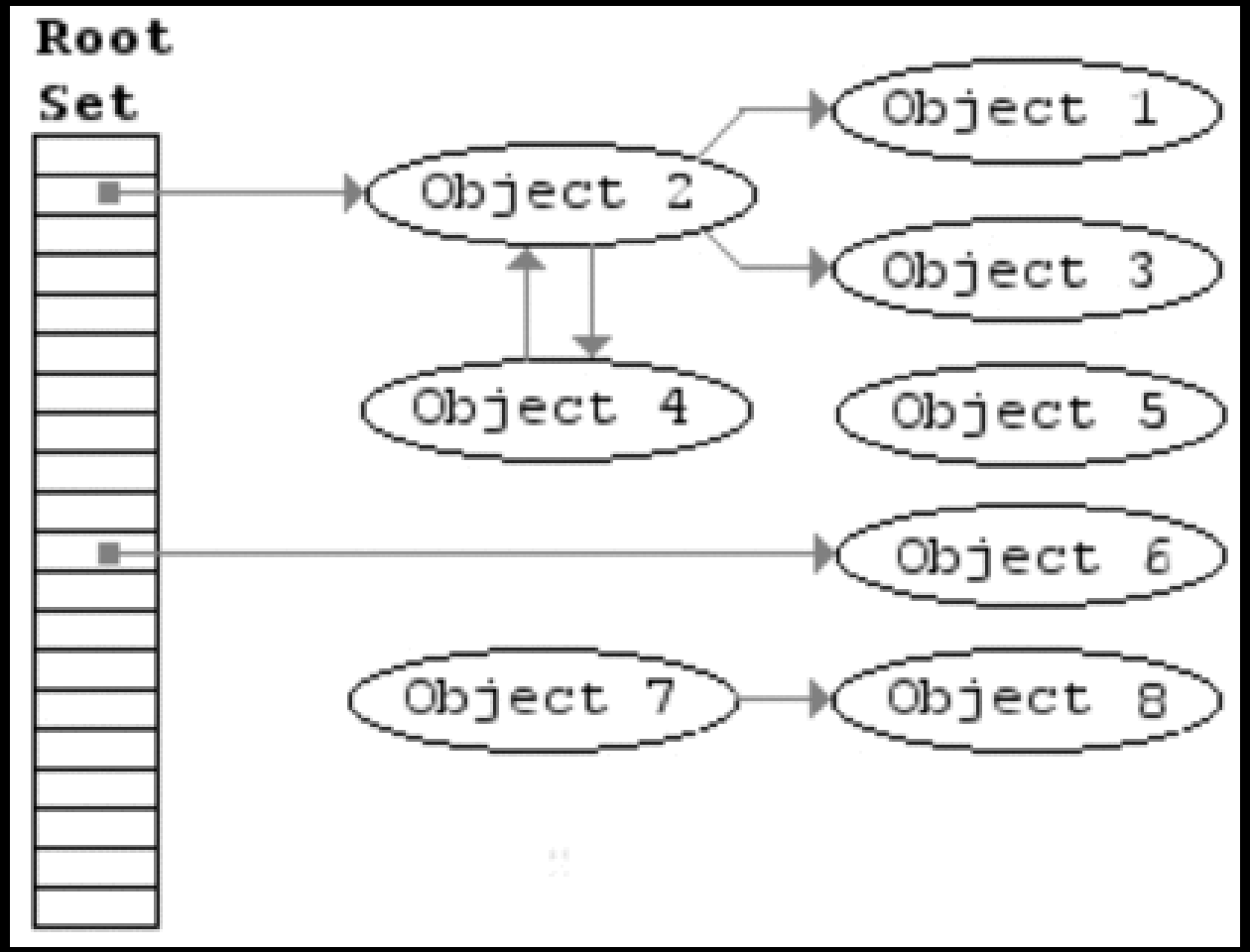


2. Naïve mark-and-sweep

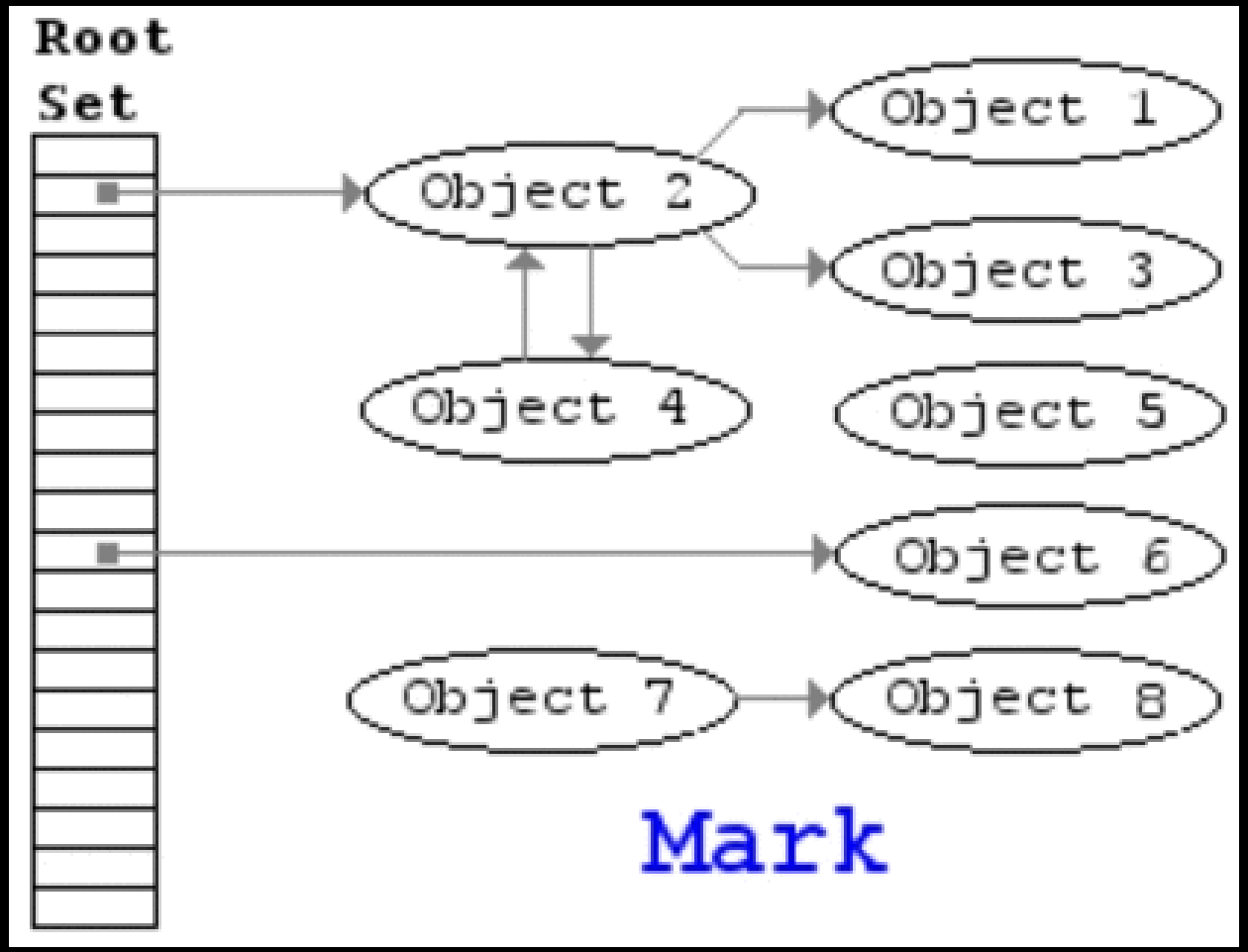
Naïve mark-and-sweep

- 메모리에는 '사용 중'을 나타내는 플래그가 있다
- Mark 단계 : Root Set에서 접근 가능한 객체의 플래그를 '사용 중'으로 마킹한다
- Sweep 단계 : 모든 메모리를 탐색하며 '사용 중'이 아닌 객체의 메모리를 해제한다. '사용 중'인 객체는 플래그를 초기화한다.
- Root Set : Stack이나 Static 공간에 선언된 변수들의 모음 정도로 생각하면 이해가 쉽다(UE4의 Root Set과는 다름)

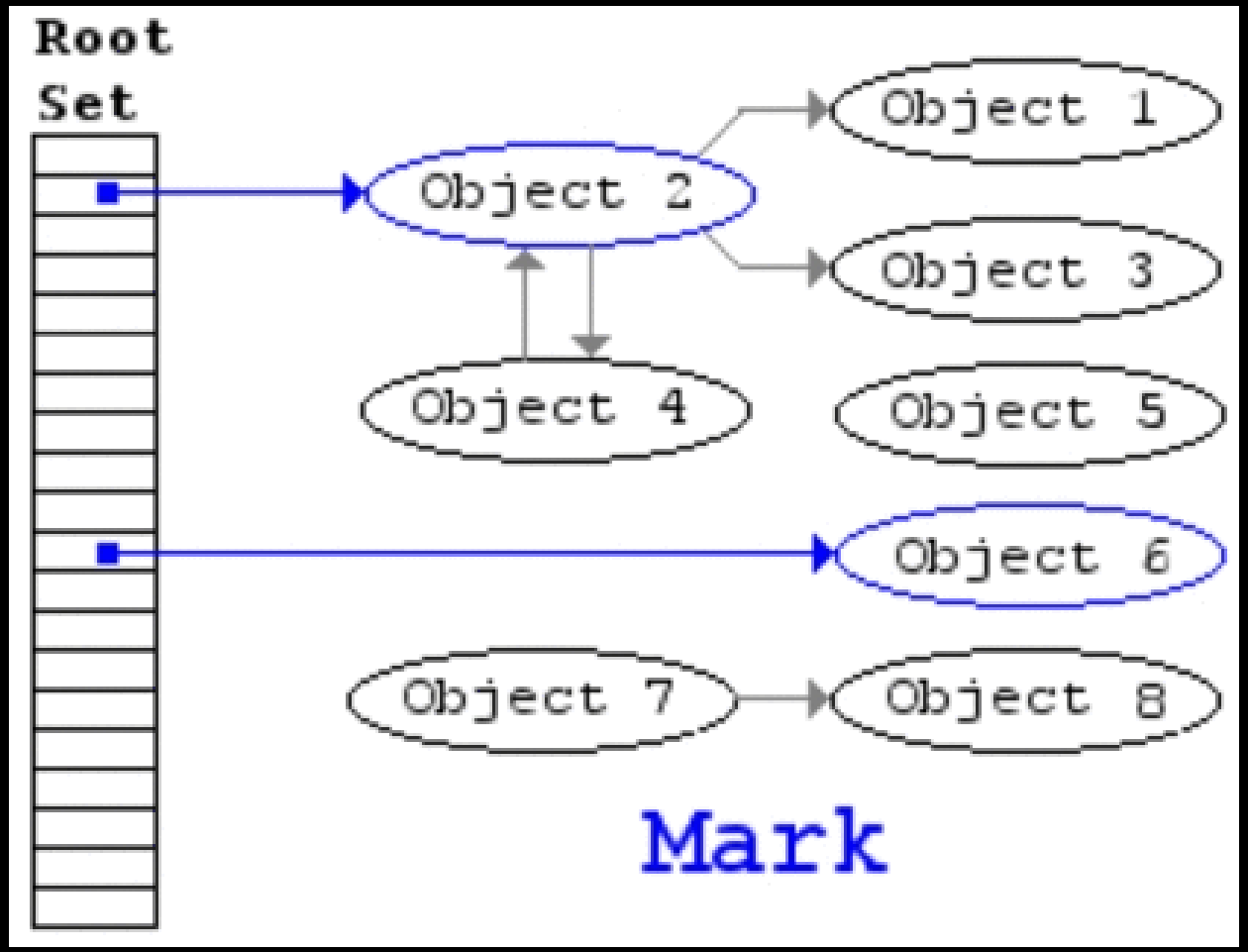
Naïve mark-and-sweep



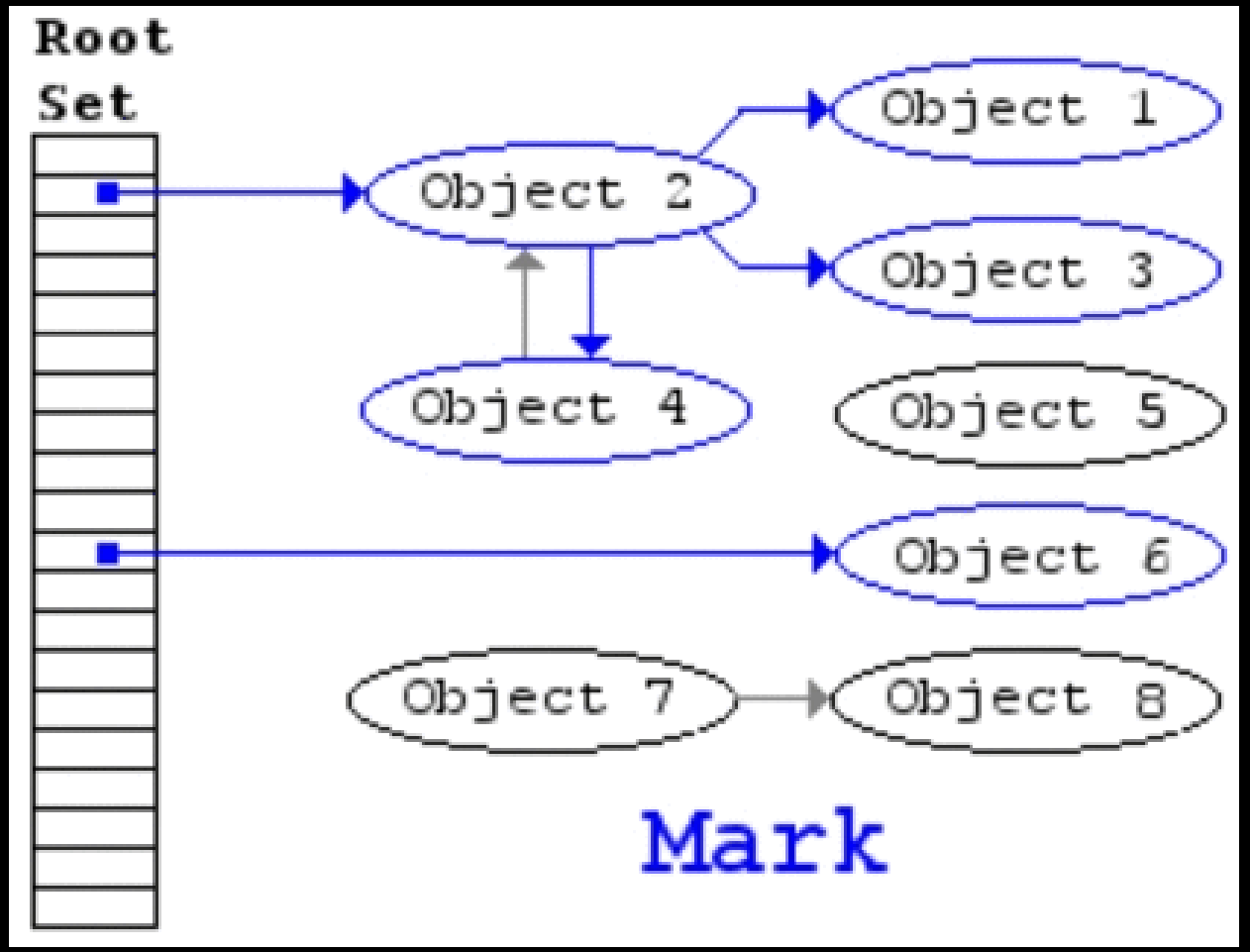
Naïve mark-and-sweep



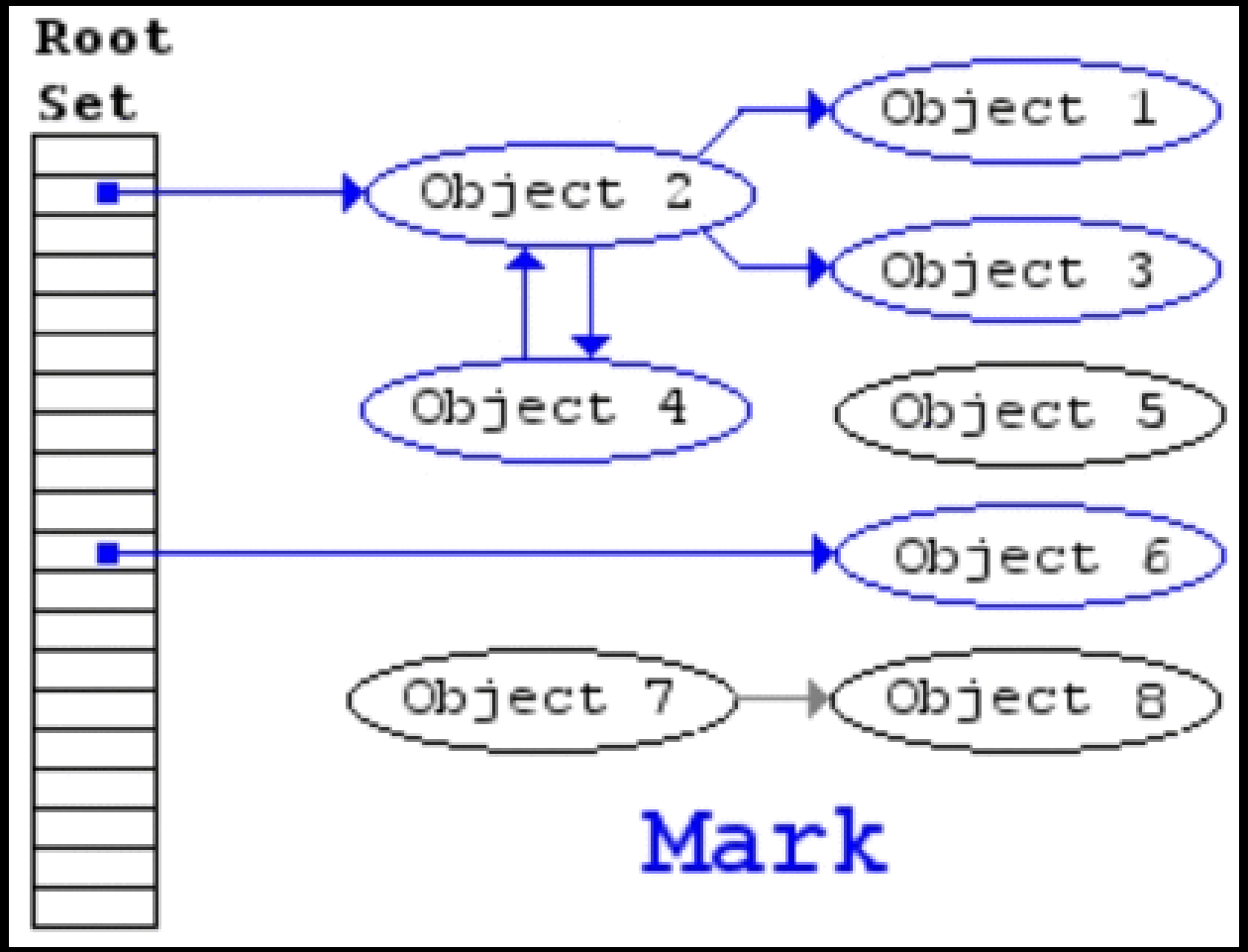
Naïve mark-and-sweep



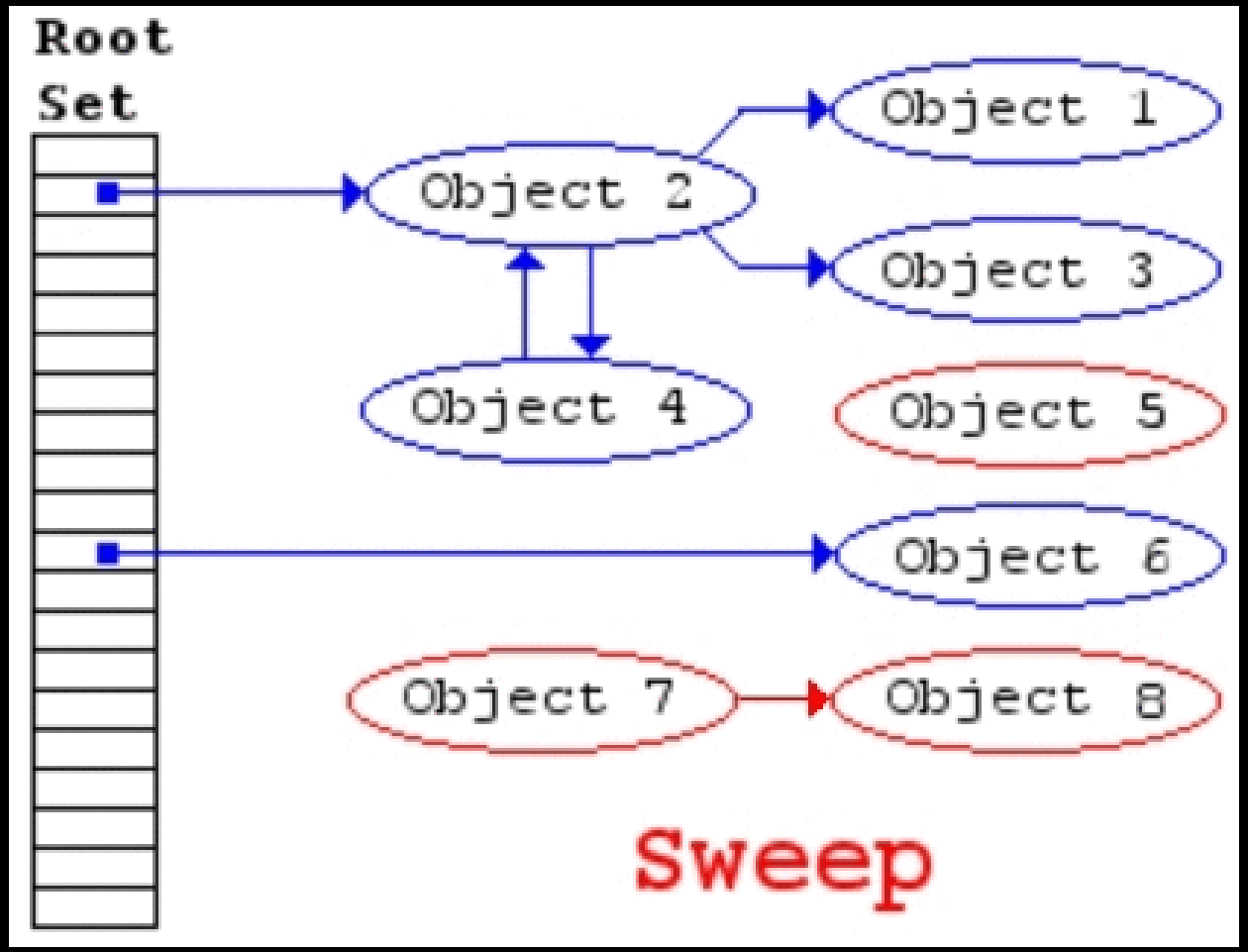
Naïve mark-and-sweep



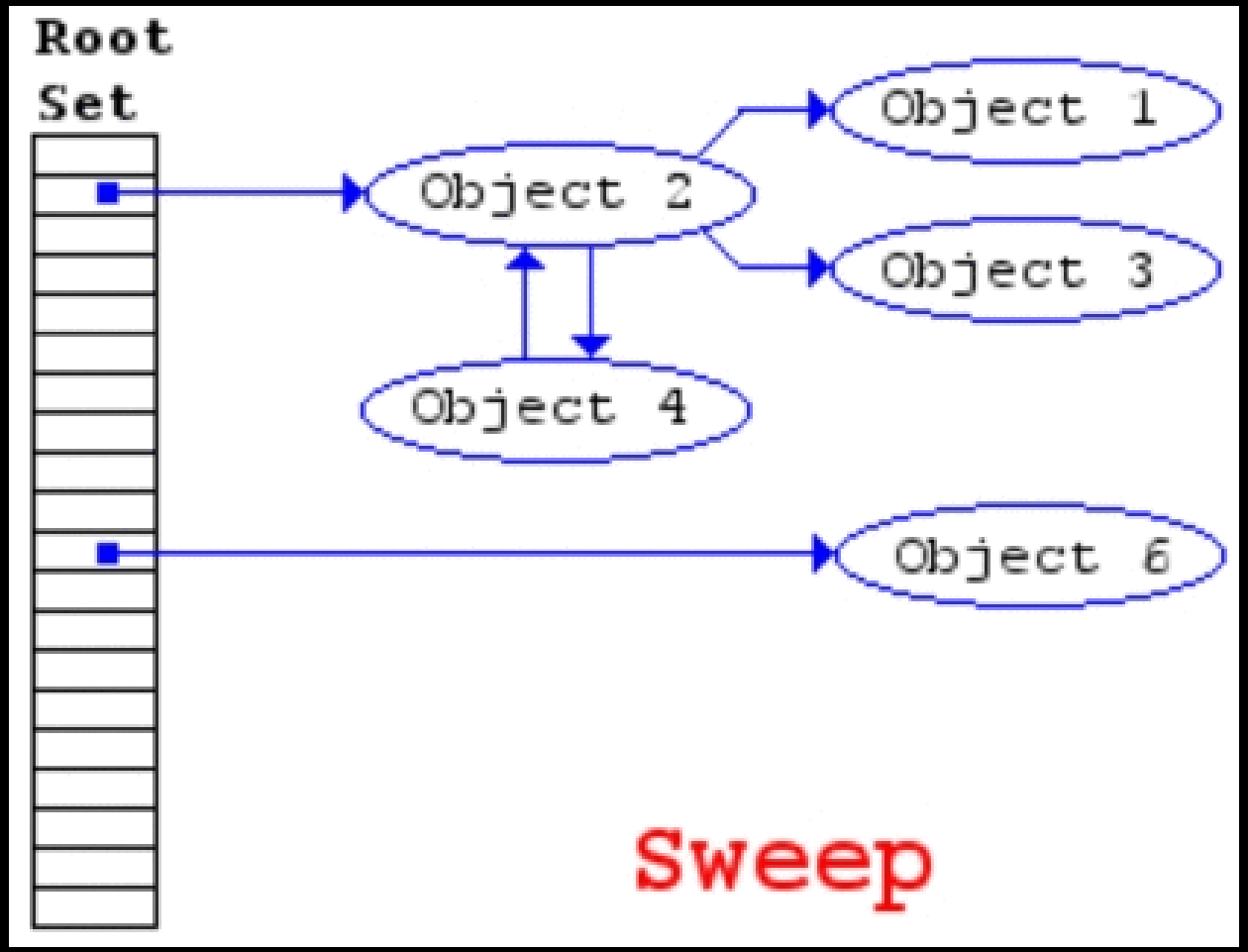
Naïve mark-and-sweep



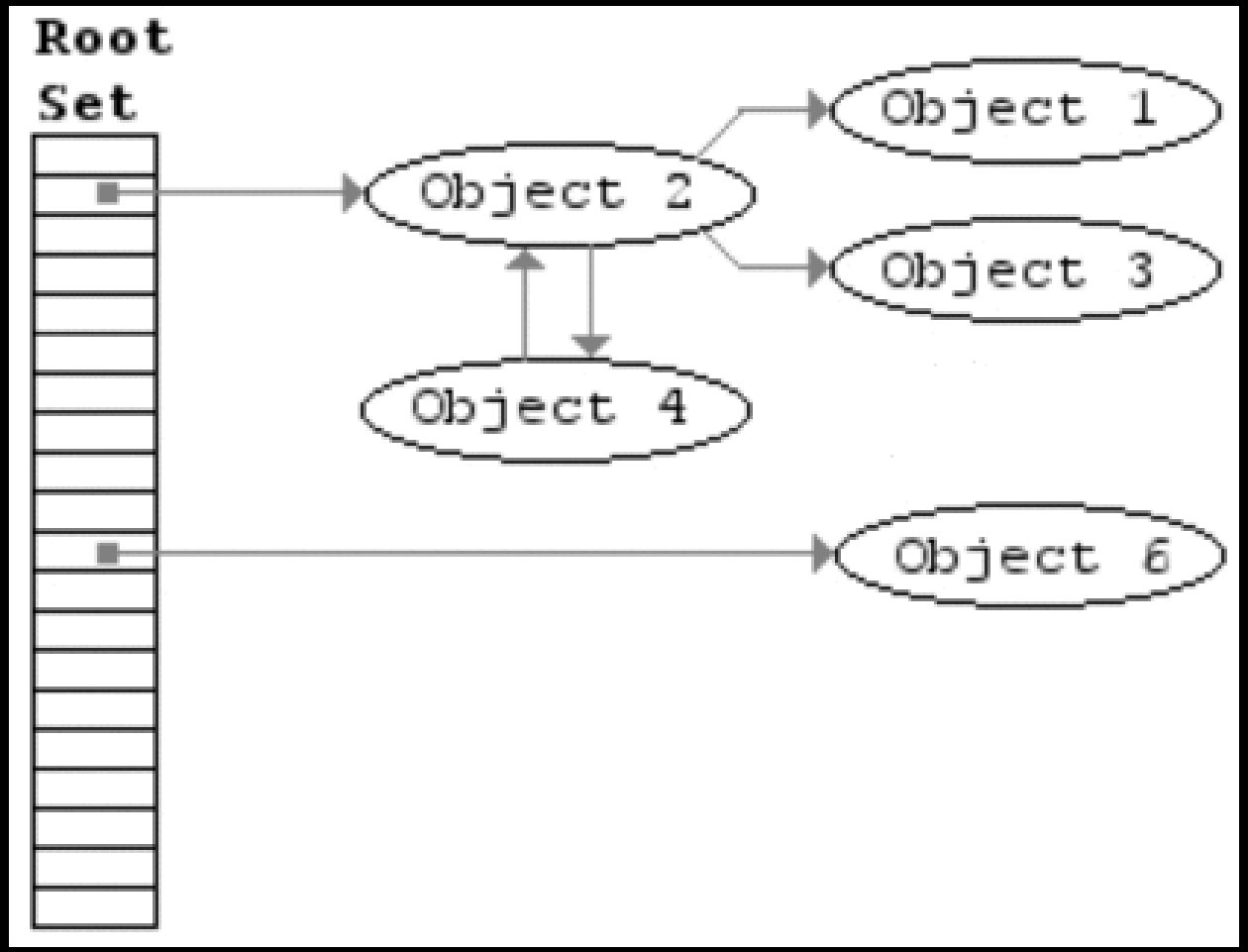
Naïve mark-and-sweep



Naïve mark-and-sweep



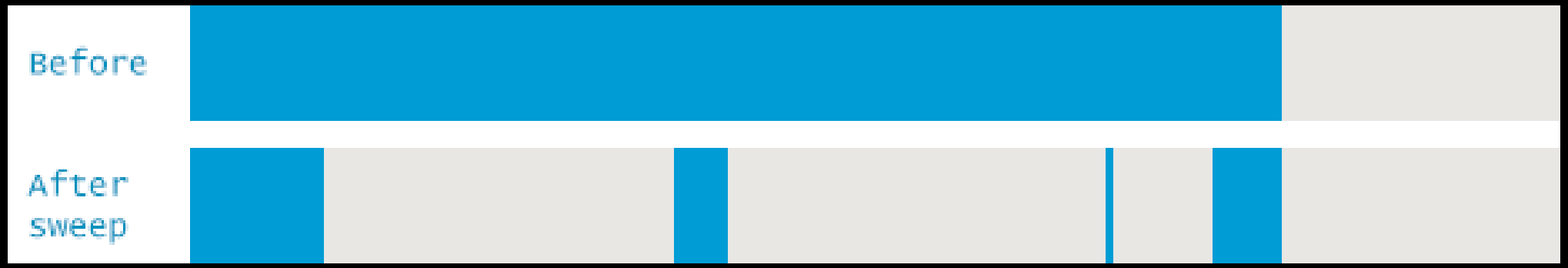
Naïve mark-and-sweep



Naïve mark-and-sweep

- 작업 도중 메모리 변경이 일어나지 않아야 하기 때문에 시스템 전체가 중단되어야 한다

- 메모리 단편화가 발생할 수 있다



3. Tri-color marking

Tri-color marking

- Mark-and-sweep의 단점을 보완

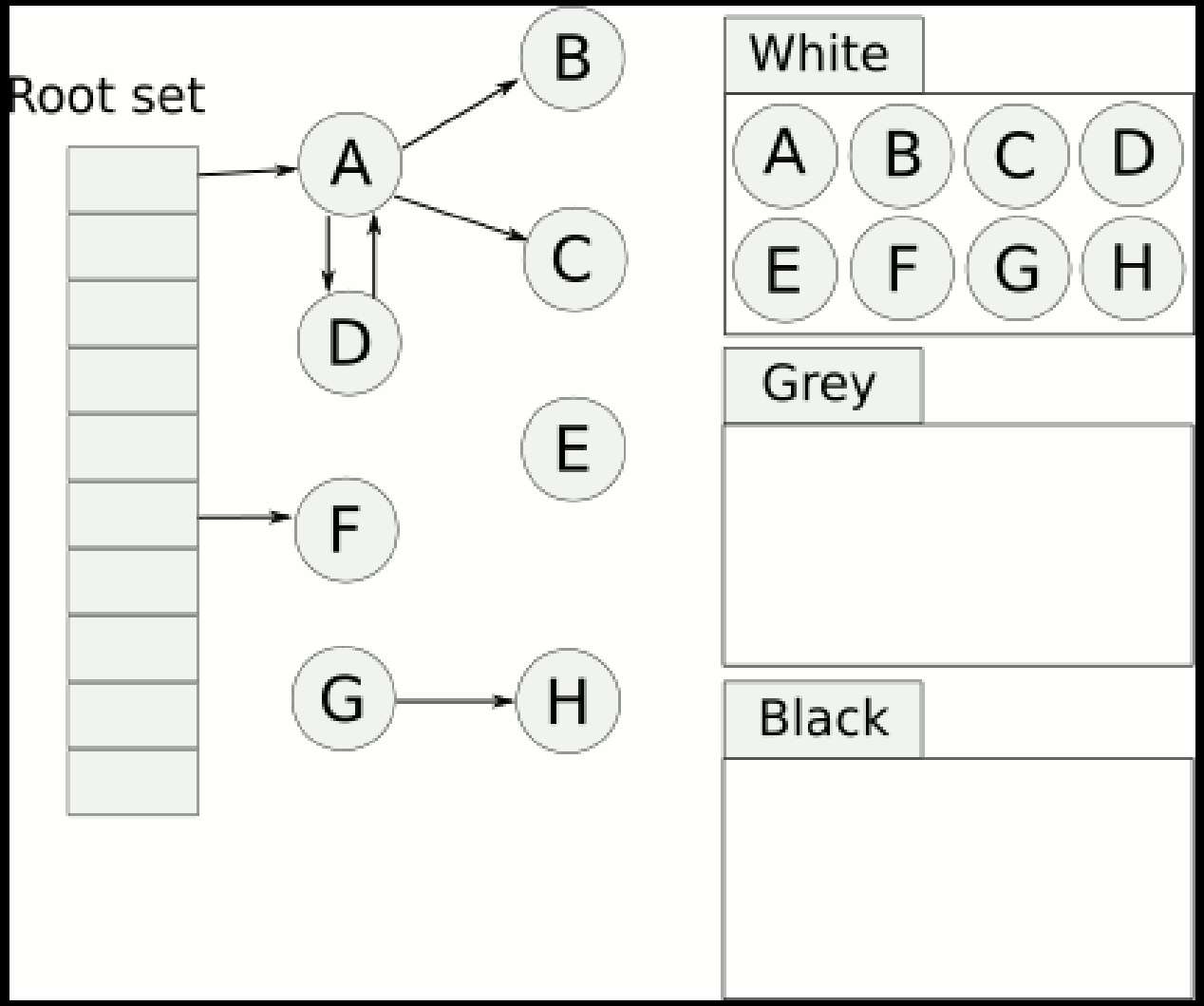
- 객체를 흰색, 회색, 검정색으로 분류

- 흰색 : 메모리 해제 해야할 객체
- 회색 : Root Set에서 접근 가능하지만 이 객체에서 가리키는 객체들은 아직 검사하지 않음
- 검은색 : Root Set에서 접근 가능하고 흰색 객체를 가리키지 않음

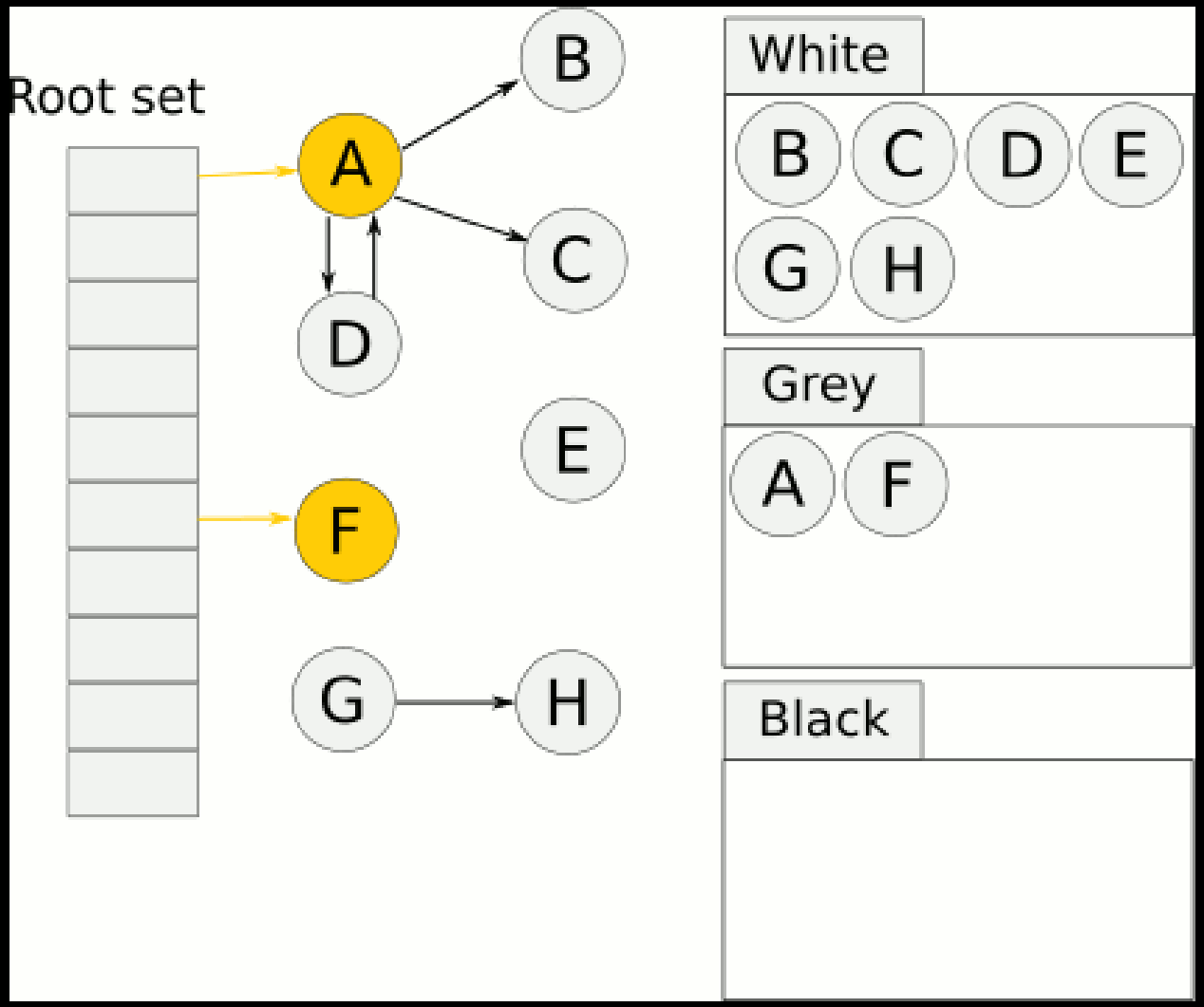
Tri-color marking

1. 회색 객체를 검은색으로 바꾼다
2. 참조하는 객체를 흰색에서 회색으로 바꾼다
3. 1, 2를 반복한다
4. 회색 객체가 존재하지 않으면 흰색 객체를 모두 해제한다

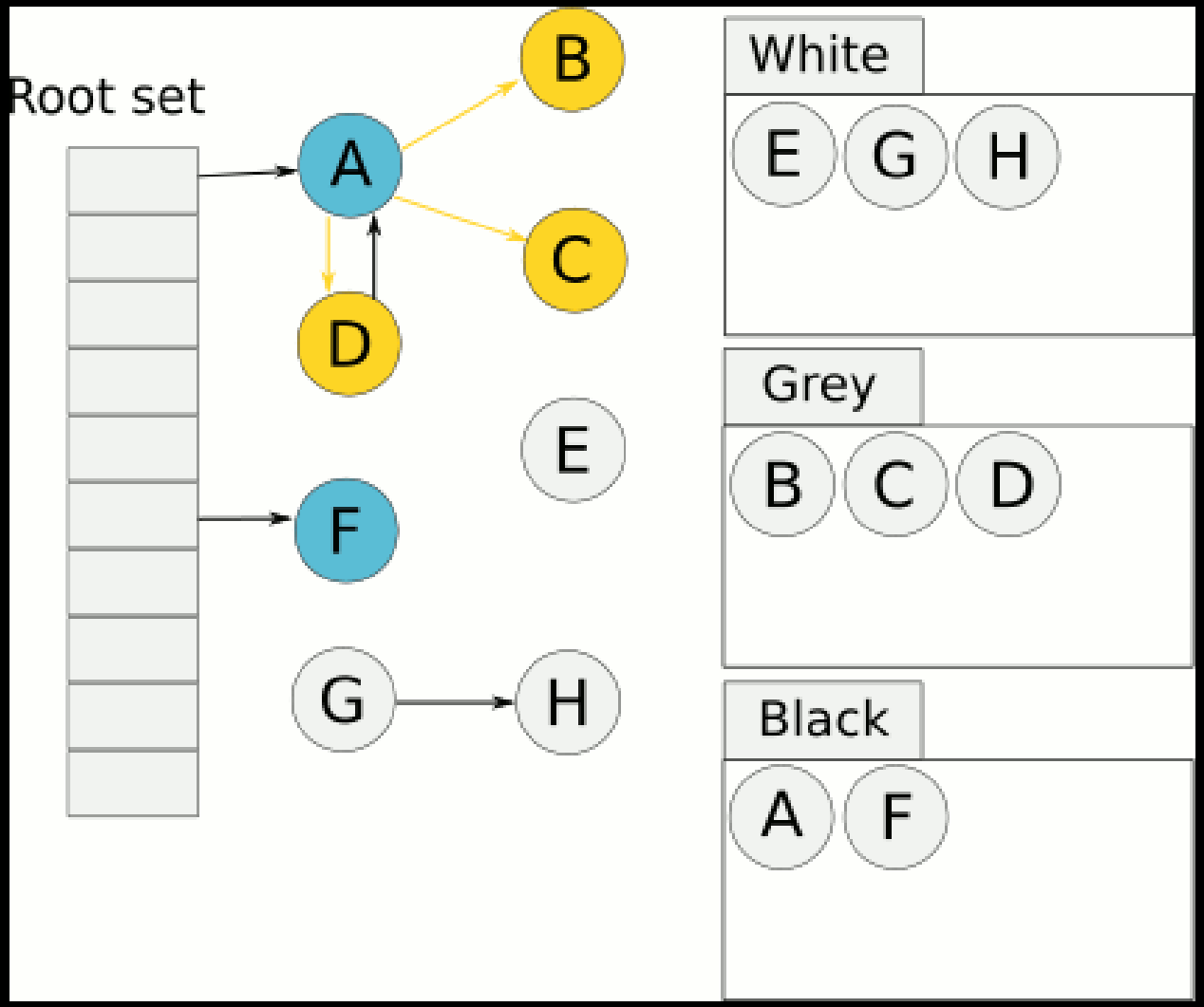
Tri-color marking



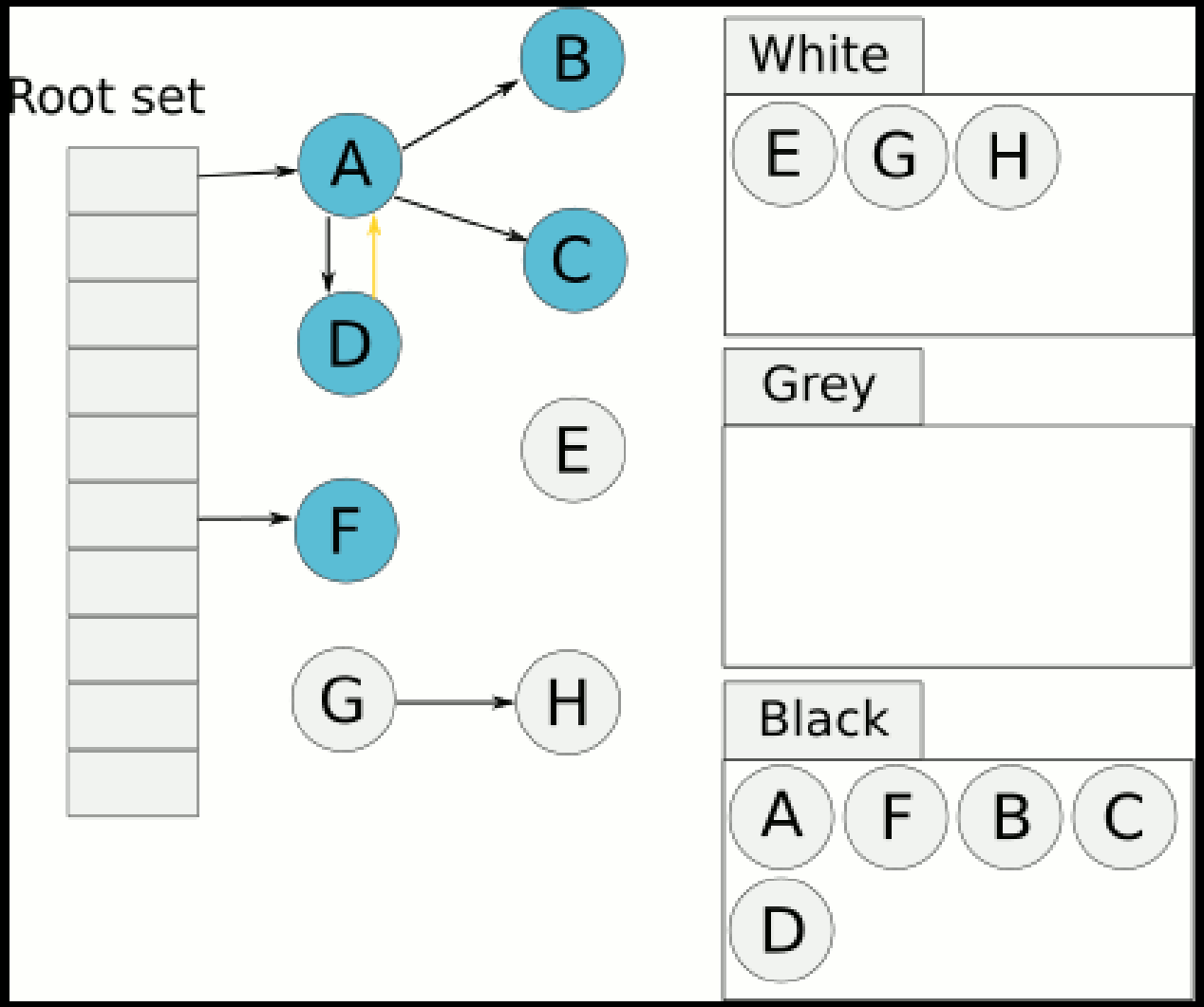
Tri-color marking



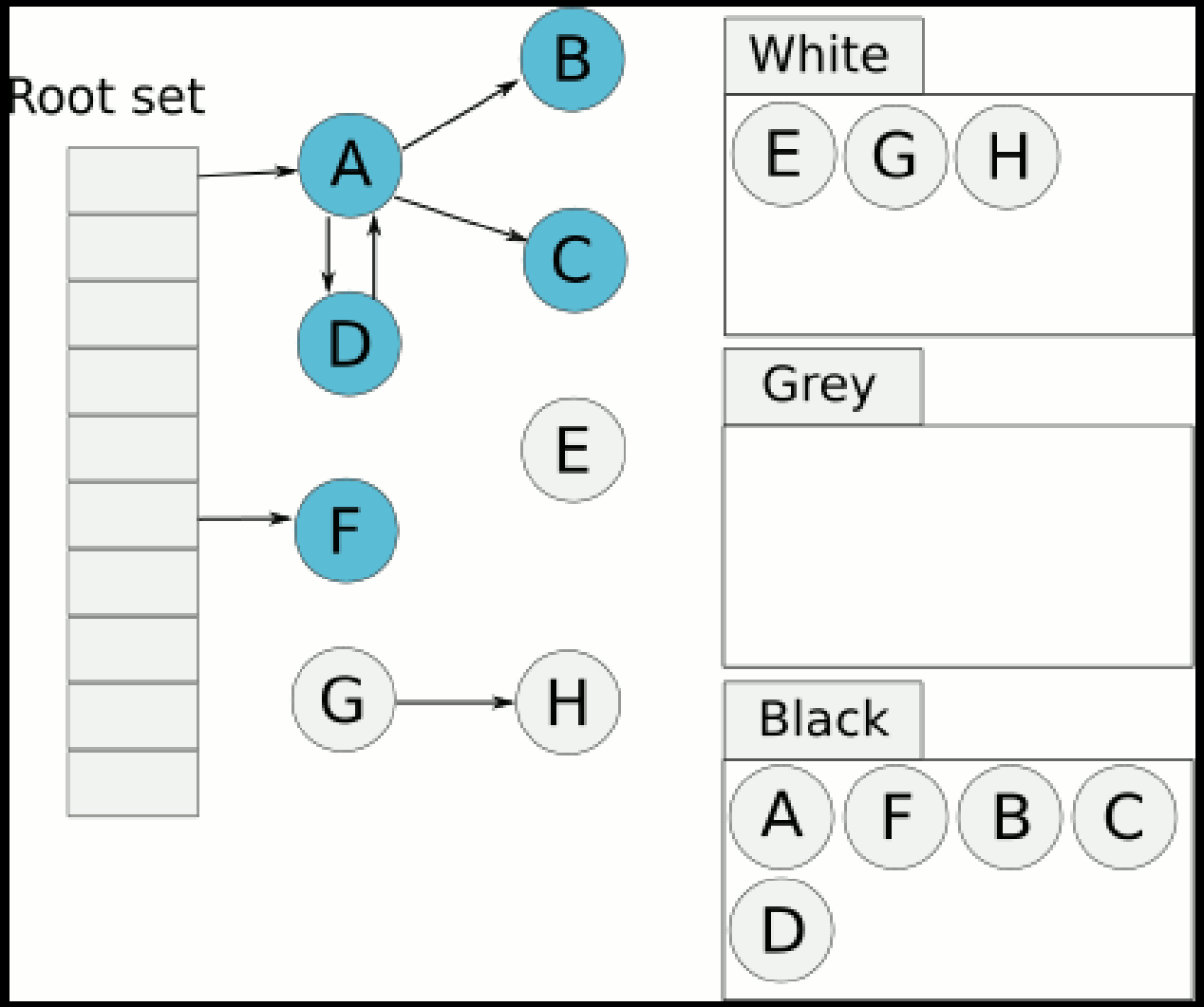
Tri-color marking



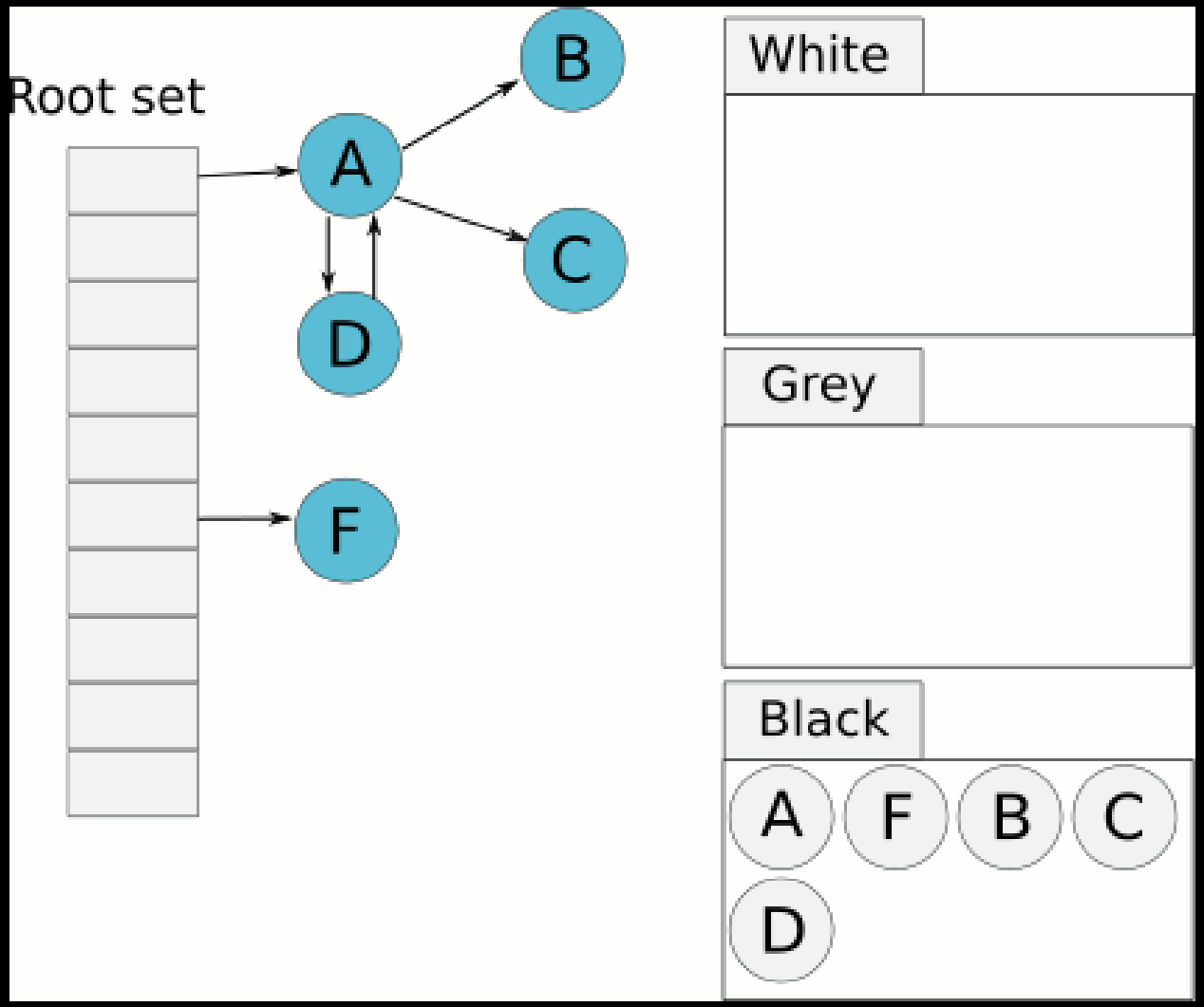
Tri-color marking



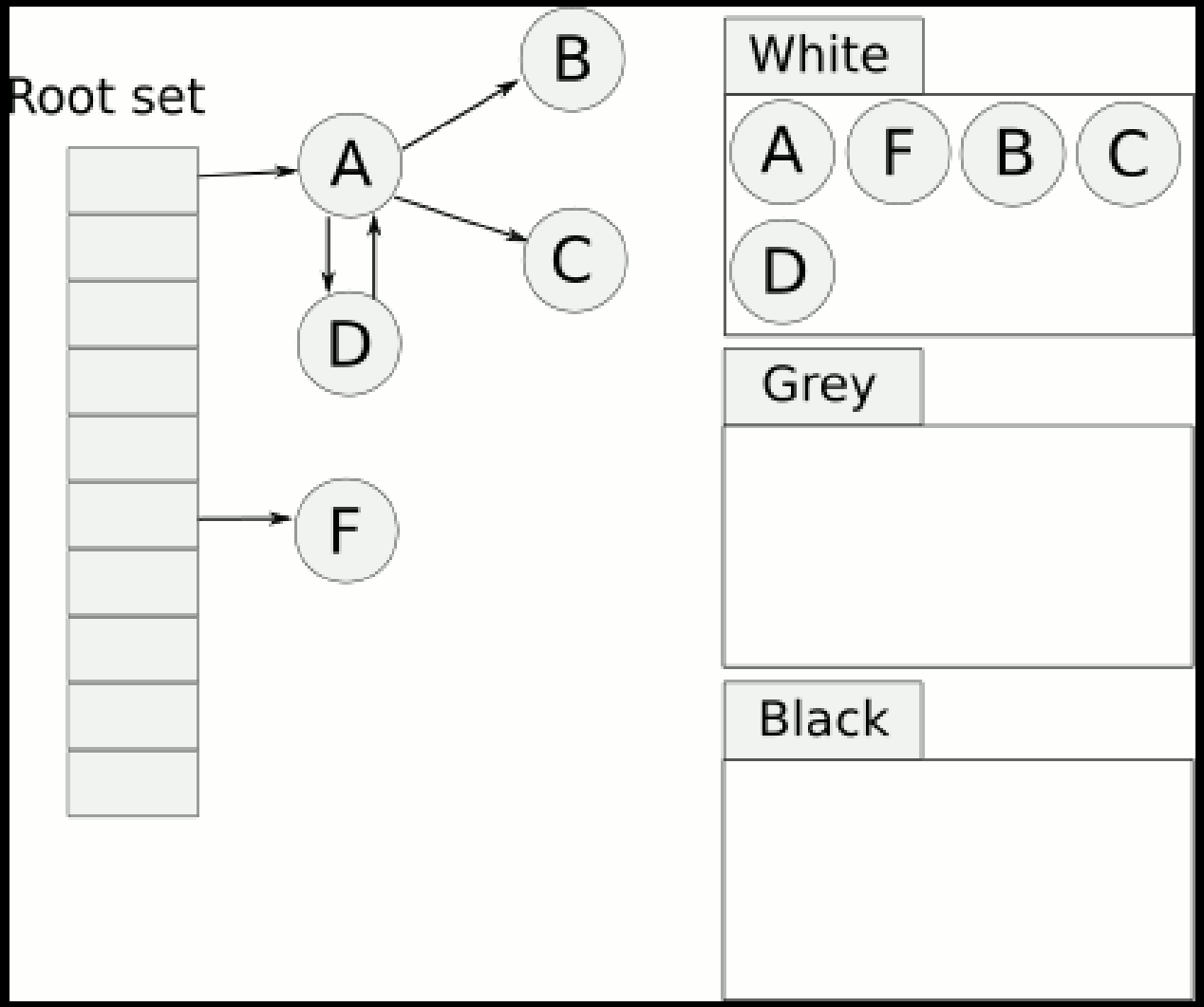
Tri-color marking



Tri-color marking



Tri-color marking



Tri-color marking

- 메모리를 탐색하는 위치를 알고 있기 때문에 시스템을 중지하지 않아도 된다(회색 위치부터 다시 탐색하면 됨)
- 따라서 주기적으로 Gabage Collection을 할 수 있으며 한 번 탐색할 때 메모리 전체를 탐색하지 않아도 된다
- 잘못 마킹 되는 경우가 생길 수 있다

UE4
Garbage Collection

오브젝트 구성

```
// Global UObject array instance
FUObjectArray GUObjectArray;
```

UObjectHash.cpp

생성되는 모든 Uobject들은 GUObjectArray에 추가된다

```
class FUObjectArray
{
    // 다른 부분은 생략
private:
    TUObjectArray ObjObjects;
};
```

UObjectArray.h

```
class TUObjectArray
{
    // 다른 부분은 생략
private:
    FUObjectItem** Objects;
};
```

UObjectArray.h

오브젝트 구성

```
class TUObjectArray
{
    // 다른 부분은 생략
private:
    FUObjectItem** Objects;
};
```

UObjectArray.h

```
class FUObjectItem
{
    // 다른 부분은 생략
    UObjectBase* Object;
    int Flags;
};
```

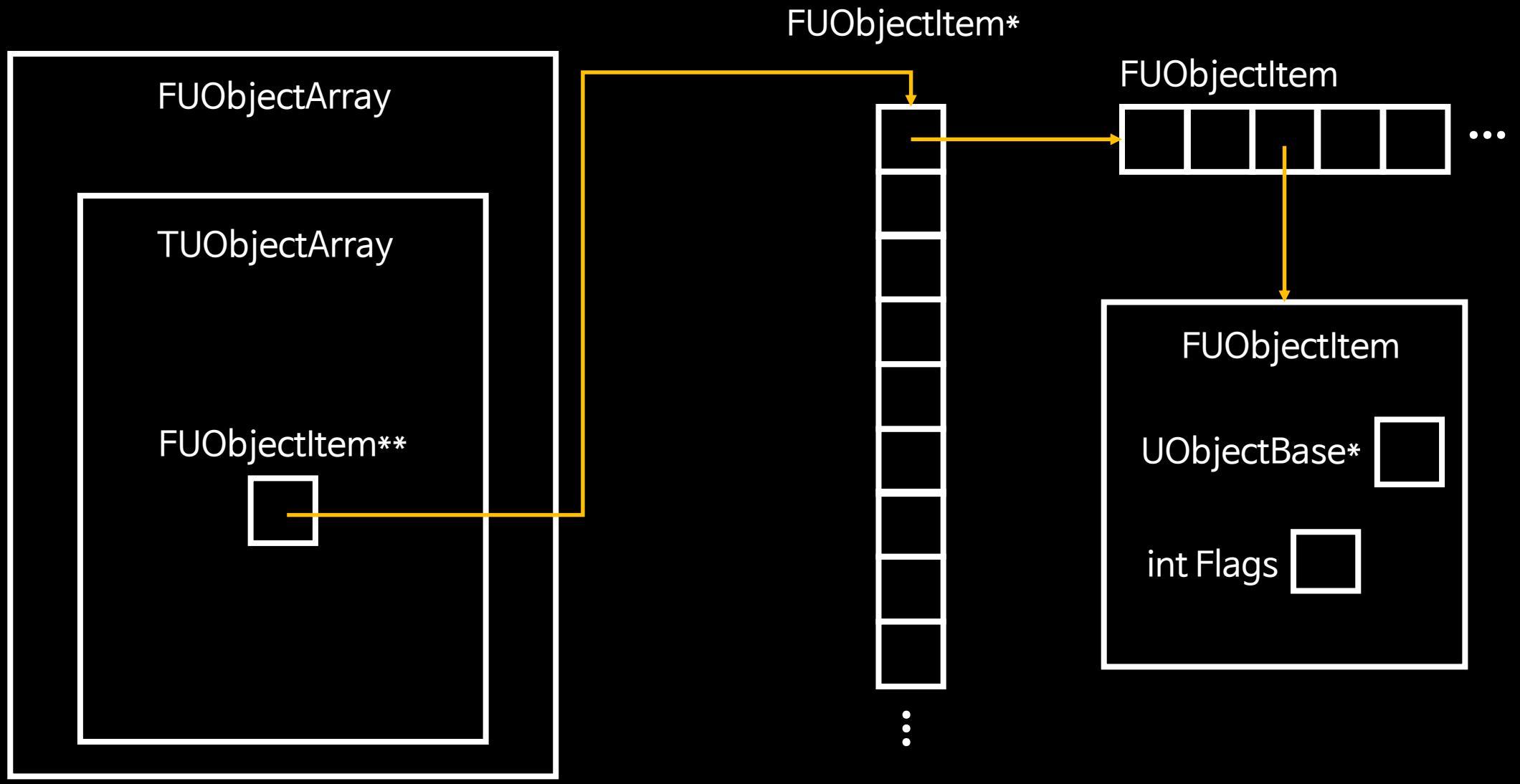
UObjectArray.h

오브젝트 구성

FUObjectItem** Objects가 할당되는 과정

```
void PreAllocate(int32 InMaxElements, bool bPreAllocateChunks) TSAN_SAFE
{
    check(!Objects);
    MaxChunks = InMaxElements / NumElementsPerChunk + 1;
    MaxElements = MaxChunks * NumElementsPerChunk;
    Objects = new FUObjectItem*[MaxChunks];
    FMemory::Memzero(Objects, sizeof(FUObjectItem*) * MaxChunks);
    if (bPreAllocateChunks)
    {
        // Fully allocate all chunks as contiguous memory
        PreAllocatedObjects = new FUObjectItem[MaxElements];
        for (int32 ChunkIndex = 0; ChunkIndex < MaxChunks; ++ChunkIndex)
        {
            Objects[ChunkIndex] = PreAllocatedObjects + ChunkIndex * NumElementsPerChunk;
        }
        NumChunks = MaxChunks;
    }
}
```

오브젝트 구성


FUObjectItem*
FUObjectArray
TUObjectArray
FUObjectItem**
FUObjectItem
FUObjectItem
UObjectBase*
int Flags

오브젝트 구성

FUObjectItem의 Flags enum 값

```
/** Objects flags for internal use (GC, low level UObject code) */
enum class EInternalObjectFlags : int32
{
    None = 0,
    //~ All the other bits are reserved, DO NOT ADD NEW FLAGS HERE!

    ReachableInCluster = 1 << 23, ///< External reference to object in cluster exists
    ClusterRoot = 1 << 24, ///< Root of a cluster
    Native = 1 << 25, ///< Native (UClass only).
    Async = 1 << 26, ///< Object exists only on a different thread than the game thread.
    AsyncLoading = 1 << 27, ///< Object is being asynchronously loaded.
    Unreachable = 1 << 28, ///< Object is not reachable on the object graph.
    PendingKill = 1 << 29, ///< Objects that are pending destruction (invalid for gameplay but valid objects)
    RootSet = 1 << 30, ///< Object will not be garbage collected, even if unreferenced.
    //~ UnusedFlag = 1 << 31,

    GarbageCollectionKeepFlags = Native | Async | AsyncLoading,

    //~ Make sure this is up to date!
    AllFlags = ReachableInCluster | ClusterRoot | Native | Async | AsyncLoading | Unreachable | PendingKill | RootSet
};
ENUM_CLASS_FLAGS(EInternalObjectFlags);
```

Cluster

- 위 구성대로라면 GC가 동작할 때마다 모든 오브젝트를 탐색해야 한다
- 이걸 Cluster 단위로 나눠 모든 오브젝트가 아닌 Cluster 단위로 탐색하게 하기 위해 만든 것 같다… (확실하지 않음)
- 적절한 Cluster 분배는 GC 성능 향상을 기대할 수 있다

조건

- GC를 활용하려면 오브젝트를 선언할 때 반드시 마크업을 추가해야 한다
  Ex) UClass(…)

- UE4의 GC는 리플렉션 시스템에 의해 동작하는데 마크업을 추가해야 오브젝트의 리플렉션 데이터를 생성하기 때문

- 클래스는 반드시 UObject의 자식 클래스여야 한다

```
UCLASS()
class MyGCType : public UObject
{
    GENERATED_BODY()
};
```

알고리즘

- 전체 메모리 탐색은

```
// Global UObject array instance
FUObjectArray GUObjectArray;
```

여기에서 이루어짐

- Flag가 RootSet인 경우 무조건 GC에서 제외
- 오브젝트가 삭제 대기 중(IsPendingKill 함수)이면 Flag를 Unreachable로 변경
- ClusterRoot가 삭제 대기 중이면 해당 클러스터의 오브젝트들과 참조하는 클러스터 오브젝트들의 Flag를 Unreachable로 변경
- GC가 작동하면 Flag가 Unreachable인 오브젝트의 메모리를 해제
- 한 번에 모든 오브젝트를 탐색하는 것 같다… 전 범위를 스레드로 병렬 분산 처리 하고 있었음

참고

- 위 알고리즘은 GarbageCollection.cpp의 MarkObjectsAsUnreachable 함수 IncrementalPurgeGarbage 함수를 참고하였음

- 사실 코드만 봐서 실제로 두 함수가 동작하는지는 모르겠다…

참고

- Tracing garbage collection / 위키피디아(영문)
https://en.wikipedia.org/wiki/Tracing_garbage_collection#Weak_collections

- 쓰레기 수집 / 위키피디아(한글)
https://ko.wikipedia.org/wiki/%EC%93%B0%EB%A0%88%EA%B8%B0_%EC%88%98%EC%A7%91_(%EC%BB%B4%ED%93%A8%ED%84%B0_%EA%B3%BC%ED%95%99)

- [JVM] Garbage Collection Algorithms
https://medium.com/@joongwon/jvm-garbage-collection-algorithms-3869b7b0aa6f

- 쓰레기 수집 / 나무위키
https://namu.wiki/w/%EC%93%B0%EB%A0%88%EA%B8%B0%20%EC%88%98%EC%A7%91

- Reference Counting과 Mark and Sweep
https://medium.com/@leeyh0216/reference-counting%EA%B3%BC-mark-and-sweep-2d046f73da4f

참고

- UE4 C++ 프로그래밍 입문 / 언리얼 문서
https://docs.unrealengine.com/ko/Programming/Introduction/index.html

- Garbage Collection & Dynamic Memory Allocation / 언리얼 위키
- https://wiki.unrealengine.com/Garbage_Collection_%26_Dynamic_Memory_Allocation

- [UE4] Garbage Collection Overview / egloos
http://egloos.zum.com/sweeper/v/3205731

- 언리얼 오브젝트 처리 / 언리얼 문서
https://docs.unrealengine.com/ko/Programming/UnrealArchitecture/Objects/Optimizations/index.html

- Uobject 인스턴스 생성 / 언리얼 문서
https://docs.unrealengine.com/ko/Programming/UnrealArchitecture/Objects/Creation/index.html

Q/A