Malloc Lab

20210774 김주은

< Lab 시간에 배운 내용 >

malloc 함수

* Size 만큼 할당하고, 할당한 주소를 리턴하는 함수

Free 함수

* 포인터가 가리키는 곳의 메모리 space를 할당 해제시키는 함수

Malloc design

* Execution speed와 memory space consumption이 중요하다. Execution speed의 경우 free memory block을 찾는 것과 memory block을 releasing하는 것에 달려 있고, memory space consumption의 경우 internal fragmentation과 external fragmentation이 결정을 하고 data structure overhead도 영향을 준다.

Implicit free list

* Speed의 경우 malloc은 linear search이기 때문에 memory block수 만큼 걸린다. O(# memory blocks).
* Free의 speed의 경우 constant time이 걸린다.
* Space 효율성을 봤을 때 next와 prev displacement때문에 2word의 overhead가 발생하며 first fit을 사용할 경우 external fragmentation이 발생한다.

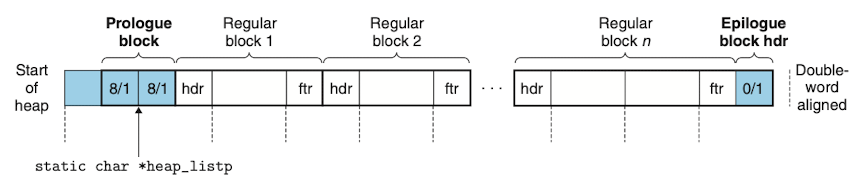
<malloc lab 코드 구현 과정 및 설명>

기본 상수 및 매크로 정의 by 교과서

텍스트, 스크린샷, 화면, 은색이(가) 표시된 사진

자동 생성된 설명

mm\_init 함수를 통해 힙을 초기화한다.



텍스트, 모니터, 화면, 스크린샷이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

Extend\_heap 선언 및 함수 정의

Extend\_heap 함수는 두가지 다른 경우에 호출되는데, 힙이 초기화될 대와 mm\_malloc이 적당한 fit을 찾지 못했을 때이다. 먼저, extend\_heap은 요청한 크기를 인접 더블워드의 배수로 반올림 해야하기에, 아래와 같은 코드를 작성한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이후에, 메모리 시스템으로부터 size만큼의 추가적인 힙 공간을 요청하는 것이므로, 아래와 같이 코드를 작성한다.

텍스트, 화면, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Mem\_sbrk를 호출하고 bp에 저장된 리턴값은 epilogue block의 헤더에 곧이어서 더블 워드 정렬된 메모리 블록의 주소이다. 그러므로, 이 블록을 새로운 free block으로 초기화하며 새로운 free block의 header, footer을 초기화하고 그 다음 블록으로 epilogue header을 새로 초기화한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이후에, 주변의 Free block과 합치기 위해 coalesce 함수를 호출하고 통합된 블록의 포인터를 리턴한다.

여기서 coalesce 함수를 호출하고 사용하기 위해 이를 정의할 필요가 있으므로 이를 아까의 extend\_heap 함수 아래 위치에 선언해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이제 coalesce 함수를 정의해주는데, 먼저 coalesce를 하기 위해 이전 블록과 다음 블록이 allocated인지, free인지를 알 필요가 있으며 현재 블록의 size도 알 필요가 있으므로, 처음에 정의한 GET\_ALLOC과 GET\_SIZE 두 개의 매트로를 사용하여 구한 후 각 변수에 저장한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그리고 coalesce는 총 4가지의 경우에 대해 다르게 동작한다.

첫번째 경우는 이전 블록과 다음 블록이 모두 allocated된 경우는 통합되는 동작이 일어나지 않고 바로 블록 포인터를 반환한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이후 나머지 3 경우에 대해서도 따로 조건문을 통해 동작하도록 한다. 통합될 블록에 해당되는 size를 계산하는 것을 모든 경우에 대해 적는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이제 각 계산된 size값들을 각 통합된 블록의 header와 footer에 기록한다. 그리고 이때, 이전 블록과 합치는 경우에는 block pointer를 이전 블록의 포인터로 업데이트 시켜주어야 한다. 다 작성한 코드는 아래와 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이로써 coalesce 함수가 완성되었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Extend\_heap의 마지막 부분에 coalesce(bp)를 호출하고 리턴값을 함수의 리턴값으로 작성할 수 있다.

이제 mm\_init 함수를 완성했으므로, mm\_malloc 함수를 구현한다.



먼저, size가 0으로 들어온 경우는 더이상 할당 관련 작업이 필요가 없으므로 예외처리를 해주며 NULL을 리턴하게 한다.

이제 할당하기 전에 alignment를 해주어야 하므로 위에서 선언했던 매크로 ALIGN을 사용하여 size를 alignment한다. 헤더와 푸터 오버헤드를 위해 DSIZE만큼을 더해서 alignment를 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이제, 할당할 size를 계산했으므로, 이에 적합한 Free block을 찾을 것이다.

텍스트, 스크린샷, 화면이(가) 표시된 사진

자동 생성된 설명

위의 코드 구성은 교과서 코드를 참조했으며 위의 흐름으로 free block을 찾고 알맞은 free block에 블록을 할당하려고 한다.

먼저, find\_fit함수를 호출하여 free block을 찾고, 적합한 것을 찾았을 시 asize의 크기의 블록으로 할당한다. 그러나 find\_fit 함수의 리턴값으로 NULL을 리턴할 시, free block을 찾은 것이 아니므로 extend\_heap을 호출하여 heap영역을 확장하고 블록을 할당한다.

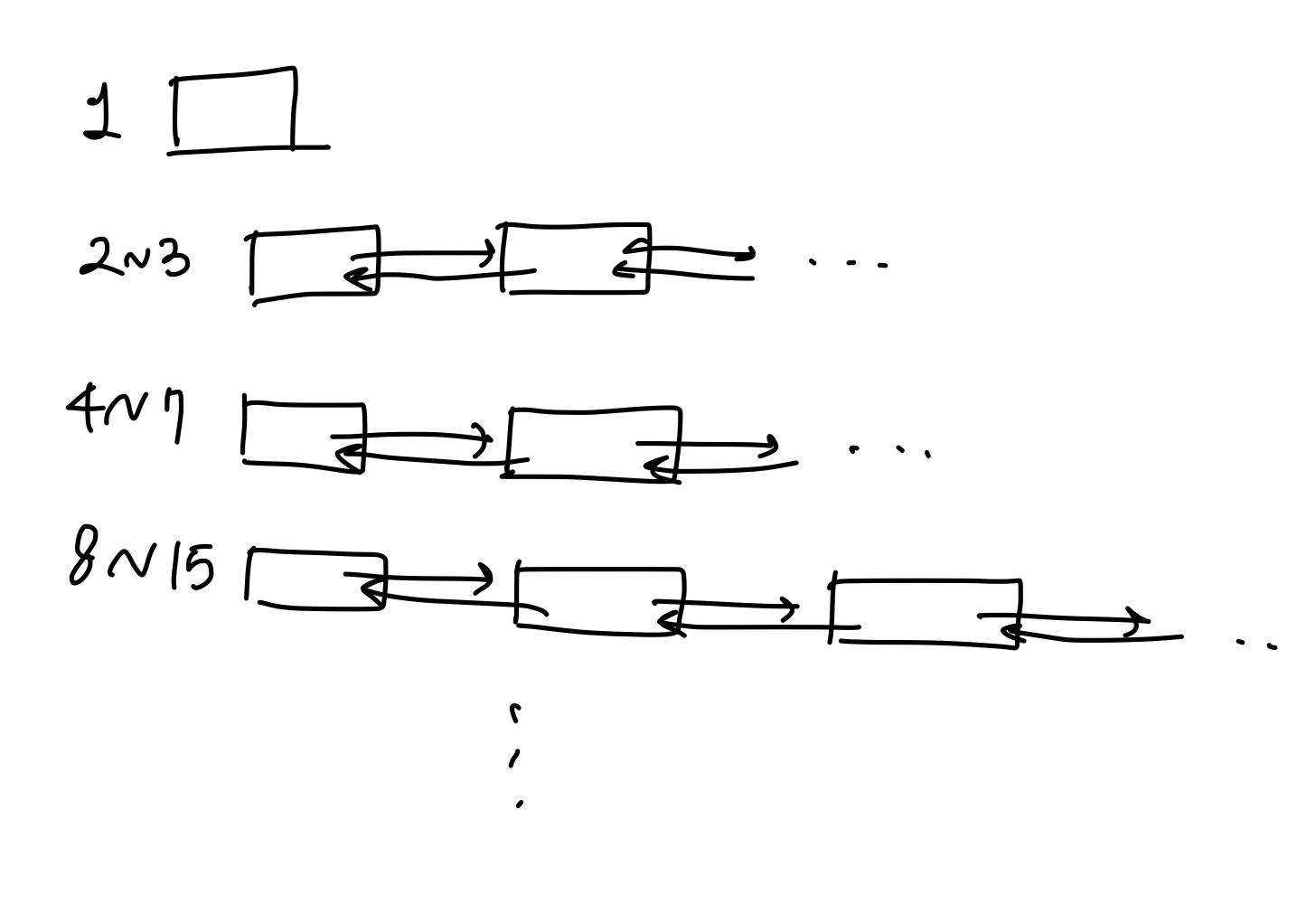
이제 위 코드에 필요한 find\_fit 함수와 place 함수를 정의하고자 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Find\_fit 함수 구현을 위해서는 free list를 어떻게 관리할지 고려해야 한다. 교과서에는 implicit free list로 구현이 되어 있으나 이는 list에 allocated list 와 free list가 모두 들어있기에 비효율적이며 시간이 오래 걸린다. 오직 free list들을 크기로 분류하여 다중 리스트를 통해 블록들을 저장하는 “segregation free list”로 구현 해야 겠다는 판단을 내렸다.

Segregation free list의 경우 list들이 여러 개 있는 것으로 2차원 배열과 같은 형태로 생각할 수 있다.



이렇게 크기로 분류하여 각 종류 별로 리스트가 linked list로 구현되는 형식을 segregation free list라고 한다. 그리고 이를 크기 별로 분류하여 리스트를 구성하고, 리스트 내부에서도 크기 순으로 정렬하여 블록을 할당하고 반환하는 것이 빠르게 되도록 할 수 있다. 또한 교과서에 따르면 이러한 블록 할당 및 반환이 상수 시간 연산이라고 한다. 리스트에 연결되는 블록의 경우 allocated block은 포함되지 않고 오직 free block들만 연결하기 때문에 할당 시간이 더 짧아지는 것이며 리스트에 대한 할당과 반환 연산에서 필요한 포인터들도 추가로 만들어줘야 한다고 생각하게 되었다.

그리고 이제 다시 find fit을 구현하려고 했을 때, find\_fit 함수 뿐만 아니라 모든 함수들에서 segregation free list를 구현하기 위한 코드들을 추가하고 수정해야 한다는 것을 깨달았다.

그래서 일단 먼저, segregation free list을 선언한다. 이는 모든 함수에서 쓰일 예정이기 때문에 지역변수가 아닌 global 변수로 seg\_free\_list를 선언한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이는 mm\_init함수에서 초기화 될 필요가 있으므로, 다시 해당 함수로 돌아가 코드를 추가해준다. 그리고 2차원배열처럼 선언하기 위해서 총 분리된 리스트가 몇개가 필요한지 maximum number이 필요하다. num\_list를 20으로 선언해둔다.

텍스트이(가) 표시된 사진

자동 생성된 설명

mm\_init 함수에서 먼저 mem\_sbrk 함수를 호출하여 num\_list 크기의 list들의 시작 주소를 가지고 있을 만큼의 메모리 영역을 확보한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

말그대로 2차원 배열이므로 다음과 같이 index를 하나씩 증가해가며 각 리스트를 NULL로 초기화해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위의 코드가 복잡하므로 매크로로 정의해준다.

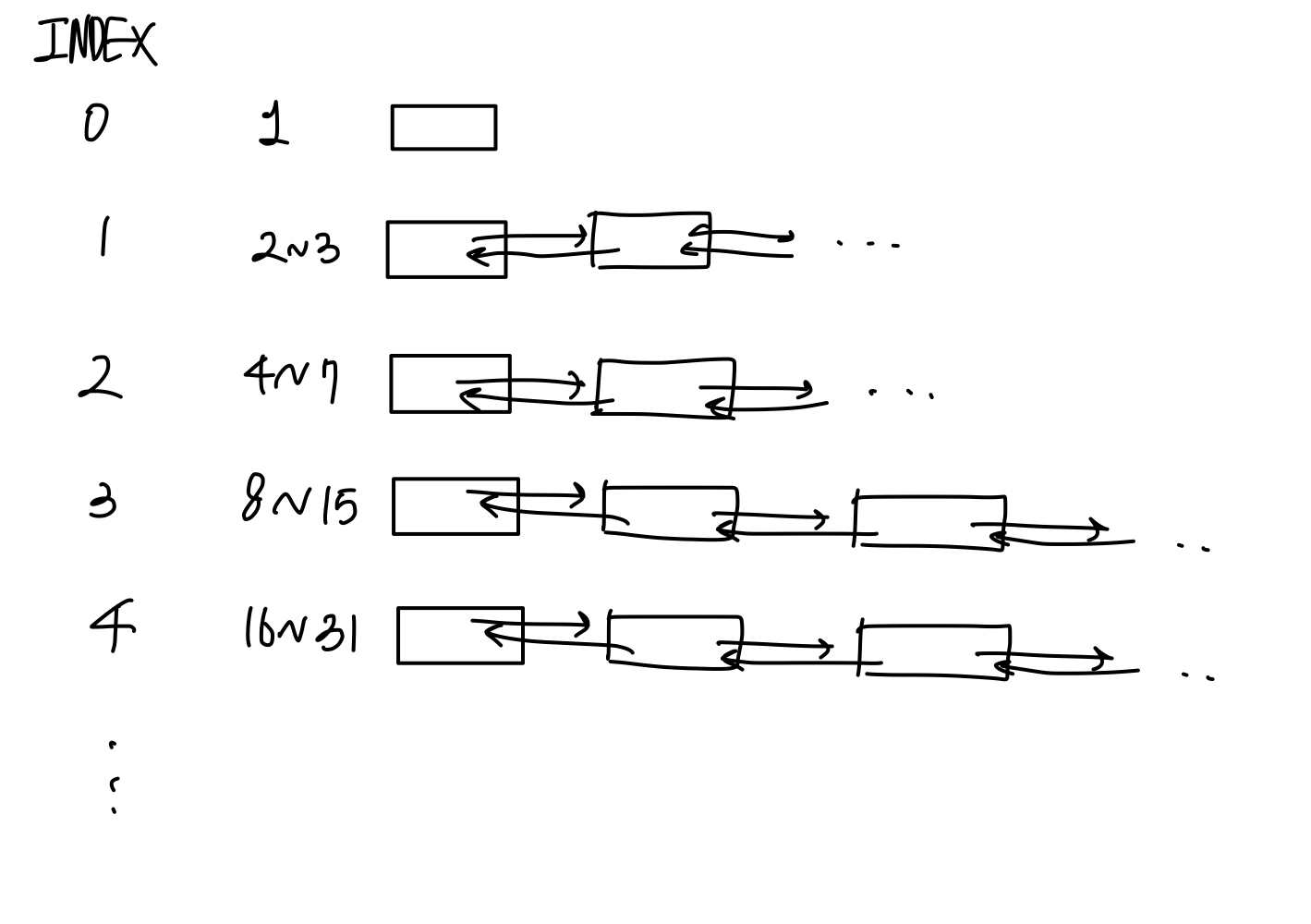
텍스트이(가) 표시된 사진

자동 생성된 설명

보통 int\*\* arr 을 선언한 후 int[3]을 하면 index가 3인 배열을 가리키는 것처럼 쓰기 위해 SEG\_INDEX\_LIST(i)의 형식으로 선언한다.

이제 find\_fit 을 다시 구현하고자 한다.

Find\_fit의 경우 먼저 블록 size에 해당되는 리스트를 먼저 찾고, 거기에 free block이 하나도 없는 경우가 있다면 더 큰 size list에 해당하는 리스트로 들어가서 찾아보고, 거기에 free block이 없다면 또 그 다음 size list에 해당하는 리스트에 들어가 찾는 것이 맞을 것이다.



위의 사진처럼 Index가 매겨진다고 하자. 여기서 말하는 index는 각 list 별로의 Index를 말한다. 즉 segregation free list의 경우 list가 여러 개이므로 그 각 각 list의 순서를 말한다.

여기서 이를 수식으로 나타내면 log값이 될 것이며 로그값을 취한 후 버림을 한 값이 될 것이다.

그래서 이를 구현하기 위해 find\_fit함수의 인자로 넘겨 받은 asize를 1이 될 때 까지 계속해서 나누고, 나눈 횟수를 index로 하는 것이다.

예를 들어, 8이 있다고 하자.

텍스트, 옷걸이이(가) 표시된 사진

자동 생성된 설명

총 3번의 나눗셈이 이루어지며 내가 구하고자 하는 list의 인덱스값도 정확하게 나온다. 그래서 반복문을 돌려가며 만들 것이다.

대략적인 슈도코드를 작성해보면,

텍스트이(가) 표시된 사진

자동 생성된 설명

이러하다.

즉, index를 0으로 초기화한 후에 NUM\_LIST까지 차근차근 올라가며 각 list를 test하는 것이다.

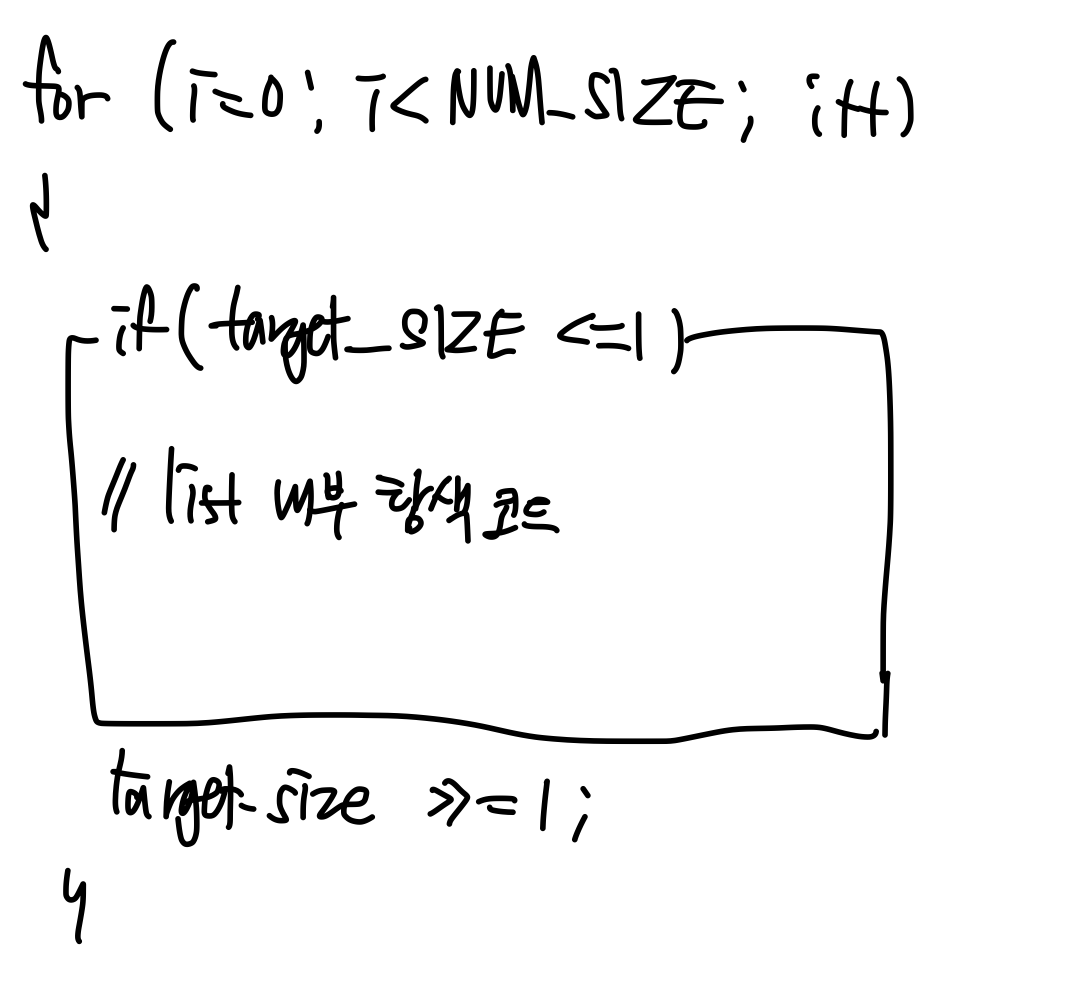
근데, 이제 list index의 값보다 적은 index인 경우에는 굳이 list 안까지 탐색해볼 필요가 없다. 그러므로, 이를 조건문으로 처리해주어야 할 것이다. List의 index값보다 크거나 같은 Index 부터는 list안까지 탐색해보며 찾자마자 block pointer을 리턴해야 할것이다.

먼저, 실질적으로 요구되는 asize값과 비교해야 하는 부문이 필요하기에 size를 임시적으로 저장하여 계속해서 2로 나누어줄 변수가 필요하므로 target\_size를 설정하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이제 target\_size가 결국 1이 될때 까지는 if 구문 안으로 들어가지 못하도록 막는다.



허나, 여기서 하나 중요한 포인트는 드디어 target\_size가 1이 되더라도 SEG\_INDEX\_LIST(i)가 NULL이면, 즉 그 Index에 해당하는 리스트에 free block이 없다면 list 내부를 탐색할 필요가 없다는 생각이 든다. 그러므로, if 조건문의 test부분에 SEG\_INDEX\_LIST(i)이 NULL인지를 체크하는 부분을 넣어준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이제는 if 문 안에 있는 블록에 어떻게 코드를 채워나가야할 것인지가 관건이다.

먼저, SEG\_INDEX\_LIST(i)의 시작주소를 bp라는 블록포인터에 대입을 하고, 여기서부터 while 반복문을 돌려 SEG\_NEXT(bp)로 다음 블록을 가리키도록 계속 업데이트하며 asize보다 블록사이즈가 더 큰 경우에는 반복문을 나오게 하여 적절한 블록을 찾게 하며, 이 블록이 NULL이 아닌 경우, 이 블록을 return하게 한다.

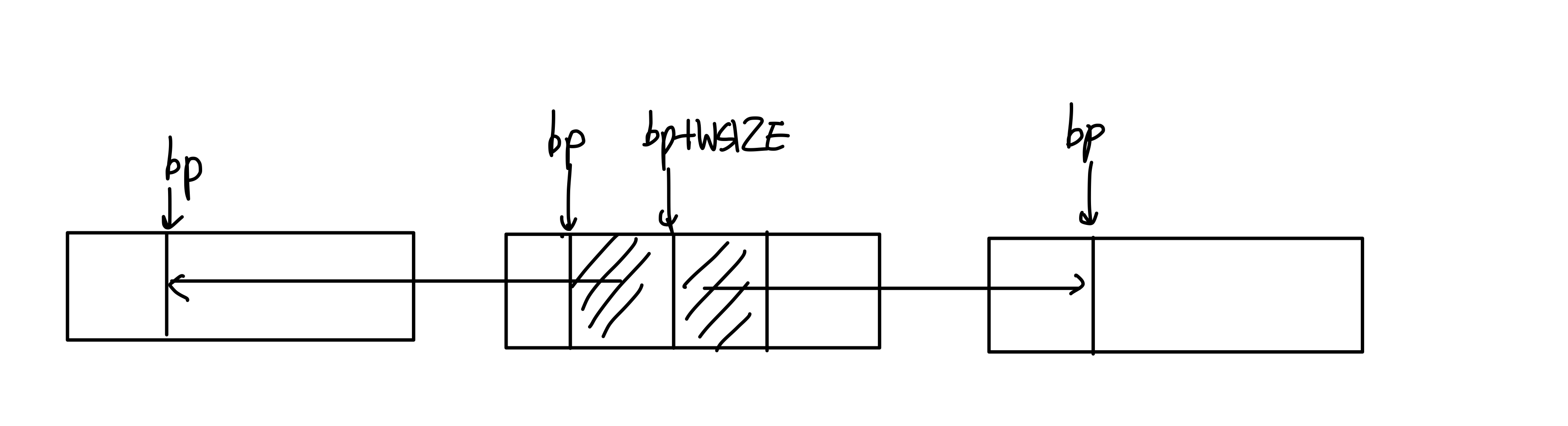
텍스트이(가) 표시된 사진

자동 생성된 설명

여기서 bp의 포인터를 리스트 내의 다음 포인터로 연결해줄 필요가 필요하다. 그래서 여기서 매크로를 하나 더 선언하여 다음 블록을 가리킬 수 있도록 한다.

나는 segregation free list를 double linked list로 구현하기에 이전과 다음 블록을 가리키는 포인터가 각 현재 블록의 payload 안에 저장된다.

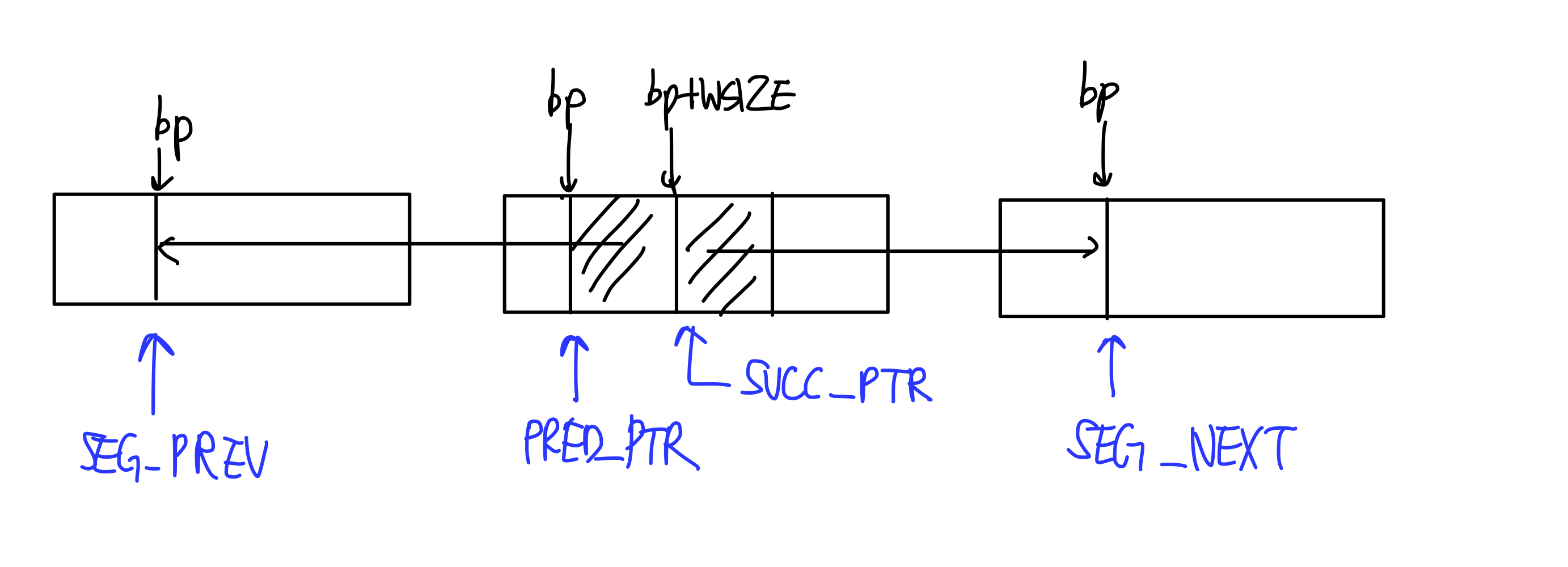
그러므로, 이러한 원리를 활용한 매크로를 선언할 것이다.



이렇게 링크드 리스트가 구현되며 하나의 블록에서 이전 블록과 다음 블록을 가리키는 각각의 포인터값이 블록안에 저장된다.

그러므로 이전블록을 가리키는 포인터를 담고 있는 곳을 가리키는 값, 다음 블록을 가리키는 포인터를 담고 있는 곳을 가리키는 값을 선언한다.

이후에는 이전블록을 가리키는 포인터, 다음 블록을 가리키는 포인터까지 선언한다.



아래에 첨부된 4개의 블록이 각각에 대응된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그래서 이제 SEG\_NEXT까지 사용하여 최종적으로 코드를 완성하면 아래와 같다.

텍스트, 스크린샷, 화면, 은색이(가) 표시된 사진

자동 생성된 설명

그리고, place를 짜야 한다. 여기서 드는 생각은 place를 짜기 위해 전체적으로 segregation free list의 insert와 delete하는 원리 구현이 우선이어야 할 것 같다.

사실, block을 insert하는 함수의 경우 원리 자체는 double linked list와 동일하기에 쉽게 구현할 수 있다고 생각한다. Block\_add함수라고 정의할 것이다.

추가할 free block의 포인터 값과 추가할 블록의 사이즈를 입력 받는 함수를 구현할 것이다.

그러므로, 다음과 같이 함수를 시작한다.



Find\_fit을 만들 때 사용하는 원리를 사용하면, 비슷하게 구현할 수 있을 거라는 생각이 들었다.

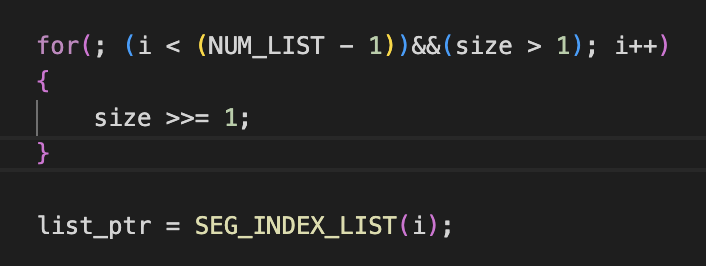
먼저, 알맞은 Index를 찾고 그 리스트에 들어가서 넣는 방식이다.

알맞은 index를 계산하는 것은 위에서 find\_fit을 구현할 때 사용했던 알고리즘을 사용하면 된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

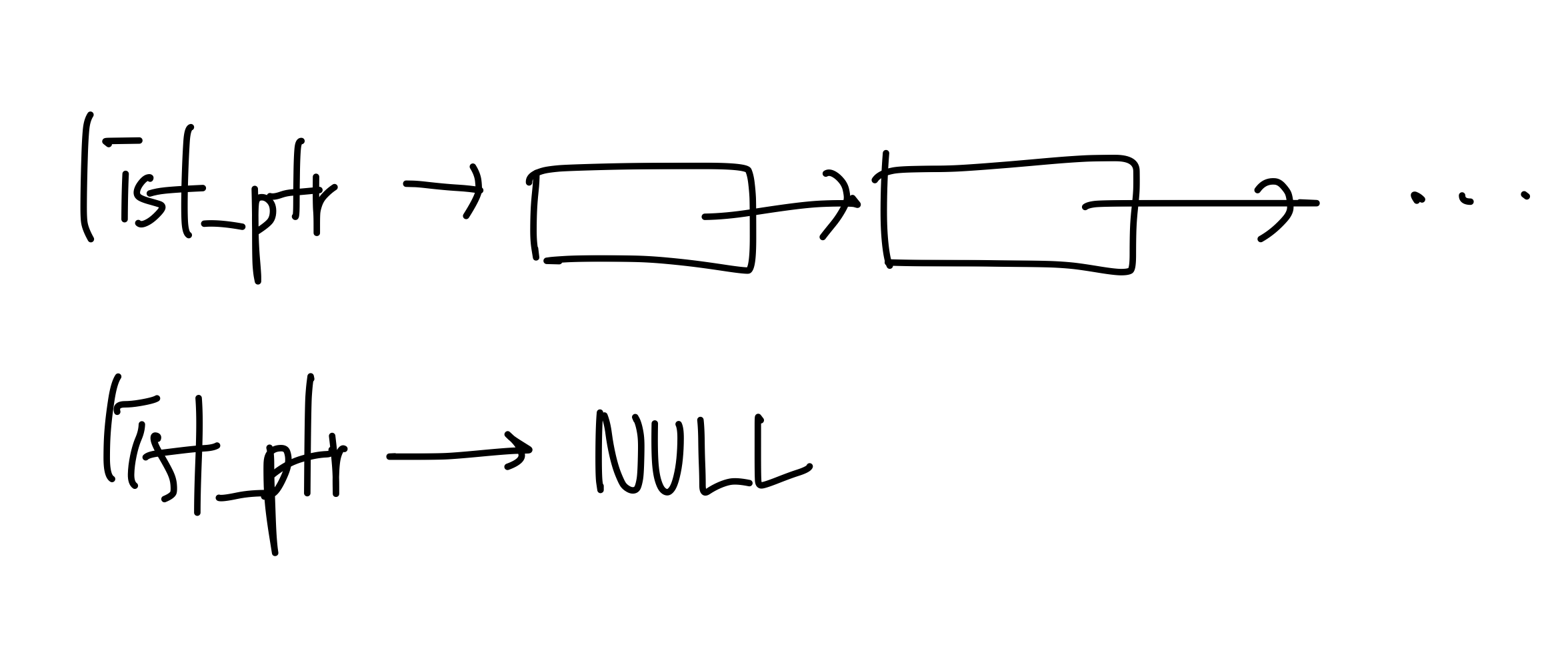
즉, NUM\_LIST -1 까지 반복문을 돌려가며 size가 1이 될 때까지 size를 2씩 나눠주는 방법이다.



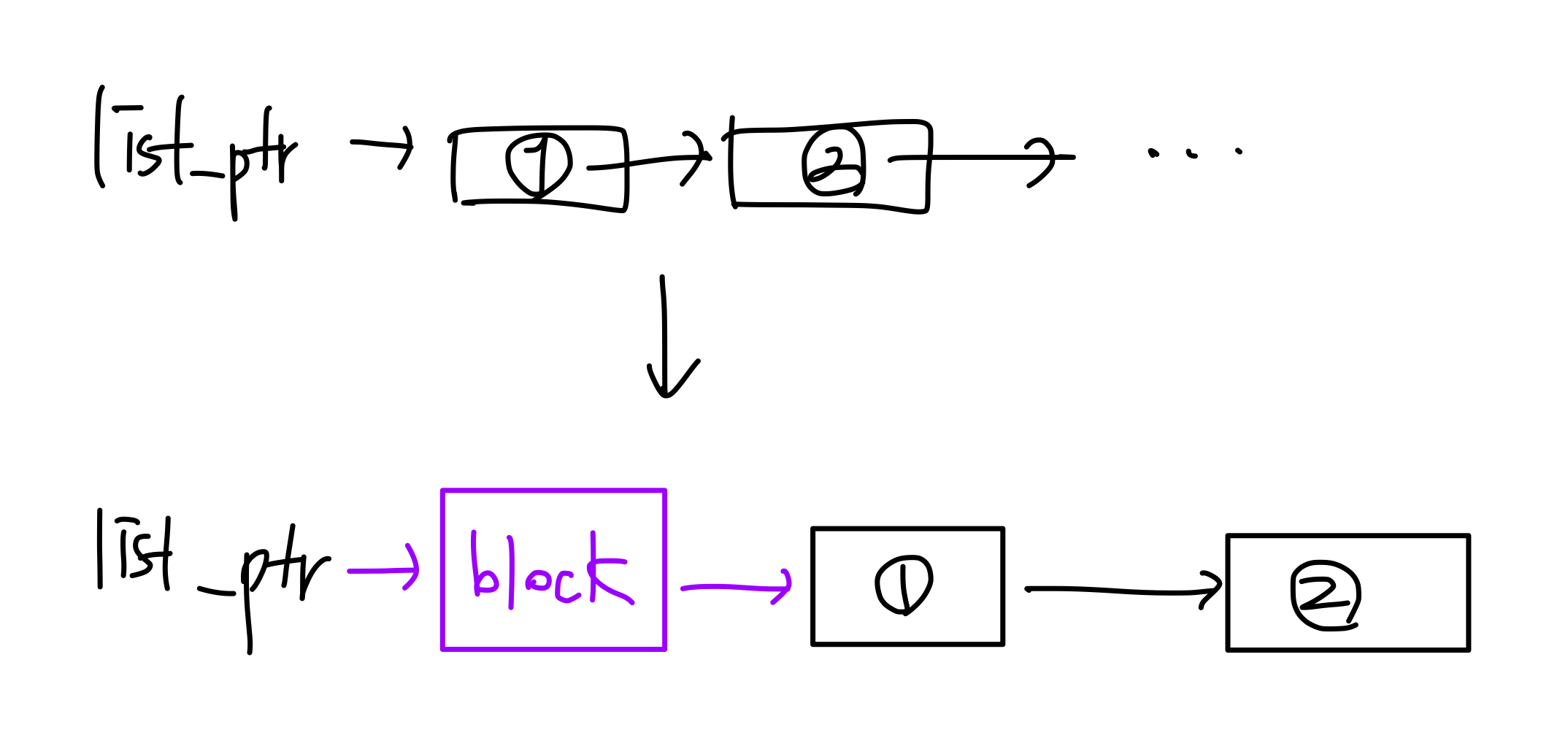
그리고 이제 index를 계산한 후 해당 리스트의 첫 주소를 가지고 블록을 넣을 것이다. 블록은 무조건 list의 가장 앞부분에 넣는 규칙을 세우려고 한다. Size를 하나씩 비교하여 list에 하나씩 넣기에는 너무 복잡하다는 판단을 내렸다.

이제 list의 가장 앞부분에 넣을 경우 두 가지 경우로 나뉠 수 있다. 그리고 그 두 가지 경우에 맞추어 블록을 삽입해야 한다.

리스트에 블록이 있는 경우, 리스트에 블록이 없는 경우이다.

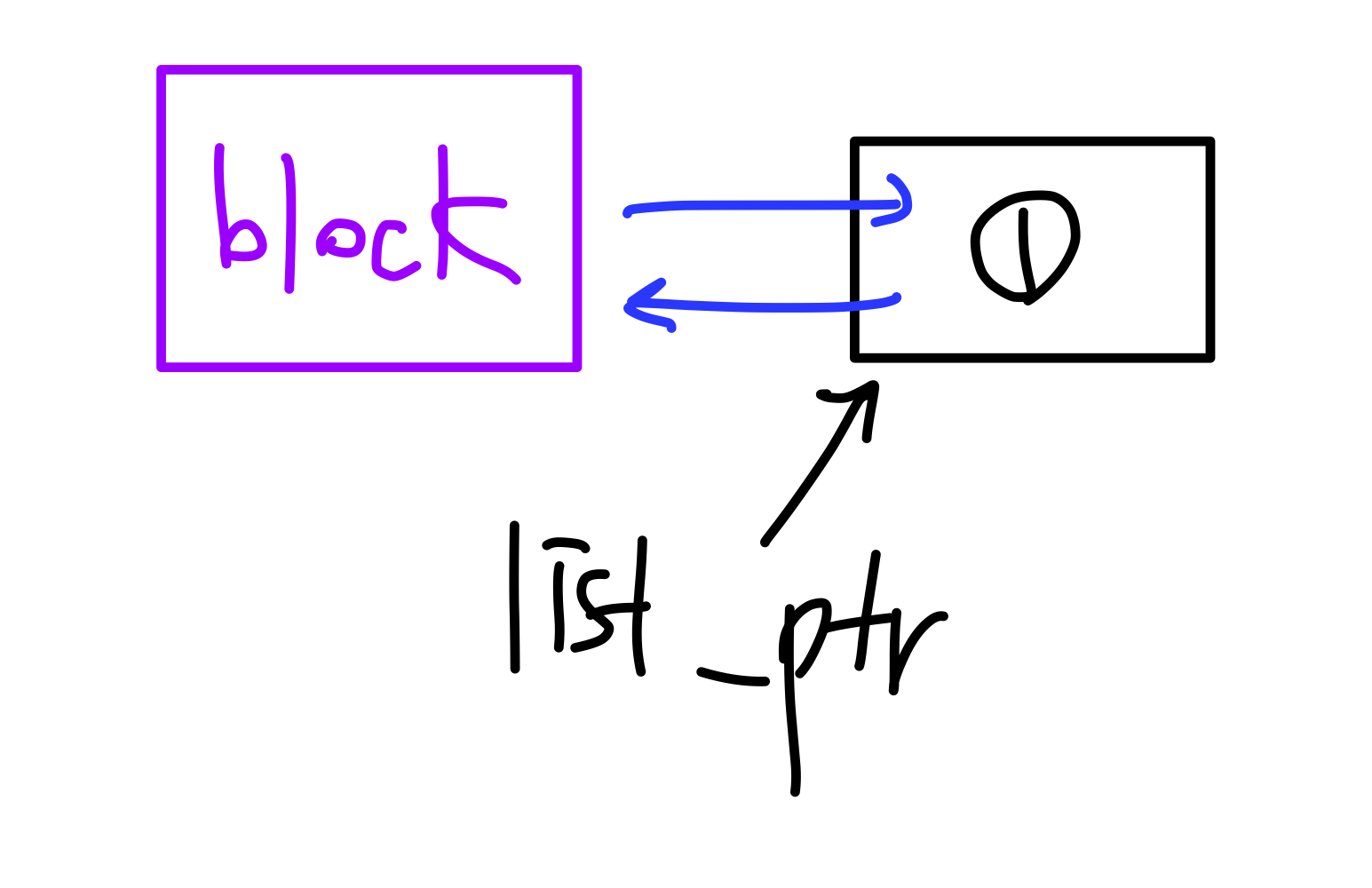


먼저, 리스트에 블록이 있는 경우다.

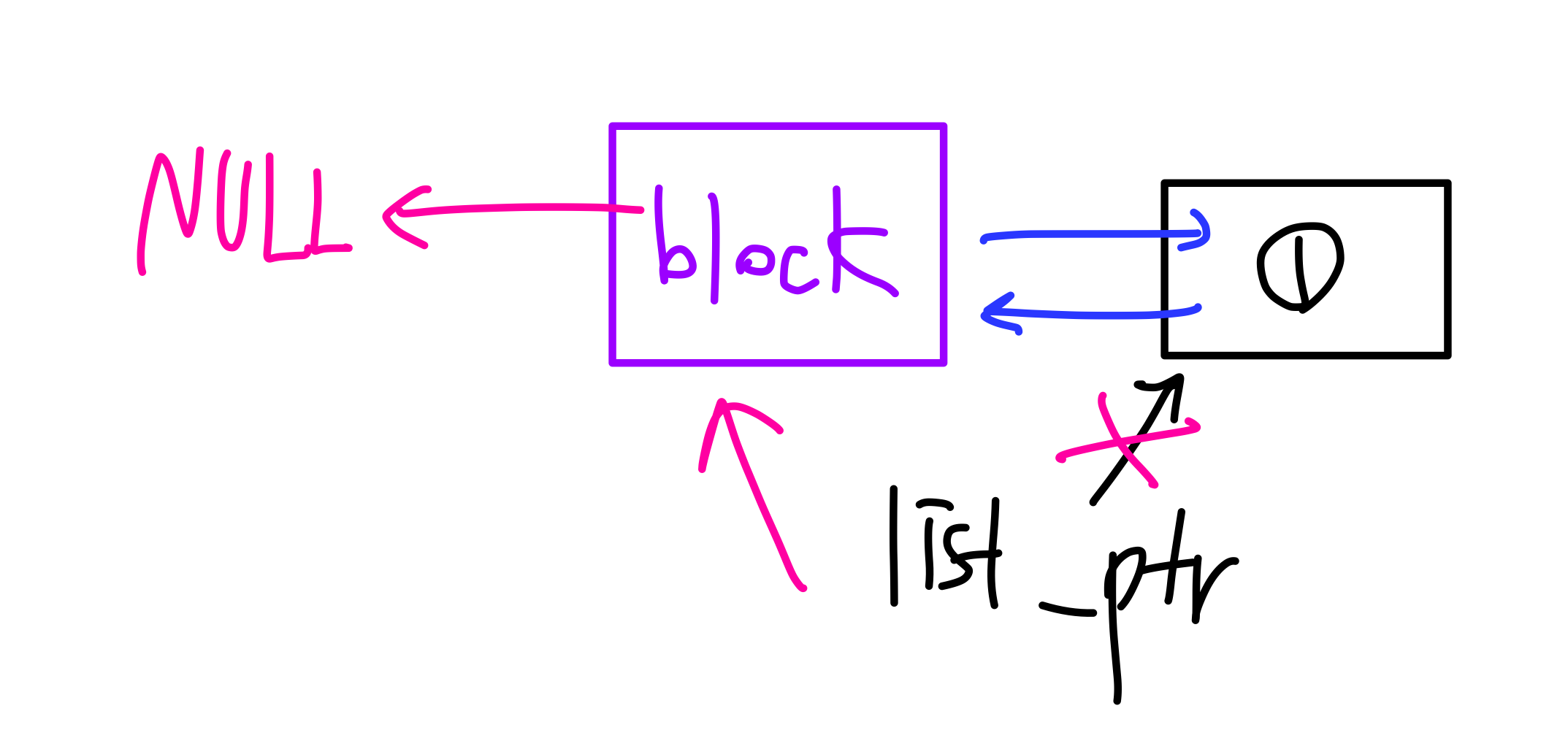


이렇게 넣으려고 한다.

하지만 double-linked-list이므로 block의 next pointer는 원래의 처음 블록을 가리키고, 원래의 처음 블록의 prev pointer은 지금 넣는 블록을 가리키도록 한다.

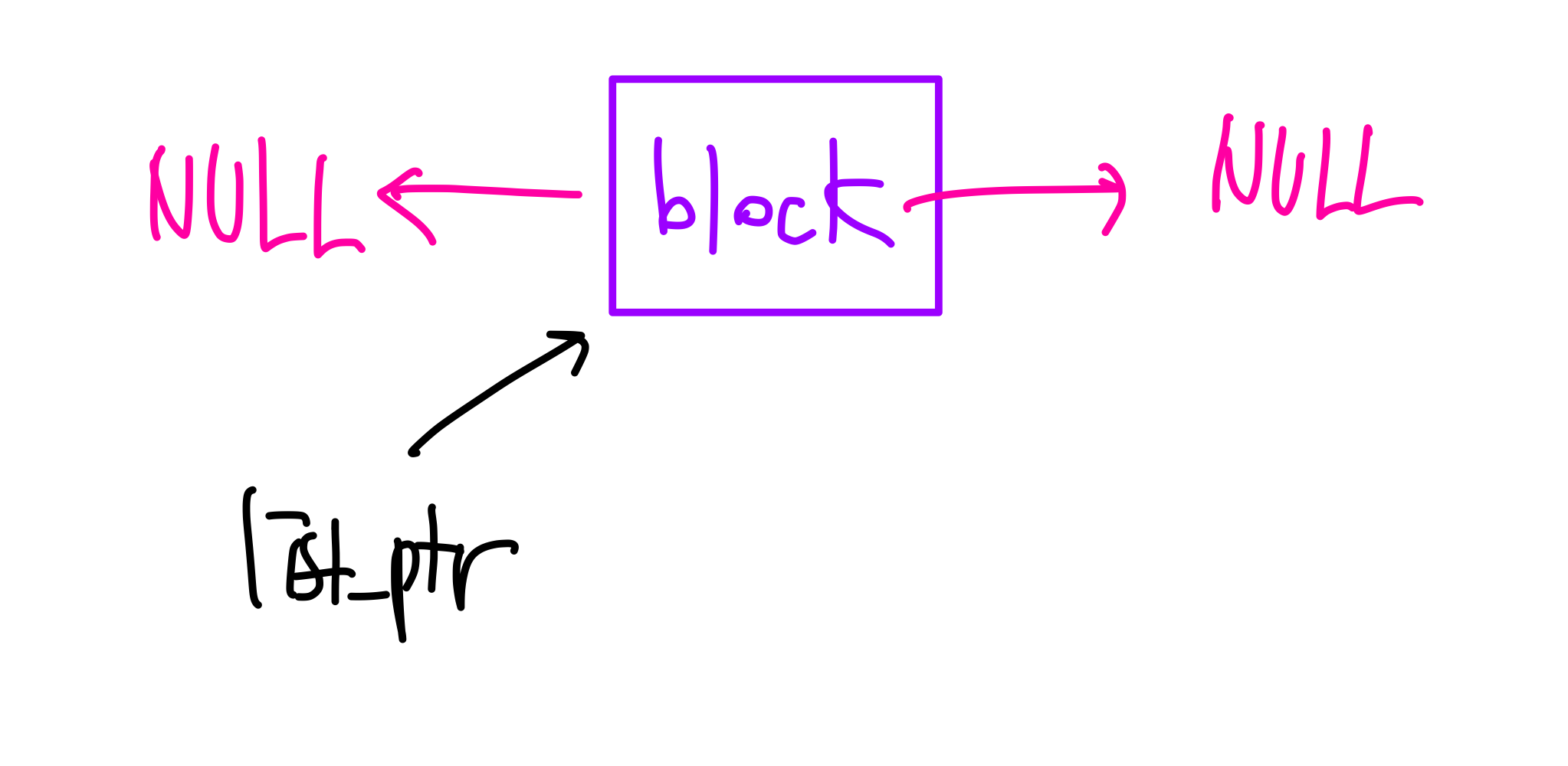


그리고 block의 이전 블록은 없으므로 prev pointer은 NULL로 초기화하고, seg\_index\_list(i) 즉, 리스트의 시작 주소에는 block의 주소로 업데이트한다.



다음으로는 리스트에 블록이 없는 경우이다.

이경우에는 굳이 block의 다음 포인터를 업데이트 시켜줄 필요가 없으며 그냥 NULL로 초기화한다.



이를 반영한 코드는 쉽게 구현할 수 있었다.

공통적인 부분이 있어서 공통적인 부분은 조건문 밑으로 따로 뺐다.

텍스트이(가) 표시된 사진

자동 생성된 설명

여기서 PUT\_PTR을 쓴 이유는 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

사실 블록안에 내용을 대입하는 것은 동일하다고 생각하여 매크로로 지정한 PUT을 사용했는데, 위와 같은 warning이 떴다. 즉, pointer에서 casting이 일어나는 것 같아서 이러한 warning을 안뜨게 하기 위해

텍스트이(가) 표시된 사진

자동 생성된 설명

위와 같이 포인터를 대입하는 매크로를 하나 더 지정하였다.

그 다음은 block\_delete 함수이다.

Block\_add와 마찬가지로 size를 통해 index값을 계산한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

블록을 삭제하는 것도 삽입하는 것과 마찬가지로 여러 경우로 나눌 필요가 있다.

총 4가지 경우로 나눌 수 있다.

1. 이전 블록과 다음 블록이 있는 경우

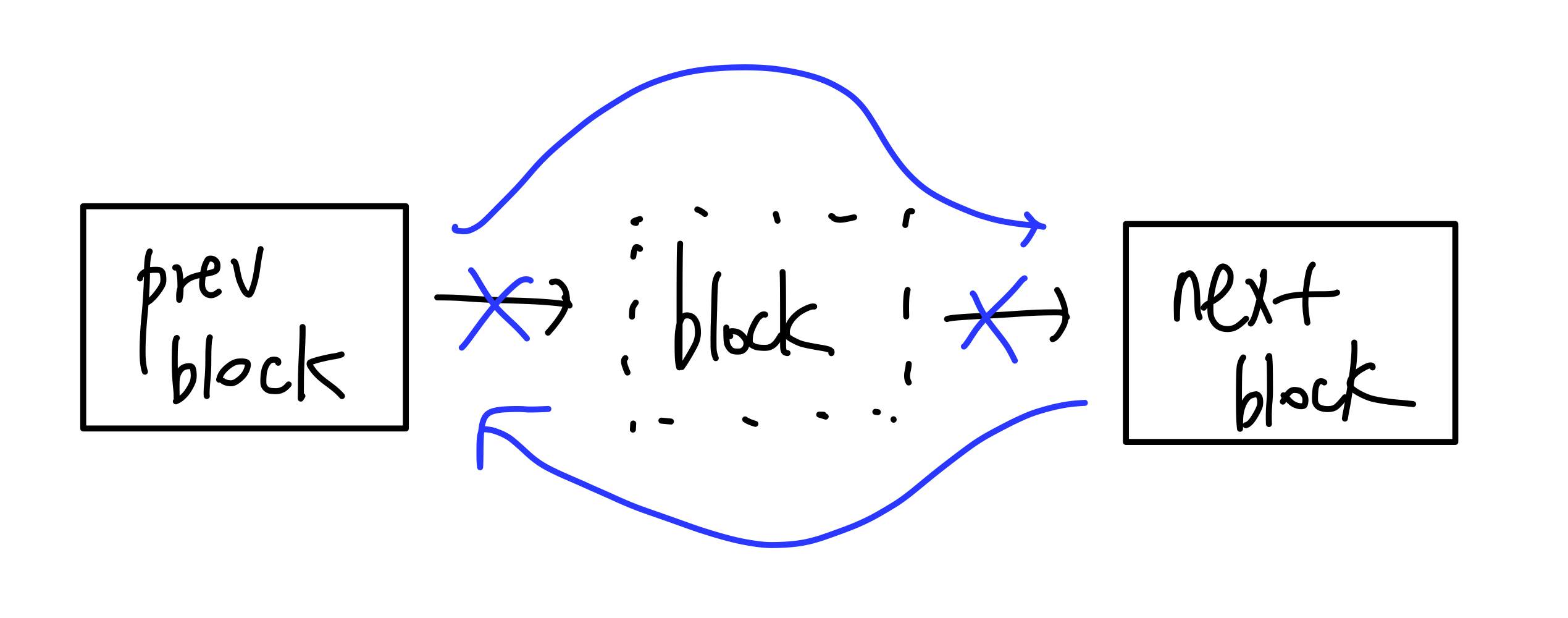
2. 다음 블록만 있는 경우

3. 이전 블록만 있는 경우

4. 이전 블록과 다음 블록이 없는 경우

먼저, 이전블록과 다음 블록이 있는 경우에는

이전블록의 next pointer을 다음 블록으로 두고 다음 블록의 prev pointer을 이전 블록으로 둔다.



다음으로 다음 블록만 있는 경우에는

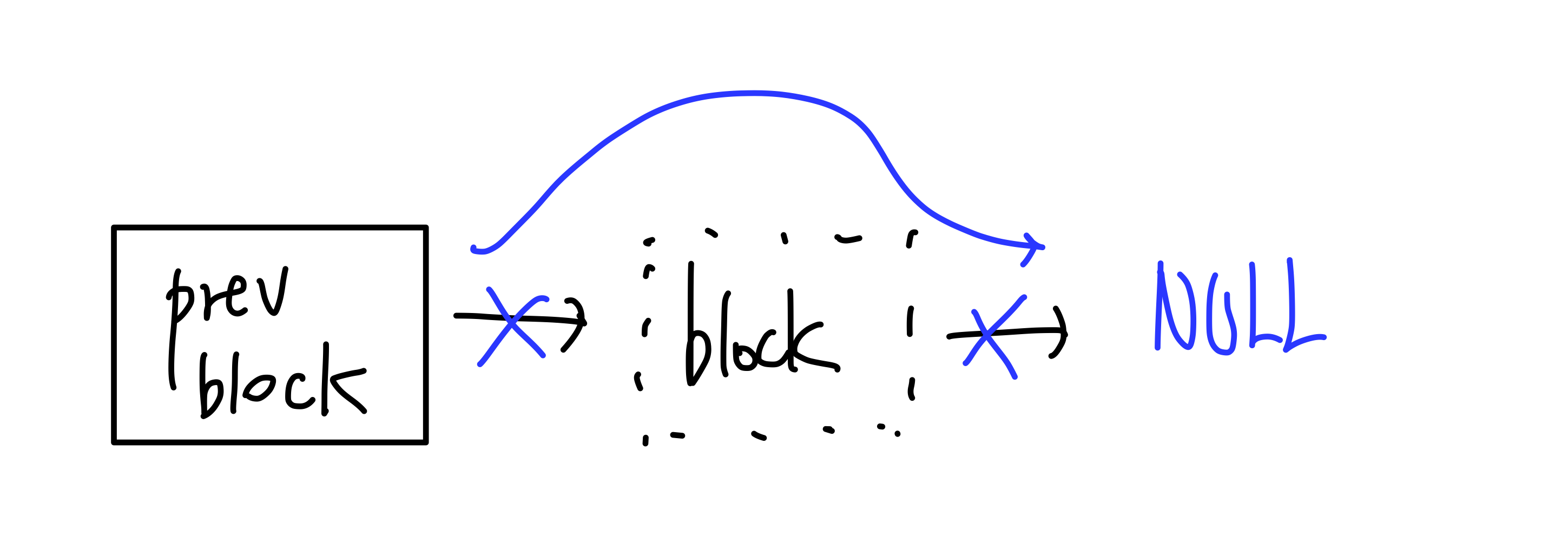
다음 블록의 prev pointer을 NULL로 설정하고, 리스트의 시작부분을 next\_block으로 처리해준다.

화이트보드이(가) 표시된 사진

자동 생성된 설명

다음으로 이전 블록만 있는 경우에는

이전 블록의 next pointer을 NULL로 설정한다.



둘다 없는 경우는

텍스트이(가) 표시된 사진

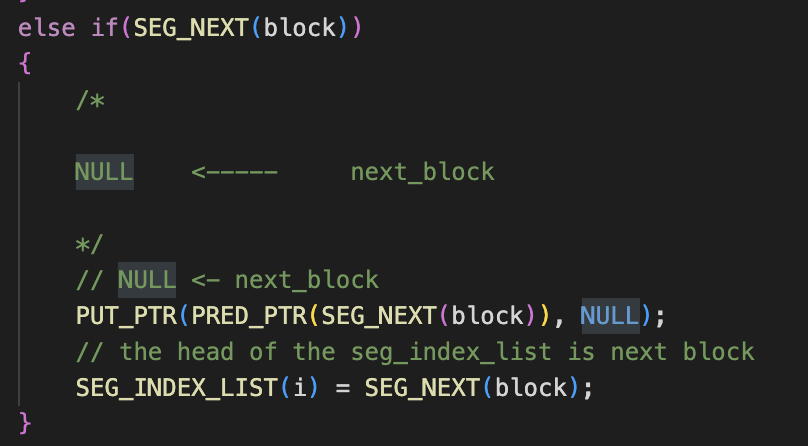
자동 생성된 설명

Seg\_index\_list 시작 부분을 NULL로 처리하여 리스트 하나를 삭제시킨다고 할 수 있다.

이 모든 경우를 구현한 코드가 바로 아래와 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명



텍스트, 스크린샷, 화면이(가) 표시된 사진

자동 생성된 설명

이제 insert와 delete 구현을 끝낸 리스트가 있으니, 이 원리를 적용하여 Place를 하는 함수를 구현해본다.

Place의 인자부분과 리턴 타입의 경우 교과서를 참조하였다.

이제 place를 어떻게 하냐가 관건이다.

Fragmentation을 고려하였을 때, 아주 큰 free 블록을 배정받았을 때 그 블록을 allocated block으로 바꾼다면, fragmentation이 아주 많이 발생할 것이라는 생각이 들었다.

그래서 두 가지의 경우로 나누어 구현하였다.

찾은 블록사이즈와 allocate하려는 사이즈를 비교하였을 때 그 size의 크기가 최소 2\*double word size라면 allocate하려는 사이즈만 allocate를 하고 나머지 부분은 free block으로 한다. 즉, block을 반으로 쪼개는 것이다.

여기서 allocate할 때는 put 매크로를 사용하여 size와 allocated인지 free인지에 대한 bit을 넘겨주는 것이다.

그리고 자르고 나온 뒷부분의 free block은 다시 segregation free list에 넣어준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

찾은 블록사이즈와 allocate하려는 사이즈를 비교했을 때 차이가 크지 않으면 그냥 allocate해준다.

이를 반영한 코드가 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

사실 여기서 거의 모든 함수가 구현되었지만 본래는 implicit list를 바탕으로 구현해놨던 부분의 함수들을 segregation free list에 맞게 수정할 필요가 있다.

그래서, 먼저 extend\_heap 함수의 경우에도 heap list을 새로 초기화 할때 segregation free list에 free block을 추가하는 코드가 필요하다.

텍스트이(가) 표시된 사진

자동 생성된 설명

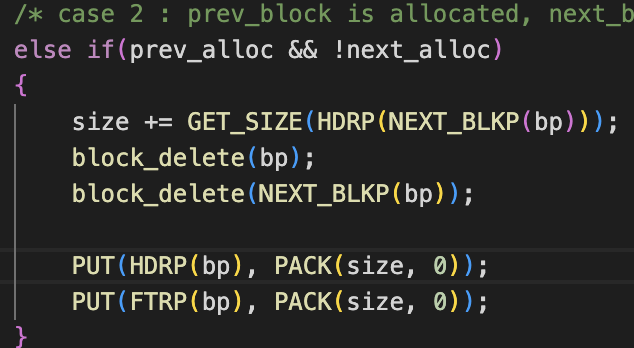
이와 같이 고쳤다.

그 다음 고칠 함수는 coalesce 함수이다.

Free block들을 연결했으므로 free b lock들이 업데이트가 되었기 때문에 이를 segregation free list에서 먼저 제외하고 다시 넣어줄 필요가 있다.

먼저, case2의 경우는 다음 블록과 연결한다.

그러므로, 현재 블록과 다음 블록을 segregation free list에서 삭제시킨다.



먼저, case 3의 경우는 이전 블록과 연결한다.

그러므로, 현재 블록과 이전 블록을 segregation free list에서 삭제시킨다.

텍스트이(가) 표시된 사진

자동 생성된 설명

마지막 case4 의경우에는 이전블록과 다음 블록 모두와 연결하므로 이전 블록, 현재 블록, 다음 블록을 모두 segregation free list에서 삭제시킨다.

텍스트, 검은색, 화면, 스크린샷이(가) 표시된 사진

자동 생성된 설명

그리고 모든 경우에 대해 새로 업데이트된 블록을 segregation free list에 추가해준다.



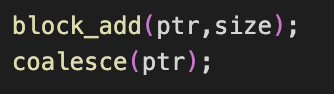
이렇게 구현한 상태에서 mm\_free 함수도 구현해야 한다.

먼저 지금 가리키고 있는 블록의 사이즈를 알아낸다음 다시 free bit인 0과 packing을 하여 ㅎㅔ더와 푸터에 기록한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그리고, segregation free list에 free block을 추가해주고, coalesce를 호출하여 주변에 free block이 있다면 합쳐준다.



그리고 일단 mm\_realloc은 원래 코드를 그대로 둔 상태에서 make를 하고 ./mdriver -V 명령어를 통해 테스트 해보았다. 이때 여러 차례의 segmentation 오류가 떴는데, 단순한 오타 문제가 많았기에 이를 해결해나가며 실행시킬 수 있었다.

결국 실행이 되었을 때 결과는 다음과 같았다.

텍스트이(가) 표시된 사진

자동 생성된 설명

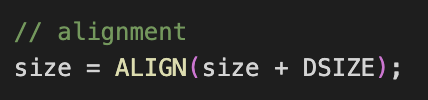
48점이 나왔는데, 여기서 성능을 올리기 위해 realloc 함수를 수정할 필요가 있다고 생각했다.

Pdf에 나와있는 대로 예외처리를 우선적으로 해주어야 한다고 생각했다. 그래서 먼저 ptr이 NULL일 때는 mm\_malloc(size)를 호출한다. 그리고 size 가 0일 때는 mm\_free(ptr)을 호출하고 NULL을 return한다.

텍스트이(가) 표시된 사진

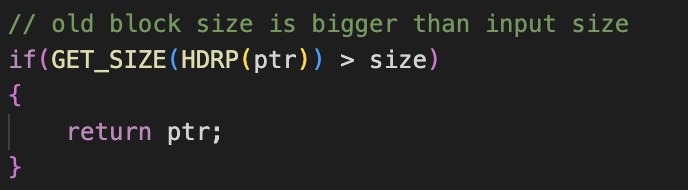
자동 생성된 설명

이제 함수에서 입력 받은 size인자에 대해 realloc 을 해주어야 하는데 이전에 미리 size에 대해 alignment를 한다.



Realloc 할 때는 다양한 경우가 발생할 수 있다.

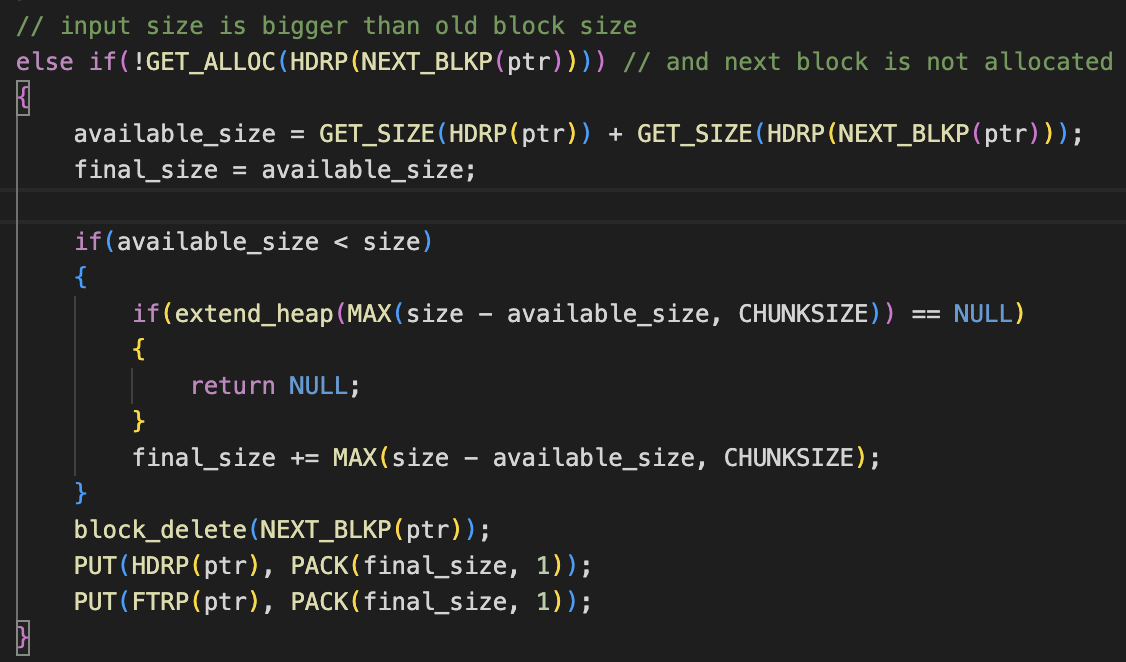
만약 현재 포인터가 가리키는 블록의 사이즈가 입력 받은 size보다 크면 그냥 return ptr을 가리킨다. 즉, 다시 재할당을 해줄 필요가 없기 때문이다.



그 다음 경우는 현재 가리키고 있는 블록의 사이즈가 입력 받은 size보다 작은 경우에는 다르게 처리해주어야 한다. 새로 입력 받은 크기에 대해 다시 할당해주어야 하기 때문이다. 그러나 만약에 지금 가리키고 있는 블록의 다음 블록이 할당되어 있지 않는다면, 즉 free block이라면 그 다음 블록까지 걸쳐가며 allocate하면 될 것이다.

그래서 현재 가리키고 있는 블록의 사이즈가 입력 받은 size보다 작은 경우를 다시 2가지 경우로 나눌 수 있을 것이다. 현재 가리키고 있는 블록의 다음 블록이 할당되어 있는 경우와 free되어 있는 경우이고, 이 경우들에 대해 각각 따로 처리해줘야 한다.

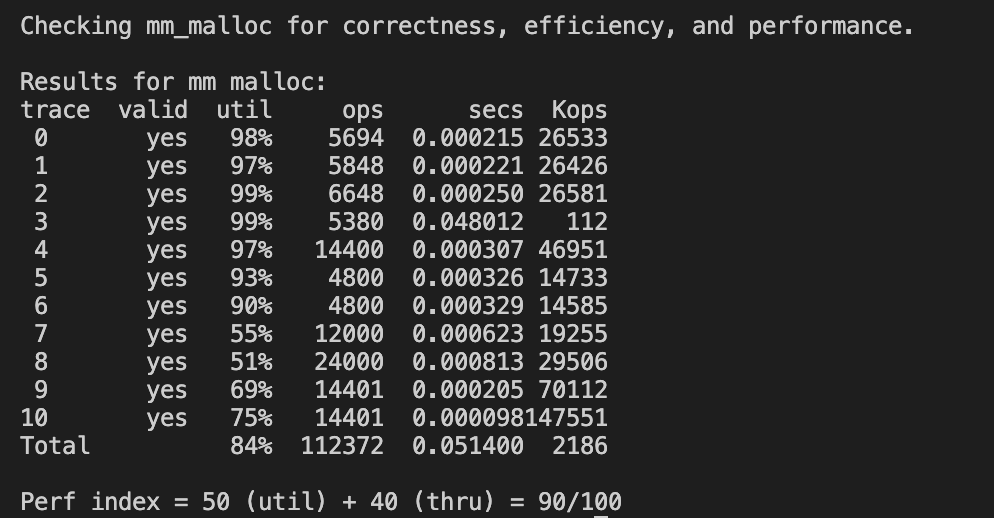
여기서 변수를 하나 사용하고 싶어서 available size를 설정했는데 이는 현재 블록과 다음 블록의 사이즈들을 더 한 것이다. Available size가 근데 입력 받은 size보다도 작을 수 있다. 그렇게 도면 extend\_heap을 불러서 free block 사이즈를 늘려줄 필요가 있다. 만약 extend\_heap을 호출하여 사이즈를 늘려주었다면 그 때는 final size라는 변수를 한개 더 사용하여 final size라는 최종 사이즈에 대입해준다. 그리고 이 final size를 활용하여 pack을 통해 헤더랑 푸터에 넣을 값을 계산하고 헤더와 푸터에 넣으며 next block의 경우는 free block list에서 없애주어야 한다.



이제 그 다음 경우로 next block이 allocated되어 있다면 free block이 아니므로 바로 같이 쓸 수 없다. 그러므로 새로운 곳에 할당하는 수 밖에 없을 것이다.



결국 realloc을 수정한 결과, 아래와 같이 90점이라는 성능이 나왔다.



이제 모든 malloc allocator을 구현하였으므로 mm\_check함수를 구현한다.

Pdf에서 체크를 요구하는 case를 모두 테스트하여 잘못된 부분이 있다면 에러를 출력하게 한다. 그리고 각각에 대해 체크해주는 함수를 하나씩 선언하고 정의해주기로 하였다. 아래는 선언한 함수들의 list이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그리고 아래와 같이 mm\_check 함수에서 함수들을 호출하여 리턴값에 따라 다르게 수행하도록 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

먼저, check mark free 함수의 경우는 free block들만 모아둔 모둔 segregation free list에 있는 free block들에서 header와 footer에 모두 0이라는 Free bit으로 이루어져있는지 체크해야한다. 즉, 여기서는 둘다 0이 아닌 모든 경우에 대해 에러 처리를 해주어야 한다.

이때, free list의 모든 free block에 대해 테스트를 해주어야 하므로 위에서 다른 함수들을 사용했을 때 search할 때 돌았던 loop처럼 2차원 배열로 가정을 하여 모든 Index에 걸치고, 모든 리스트에 걸쳐 반복하여 모든 free block에 대해 test를 해보면 된다.

즉, index를 담을 수 있는 변수와 블록을 테스트할 때마다 해당 블록을 가리킬 포인터변수를 먼저 만들어놓고 각 index별 리스트를 방문하여 그 리스트의 처음부터 끝까지 연결된 모든 블록들에 대해 GET\_ALLOC을 사용하여 allocated bit인지 free bit인지 확인하면 되는 것이다.

그리고 각 Test이후에는 SEG\_NEXT 등을 사용하여 블록 포인터를 업데이트 해주고 index도 업데이트 해준다.

코드는 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

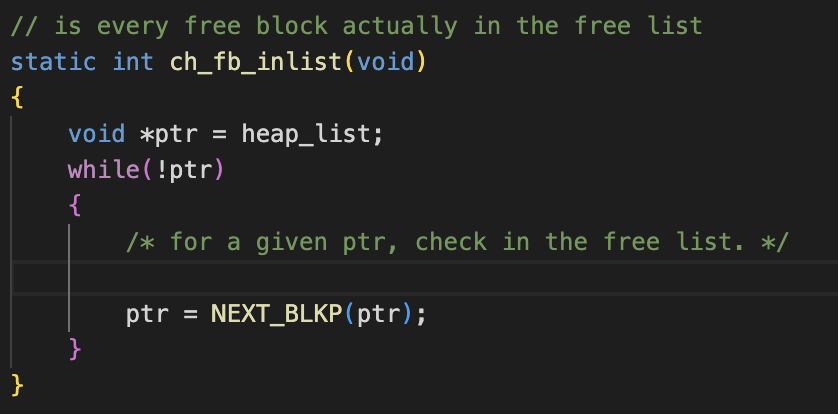
그 다음 함수는 ch\_\_coalesce 함수로 free block의 앞 뒤에 free block이 있으면 안되고, allocated block이 있어야 하는 것을 체크해주어야 한다. Coalescing이 제대로 되었다면 free block들끼리 합쳐졌을 것이므로 앞뒤에 free block이 존재하지 않을 것이다. 반복문은 위의 함수와 똑같이 돌아가게 두고,

이전 블록이 allocated인지 확인한다. Allocated 인지 확인하려면 마찬가지로 GET\_ALLOC()을 쓰면 된다. 코드는 다음과 같다.

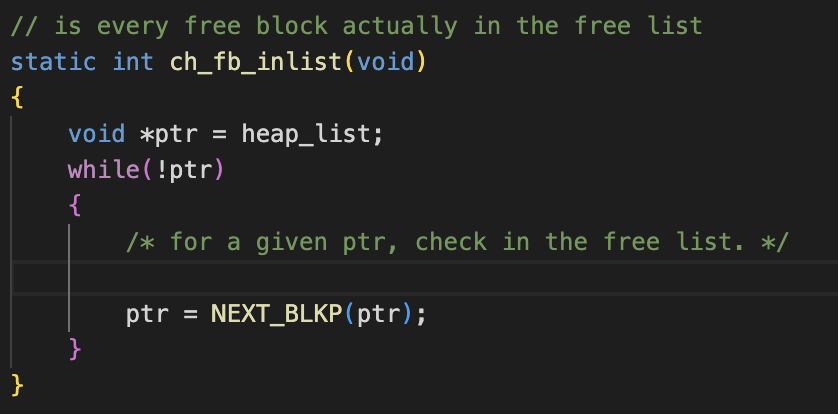
텍스트, 스크린샷, 화면, 은색이(가) 표시된 사진

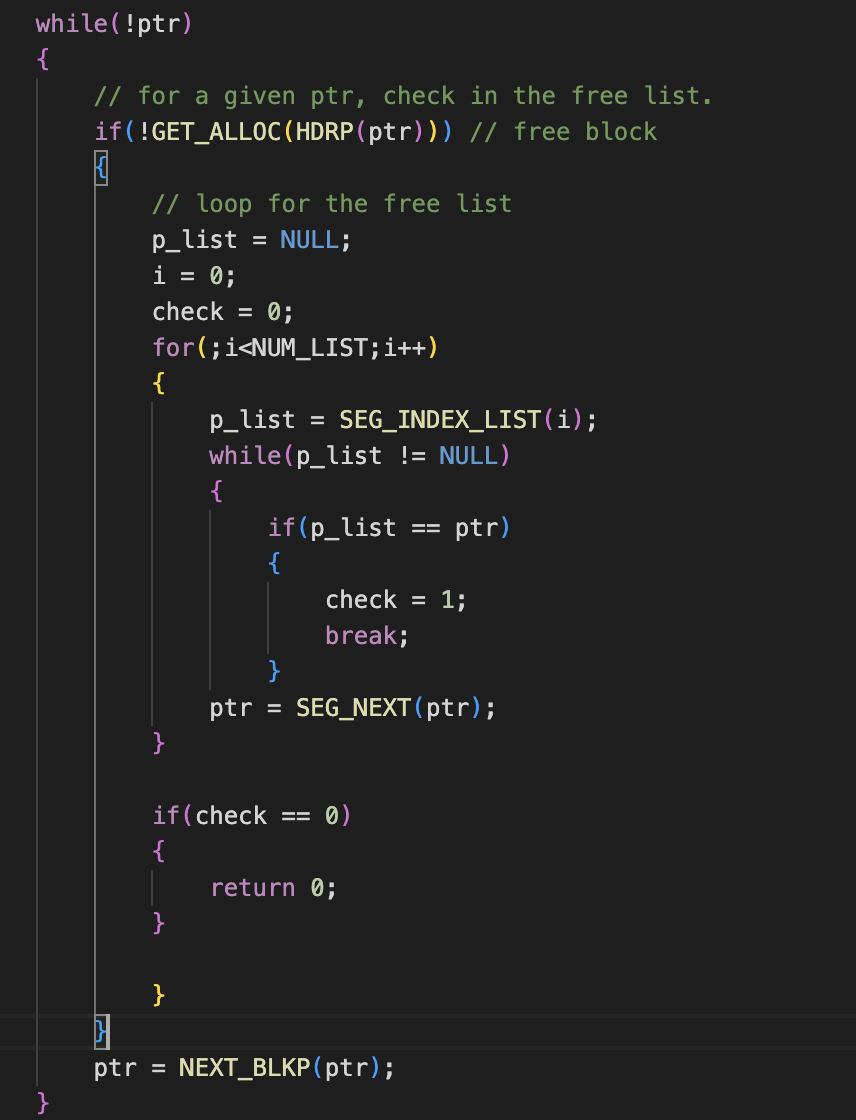
자동 생성된 설명

그 다음 함수로는 ch\_fb\_inlist인데 이는 heap\_list에 있는 모든 free block이 free list에 있는지 확인하는 것이다.



이런식으로 모든 heap\_list에 있는 freeblock들에 대해서 free list에 있는지 확인한다.





즉 하나의 블록 포인터를 가리킨 상태에서 그 블록이 free block일 때만 segregation free list에 대해 loop를 돌게 한다.

그리고 loop를 도는 방식은 위에서의 다른 함수에서의 방식과 동일하고 거기서 같은 블록이 잇는 경우 check 변수에 1을 저장한다.

그리고 loop를 다 돌고 나온 후에 checkㄱㅏ 여전히 0이면 없다는 뜻이므로 이때 return 0;을 하여 에러처리를 한다.

그 다음 함수는 ch\_valid\_fb라는 함수이다.

free list에 있는 pointer가 free block을 잘 가리키고 있는지 체크하는 함수이다.

그러므로, 위의 함수와 반대로 가는 함수라고 생각하면 된다. 직접 segregation free list에서 모든 블록을 거쳐가며 이가 heap list에 있는 블록인지 체크해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이로써 모든 malloc allocator가 완성되었다.