

REPORT

[목 차]

1. Malloc Lab?.....	
3	
1) MallocLab 소개.	
2) MallocLab 구성.	
3) 실습 이해를 위한 사전 지식.	
2. mm-naive.	5
1) mm-naive 상수 및 매크로.	
2) mm-naive 사용된 함수.	
3) mm-naive 성능 측정.	
3. mm-implicit.	
7	
1) mm-implicit 상수 및 매크로.	
2) mm-implicit 사용된 함수.	
3) 구현한 mm-implicit 의 특징 및 느낀점.	
4) mm-implicit 성능 측정.	
4. mm-explicit.	1
5	
1) mm-explicit 상수 및 매크로 와 블록.	
2) mm-explicit 사용된 함수.	
3) 구현한 mm-explicit 의 특징 및 느낀점.	
4) mm-explicit 성능 측정.	
5 . 각 l i s t 방식에 대해 참고한 추가자료 .	
.....	19

1. Malloc Lab?

1) MallocLab 소개.

Malloc Lab은 Dynamic allocation을 구현하는 것이다!

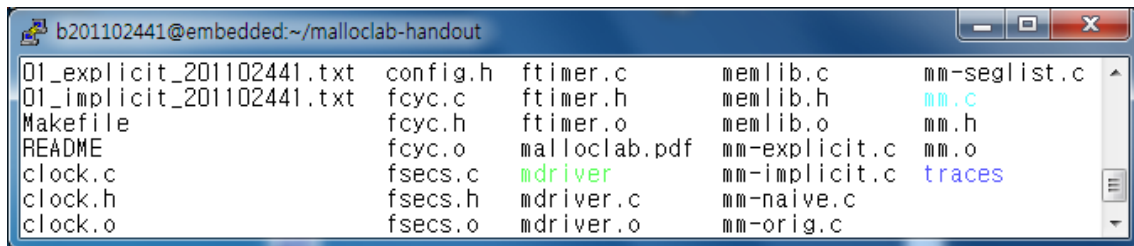
[진행 과정]

- malloc, realloc, free를 구현하되 높은 성능을 갖도록 구현해야한다.
- 구현하는 알고리즘에 따라 다른 성능을 가지게 된다.

iii. 성능 측정 및 테스트는 테스트 프로그램을 이용하여 확인할 수 있다.

2) MallocLab 구성.

본 실습의 구성은 텍스트파일을 제외한 아래의 파일과 같다.



mm-implicit.c : implicit list 방식으로 malloc, free, realloc을 구현한다.

mm-explicit.c : explicit list 방식으로 malloc, free, realloc을 구현한다.

mm-seglist.c : segregated free list 방식으로 malloc, free, realloc을 구현한다.

mm-naive.c : 제공된 코드로 malloc과 realloc이 구현되어 있다. 이 코드를 통해 기본적인 형식에 대해서 파악하고, 각 함수에 대해서 알아보도록 한다.

memlib.c : dynamic memory allocation을 위한 가상 메모리 시스템이다.

mdriver.c : malloclab을 테스트할 프로그램이다.

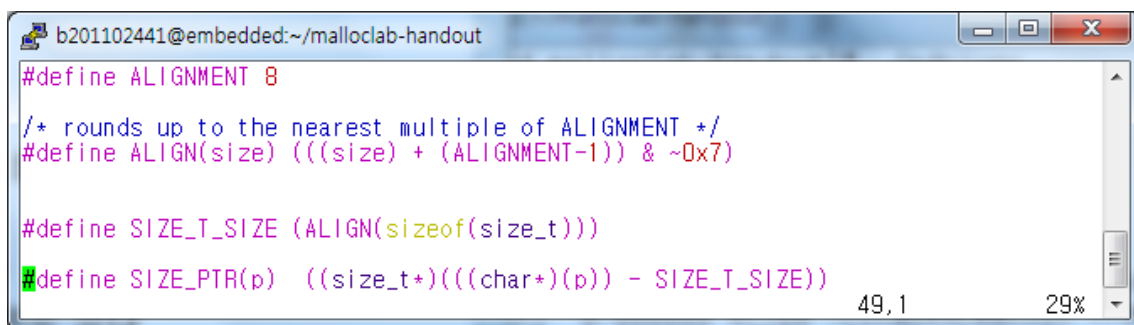
trace : malloclab 테스트 케이스가 있는 디렉토리이다.

3) 실습 이해를 위한 사전 지식.

커다란 데이터 공간이 있다고 하자. 우리는 이 공간에 데이터들을 저장하여 사용한다. 그런데 어떻게 저장 할 것 인가? 문제점들을 고려하지 않는다면, ‘평범하게 데이터 공간의 처음위치부터 순차적으로 쭉 저장하며 쓰면 된다.’는 단순한 사고를 낼 것 이다. (본 사고는 실습에서 제공된 mm-naive의 해당하는 알고리즘이다.) 이는 데이터를 저장하는 면만 본다면 최고의 속도를 가져온다. 하지만 커다란 데이터 공간을 가장 비효율적이게 사용하는 방법이기도 하다. 우리는 이 공간에 저장했던 데이터들을 MASK-ROM처럼 저장하여 변함없이 평생 사용하는 것이 아니라, 필요에 따라 데이터들을 수정도 하고, 삭제를 하기도 한다. 먼저 수정을 한다고 가정하여 보면, 수정하는 데이터의 크기는 원래의 데이터보다 커질 수도 있으며 작아질 가능성도 있

다. 만약 더 커진다면 위와 같은 단순한 사고를 기반으로 할 때, 수정할 데이터의 다음위치에 저장되어 있는 데이터와 충돌이 일어나며, 축소 할 경우에는 앞 데이터와의 사이에 빈 공간이 생긴다. 삭제를 한다고 가정하면 [데이터-데이터(삭제)-데이터] 삭제 할 데이터의 앞 뒤 사이에 더욱 커다란 빈 공간이 생겨버린다. 문제점이 보이는가? 우리는 때론 데이터와의 충돌이 일어나지 않도록 확장을 하여야 하며, 때론 차후 발생하는 빈 공간에 대한 활용을 해야한다. 그것이 MallocLab 실습이다! 한가지 아이디어를 제공하여 준다. 데이터가 들어있는 블록이라는 집을 짓자. 집은 사이즈는 물론, 데이터의 번지수를 담을 수도 있으며, 이웃과의 거리에 대한 정보도 담을 수 있고, 필요에 따라 여러 가지 정보를 임의로 데이터에 붙여 담을 수 있다. 이는 데이터의 크기가 매우 작다면 쓸대없이 크기를 부풀리는 것으로 보일 수 있지만, 일반적으로 블록의 틀 자체는 데이터의 크기에 비하면 매우매우 작은 크기의 데이터에 불과하다. 100MB의 데이터를 예로, 100 MB = 104857600 Byte 이다. 이곳에 주소, 이름, 이웃과의 거리, 기타 등등 원하는 정보를 담아도 100Byte 채 되지않는데, 이러한 정보를 붙임으로써 104857600Byte 같은 데이터들을 관리 할 수 있다면 얼마나 비교조차 할 수 없는 큰 이득인가. 어떤 정보들을 담아, 어떤 방식으로 관리를 할 것인지는 우리의 몫이다. 이 보고서는 내가 어떤 정보를 담아, 어떤 식으로 블록들을 처리하고 활용했는지 담았다.

2. mm-naive.



```

b201102441@embedded:~/malloclab-handout
#define ALIGNMENT 8

/* rounds up to the nearest multiple of ALIGNMENT */
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)

#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
#define SIZE_PTR(p) ((size_t)(((char*)(p)) - SIZE_T_SIZE))
49,1 29%

```

1) mm-naive 상수 및 매크로.

```

/* single word (4) or double word (8) alignment */
#define ALIGNMENT 8

```

블록을 8의 배수로 할당하므로 자주 연산에 사용되기 때문에 상수를 정의하여 사용한다.

```

/* rounds up to the nearest multiple of ALIGNMENT */

```

정렬의 가장 가까운 배수로 반올림까지

```
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)
```

블록이 8의 배수로 할당되도록 연산한다.

연산의 예로 size 가 1이면, $(1+8-1) \& \sim 0x07$ 이고, $001000 \& 1111000 = 1000$ 이다.

따라서 size가 무엇이 들어오든 8의 배수를 반환한다.

```
#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
```

sizeof 명령은 안의 데이터 타입의 크기를 반환한다. int형일 경우 4 double 8 등.

size_t 는 int형과 같이 4를 반환한다.

따라서 블록이 최소 단위로 8을 리턴해야 하기 때문에 이 매크로는 블록의 사이즈를 명시 하며 결과적으로 값은 8을 나타낸다.

```
#define SIZE_PTR(p) ((size_t*)((char*)(p)) - SIZE_T_SIZE)
```

블록의 사이즈를 가르키고 있는 포인터에 접근한다.

2) mm-naive 사용된 함수.

(MallocLab에서 모두 제공되기 때문에 소스는 따로 첨부하지 않습니다.)

int mm_init(void); 사용되지 않았다.

void *malloc(size_t size);

단순하게 heap을 늘려가며 공간을 할당한다.

알고리즘없이 계속 늘리기만 하기 때문에 중간에 발생하는 빈 공간을 하나도 활용 하지 않기 때문에 매우 비효율적으로 공간을 사용하게 된다.

void free(void *ptr); 사용되지 않았다.

void *realloc(void *oldptr, size_t size);

realloc의 본래 기능은 이미 할당되었는 메모리를 재할당하는 함수이다.

naive에서는 size == 0, 경우 oldptr을 바로 free하나, free는 구현하지 않았기에 실질적으로 아무것도 하지 않는다. 양수의 size가 입력이 되면 해당 size 만큼의 공간을 malloc을 통해 할당한 뒤, size 만큼 복사하여 그대로 붙여넣는다.

void *calloc (size_t nmemb, size_t size);

calloc의 본래 기능은 nmemb*size 만큼의 메모리를 확보하고, 그 영역을 초기화 한뒤, 시작주소를 반환해주는 함수이다.

naive에서는 size_t 타입의 변수를 만들어 nmemb*size 의 값을 넣고, 이 변수를 이용하여 malloc() 호출해 할당한뒤, memset을 통해 0으로 초기화 시킨다.

void mm_checkheap(int verbose); 사용되지 않았다.

3) mm-naive 성능 측정.

```
b201102441@embedded:~/malloclab-handout
[b201102441@embedded malloclab-handout]$ ./mdriver
Using default tracefiles in ./traces/
Measuring performance with a cycle counter.
Processor clock rate ~= 1596.0 MHz

Results for mm malloc:
  valid  util   ops   secs    Kops  trace
  yes    94%    10  0.000000  53200  ./traces/malloc.rep
  yes    77%    17  0.000000  57728  ./traces/malloc-free.rep
  yes   100%    15  0.000000  40576  ./traces/corners.rep
* yes    71%   1494  0.000039  37932  ./traces/perl.rep
* yes    68%    118  0.000003  46386  ./traces/hostname.rep
* yes    65%   11913  0.000167  71339  ./traces/xterm.rep
* yes    23%   5694  0.000073  77898  ./traces/amptip-bal.rep
* yes    19%   5848  0.000074  79487  ./traces/cccp-bal.rep
* yes    30%   6648  0.000083  79920  ./traces/cp-decl-bal.rep
* yes    40%   5380  0.000067  79956  ./traces/expr-bal.rep
* yes     0%   14400  0.000169  85440  ./traces/coalescing-bal.rep
* yes    38%   4800  0.000073  66138  ./traces/random-bal.rep
* yes    55%   6000  0.000069  87396  ./traces/binary-bal.rep
10      41%  62295  0.000816  76358

Perf index = 26 (util) + 40 (thru) = 66/100
[b201102441@embedded malloclab-handout]$ █
```

3. mm-implicit.

1) mm-implicit 상수 및 매크로.

```
b201102441@embedded:~/malloclab-handout
/* single word (4) or double word (8) alignment */
#define ALIGNMENT 8

/* rounds up to the nearest multiple of ALIGNMENT */
#define ALIGN(p) (((size_t)(p) + (ALIGNMENT-1)) & ~0x7)

#define WSIZE      4
#define DSIZE      8
#define CHUNKSIZE  (1<<12)
#define OVERHEAD    8

#define MAX(x, y) ((x) > (y)? (x) : (y))

#define PACK(size, alloc) ((size) | (alloc))

#define GET(p)      ( *(unsigned int *) (p) )
#define PUT(p, val) ( *(unsigned int *) (p) = (val) )

#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

#define HDRP(bp) ((char *) (bp) - WSIZE)
#define FTRP(bp) ((char *) (bp) + GET_SIZE(HDRP(bp)) - DSIZE)

#define NEXT_BLKP(bp) ((char *) (bp) + GET_SIZE( ((char *) (bp) - WSIZE) ))
#define PREV_BLKP(bp) ((char *) (bp) - GET_SIZE( ((char *) (bp) - DSIZE) ))
```

(맨 위 두줄은 naive와 동일하다.)

상수를 먼저 보면,

```
#define WSIZE      4           // word 크기를 지정.
#define DSIZE      8           // double word의 크기 지정.
#define CHUNKSIZE  (1<<12)    // 초기 Heap의 크기를 설정해 준다. (4096)
#define OVERHEAD    8         // header + footer의 사이즈.
```

매크로도 위에서부터 순서대로.

```
#define MAX(x, y) ((x) > (y)? (x) : (y))
// x와 y를 비교하여 더 큰 값을 반환한다.
```

```
#define PACK(size, alloc) ((size) | (alloc))
// size와 alloc(a)의 값을 한 word로 묶는다.
// 편리하게 header와 footer에 저장하기 위함.
```

```
#define GET(p)      ( *(unsigned int *) (p) )
//포인터 p가 가리키는 곳의 한 word의 값을 읽어온다.
```

```
#define PUT(p, val) ( *(unsigned int *) (p) = (val) )
//포인터 p가 가리키는 곳의 한 word의 값에 val을 저장한다.
```

```
#define GET_SIZE(p) (GET(p) & ~0x7)
//포인터 p가 가리키는 곳에서 한 word를 읽고 하위 3bit를 버린다.
```

//Header에서 block size를 읽기위함.

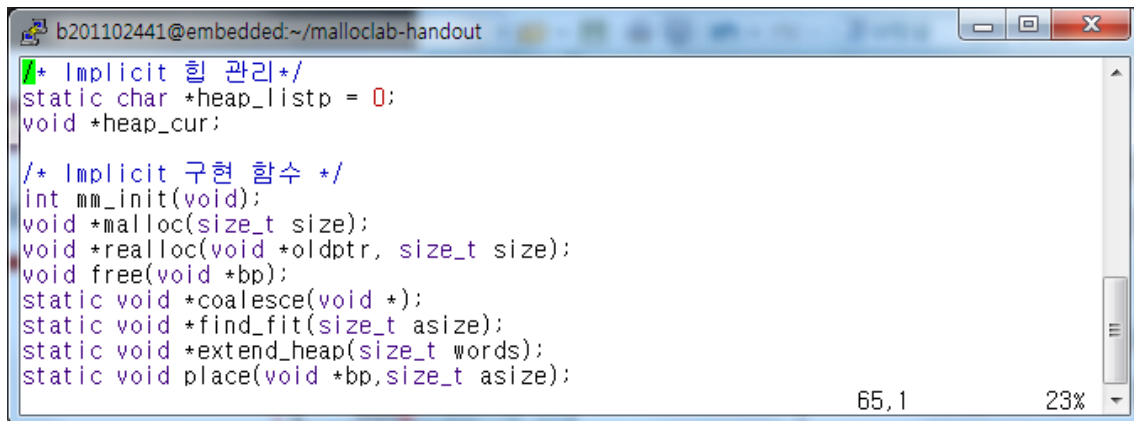
```
#define GET_ALLOC(p)      (GET(p) & 0x1)
//포인터 p가 가리키는 곳에서 한 word를 읽고 최하위 1bit를 가져온다.
//block의 할당여부 체크에 사용된다.
//할당된 블록이라면 1, 아니라면 0.
```

```
#define HDRP(bp)          ((char *) (bp) - WSIZE)
//주어진 포인터 bp의 header의 주소를 계산한다.
```

```
#define FTRP(bp)          ((char *) (bp) + GET_SIZE(HDRP(bp)) - DSIZE)
//주어진 포인터 bp의 footer의 주소를 계산한다.
```

```
#define NEXT_BLKP(bp)     ((char *) (bp) + GET_SIZE( ((char *) (bp) - WSIZE)) )
//주어진 포인터 bp를 이용하여 다음 블록의 주소를 얻어 온다.
```

```
#define PREV_BLKP(bp)     ((char *) (bp) - GET_SIZE( ((char *) (bp) - DSIZE)) )
//주어진 포인터 bp를 이용하여 이전 블록의 주소를 얻어 온다.
```

A screenshot of a terminal window with a blue title bar. The title bar text is "b201102441@embedded:~/malloclab-handout". The terminal shows C code for memory management. The code includes comments in Korean and defines several functions: mm_init, malloc, realloc, free, coalesce, find_fit, extend_heap, and place. The code is color-coded with green for comments, blue for static variables, and black for function signatures and other code. The terminal window has standard window controls (minimize, maximize, close) in the top right corner. The bottom right corner of the terminal shows "65,1" and "23%".

```
b201102441@embedded:~/malloclab-handout
/* Implicit 힙 관리*/
static char *heap_listp = 0;
void *heap_cur;

/* Implicit 구현 함수 */
int mm_init(void);
void *malloc(size_t size);
void *realloc(void *oldptr, size_t size);
void free(void *bp);
static void *coalesce(void *);
static void *find_fit(size_t asize);
static void *extend_heap(size_t words);
static void place(void *bp, size_t asize);
```

2) mm-implicit 사용된 전역변수와 함수.

```
static char *heap_listp = 0; // 처음 first block의 포인터를 선언.
void *heap_cur;              // 현재 block의 위치를 가리키는 포인터.
int mm_init(void);
```



```
b201102441@embedded:~/malloclab-handout
int mm_init(void) {
    /* 초기 empty heap 생성 */
    /* heap_listp - 새로 생성되는 heap영역의 시작 address */
    if( (heap_listp = mem_sbrk(4*WSIZE) ) == NULL) return -1;

    PUT(heap_listp, 0);                          // 정렬을 위한 의미없는 값
    PUT(heap_listp+WSIZE, PACK(OVERHEAD, 1));    // prologue header
    PUT(heap_listp+DSIZE, PACK(OVERHEAD, 1));    // prologu footer
    PUT(heap_listp+WSIZE+DSIZE, PACK(0, 1));     // epilogue header
    heap_listp += DSIZE;

    heap_cur = heap_listp;

    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
    /* 생성된 empty heap을 free block으로 확장 */
    if( extend_heap(CHUNKSIZE/WSIZE) == NULL) return -1; //WSIZE로 align 되어있지 않았으면 에러
    return 0;
}
```

시작할 때 Heap을 생성해주는 역할을 한다.
전역변수로 선언한 heap_listp 포인터 변수를 이용하여 첫 블록을 구성한다. 블록 구조에 맞게 Heap을 설정하며, 본 소스는 ppt에 첨부되어 있다. ppt와 다른점이 하나 있는데 추가로 선언했던 heap_cur 전역변수를 heap_listp 위치와 맞춰준다.

void *malloc(size_t size);

```
b201102441@embedded:~/malloclab-handout
void *malloc (size_t size) { // 블록 할당
    size_t asize;
    size_t extend_size;
    char *ptr;

    if(size <= 0) return NULL;
    if(size <= DSIZE) asize = DSIZE + OVERHEAD; // footer 4, header 4, pay 8
    else asize = DSIZE * ((size + (DSIZE) + (DSIZE - 1)) / DSIZE); // DSIZE * n

    if( (ptr = find_fit(asize)) != NULL){
        place(ptr, asize); // asize 만큼 현재 ptr 가리키는곳에 메모리 위치.
        return ptr;
    }

    extend_size = MAX(asize, CHUNKSIZE);

    if( (ptr = extend_heap(extend_size/WSIZE) ) == NULL) return NULL;

    place(ptr, asize); // asize 만큼 현재 ptr 가리키는곳에 메모리 위치.
    heap_cur = NEXT_BLKPTR(ptr); // 현재 힙 NEXT.

    return ptr;
}
```

메모리에 블록을 할당해주는 함수이다. 할당할 사이즈가 0이하이면 아무것도 하지 않고 빠져나오며, 사이즈가 양의정수로 입력되었다면 공간을 할당한다. 블록은 8의 배수로 할당되며, 블록은 모두 OVERHEAD(header 4bit, footer 4bit로) 8bit을 기본적으로 갖는다. asize는 블록의 사이즈를 담는데 DSIZE(8bit)보다 클 경우 else 문을 통해 DSIZE가 몇 개가 필요한지로 연산을 한다. (8의 배수로 맞추기 위함.)

사이즈를 정했으면 find_fit() 함수를 통해, 효율적인 곳에 배치하기위해 블록을 둘 곳을 서치하며, 만약 찾지 못할 경우 현재 포인터가 위치한 곳에 힙을 할당하여 배치한다. 위 과정을 모두 마치면 현재 힙을 가리키고 있는 포인터를 NEXT(다음블록)에 위치시킨뒤, 종료된다.

```
b201102441@embedded:~/malloclab-handout
void *realloc(void *oldptr, size_t size){ // 메모리 재할당
    size_t oldsize;
    void *newptr;

    if(size == 0){
        free(oldptr);
        return 0;
    }

    if(oldptr == NULL){
        return malloc(size);
    }

    newptr = malloc(size);

    if(!newptr) return 0;

    oldsize = GET_SIZE(HDRP(oldptr));
    if(size < oldsize) oldsize = size;
    memcpy(newptr, oldptr, oldsize);

    free(oldptr);
    return newptr;
}
```

`void *realloc(void *oldptr, size_t size);`
naive와 동일합니다. (설명은 4page.)

`void free(void *bp);`

```
b201102441@embedded:~/malloclab-handout
void free (void *bp) { // 메모리 해제
    if(bp == 0) return;
    size_t size = GET_SIZE(HDRP(bp)); // bp의 헤더에서 block size 읽어옴

    /* 실제로 데이터를 지우는 것이 아니라
     * header와 footer의 최하위 1비트(활용된 상태)만을 수정 */
    PUT(HDRP(bp), PACK(size, 0)); // bp의 header에 block size와 alloc = 0 저장.
    PUT(FTRP(bp), PACK(size, 0)); // bp의 footer에 block size와 alloc = 0 저장.
    coalesce(bp); // 주의 빈 블록이 있을 시 병합.
}
```

교과서에 나온 내용과 동일하다. 모든 라인에 주석이 달려있으므로 따로 설명하지 않습니다.

`static void *find_fit(size_t asize);`

```
b201102441@embedded:~/malloclab-handout
static void *find_fit(size_t asize){ // free block 검색.

    char *p;

    /* 힙의 현재 위치에서 부터 find */
    for( p = heap_cur; GET_SIZE(HDRP(p)) > 0; p=NEXT_BLK(p) )
        if( !(GET_ALLOC(HDRP(p))) && (asize<=GET_SIZE(HDRP(p))) ) return p;

    return NULL;
}
```

heap_cur이 위치한 가장 최근의 생성된 블록부터 Heap 끝까지 서치하며, asize만큼의 데이터를 기록할 수 있는 free block을 찾아내어 해당 지점의 포인터를 반환한다. 찾지 못할 경우 NULL 반환.

`static void *coalesce(void *);`

```
b201102441@embedded:~/malloclab-handout
void *coalesce(void *bp) // 빈 공간을 매꾸는 함수.
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKPTR(bp))); // 이전블록 할당 여부 0 = No, 1 = Yes
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKPTR(bp))); // 다음 블록 할당 여부 0 = no, 1 = Yes
    size_t size = GET_SIZE(HDRP(bp)); // 현재 블록의 size

    /* case 1: 이전 블록, 다음 블록 최하위 bit 둘 다 1(할당)
    * 블록 병합 없이 bp return */
    if(prev_alloc && next_alloc) return bp;

    /* case 2: 이전블록 최하위 bit 1(할당, 다음블록 최하위 bit 0(비할당)
    * 다음 블록과 병합한 뒤 bp return */
    else if(prev_alloc && !next_alloc){
        size += GET_SIZE(HDRP(NEXT_BLKPTR(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }

    /* case 3: 이전 블록 최하위 bit 0(비할당), 다음블록 최하위 bit 1(할당)
    * 이전 블록과 병합한 뒤 새로운 bp return */
    else if(!prev_alloc && next_alloc){
        size += GET_SIZE(HDRP(PREV_BLKPTR(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
        bp = PREV_BLKPTR(bp);
    }

    /* case 4: 이전블록, 최하위 bit 0(비할당), 다음블록 최하위 bit 0(비할당)
    * 이전 블록, 현재 블록, 다음 블록 모두 병합한 뒤 새로운 bp return */
    else{
        size += GET_SIZE(FTRP(PREV_BLKPTR(bp))) + GET_SIZE(HDRP(NEXT_BLKPTR(bp)));
        PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKPTR(bp)), PACK(size, 0));
        bp = PREV_BLKPTR(bp);
    }

    if(heap_cur > bp) heap_cur = bp;

    return bp; // 병합된 블록의 주소 bp return
}
```

coalesce 함수는 빈 공간을 매꾸주는(메모리 낭비를 줄이기 위해) 함수이다. 주석에서의 설명과 같이 총 4가지 케이스가 존재하며, 정리하여 보면 아래와 같다. 기본적인 원리는 이전블록과 다음블록의 할당여부(블록의 최하위 1 bit를 보면 알 수 있음.)를 체크하여 각각의 케이스에 맞도록 병합한 뒤 그 포인터를 반환한다.

- Case 1 : 1 & 1, 모두 사용 중이므로 병합 불가.
- Case 2 : 1 & 0, 다음 블록과 병합 하여 반환.
- Case 3 : 0 & 1, 이전 블록과 병합 하여 반환.
- Case 4 : 0 & 0, 앞 뒤 블록 모두 병합하여 반환.

*1 = 할당, 0 = 비할당 이다.

```
static void *extend_heap(size_t words);
```

```
b201102441@embedded:~/malloclab-handout
static void *extend_heap(size_t words){ // 요청받은 size의 빈 블록을 만든다.
    char *bp;
    size_t size;

    size= (words%2) ? ((words+1)*WSIZE) : (words * WSIZE);
    if((long)(bp=mem_sbrk(size))==-1) return NULL;

    PUT(HDRP(bp), PACK(size,0));
    PUT(FTRP(bp), PACK(size,0));
    PUT(HDRP(NEXT_BLKP(bp)),PACK(0,1));

    return coalesce(bp);
}
```

요청받은 size의 빈 블록을 만들어주는 함수이다.

malloc()에서 블록을 생성할때 find_fit()을 통해 기존 공간을 활용하여 요청받은 size의 크기의 블록을 넣을 수 없을 때 extend_heap()을 통해 블록을 생성하는 것이다.

매크로 이외 사용된 함수로 mem_sbrk가 있는데 이것은 제공된 파일 memlib.c 안에

```
b201102441@embedded:~/malloclab-handout
/* mem_sbrk - simple model of the sbrk function. Extends the heap
 * by incr bytes and returns the start address of the new area. In
 * this model, the heap cannot be shrunk.
 */
void *mem_sbrk(int incr)
{
    char *old_brk = mem_brk;

    if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
        errno = ENOMEM;
        fprintf(stderr, "ERROR: mem_sbrk failed. Ran out of memory...\n");
        return (void *)-1;
    }
    mem_brk += incr;
    return (void *)old_brk;
}
```

들어있는 함수이다.

heap영역에 메모리를 추가시키도록 구성되어 있다.

매크로를 이용하여 블록의 헤더와 풋터를 저장하고 해당 블록의 포인터를 인자로하여 coalesce() 함수를 부른다. (병합 할 수 있는지 체크)

```
static void place(void *bp, size_t asize);
```

```
b201102441@embedded:~/malloclab-handout
static void place(void *bp, size_t asize){ // 해당 위치에 사이즈만큼 메모리를 위치 시킴.
    size_t size = GET_SIZE(HDRP(bp));

    if( (size-asize) >= (DSIZE + OVERHEAD) ){
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));

        bp = NEXT_BLK(bp);

        PUT(HDRP(bp), PACK(size - asize, 0));
        PUT(FTRP(bp), PACK(size - asize, 0));
    }else{
        PUT(HDRP(bp), PACK(size, 1));
        PUT(FTRP(bp), PACK(size, 1));
    }
}
```

해당 위치에 사이즈만큼 메모리를 위치 시켜주는 함수이다.
할당중이던 블록의 크기가 asize보다 크게 되면 뒤에 남게 되는 블록을 처리해 주어야 메모리를 효율적으로 관리 할 수 있다. asize 까지 할당을 하며, 뒤에는 asize의 뺀 값(나머지) 만큼은 비할당 상태로 전환하여준다.

```
b201102441@embedded:~/malloclab-handout
static int in_heap(const void *p) {
    return p < mem_heap_hi() && p >= mem_heap_lo();
}

/*
 * Return whether the pointer is aligned.
 * May be useful for debugging.
 */
static int aligned(const void *p) {
    return (size_t)ALIGN(p) == (size_t)p;
}

/*
 * mm_checkheap
 */
void mm_checkheap(int verbose) {
}
```

기본적으로 구현되어 있지만 사용되지 않은 함수.

3) 구현한 mm-implicit 의 특징.

구현되어있던 naive는 단순히 쪼개 할당만 하는 것이었다면, 구현한 implicit은 데이터에 헤더와 풋터(시작과 끝을 알리는)등으로 앞뒤를 표시하여 데이터를 블록이란 단위로 쪼개었고, 블록의 사용 유무를 체크(최하위 1bit)하는 등 데이터들을 관리 할 수 있게 하며, 이렇게 블록을 관리중에 있어서 할당했던 데이터를 중간에 free()하여 생기는 빈 공간을 find()하여 메모리의 낭비를 어느정도 보완 할 수 있게 되었다. 또한 coalesce()통해 사용중이지 않은 블록을 병합하여 큰 size의 데이터를 할당된 블록들을 조합해 담을 수 있도록 하는 기술을 사용하여 더 메모리의 사용 효율을 높였다. 전역변수를 남들보다 하나 더 사용한 것 같은데(heap_cur), 이것을 사용한 이유는 find() 함수에서 블록들을 찾을 때 비어있는 블록에서부터 서치를 하기 위함이었다(speed향상을 위함). 성능측정 툴 mdriver에서 좀 더 고득점에 접근하려면 빠른 처리속도를 요구했기 때문이다. 그러나 속도와 메모리공간을 둘 다 고효율로 처리하는 것은 어려웠다. heap_cur의 보다 앞쪽에 남아있는 빈 공간들이 있을 수 있기 때문에 메모리활용 면에선 완벽하지 못했던 것 같다.

```
b201102441@embedded:~/malloclab-handout
Measuring performance with a cycle counter.
Processor clock rate ~= 1596.0 MHz

Results for mm malloc:
  valid  util   ops   secs   Kops  trace
  yes    34%    10   0.000001 14916 ./traces/malloc.rep
  yes    28%    17   0.000001 29491 ./traces/malloc-free.rep
  yes    96%    15   0.000001 20288 ./traces/corners.rep
* yes    81%   1494   0.000336  4452 ./traces/perl.rep
* yes    50%    118   0.000018  6517 ./traces/hostname.rep
* yes    87%  11913   0.007892  1509 ./traces/xterm.rep
* yes    97%   5694   0.001169  4869 ./traces/amptjp-bal.rep
* yes    95%   5848   0.003668  1594 ./traces/cccp-bal.rep
* yes    94%   6648   0.001326  5012 ./traces/cp-decl-bal.rep
* yes    93%   5380   0.000741  7257 ./traces/expr-bal.rep
* yes    66%  14400   0.000320 44952 ./traces/coalescing-bal.rep
* yes    90%   4800   0.004684  1025 ./traces/random-bal.rep
* yes    55%   6000   0.000283 21202 ./traces/binary-bal.rep
10      81%  62295   0.020438  3048

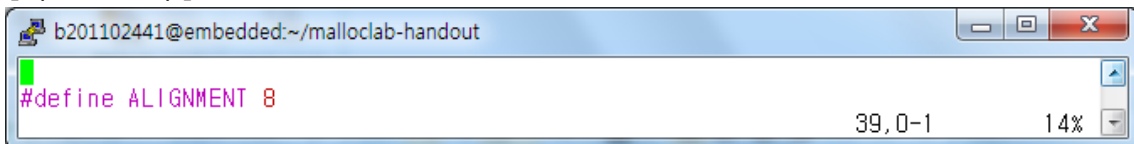
Perf index = 52 (util) + 40 (thru) = 92/100
[b201102441@embedded malloclab-handout]$
```

4) mm-implicit 성능 측정.

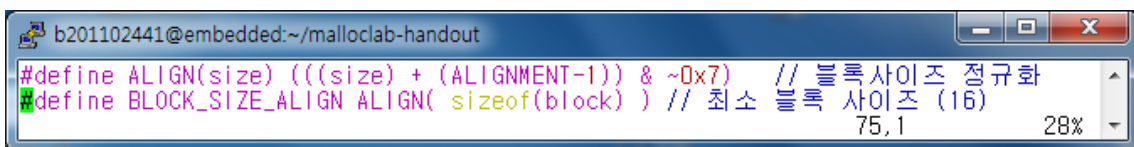
4. mm-explicit.

1) mm-explicit 상수 및 매크로 와 블록.

[사용한 상수]



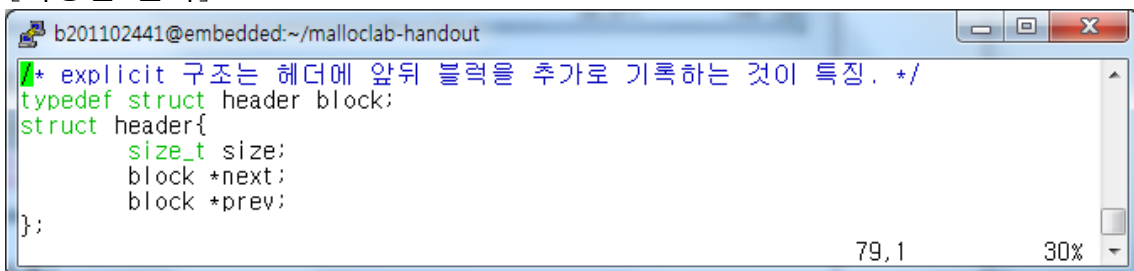
```
b201102441@embedded:~/malloclab-handout
#define ALIGNMENT 8
```



```
b201102441@embedded:~/malloclab-handout
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7) // 블록 사이즈 정규화
#define BLOCK_SIZE_ALIGN ALIGN( sizeof(block) ) // 최소 블록 사이즈 (16)
```

[사용한 매크로]

[사용한 블록]



```
b201102441@embedded:~/malloclab-handout
/* explicit 구조는 헤더에 앞뒤 블록을 추가로 기록하는 것이 특징. */
typedef struct header block;
struct header{
    size_t size;
    block *next;
    block *prev;
};
```

- 헤더는 사이즈뿐만 아닌, 앞의 블록과 뒤의 블록의 위치를 담는 포인터 변수 2개를 포함하고 있습니다.

2) mm-explicit 사용된 함수.

```
b201102441@embedded:~/malloclab-handout
int mm_init(void) {
    /* 블록의 사이즈는 헤더(4), 푸터(4), 다음블록 포인터(1), 이전블록 포인터(1)
     * 한 10이나 블록은 8배수이므로 최소 16의 크기를 갖는다. */
    size_t size = BLOCK_SIZE_ALIGN;

    /* mem_sbrk()는 memlib.c 에 포함되어 있는 함수로 힙에 메모리를 추가시킨다. */
    block *bp = mem_sbrk(size);

    heap_listp = bp;

    /* 소스 보시면 size 옆에 1 붙인게 많은데, 최하위 비트가 블록을 사용중인지 아닌지의
     * 용도로 쓰이는 비트라서 블록을 쓰면 1, 블록 해제 0을 셋하는 용도입니다. */
    bp->size = size|1;
    bp->next = bp;
    bp->prev = bp;
    // 전부 생성한 블록 bp로 초기화 시킵니다.

    return 0;
}
```

- init, 초기화를 해주는 함수이다. 코드에 대해 생략된 주석이 없으므로 추가 설명은 달지 않습니다.

```
b201102441@embedded:~/malloclab-handout
void *malloc (size_t size) {
    /* 인자로 받은 사이즈에 맞는 블록 사이즈 */
    size_t newSize = ALIGN(size + BLOCK_SIZE_ALIGN);

    block *bp;

    if(size <= 0) return NULL; // 잘못된 사이즈의 블록생성 요구하면 처리안함.

    bp = find_fit(newSize); // 블록중에 사이즈 만큼의 공간이 있는 블록 있는지 체크.

    if(bp == NULL){ // 그런 블록 못찾았을때 뉴블록 생성.
        bp = mem_sbrk(newSize);
        if( (long)bp == -1 ) return NULL;
        else bp->size = newSize|1;
    }else{ // 찾았으면 사용중으로 바꾸고 블록연결.
        bp->size |= 1;
        bp->prev->next = bp->next;
        bp->next->prev = bp->prev;
    }

    return (char*)bp + BLOCK_SIZE_ALIGN;
}
```

명은 달지 않습니다.

- malloc, 요구하는 데이터 크기(size)만큼의 메모리를 할당하는 함수이다. 중간에 bp == -1 에 대한 비교문은 뉴블록을 생성하지 못했을 때의 에러에 대한 처리입니다. 마지막 리턴문은 최종적으로 블록을 생성했기에 생성한 블록의 크기만큼 증가시켜 블록포인터를 다음으로 넘깁니다. 이외의 코드에 대해선 생략된 주석이 없으므로 추가 설명은 달지 않습니다. 전체적으로 정리해 보면, 우선 잘못된 사이즈가 요구 되었는지 체크하고, find_fit() 함수를 통해 기존 블록을 활용하여 할당이 가능한지 체크합니다. 체크 결과 그런 블록을 찾지 못했을때 새로운 블록을 생성하며, 찾았으면 그곳에 연결 되었던 프리블록을 서로 연결 시킵니다.


```
b201102441@embedded:~/malloclab-handout
void free (void *ptr) { // 블록 할당 해제

    if(!ptr) return; // 주소 널 오면 아무것도 안함.
    if(heap_listp == 0) mm_init(); // 힙없으면 초기화(전역변수 선언되어있다).

    block *bp
    /* mem_heap_lo() 함수는 memlib.c 에 구현되어있는 함수로
     * 힙의 첫번째 바이트 주소를 반환하는 함수이다. 즉 헤더가리키는 */
    block *head = mem_heap_lo();

    bp->size = bp->size/2; //사용안함으로
    bp->next = head->next;
    bp->prev = head;
    head->next = bp;
    bp->next->prev = bp;

145,1 58%
```

- free, 할당된 블록을 비할당 상태로 즉, 할당 해제 시켜주는 함수이다. ptr - BLOCK_SIZE_ALIGN 는 해당 블록에 접근하는(포인터를 구해 옴으로써) 연산이며, 아래는 프리한 블록을 매꾸는 것이다. 원래 위에서 구현한 mm-implicit 와 같이 병합의 과정을 하여야 메모리를 더 좋게 활용 할 수 있으나, 세그먼테이션 오류의 원인을 찾는 것도 매우 힘들고, 시험이 겹치는 등의 시간 문제로 제한 시간내 구현하지 못하였습니다. 메모리 활용면에서 매우 점수가 떨어지면 딜레이를 감안하면서라도 구현하였을텐데, 생각 외로 괜찮게 메모리 관리가 되어 이렇게 처리하여 보고서에 담습니다. mem_heap_lo() 에 대해선 주석과

```
b201102441@embedded:~/malloclab-handout
/*
 * mem_heap_lo - return address of the first heap byte
 */
void *mem_heap_lo()
{
    return (void *)heap;
}

66,1 72%
```

아래에 추가 소스 스크린 샷 참고.

- mem_heap_lo(), memlib.c 의 내용 중 일부.

```
b201102441@embedded:~/malloclab-handout
void *realloc(void *oldptr, size_t size) { //naive와 동일하나 블록 사이즈가 다르기 때문에 약간 수정.
    size_t oldSize;
    void *newPtr;

    block *bp = oldptr - BLOCK_SIZE_ALIGN; // 블록을 가르키는 포인터.

    if(size == 0){
        free(oldptr);
        return 0;
    }

    if(oldptr == NULL) return malloc(size);

    newPtr = malloc(size);
    if(!newPtr) return 0;

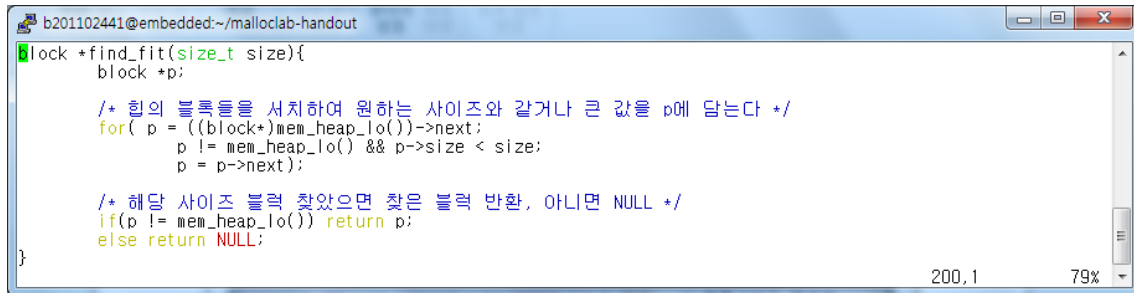
    oldSize = bp->size; // 블록의 사이즈 만큼.
    if(size < oldSize) oldSize = size;
    memcpy(newPtr, oldptr, oldSize);

    free(oldptr);

    return newPtr;

187,1-8 68%
```

- realloc, 제공된 소스 naive.c의 내용과 동일하며, 수정된 부분은 주석을 달았습니다. (naive.c의 realloc의 내용 6 page.)



```
b201102441@embedded:~/malloclab-handout
block *find_fit(size_t size){
    block *p;

    /* 힙의 블록들을 서치하여 원하는 사이즈와 같거나 큰 값을 p에 담는다 */
    for( p = ((block*)mem_heap_lo())->next;
        p != mem_heap_lo() && p->size < size;
        p = p->next);

    /* 해당 사이즈 블록 찾았으면 찾은 블록 반환, 아니면 NULL */
    if(p != mem_heap_lo()) return p;
    else return NULL;
}
```

- find_fit, mem_heap_lo()를 사용하여 힙의 헤더를 가져와, free 상태의 블록을 찾습니다. 찾는 사이즈보다 크거나 같은 경우, 해당 프리한 블록을 찾은 것 이므로 해당 블록이 담긴 포인터를 반환하며, 그렇지 못한 경우 NULL을 반환합니다. 주의 할 점은 만약 포인터가 다시 제자리인 힙의 헤더로 왔다면, 한바퀴 돌아서 찾지 못한 것 이기에 NULL을 반환합니다.

3) 구현한 mm-explicit 의 특징 및 느낀점.

헤더에 블록의 위치를 가리키는 포인터 변수를 만들어, 이웃하는 블록들과 연결을 하여 데이터들을 관리합니다. 프리한 블록이 발생하면, 매번 앞쪽에 삽입하고, 프리블록을 통해 데이터가 할당을 요구하면 요구하는 해당 사이즈의 크기의 프리블록이 존재 할 시, 그곳에 넣은뒤 단순히 앞 뒤 블록을 연결하는 구조입니다. (LIFO와 흡사) 병합을 하지 않기 때문에, 공간을 활용하는 면에 있어선 부족한 점이 많으나 (다수의 프리블록들을 합쳐 커다란 프리블록을 만들어 요구하는 사이즈에 부합하는 블록을 얻어낼 수 있기에), 프리한 블록에 메모리를 할당하는 속도는 가장 빠를 것이라 생각합니다. 이론적으로 무엇을 해야 할지, 어떻게 해야 할지는 이해하였으나, 포인터를 이용한 메모리 접근을 하는 프로그래밍은 비교적 원인을 찾기 힘든 오류를 발생시켜 난해한 문제에 시달리게 되는 것 같습니다. 아직 Seglist문제를 보진 못하였으나, Seglist 구현에서는 최대한 메모리 활용에 중점을 두고 짜 볼 계획입니다.

4) mm-explicit 성능 측정.

```

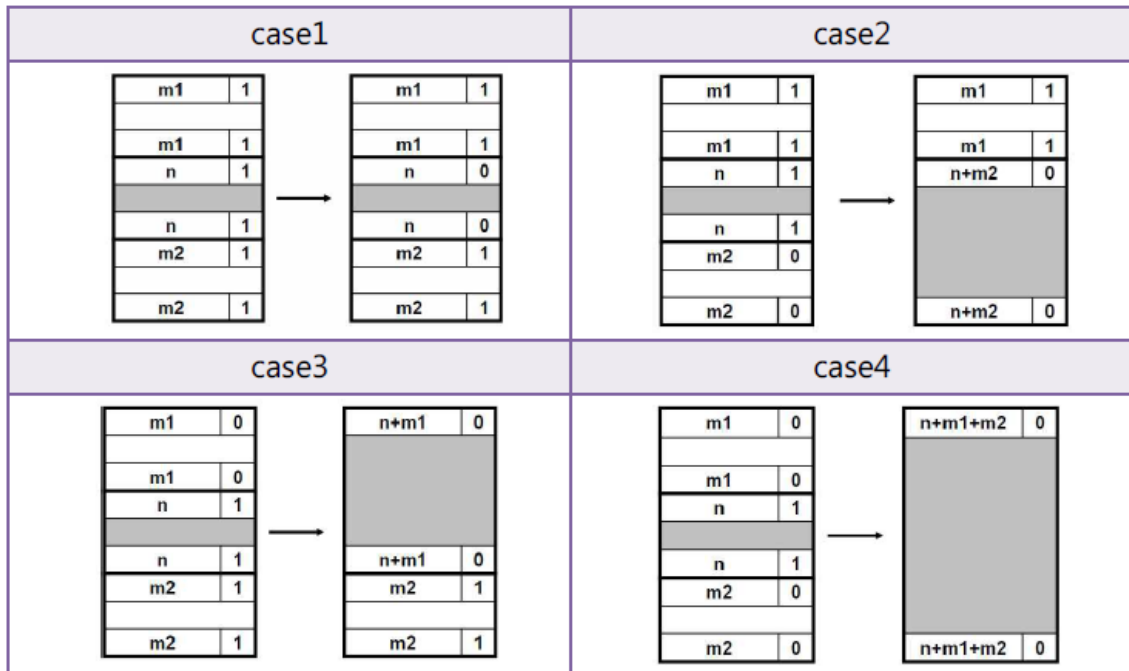
b201102441@embedded:~/malloclab-handout
Results for mm malloc:
valid util ops secs Kops trace
yes 88% 10 0.000001 19950 ./traces/malloc.rep
yes 84% 17 0.000001 26088 ./traces/malloc-free.rep
yes 100% 15 0.000001 14687 ./traces/corners.rep
* yes 73% 1494 0.000080 18743 ./traces/perl.rep
* yes 67% 118 0.000005 24021 ./traces/hostname.rep
* yes 75% 11913 0.000355 33583 ./traces/xterm.rep
* yes 71% 5694 0.000133 42890 ./traces/amptip-bal.rep
* yes 71% 5848 0.000220 26573 ./traces/cccp-bal.rep
* yes 83% 6648 0.000159 41682 ./traces/cp-decl-bal.rep
[ 병합 케이 0.000129 41781 ./traces/expr-bal.rep
스기 0.000260 55327 ./traces/coalescing-bal.rep
0.000201 23846 ./traces/random-bal.rep
0.008779 683 ./traces/binary-bal.rep
0.010321 6036
Perf index = 46 (util) + 40 (thru) = 86/100

```

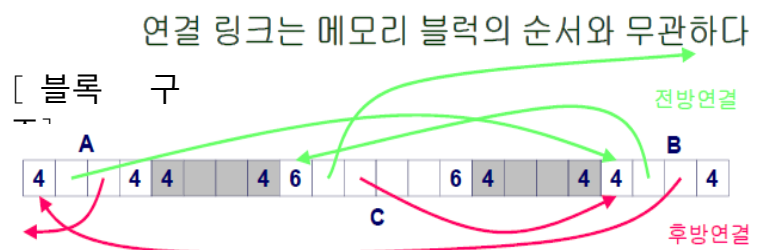
5. 각 list 방식에 대해 참고한 추가자료.

[Implicit list]





[Explicit list]



데이터 내용