HOMEWORK #2: DYNAMIC MEMORY MANAGEMENT

Due: October 15, 2017 23:59:59

PART 1. DOUBLY LINKED LIST 구현하기

DLinkedList.h 파일을 완성하시오.

유의사항

(1) DLinkedList.c 에 선언된 함수의 signature 는 바꿀 수 없다.
(2) 필요에 따라 추가적으로 함수를 작성할 수 있다.
(3) setHeaderInfo 와 setTrailerInfo 는 검색 함수 등에서 사용할 정보를 설정하는 용도로 사용한다.
(4) 모든 함수를 **반드시** 구현해야 합니다. (함수마다 구현 점수 있음)

PART 2. (PART 1.)에서 구현한 DLINKEDLIST 를 이용하여 MEMORY MANAGER 구현하기

MemoryManager.h 파일을 완성하시오.

(1) MemoryManager.h 에 선언된 함수의 signature 는 바꿀 수 없다.
(2) Block *m_alloc(int size): free store (=heap)로부터 size 의 크기를 가지는 Block 를 생성하여 반환한다.
할당이 실패할 경우, NULL 을 반환하며, 오류 메시지를 출력한다.(에러메시지 출력 후 exit(1))
(3) void m_free(Block block): block 를 free store 에 돌려 준다.
내부적으로 Memory Manager 는 heap(=DLinkedList)의 상태를 갱신한다.
(4) 필요에 따라 추가적으로 함수를 작성할 수 있다.
(5) getHeap() 함수는 수정할 수 없다.
(6) 구조체 포인터 변수를 할당받을 때에는
Type *variable = (type*) malloc(sizeof(type)) // type 은 해당 구조체의 타입
을 이용한다.

PART 3. MAKEFILE 만들기

Makefile 을 만들어 여러 파일 (Main.c, MemoryManager.h, DLinkedList.h, Node.h, Block.h)이 한번에 컴파일 되어 하나의 실행파일로 만들어지도록 Makefile 을 작성하여 사용한다.

***추가 사항**

실행시 heap 의 size 를 입력하게 되어있으며, 옆의 그림과 같이 나타난다.

```
====Memory Manager====
Set heap size: 1234

====================
size: 1234
Header] <-> [0<-(1234)->1234] <-> [Trailer
Choose a number to operate
1. malloc
2. free
or, exit
```

(초기 block 을 생성할 때, addFirst()를 사용함.)

header 와 trailer 에는 heap 의 시작 주소, 끝 주소, 크기를 저장한다.

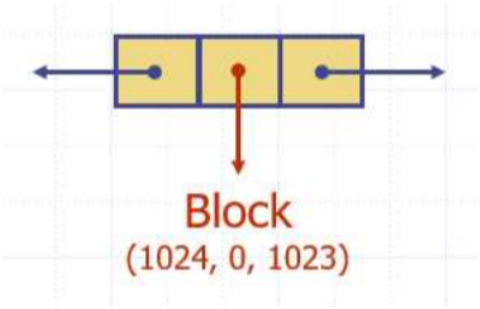
m_alloc 하여 할당받은 block 은 allocatedBlocks[100]에 저장되며, m_free()시 index 로 찾아서 block 을 해제한다.

전체 코드를 꼭 읽어보고, 각 함수와 변수의 역할이 무엇인지 숙지하고 구현을 시작하시오.

(각 코드 상단부에 이름, 학번, 학과 기입할 것.)

Heap 관리 방법

Heap 은 free storage block 의 정보를 나타내는 Node 를 Doubly Linked List 로 관리한다. 각 Node 의 구조는 다음과 같다: 실제 Node 의 구조는 (item, prev, next)이지만, 편의상 아래와 같이 도식화 한다. Item field 는 Block 정보를 나타낸다.



각 Block 은 (size, free block start address, free block end address)의 tuple 구조를 갖는다.

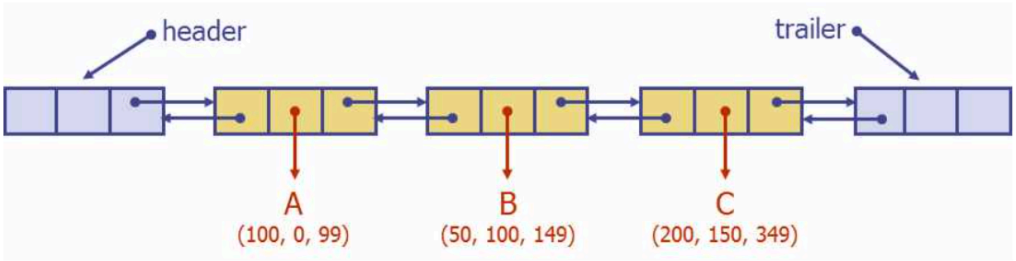
Free Block 관리 규칙은 다음과 같다.

(1) Linked List 의 각 Node 는 block 주소가 증가하는 순서로 관리한다.

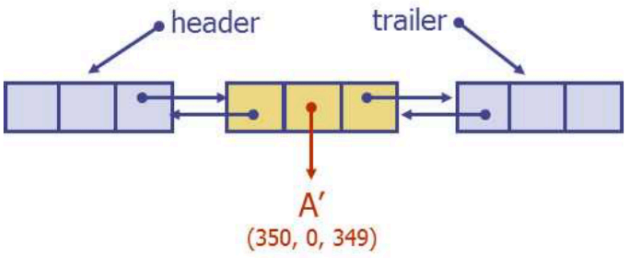
(2) m_alloc(size)가 호출된 경우, 최소 size 이상의 크기를 갖는 free block 을 first-fit 방법으로 찾는다. First-fit 이란 list 의 첫 번째 node 부터 차례로 검색하여 맨 먼저 찾은 node (즉, 크기가 size 이상인, node)를 선택함을 의미한다. 이 경우 선택된 free node 의 block 의 크기와 start address 가 갱신된다.

(3) m_free()에 의해 반납된 block 이 free list 에 추가될 경우, 반납된 free block 을 free list 의 해당 위치에 삽입한다. 단, 새로 삽입된 free block 이 기존의 free block 과 인접한 경우, 하나의 free block 으로 병합(merge)되어야 한다..

Before merge:



After merge



실행 시나리오.

1. Free block list 의 초기화

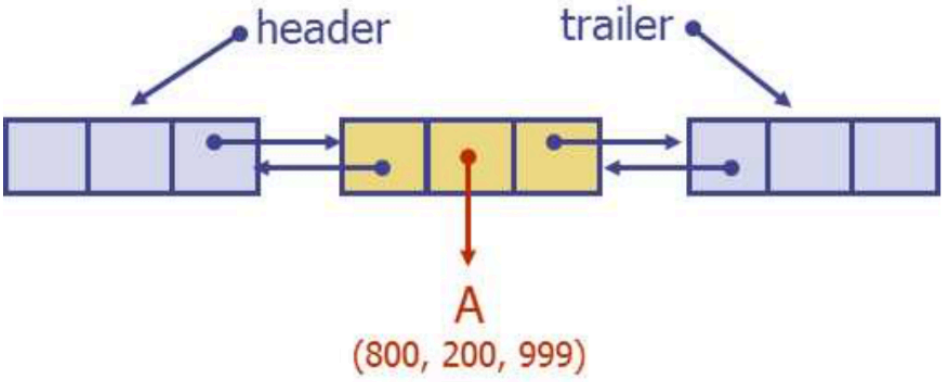
```
printf("====Memory Manager====\n");
printf("Set heap size: ");
scanf("%d", &heapSize);

initManager(heapSize);
```

2. 메모리 크기 200 을 요청한다.

   Block *b1 = m_alloc(200) // b1 = Block(200, 0, 199)

   메모리(200)의 요청이 성공한 이후 Free block list 의 상태:

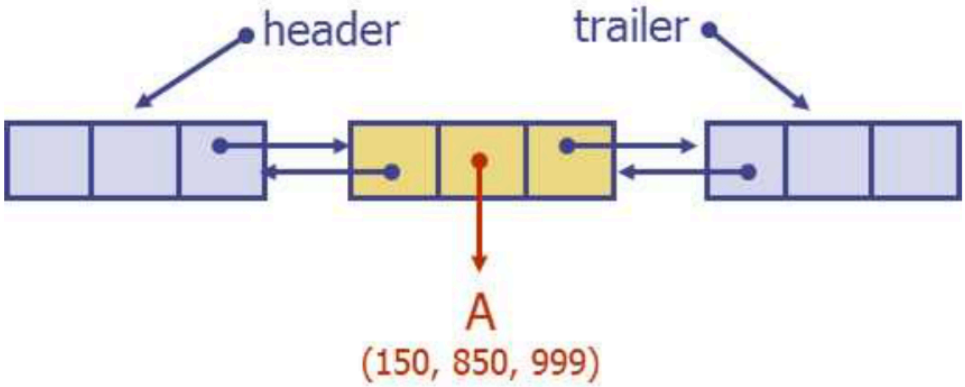


3. 메모리 크기 300, 150, 200 을 연속으로 요청 받은 이후 free block 의 상태

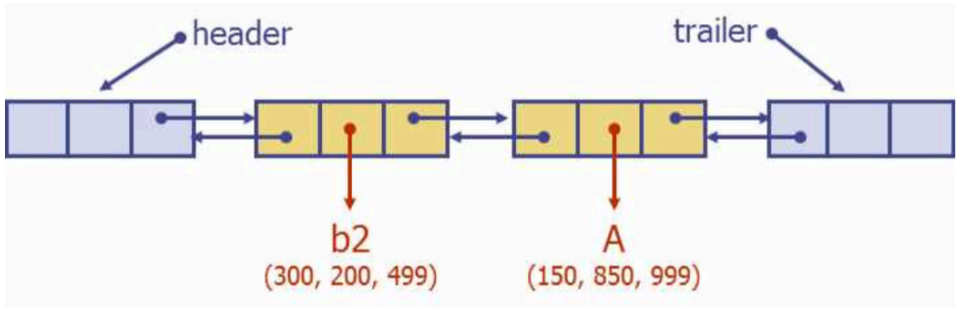
   Block *b2 = m_alloc(300) // b2 = Block(300 200, 499)

   Block *b3 = m_alloc(150) // b3 = Block(150, 500, 649)

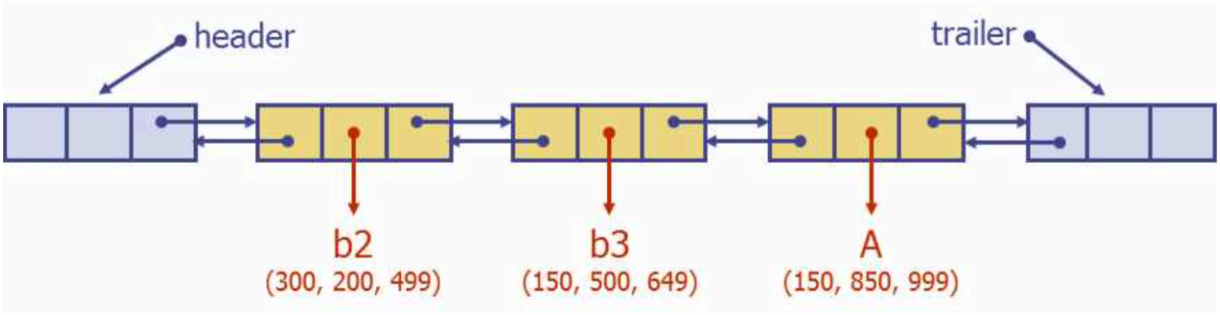
   Block *b4 = m_alloc(200) // b4 = Block(200, 650, 849)



4. 할당 받았던 block b2 (Block(300, 200, 499)) 반환 처리 이후 free block list 의 상태:
   m_free(*b2)



5. 할당 받았던 block b3 (Block(10, 500, 649)) 반환 처리 이후 free block list 의 상태:
   m_free(*b3)



b2 block 과 b3 block 이 주소가 연속된 block 이므로 하나의 free block 으로 merge 함.