HeapMemInit()

↳ Set up one block of memory in used section and one block of memory in the free section.

→ Also makes sure that heap memory initialization is done only once.

FindAllocSize (int iNumBytes)

{ Count = iNumBytes / HeapUnitSize;

Let iNumBytes = 127 //

Let HeapUnitSize = 64

Count = $\frac{127}{64} = 1$ /

FindAllocSize (iNumBytes)

↳ This function makes sure that we always return a chunk of memory that is greater than or equal to the requested memory size (iNumBytes), but never below it.

↳ 'iNumBytes' is the size of requested memory in bytes.

HeapMemInit()

→ Let iHeapSize = 512 = HeapSize;

→ Let iUnitSize = 16 = HeapUnitSize;

→ Let pStartAddr = 200 → pHeapStartAddr

→ psFirstUsedBlock = 200

↳ BlockID is incremented
↳ parentblock is set to NULL
↳ childblock is set to NULL
↳ uiSize = 0.

BlockHeaderSize = sizeof (BLOCK_HEADER)

BlockHeaderSize = 32

↳ When compiled in VS Code

psFirstFreeBlock = (BLOCK_HEADER *)(pHeapStartAddr + BlockHeaderSize);
= 200 + 32
= 232

- BlockID of psFirstFreeBlock is incremented
- ParentBlock is set to NULL
- ChildBlock is set to NULL
- psFirstFreeBlock → uiSize = HeapSize − (BlockHeaderSize * 2)

→ psFirstFreeBlock → uiSize = 512 − (32 * 2) = 448

- Available memory in Heap after 2 Block Header structures.

---

void* MemAlloc(int iNumBytes)

- BLOCK_HEADER *FreeBlock, *NewFreeBlock, *NewUsedBlock;
- void* pData; int iSize;

iSize = FindAllocSize(iNumBytes);

- iNumBytes is the memory size we request for.
- iSize is the memory size that is allocated to us.
- iSize is in multiples of HeapUnitSize.
- iSize ≥ iNumBytes [This holds true always]
- Let iNumBytes = 16 ⇒ iSize = 16

FreeBlock = (BLOCK_HEADER*)(FindUsableFreeBlock(iSize))

- FreeBlock = (BLOCK_HEADER*)(FindUsableFreeBlock(16))
- BLOCK_HEADER* CurrentBlock;
- CurrentBlock = psFirstFreeBlock = 232

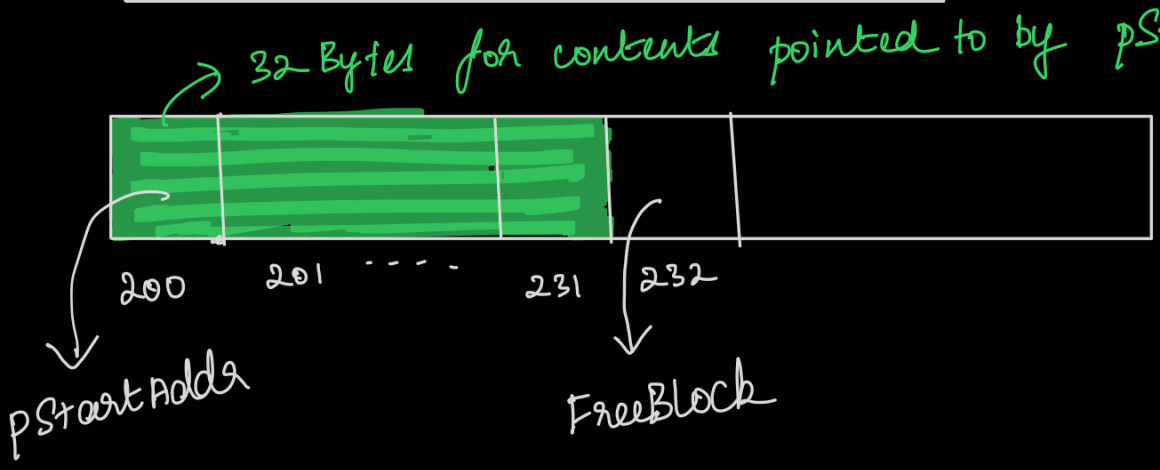
CurrentBlock → uiSize = psFirstFreeBlock → uiSize

= 448 > 16

→ for the first iteration

CurrentBlock → uiSize > iSize ⇒ return CurrentBlock

⇒ FreeBlock = CurrentBlock = 232



→ Using the FindUsableFreeBlock(iSize), we set FreeBlock to point to "232". The first 32 bytes are occupied by contents pointed to by pStartAddr.

if (FreeBlock → uiSize >= (iSize + BlockHeaderSize))

↳ FreeBlock → uiSize = CurrentBlock → uiSize = psFirstFreeBlock → uiSize = 448

↳ iSize + BlockHeaderSize = 16 + 32 = 48

↳ FreeBlock → uiSize > (iSize + BlockHeaderSize)

↳ NewFreeBlock = (BLOCK_HEADER*) (232 + 16 + 32)

FreeBlock, iSize, ↳ BlockHeaderSize

NewFreeBlock = 280

InsertBlock (FreeBlock, NewFreeBlock, FreeBlock->uiSize - (uiSize + BlockHeaderSize)))

InsertBlock (232, 280, 4K8 - (16+32))

InsertBlock (232, 280, 400)

CurrentBlock = 232 ; NewBlock = 280

Increment BlockID of the new block. (BlockID of the CurrentBlock would have been updated in HeapMemInit() function).

Parent (NewBlock) = CurrentBlock

Child (NewBlock) = Child (CurrentBlock)

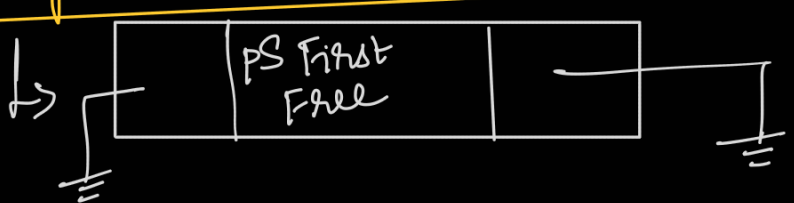
Child (CurrentBlock) is NULL (Child of psFirstFreeBlock was set to NULL in HeapMemInit()).

So Child (NewBlock) = NULL

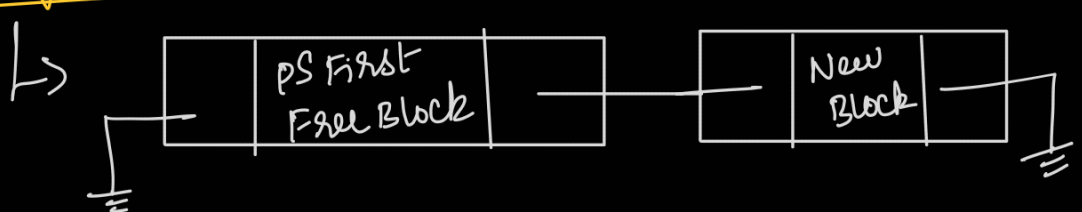
NewBlock -> uiSize = 400 → from the third argument in the function.

Child (CurrentBlock) = NewBlock

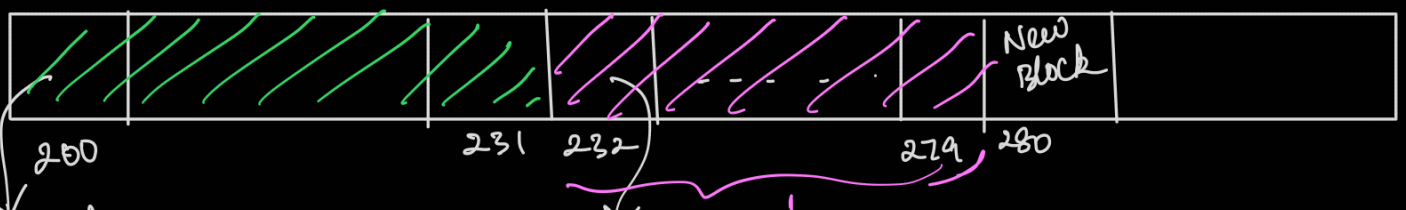
Before InsertBlock()



After InsertBlock



→ for this first time. After the first time, there will be more nodes present in the linked list.



↳ space for Block Header and content of data that the user can put in

(32 + n)
↓ Size for BLOCK_Header structure
↳ Amount of space requested by user

DeleteBlock (FreeBlock, FREE_BLOCK)

→ BLOCK_HEADER *Block, *ParentBlock, *ChildBlock;

→ In the function, void* pBlock and int BlockType are the arguments. ⇒ pBlock = FreeBlock; BlockType = FREE_BLOCK.

→ Block = (BLOCK_HEADER*) pBlock

⇒ Block = FreeBlock = 232

Parent (FreeBlock) = Parent (ChildBlock) = NULL

↳ ChildBlock = Block → pChildBlock = NewBlock = 280

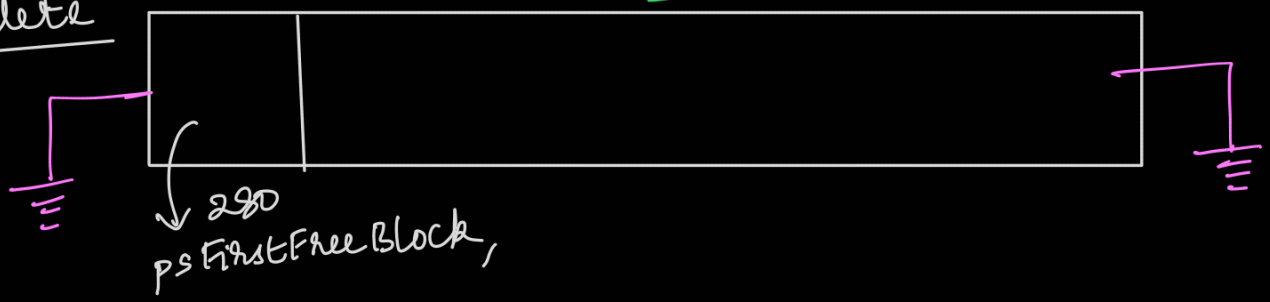
↳ ChildBlock → pParentBlock = NULL

↳ BlockType = FREE_BLOCK

↳ ps FirstFreeBlock = ChildBlock = 280

→ Free List (Available List)

After delete



New Used Block = Free Block = 232

InsertBlock (psFirstUsedBlock, NewUsedBlock, iSize)

- InsertBlock (200, 232, 16)
- BLOCK_HEADER *CurrentBlock, *NewBlock, *ChildBlock;
- CurrentBlock = (BLOCK_HEADER *) pCurrentBlock
  = (BLOCK_HEADER *) psFirstUsedBlock

→ CurrentBlock = 200

→ NewBlock = (BLOCK_HEADER*) pNewBlock = 232

→ Increment BlockID of NewBlock.

→ Parent (NewBlock) = CurrentBlock

⇒ Parent (232) = 200

→ Child (NewBlock) = Child (Current Block)

Child (NewBlock) = Child (psFirstUsedBlock) = NULL

[Child of psFirstUsedBlock was set to NULL in HeapMem Init() function].

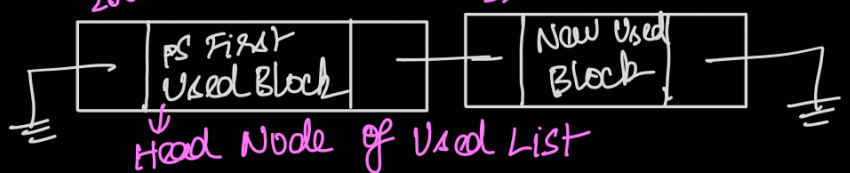
→ NewBlock → uiSize = iSize = 16 → (A)

→ Child (Currentblock) = Newblock.

Before InsertBlock()

After InsertBlock()

Used List [Allocated List]



pData = (void *)((unsigned int) NewUsedBlock + BlockHeaderSize)

- NewUsedBlock points to location 232
- pData must point to the location from where the user can start writing data.
- Every node (chunk) carries a label with it
- This label is the size of the structure BLOCK_HEADER
- So the location that comes after the size of BLOCK_HEADER, considered from starting point NewUsedBlock is the location where the user can do whatever they want to.

→ In this case pData = 232 + 32 = 264.

⇒ 'NewUsedBlock + BlockHeaderSize' points to the location of the allocated section where the user can start making changes.

int MemFree (void* pData):

```
BLOCK_HEADER *UsedBlock, *NewFreeBlock;
int iSize;
UsedBlock = (BLOCK_HEADER *)((unsigned int)pData - BlockHeaderSize);
```

- Here UsedBlock denotes the portion of memory that was used by the task, without considering blockheader size

→ For example purposes, let pData = 264 (this was the result of pData that was obtained at the end of MemAlloc() function)

⇒ UsedBlock = 264 − 32 = 232

iSize = UsedBlock ⇒ uiSize = 232 ⇒ uiSize = 16 (check Ⓐ of pg 6)

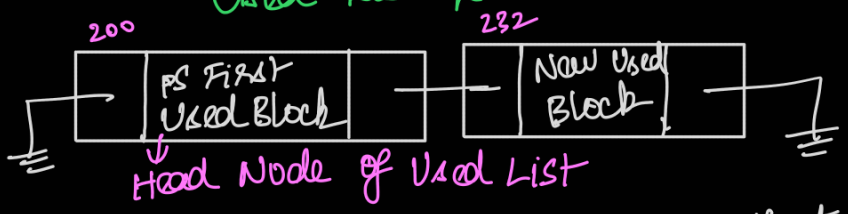
if (HEAP_SUCCESS == DeleteBlock (UsedBlock, USED_BLOCK))

→ BLOCK_HEADER *Block, *ParentBlock, *ChildBlock;

→ Block = (BLOCK_HEADER *) pBlock

Block = 232

Used List [Allocated List]



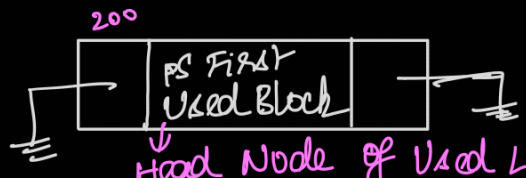
→ From the Used list, we see that child (Block) = child (New Used Block) = NULL

→ Therefore, this is similar to the case of deleting the last element in the "Used Linked List".

→ ParentBlock = Block → parentBlock = 200 (from the Used LL picture)

→ ParentBlock → ChildBlock = NULL

⇒ Updated Used List:



→ return HEAP_SUCCESS after deleting the last node of the linked list.

⇒ NewFreeBlock = UsedBlock = 232

→ InsertBlock (psFirstFreeBlock, NewFreeBlock, iSize)

→ InsertBlock (200, 232, 16)

↳ BLOCK_HEADER *CurrentBlock, *NewBlock, *ChildBlock

→ CurrentBlock = 280

→ NewBlock = 232

→ Increment blockID of NewBlock

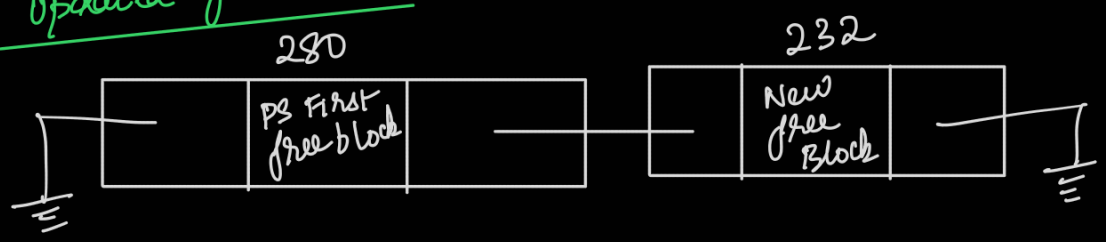
→ Parent (232) = 280

→ Child (232) = NULL (from free LL diagram)

→ NewBlock → uiSize = 16.

→ Child (280) = 232

Updated free list



MergeFreeBlocks( )

↳ BLOCK_HEADER *CurrentBlock, *NextBlock;

↳ unsigned int CurrentBlockEnd;

↳ CurrentBlock = pS First Free block = 280

↳ while (CurrentBlock != NULL)

↳ CurrentBlockEnd = CurrentBlock + BlockHeaderSize + CurrentBlock → uiSize

= 280 + 32 + 400 ← Check this part again

= 712

→ NextBlock = 280

→ while (NextBlock != NULL)

↳ NextBlock ≠ CurrentBlockEnd (280 ≠ 712) ← Check this again

→ Next Block = 232

→ Next Block = NULL

→ Work through the code one more time to see what went wrong in the mergeFreeBlocks(void) function. You seem to do well until you reach the final step.

DRY RUN PART-2

int HeapMemInit():

- Let pStartAddr = 200 = pHeapStartAddr
- iHeapSize = 512 = HeapSize
- iUnitSize = 16 = HeapUnitSize.
- BlockHeaderSize = 32

Used List

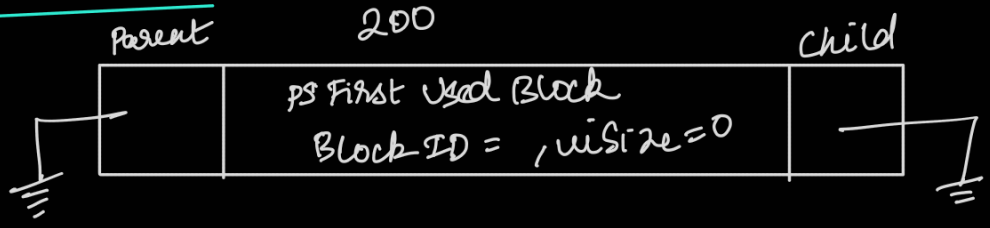
- psFirstUsedBlock = pHeapStartAddr = 200
- Increment BlockID of psFirstUsedBlock
- Parent (psFirstUsedBlock) = NULL
- Child (psFirstUsedBlock) = NULL
- uiSize (psFirstUsedBlock) = 0

Free List

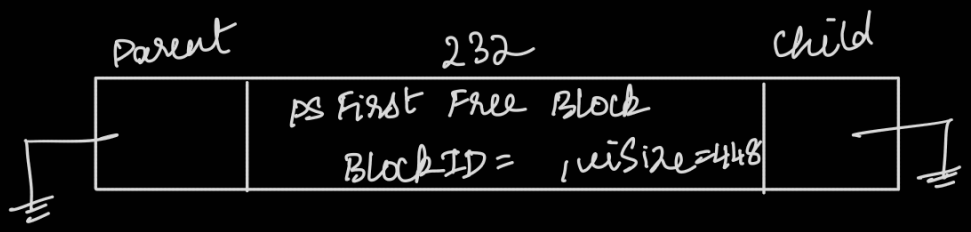
- psFirstFreeBlock = 200 + 32 = 232 (pHeapStartAddr + BlockHeaderSize)
- Increment BlockID of psFirstFreeBlock
- Parent (psFirstFreeBlock) = NULL
- Child (psFirstFreeBlock) = NULL
- uiSize (psFirstFreeBlock) = 512 − (2*32) = 448

After HeapMemInit():

Used list:



Free list:



MemAlloc()

- BLOCK_HEADER *FreeBlock, *NewFreeBlock, *NewUsedBlock;
- void* pData; int iSize;
- iSize = FindAllocSize(iNumBytes) = FindAllocSize(16) — Let iNumBytes = 16

  - Inside FindAllocSize() function:
    - Count = 16/16 = 1
    - iNumBytes = Count * HeapUnitSize ⇒ Count NOT incremented
    - return (Count * HeapUnitSize) = (1 * 16) = 16

- iSize = FindAllocSize(iNumBytes) = FindAllocSize(16) = 16
- FreeBlock = (BLOCK_HEADER*) FindUsableFreeBlock(iSize)

  - Inside FindUsableFreeBlock() function:
    - CurrentBlock = psFirstFreeBlock = 232
    - CurrentBlock->uiSize = 448 > (iSize = 16)
    - ⇒ return CurrentBlock

  ⇒ FreeBlock = CurrentBlock = psFirstFreeBlock = 232

- FreeBlock is NOT NULL
- So, we go into else part.

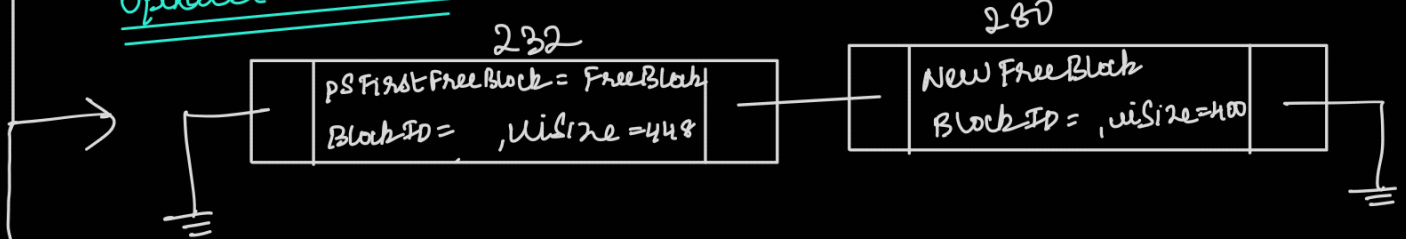
uiSize (FreeBlock) = 448 > iSize + BlockHeader Size = 16 + 32 = 48

NewFreeBlock = FreeBlock + iSize + BlockHeaderSize = 232 + 16 + 32 = 280

InsertBlock (FreeBlock, NewFreeBlock, (FreeBlock → uiSize − (iSize + BlockHeader Size)))

- InsertBlock (232, 280, (448 − (16+32)))
- InsertBlock (232, 280, 400)
- Inside InsertBlock() function:
- CurrentBlock = pCurrentBlock = FreeBlock = 232
- NewBlock = pNewBlock = NewFreeBlock = 280
- BlockID (NewBlock) ++;
- Parent (NewBlock) = CurrentBlock
  - ⇒ Parent (NewFreeBlock) = FreeBlock
- Child (NewBlock) = Child (CurrentBlock)
- But Child (CurrentBlock) = Child (FreeBlock) = NULL.
- uiSize (NewBlock) = iSize = 400
- Child (CurrentBlock) = NewBlock
- ⇒ Child (FreeBlock) = New Free Block
- return HEAP_SUCCESS

Updated Free List



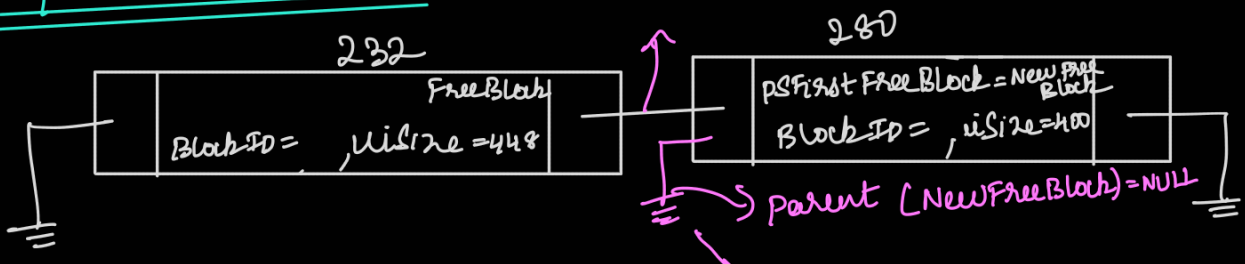
DeleteBlock (FreeBlock, FREE_BLOCK)

- Block = FreeBlock
- Parent (Block) = Parent (FreeBlock) = NULL
- So we delete the first node from free list

- ChildBlock = Child(Block) = Child(FreeBlock)
  = NewFreeBlock = 280
- Parent(ChildBlock) = Parent(NewFreeBlock) = NULL
- BlockType = FREE_BLOCK
  - psFirstFreeBlock = ChildBlock = NewFreeBlock
- return HEAP_SUCCESS

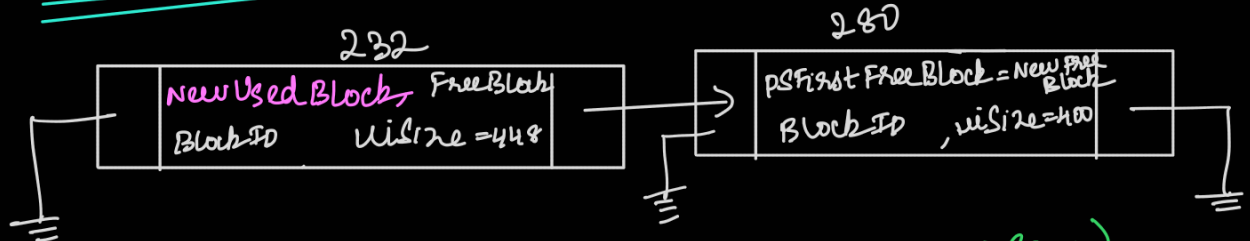
Updated Free List

Child(NewFreeBlock) = 280



NewUsedBlock = FreeBlock;

Updated Free List



Insert(psFirstUsedBlock, NewUsedBlock, iSize)

- Insert(200, 232, 16)
- Inside the Insert() function:
- CurrentBlock = pCurrentBlock = psFirstUsedBlock = 200
- NewBlock = pNewBlock = NewUsedBlock = 232
- BlockID(NewBlockID)++ ⇒ BlockID(NewUsedBlock)++
  - ⇒ BlockID(NewUsedBlock) = 2
- Parent(NewBlock) = CurrentBlock
  - ⇒ Parent(NewUsedBlock) = psFirstUsedBlock = 200
- Child(NewBlock) = Child(CurrentBlock)
  - ⇒ Child(NewUsedBlock) = Child(psFirstUsedBlock)
    = NULL

→ uiSize (NewBlock) = iSize

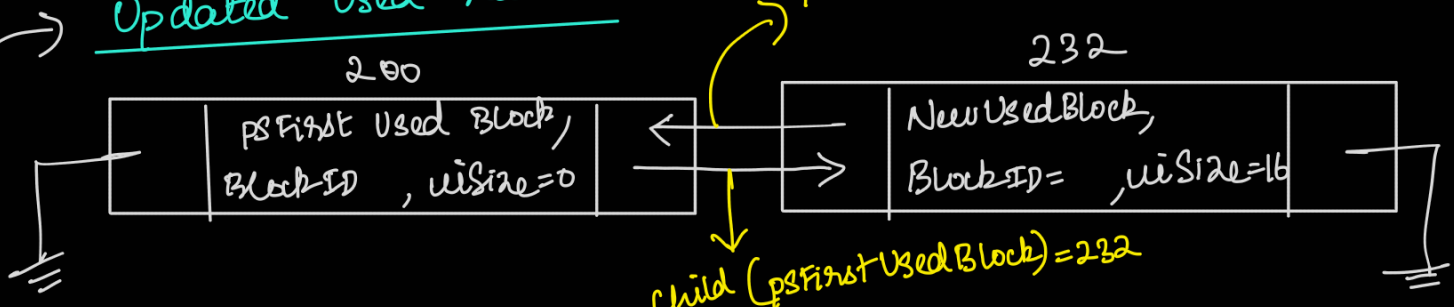
⇒ uiSize (NewUsedBlock) = 16

→ Child (Current Block) = NewBlock

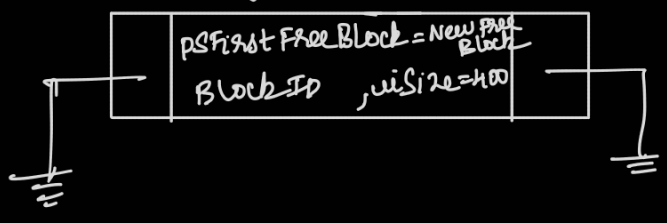
⇒ Child (psFirstUsedBlock) = NewUsed Block = 232

→ return HEAP_SUCCESS

→ **Updated Used List**



→ **Updated Free List**



→ pData = NewUsedBlock + BlockHeaderSize

= 232 + 32 ⇒ **pData = 264**

→ pData = 264 is returned to user at the end of MemAlloc() function.

int MemFree (void* pData)

→ For the purpose of dry run, let pData = 264 (result of MemAlloc() function)

→ UsedBlock = pData - BlockHeaderSize = 264 - 32 = 232

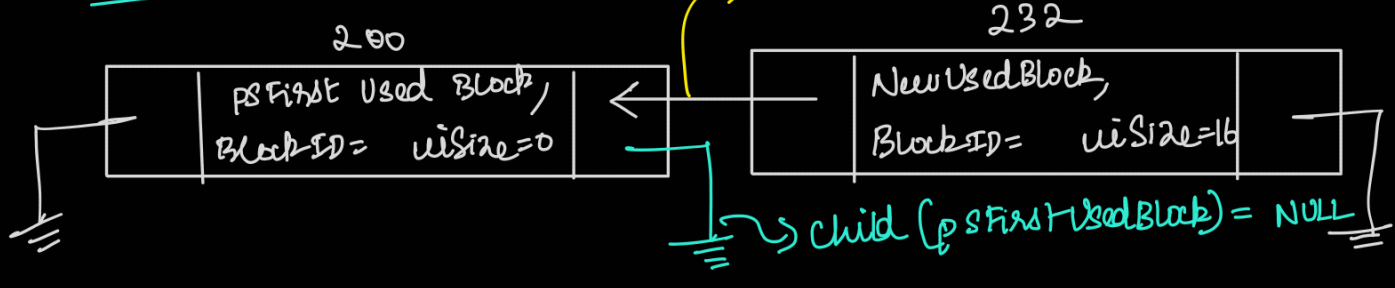
→ iSize = UsedBlock → uiSize = 16 ⇒ **iSize = 16**

if (HEAP_SUCCESS == DeleteBlock(UsedBlock, USED_BLOCK))

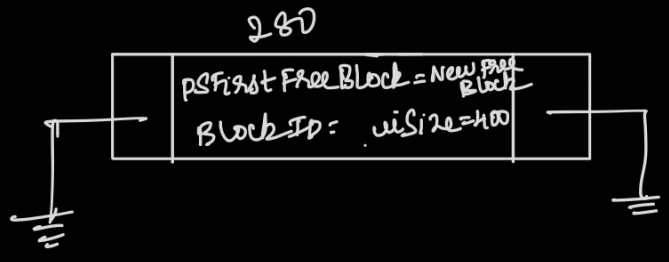
- DeleteBlock(232, USED_BLOCK)
- Inside the DeleteBlock()
- Block = pBlock = 232
- From the "updated used list" of page 14, we see that Child (NewUsedBlock = 232) = NULL ⇒ Case of deleting the last node of a linked list
- ⇒ Child (Block = 232) = NULL
  - ParentBlock = Parent (Block) = psFirstUsedBlock = 200
  - Child (ParentBlock) = NULL
  - ⇒ Child (psFirstUsedBlock) = Child (200) = NULL
- return HEAP_SUCCESS

→ Updated Used List :

parent (NewUsedBlock) = psFirstUsedBlock = 200



→ Updated Free List



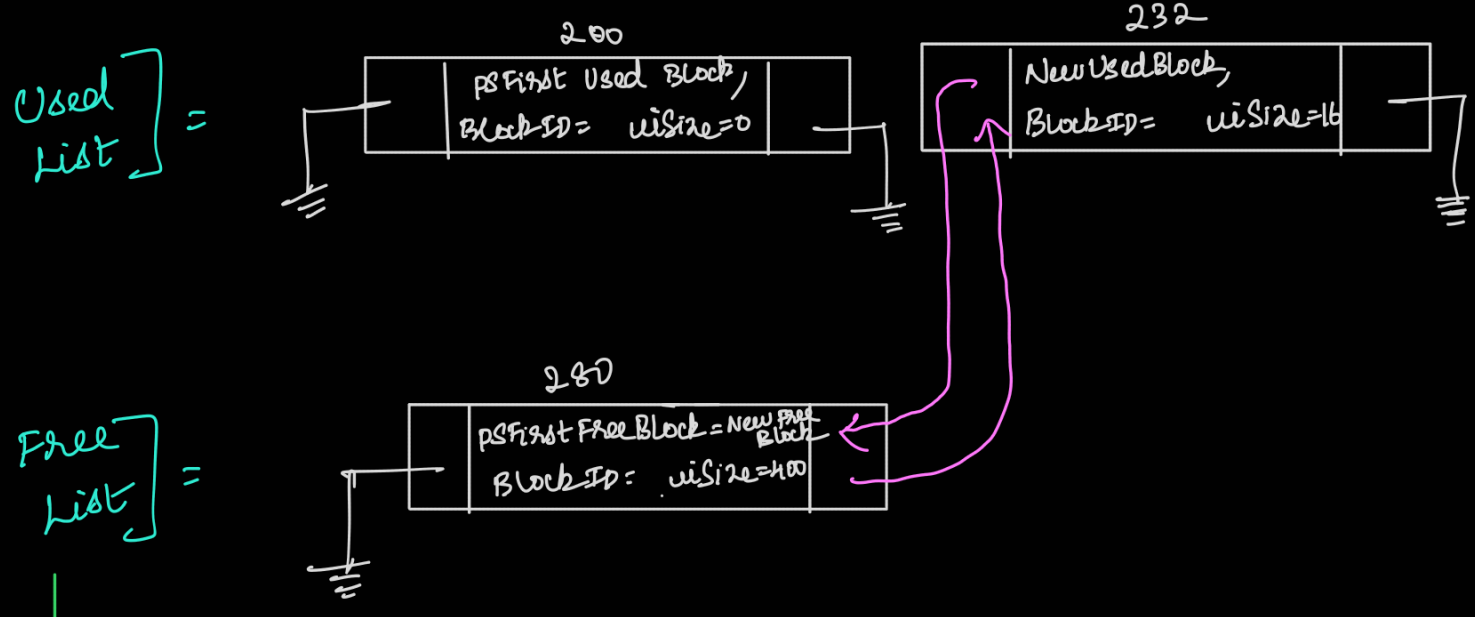
This node is unaltered so far in MemFree() function.

→ NewFreeBlock = UsedBlock = 232

→ InsertBlock (psFirstFreeBlock, NewFreeBlock, iSize)

- InsertBlock (280, 232, 16)
- Inside InsertBlock() function

- CurrentBlock = pCurrentBlock = 280
- NewBlock = pNewBlock = 232
- BlockID (NewBlock) = uiBlockID++
- Parent (NewBlock) = CurrentBlock
  - ⇒ Parent (NewFreeBlock) = psFirstFreeBlock = 280
- Child (NewBlock) = Child (CurrentBlock)
  - ⇒ Child (NewFreeBlock) = Child (psFirstFreeBlock) = NULL
- NewBlock → uiSize = iSize
  - ⇒ uiSize (pNewBlock) = uiSize (NewFreeBlock) = 16
- Child (CurrentBlock) = NewBlock
- ⇒ Child (psFirstFreeBlock) = NewFreeBlock



- Merge FreeBlocks ( )
  - Inside MergeFreeBlocks ( ) function
  - CurrentBlock = psFirstFreeBlock = 280
  - CurrentBlock ≠ NULL
    - CurrentBlockEnd = CurrentBlock + BlockHeaderSize + CurrentBlock → uiSize

⇒ CurrentBlockEnd = 280 + 32 + 400 = 712

→ NextBlock = psFirstFreeBlock = 280

→ NextBlock ≠ NULL

↳ NextBlock = 280 ≠ CurrentBlockEnd = 712

→ NextBlock = Child (NextBlock)

= Child (280) = 232

⇒ NextBlock = 232

→ NextBlock ≠ NULL

→ Next Block = 232 ≠ Current Block End = 712

→ Next Block = Child (Next Block)

= Child (232) = NULL

→ Current Block = child (CurrentBlock)

= child (280) = 232

→ CurrentBlockEnd = 232 + 32 + 16 = 280

→ NextBlock = psFirstFreeBlock = 280

→ Next Block = Current BlockEnd = 280

→ Current Block → uiSize = 400 + 32 = 432

→ DeleteBlock (NextBlock, FREE_BLOCK)

↳ DeleteBlock (280, FREE_BLOCK)

→ Block = pBlock = 280

→ Parent (Block) = Parent (280) = NULL

→ ChildBlock = Child (Block) = Child (280)

⇒ ChildBlock = 232

→ Parent (ChildBlock) = NULL

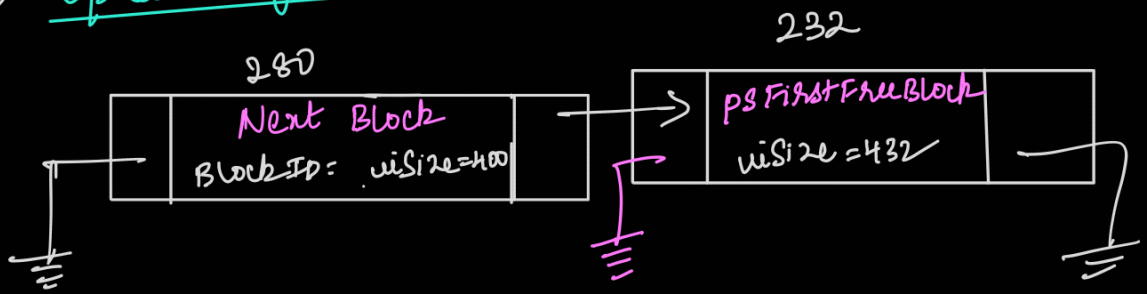
⇒ Parent (232) = NULL

→ BlockType = FREE_BLOCK

→ psFirstFreeBlock = ChildBlock = 232

→ return HEAP_SUCCESS

→ Updated free list:



→ break

→ NextBlock = Child (Next Block) = 232

→ NextBlock ≠ NULL

→ Next Block (232) ≠ Current Block End (280)

→ NextBlock = Child (Next Block) = NULL

→ Current Block = Child (Current Block)

= Child (232) = NULL