The Memory Manager Project

Objectives

- The goal of your next project is to simulate the C heap manager
- A runtime module used to allocate and de-allocate dynamic memory.
- The "heap" is a large "pool" of memory set aside by the runtime system
- The two main functions are
  - **`malloc`**, used to satisfy a request for a specific number of consecutive blocks;
  - **`free`**, used to make allocated blocks available f

Description

- Our simulation uses
  - a large block of unsigned chars as our memory pool; and
  - a doubly-linked list to keep track of allocated and available blocks of unsigned char.
  - We will refer to the nodes of this list as *blocknodes*
- The info field of each node is of type **`blockdata`**
- An object of type **`blockdata`** has attributes
  - **`blocksize`** number of bytes in the block
  - **`free`** a Boolean flag indicating the status of a block
  - **`blockptr`** a pointer to the first byte of the block

`malloc`

- The **`malloc`** algorithm has an **`int`** parameter **`request`**
- **`request`** is the size of the block to be allocated
- **`request`** scans the list until it finds the first blocknode **`B`** such that
  - **`B.free == true`**
  - **`B.size ≥ request`**
- If no such block is found, **`malloc`** returns **`NULL (0)`**

`malloc`

- If **`B.size`** is larger than **`request`**, the block is broken up into two blocks
  - The first block's size: **`request`**
  - The second's size: **`B.size-request`**
- This requires that we insert a new blocknode **`C`** after **`B`** to reference the second block (which is free)
- Then, whether we split the block or not, we
  - set **`B.free`** to **`false`**
  - set **`B.size`** to **`request`**
  - return the address **`B.bptr`**

`free`

- To implement **`free(unsigned char *p)`** we must find the blocknode whose **`bptr`** field equals **`p`**
- This is done by traversing the blocknode list
- If this fails, we terminate the program
- Otherwise we change the blocknode's **`free`** field to **`true`**
- But we don't stop there

Merging Consecutive `free` Blocks

- It should be clear that we want to maximize the size of the free blocks
- This means there should never be consecutive free blocks
- Whenever consecutive free blocks occur, they should be merged
- When we free a block, we need to check the previous and next blocks to see if they are free
- If so, we must merge the blocks into one big block
- This may involve the deletion of one or two blocknodes from our list

Doubly-Linked List Utilities

- To manage doubly-linked lists, we will use a collection of templated functions
- We will not need the apparatus of a class here, a **`struct`** suffices
- The definition of **`dlNode`** and associated functions will be supplied in the file **`dlListUtils.h`**
- We will take the approach used in the text for doubly-linked lists
- Namely, we will use dummy header and trailer nodes
- This simplifies the code for many list operations

Project Files

- The files used in this project are
  - **`dlListUtils.h`**
  - **`blockdata.h`**
  - **`blockdata.cpp`**
  - **`MemoryManager.h`**

    Do not modify, do not submit

  - **`MemoryManager.cpp`**

    Complete and submit

  - **`testMemMgr.cpp`**

    Modify and use for testing;
    Do not submit

Source Code

dlUtils.h

```
#include <iostream>
#include <cassert>
template <class T>
struct dlNode {
  T info;
  dlNode<T> *prev;
  dlNode<T> *next;
  dlNode<T>(T val, dlNode<T> *p,
            dlNode<T> *n)
                :info(val),prev(p),next(n){};
};
```

dlUtils.h

```
template <class T>

void insertAfter(dlNode<T> *trailer,
                 dlNode<T> *current, T newval)

{

  assert(current != trailer);

  current->next =
                           prev      next
    new dlNode<T>(newval,current,current->next);

  current = current->next;

  current->next->prev = current;

}
```

dlUtils.h

```
template <class T>
void printDlList(dlNode<T>* header,
                 dlNode<T> *trailer,
                 const char *sep)

{
  assert(header != NULL && trailer != NULL);
  dlNode<T> *cursor = header->next;

  while(cursor->next != trailer) {
    std::cout << cursor->info << sep;
    cursor = cursor->next;
  }

  if (cursor->next = trailer)
    std::cout << cursor->info << std::endl;
}
```

dlUtils.h

```
template <class T>

void deleteNode(dlNode<T>* header,
                dlNode<T>* trailer,
                dlNode<T>* current)

{

  assert(current!= header &&
         current != trailer);

  dlNode<T> *hold = current;

  current->prev->next = current->next;
  current->next->prev = current->prev;

  delete hold;
}
```

dlUtils.h

```
template <class T>
void deleteNext(dlNode<T>* header,
                dlNode<T>* trailer,
                dlNode<T> *current)

{

  assert(current != trailer &&
         current->next != trailer);

  deleteNode(header,trailer, current->next);

}
```

dlUtils.h

```
template <class T>
void deletePrevious(dlNode<T> * header,
                    dlNode<T> * trailer,
                    dlNode<T> *current)

{

  assert(current != header &&
         current->prev != header);

  deleteNode(header, trailer,current->prev);

}
```

dlUtils.h

```
template <class T>
void clearList(dlNode<T> *p)
{

  dlNode<T> *hold = p;

  while(p != NULL) {
    p = p->next;
    delete hold;
    hold = p;
  }
}
```

The blockdata Definition

```
// blockdata.h
#include <iostream>
class blockdata {
  friend ostream& operator<<(ostream&
                          const blockdata &);
 public:
  blockdata(unsigned int s, bool f,
            unsigned char *p);
  int blocksize;
  bool free;
  unsigned char *blockptr;
};
```

The blockdata Implementation

```
// blockdata.cpp
#include "dlUtils.h"
#include "blockdata.h"
#include <iostream>
using namespace std;
blockdata::blockdata(unsigned int s, bool f,
                     unsigned char *p)
{
  blocksize = s;
  free = f;
  blockptr = p;
}
```

The blockdata Implementation

```
// blockdata.cpp
ostream &operator << (ostream &out, const
blockdata &B)
{
  out << "[" << B.blocksize << ",";
  if (B.free)
    out << "free";
  else
    out << "allocated";
  out << "]";
  return out;
}
```

The MemoryManager Definition

```
class MemoryManager
{
  public:
   MemoryManager(unsigned int memsize);
   ~MemoryManager();
   unsigned char *
   malloc(unsigned int request);
   void free(unsigned char * ptr2block);
   void showBlockList();
```

The MemoryManager Definition

```
 private:
   unsigned int memsize;
   unsigned char *baseptr;
   dlNode<blockdata>* header;
   dlNode<blockdata>* trailer;

   void mergeForward(dlNode<blockdata> *p);
   void mergeBackward(dlNode<blockdata> *p);
   void splitBlock(dlNode<blockdata> *p,
                   unsigned int chunksize);
};
```

The MemoryManager Implementation

```
MemoryManager::MemoryManager(unsigned int memtotal)
                                    : memsize(memtotal)

{
   baseptr = new unsigned char[memsize];

   blockdata dummyBlock(0,false,0);
   blockdata originalBlock(memsize,true,baseptr);
   header = new
       dlNode<blockdata>(dummyBlock,nullptr,nullptr);

   trailer = new
       dlNode<blockdata>(dummyBlock,nullptr,nullptr);

   header->next = new
       dlNode<blockdata>(originalBlock,header,trailer);

   trailer->prev = header->next;

}
```

The MemoryManager Implementation (partial)

```
void MemoryManager::showBlockList()
{
  printDlList(firstBlock,"->");
}
```

The MemoryManager Implementation (partial)

```
void
MemoryManager::mergeForward(dlNode<blockdata> *p)
{ // Put your code here }
void
MemoryManager::mergeBackward(dlNode<blockdata> *p)
{ // Put your code here }


void
MemoryManager::free(unsigned char *ptr2block)
{ // Put your code here }
```

The MemoryManager Implementation (partial)

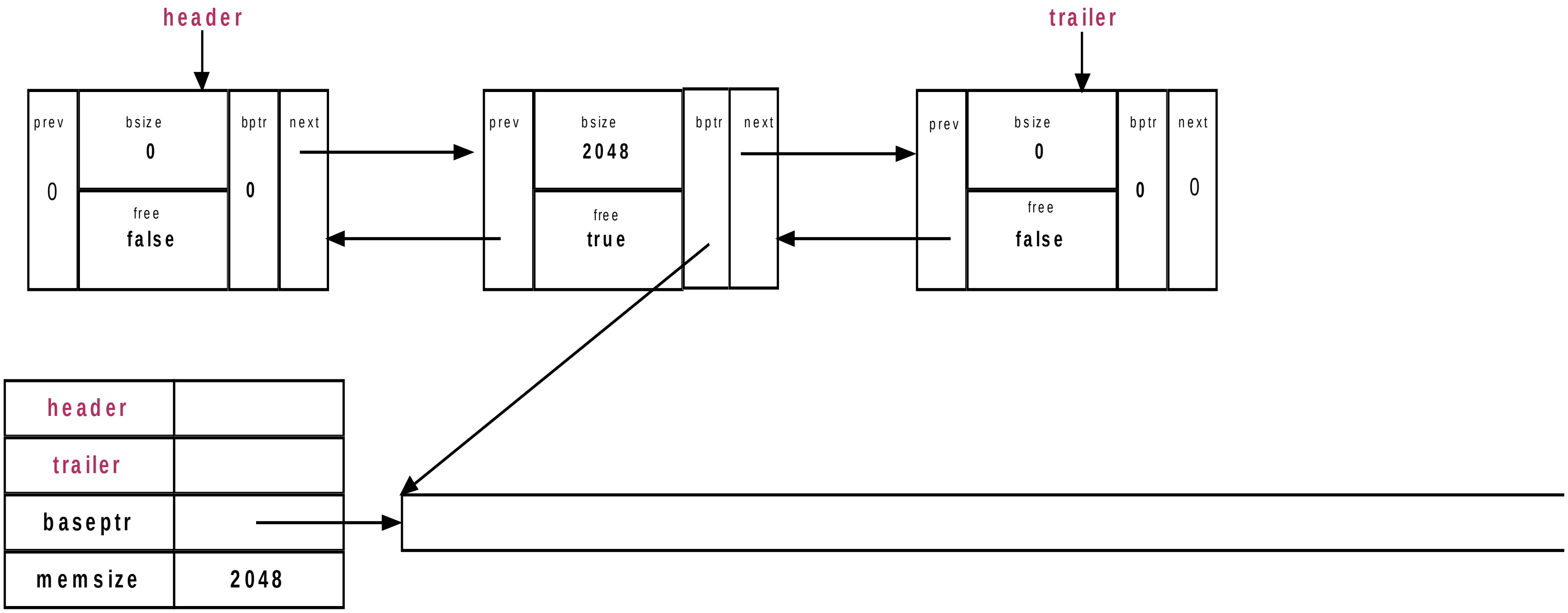
```
void
MemoryManager::splitBlock(dlNode<blockdata> *p,
                          unsigned int chunksize)
{ // Put your code here }
```

```
unsigned char *
MemoryManager::malloc(unsigned int request)
{ // Put your code here }
```

Visual Trace of Operations

The MemoryManager Constructor

```
MemoryManager M(2048);
```



header

trailer

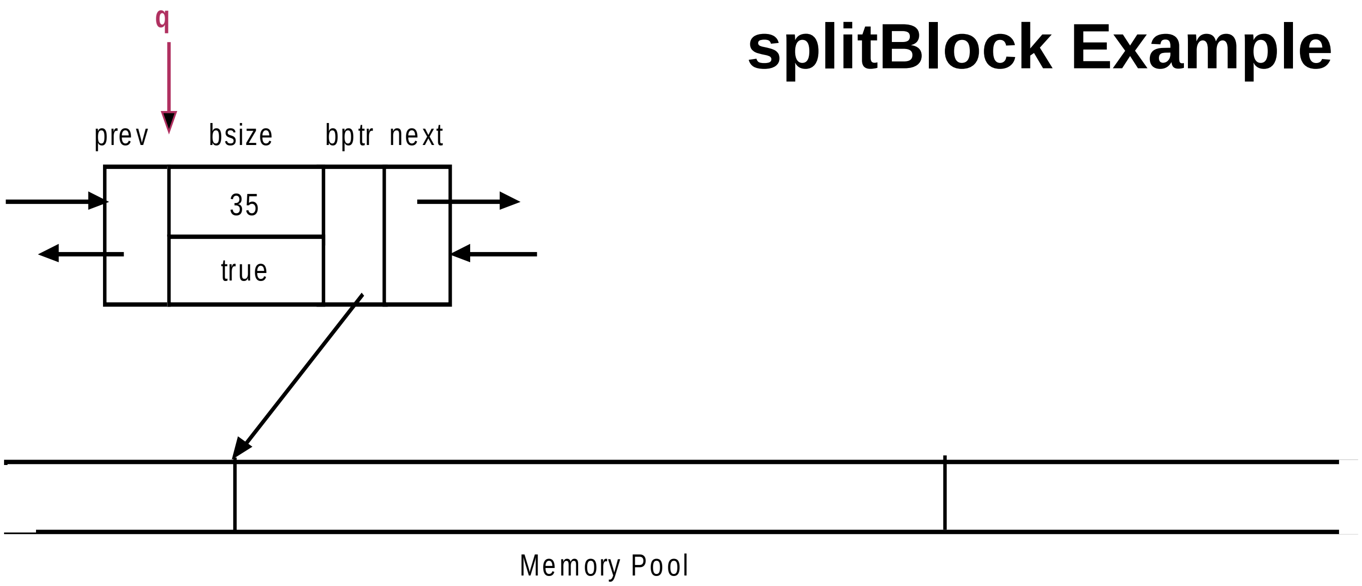
prev 0 | bsize 0 | free false | bptr 0 | next

prev | bsize 2048 | free true | bptr | next

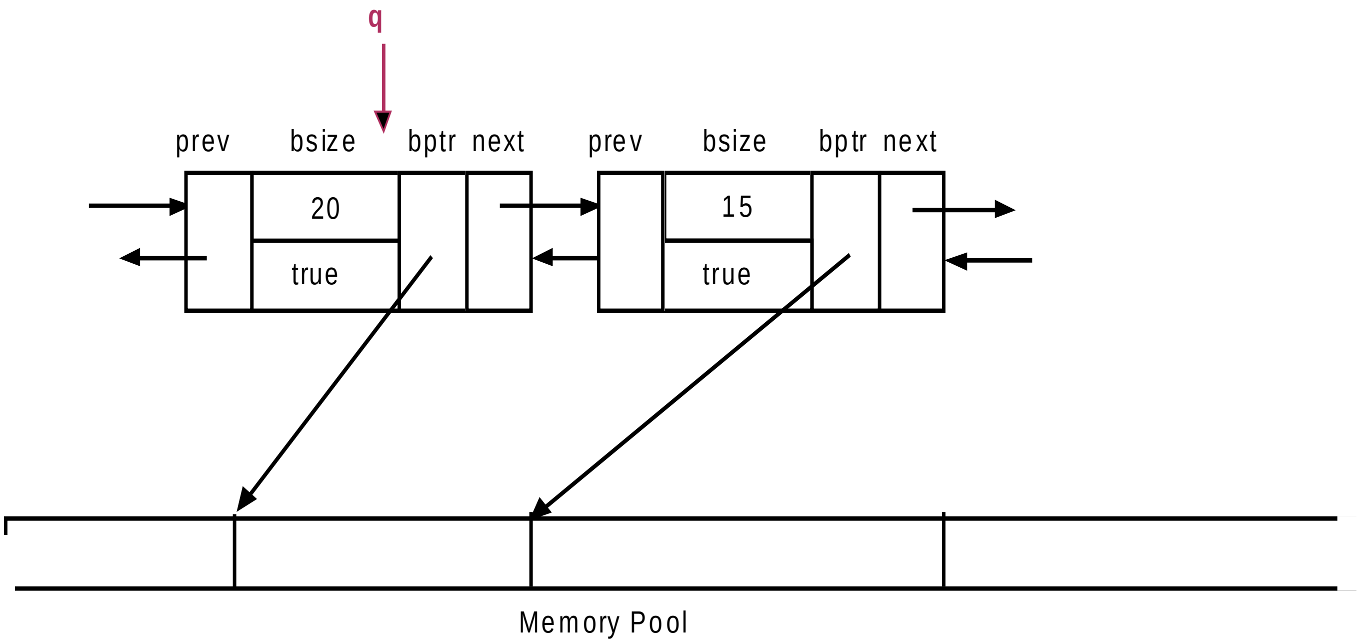
prev | bsize 0 | free false | bptr 0 | next 0

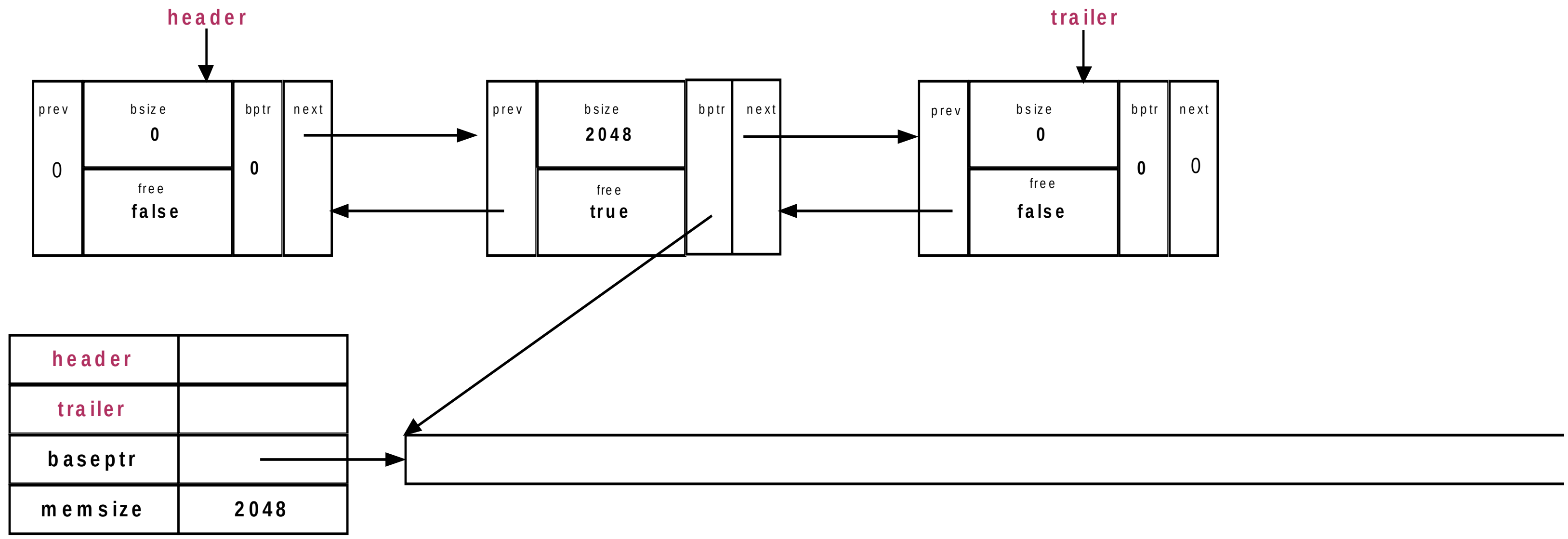
header	
trailer	
baseptr	
memsize	2048

splitBlock Example

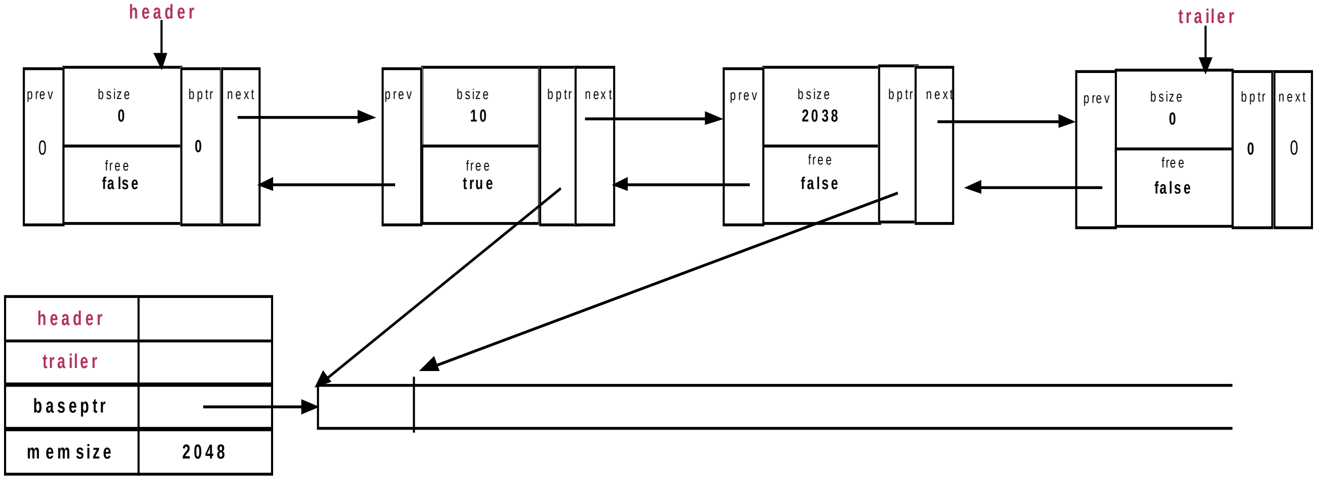


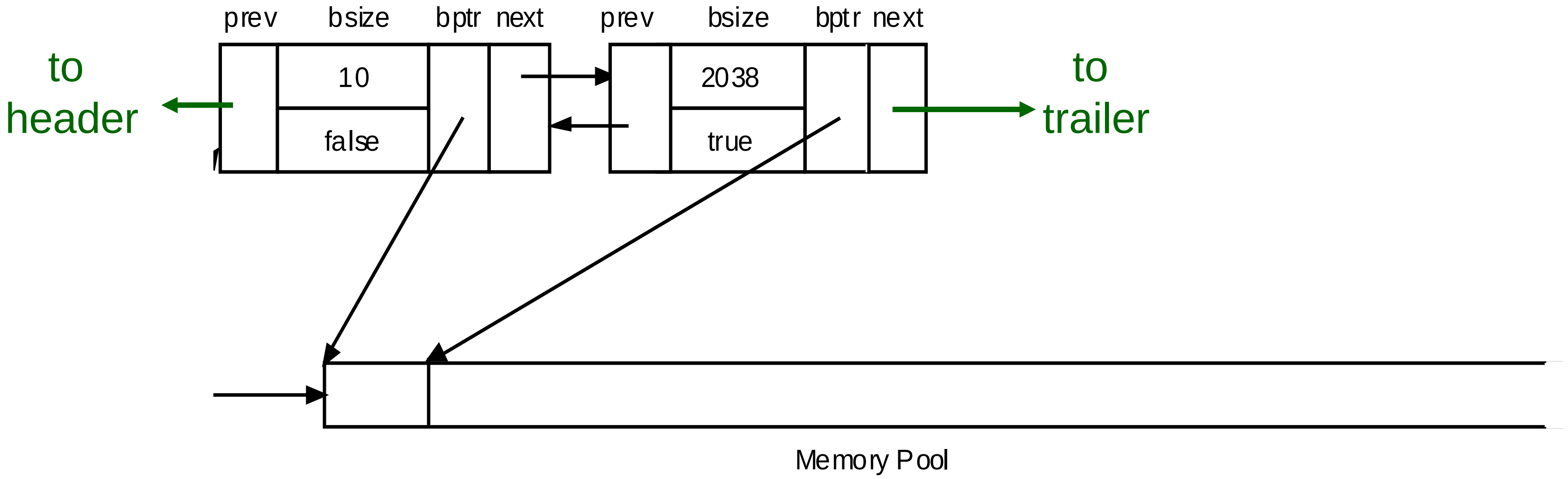
```
splitBlock(q,20);
```



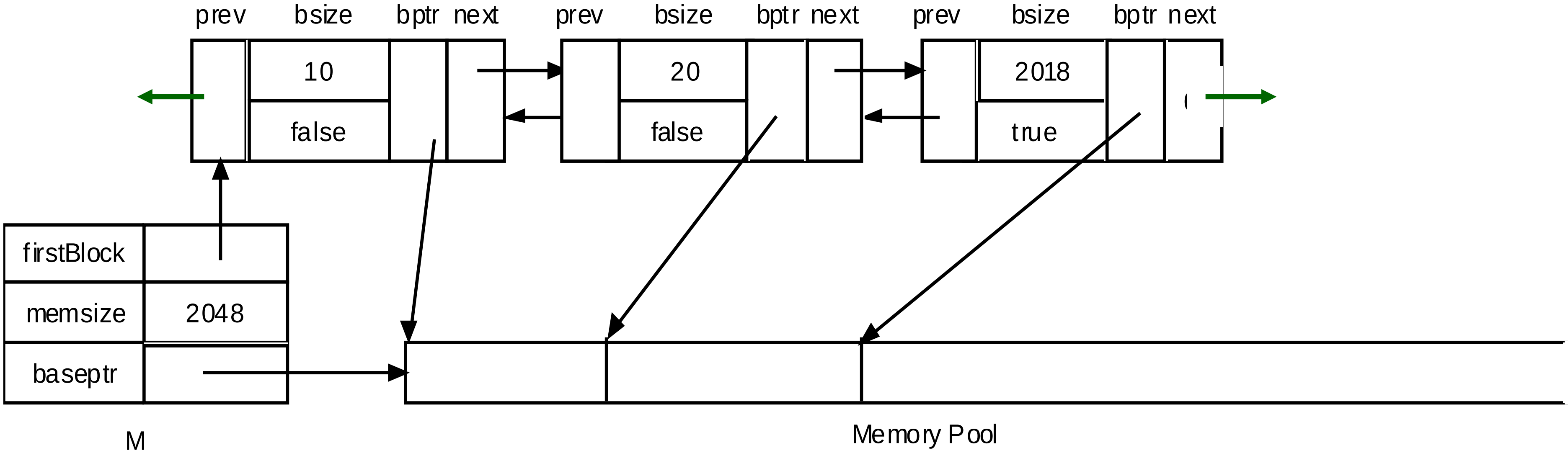


```
unsigned char *p1 = M.malloc(10);
```

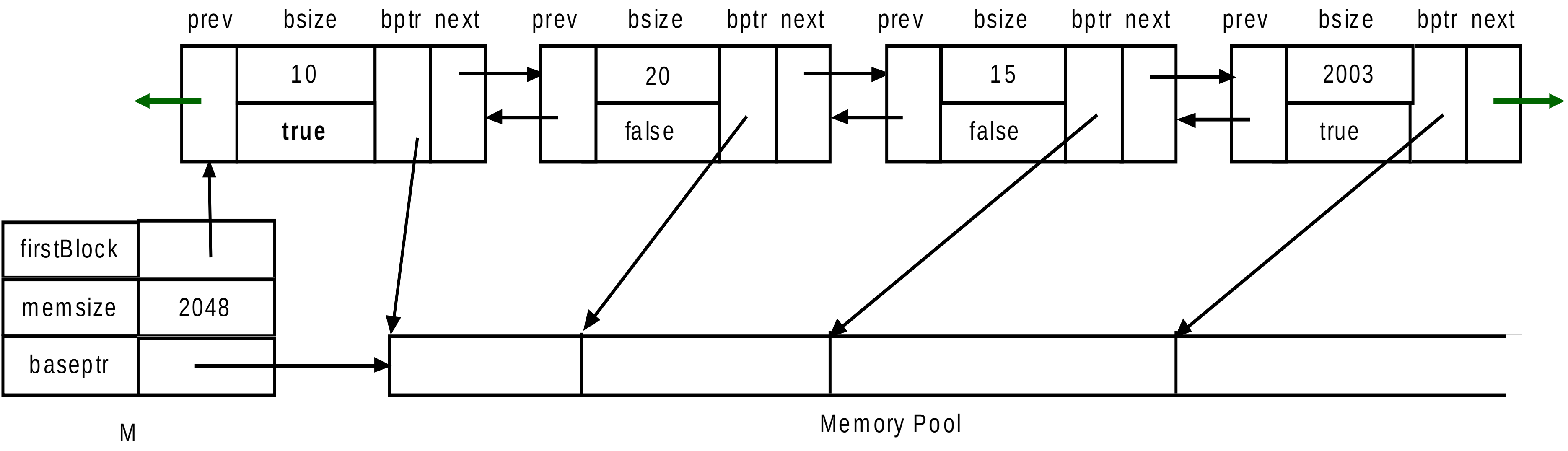




```
unsigned char *p2 = M.malloc(20);
```

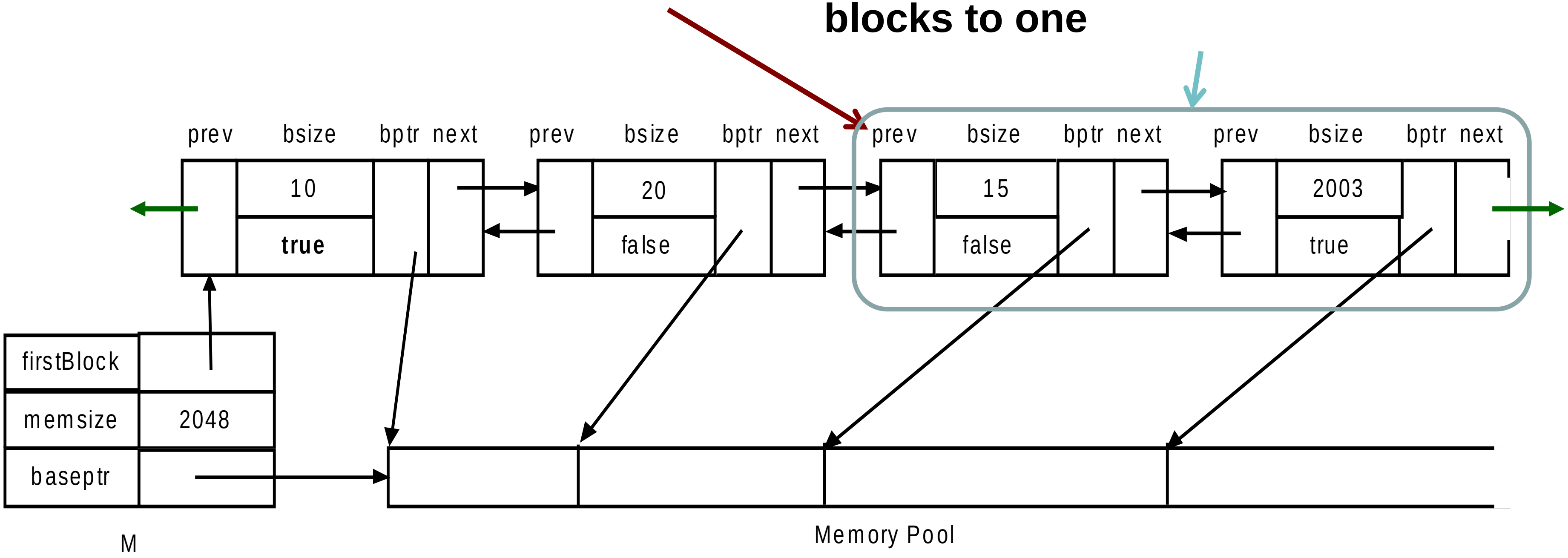


```
p1 = M.malloc(15);
```

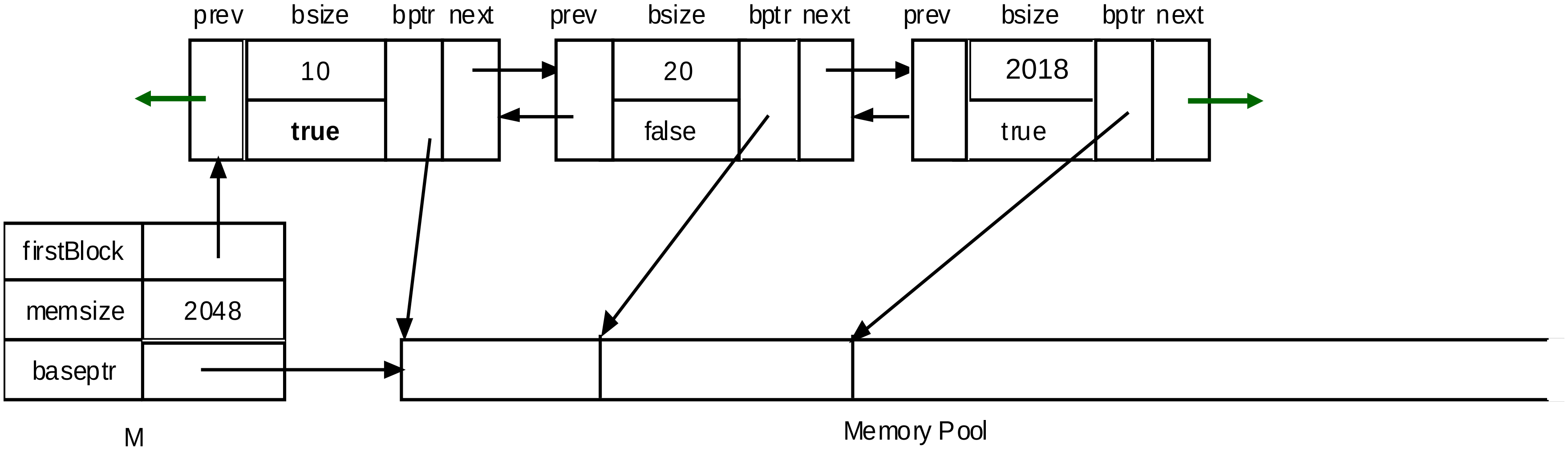


Block allocated to p1

When free is called on p1, we must merge the resulting consecutive free blocks to one



M.free(p1);



Testing Code

```
#include <iostream>
#include <cassert>
#include "MemoryManager.h"

const char * startlist =
      "\n---------BlockList start--------------\n"
const char * endlist =
      "\n---------BlockList end-------------\n"
int main()
{
   MemoryManager heaper(2048);
   cout << "heap initialized\n";

   cout << startlist;
   cout << heaper << endl;
   cout << endlist;
cout << "Doing first malloc:\n";
unsigned char * p1 = heaper.malloc(10);
cout << "malloc done\n";

cout << startlist;
cout << heaper << endl;
cout << endlist;

cout << "On to the second malloc\n";
unsigned char *p2 = heaper.malloc(20);
cout << "malloc done\n";

cout << startlist;
cout << heaper << endl;
cout << endlist;
cout << "Next free the first pointer\n";
heaper.free(p1);

cout << startlist;
cout << heaper << endl;
cout << endlist;

cout << "Now do a malloc for a block too big for "
     << "the initial open block\n";
p1 = heaper.malloc(15);
cout << "malloc done\n";

cout << startlist;
cout << heaper << endl; n\n";
cout << endlist;
```

```
   cout << "Next free the most recently "
        << "allocated pointer\n";
   heaper.free(p1);

   cout << startlist;
   cout << heaper << endl;
   cout << endlist;

   cout << "Next free the middle pointer\n";
   heaper.free(p2);

   cout << startlist;
   cout << heaper << endl;
   cout << endlist;

   return 0;
}
```

Test Output

```
heap initialized

-------------BlockList start------------------
[2048,free]
-------------BlockList end------------------

Executing p1 = malloc(10):
malloc done

-------------BlockList start------------------
[10,allocated]  -> [2038,free]
-------------BlockList end------------------

Executing p2 = malloc(20):
malloc done

-------------BlockList start------------------
[10,allocated]  -> [20,allocated]  -> [2018,free]
-------------BlockList end------------------
```

```
Executing free(p1):

-------------BlockList start------------------
[10,free]  -> [20,allocated]  -> [2018,free]
-------------BlockList end------------------

malloc for a block too big for the initial open block
Executing p1 = malloc(15)
malloc done

-------------BlockList start------------------
[10,free]  -> [20,allocated]  -> [15,allocated]  ->
[2003,free]
-------------BlockList end------------------
```

Next free the most recently allocated pointer (p1)

```
-------------BlockList start------------------
[10,free]  -> [20,allocated]  -> [2018,free]
-------------BlockList end------------------
```

Next free p2

```
-------------BlockList start------------------
[2048,free]
-------------BlockList end------------------
```