Heap Manager

Mohamed Gamal Fathy
"MemOo™"

Planning:

There are several requirements that the implementation of the
Heap manager must satisfy :

It must keep track somehow of the blocks of memory that have been allocated as well as the areas of memory that remain unallocated.

For example, in order to implement the *alloc.* operation, we must have the means to locate an unused area of memory of sufficient size in order to satisfy the request.

The approach taken in this project (ADT) is to use a singly-linked list to keep track of the free areas in the Heap.

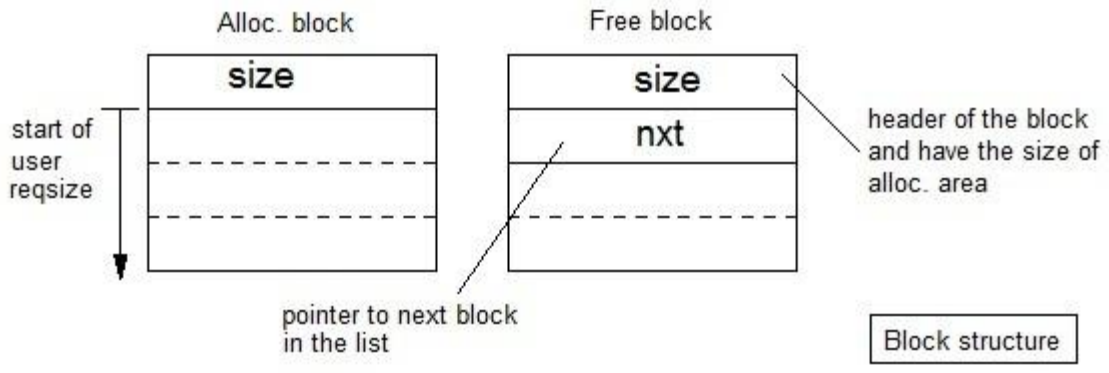
In addition to keeping track of the free areas, it is necessary to keep track of the size of each block that is allocated. This is necessary because the free operation takes only a pointer to the block of memory to be free.
I.e., the size of the block is *not* provided as an argument to the *memfree* function.

We keep the necessary information" size and next ptr " *in the Heap itself*. An area that has not been allocated to a user is available for use by the Heap itself. Specifically, the nodes of the linked list of free areas themselves occupy the free areas.

The heap is seen as constructed of a number of blocks. The block is a parameter of the heap and is passed to the heap constructor.

An area which has been allocated is said to be *allocated* . The first word of the first block in the area is used to keep track of the length of the area (in blocks). The remaining memory locations in the area are given up to the user.

An area which has not been allocated is said to be free . The first word of the first block in the area is used to keep track of the length of the area (in blocks). All of the free areas are linked together in a singly-linked list, known as the free list . The second word of the first block in the area contains a ptr to the next free area in the free list.



Implementation:

Class Design "Heap .h "

The public interface of the class comprises a constructor, the destructor, and the two member functions *memalloc* and *memfree*.
The Private of the class comprises the Block structure , number of chunks in the heap and pointers that we will use .

The Block structure contains the header"Size" has the size of the allocated area and a "union". In C++, the elements of a *union* occupy the same space. I.e., if the block is *allocated.* the "*data array"* of the block has been allocated to the user. On the other hand, if the block is free, the "nxt" field is used to point to the next Block of the free list.

The Block size is set to 16 bytes. Therefore, the size of an area is a multiple of 16 bytes. E.g., it is possible for the free list to contain many small areas. The choice of 16 is somewhat arbitrary. However, values between 8 and 16 are typical. In this implementation the size of a block (in bytes) is required to be at least sizeof(Block)+sizeof (*Block)= 8 Bytes.

Constructor and Destructor "Heap ()"

The constructor takes a single argument which specifies the desired size of the Heap manager in bytes. Initially, the entire Heap manager is one big chunk .

The **nchunk** variable records the total number of blocks in the Heap manager that are available to be allocated. The **Heapmem** variable points to the array of blocks that make up the Heap manager.
The array is itself dynamically allocated using operator new!

Finally, the **base** variable refers to an extra block that is used as the Base for the free list.
The nchunk is calculated by divide the heap manager size over block size.

The destructor is quite simple. It simply releases the heap manager that was dynamically allocated in the constructor.

Memalloc Function **“** void * memalloc (unsigned int reqsize) ”

Allocates a contiguous block of memory of size at least reqsize and returns a pointer of type void to the start address of the memory allocated. The programmer can cast the pointer it to any other data type.

The function begins by calculating the number of blocks required using the formula
nblock = (reqsize+ sizeof (block)+ sizeof(link)-1U) /sizeof (block);
I.e., enough storage is set aside to hold the requested number of bytes *plus* a header.

The function then traverses the linked list of free areas to find a free area that is large enough to satisfy the request . This is the so-called **first-fit allocation strategy** : It always allocates storage in the first free area that is large enough to satisfy the request.
If the search for a free area is unsuccessful, a **badalloc** exception is thrown.

An alternative to the first-fit strategy is **the best-fit allocation strategy** . In the best fit strategy, the Acquire function allocates storage from the free area the size of which matches most closely the requested size. Under certain circumstances, the best-fit strategy may prevent excessive fragmentation of the storage pool. However, the best-fit strategy requires that the entire free list be traversed. Since we are interested in the **fastest possible execution time**, the first-fit strategy is used here.

Otherwise, the variable **ptr** points the free area in which the allocation takes place and the variable **prevPtr** points to its predecessor in the singly-linked list. If the free area is exactly the correct size, it is simply unlinked from the free list and a pointer to the data array ”start ” of the area is returned.

Memfree Function **“**void memfree(void* allocptr)”

This function takes as its alone argument the address of the Data array ”start of alloc.area” that was previously obtained from the memalloc function. The memfree function begins by determining the block which corresponds to the given area and checking that this block is indeed a part of the heap manager.

The func. loop traverses the linked list of free areas and when it terminates, the following is true: The pointer **prevPtr** either points to the Base or it points at a free area the address of which is less than that of the area to be free. The pointer **ptr** is either zero or its points to a free area the address of which is greater than that of the area to be released. And **prevPtr** and **ptr** always point to adjacent elements of the linked list.

Testing :

Test code

```
  clock_t start = clock(); // start of count
for(long k=0;k<= 9999;k++)
{
  a=(double*)heap.memalloc(300);
  b=(double*)heap.memalloc(100);
  c=(double*)heap.memalloc(100);
  d=(double*)heap.memalloc(100);
  cout<<"a="<<a<<"\n"<<"b="<<b<<"\n"<<"c="<<c<<"\n"<<"d="<<d<<"\n";
  heap.memfree(a);
  heap.memfree(b);
  heap.memfree(c);
  e=(double*)heap.memalloc(524); //check bounce
  cout<<"e="<<e<<"\n";
  f=(double*)heap.memalloc(300);
  cout<<"f="<<f<<"\n";
}
   //Calculating the average time in nsec :
  cout<<"avg. time ="<<pow (10.0,5)*((double)clock()-start)/CLOCKS_PER_SEC<<"ns\n";
```

First,
we Allocate 4 Blocks each size of them 300,100,100,100 bytes respectively
and show on the screen its allocated addresses
then we free 3 of them 300,100,100 respectively and keep the last one
Second, we allocated 1 block with size equal the sum of last free blocks
due to aggregation function it can allocate in the start address of them
Third, we allocate the final block as there is not an empty space behind allocated block satisfy the required size so It would allocated after last allocated block in the heap not the last allocated block we allocate before .
finally,
we calculate the time average of the 6 allocation and 3 free function
We use < **time.h>** library to calculate the average time of functions ,first we put the requested code in a loop and calculates the time by using the func. **clock()**
and in the end we calculate the average time by using this formule :

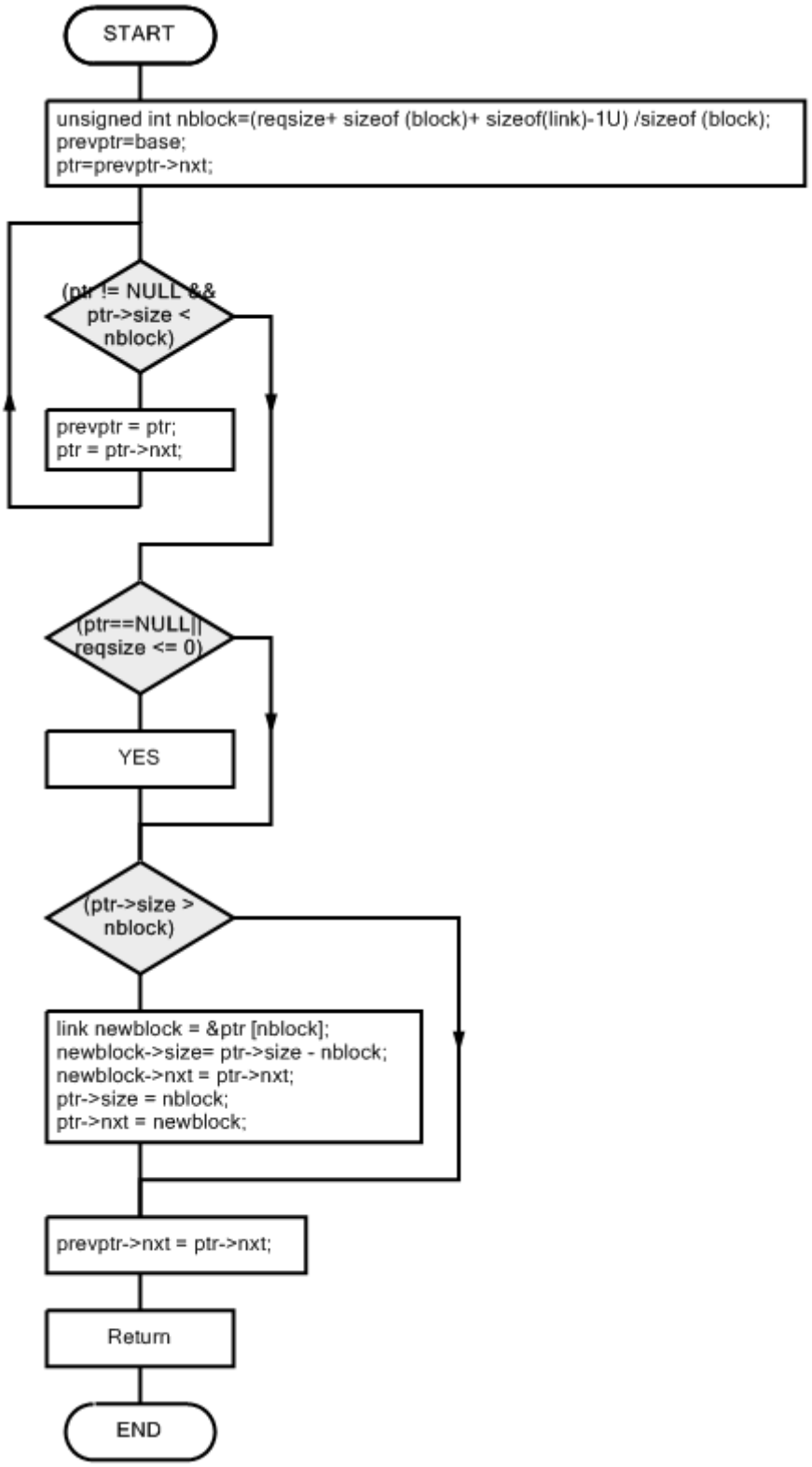
((double)clock()-start)/CLOCKS_PER_SEC.

we divide it by number of iterations of the loop to get the final avg time
and multiply by 10^9 to get time in nsec.

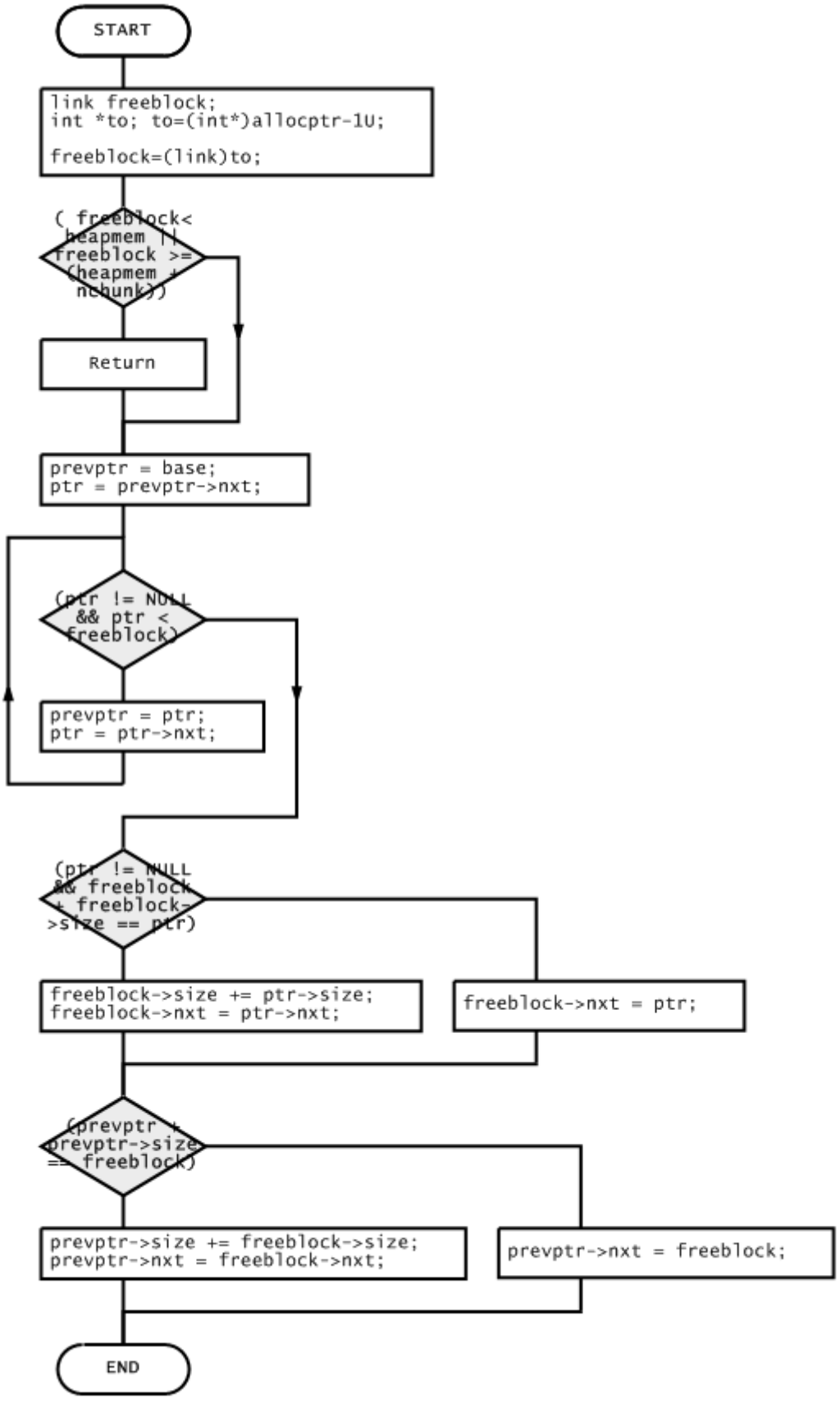
The **Running time** of the memalloc function is determined by the number of iterations of the loop on lines. All of the remaining statements in the function require a constant amount of time in the worst-case. The number of iterations of the loop is determined by three factors: the size of the free list, the size of the area requested, and the position in the free list of an area that is large to satisfy the request.
The running time of the memfree function is determined by the number of iterations required to find the position in the list at which to do the insertion. In practice, the free list is significantly shorter than *n"Heap size"*, and the running time varies accordingly

- memalloc func. Avg. time**=~ 200 ns**
- memfree func. Avg. time **=~ 100 ns**

Flow Charts :



Memalloc Function



Memfree Function