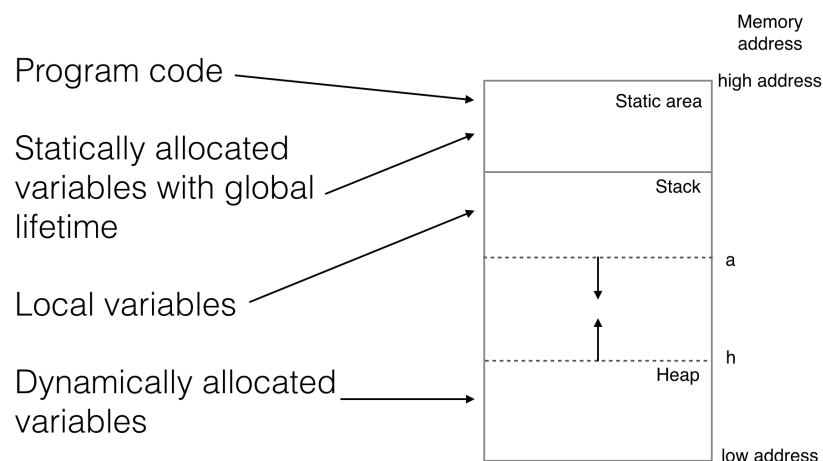


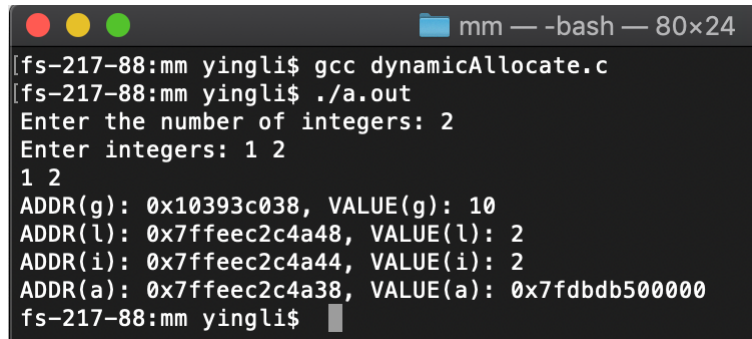
Memory Management

Composition

- The memory of a program consists of **four types of information**
 - **Program code**
 - Statically allocated **variable with global lifetime**
 - **Local variables** and **function calls**
 - **Dynamically allocated variables**
- The **memory is composed of a static area, a stack, and a heap**.
 - Although the stack is implemented as the stack data structure, but the **heap has nothing relevant to the heap data structure**. It is confusing especially if you know data structure. So it is better to clarify here heap in memory is not heap in data structure
- Generally, the **stack is in high memory** and grows down towards the heaps. The **heap is in low memory** and grows up towards the stack.
- In this figure, static area is on top of stack. But depending on the machine architecture, **static area can be either on top of stack or after heap**.



- Program static variable are in the static area. code and
- Local variables and function calls are stored in the stack.
- Dynamically allocated variables are stored in heap.
- Exercise
 - Where are the variables, g, l, i, and a in the following program stored in memory?
 - g in static area
 - l, i, and **a are in stack**
 - **Variables initialized by using malloc are stored in the stack and their values are addresses on the heap.**



```
mm — -bash — 80x24
[fs-217-88:mm yingli$ gcc dynamicAllocate.c
[fs-217-88:mm yingli$ ./a.out
Enter the number of integers: 2
Enter integers: 1 2
1 2
ADDR(g): 0x10393c038, VALUE(g): 10
ADDR(l): 0x7ffeec2c4a48, VALUE(l): 2
ADDR(i): 0x7ffeec2c4a44, VALUE(i): 2
ADDR(a): 0x7ffeec2c4a38, VALUE(a): 0x7fdbdb500000
fs-217-88:mm yingli$
```

```

#include <stdio.h>
#include <stdlib.h>

char g = 0xA; // '\n' in ASCII

int main () {
    int l, i, *a;

    printf("Enter the number of integers: ");
    scanf("%d", &l);
    a = (int*) malloc (l * sizeof(int));

    if (a == NULL) {
        printf("Error! memory not allocated");
        exit(0);
    }

    printf("Enter integers: ");
    for (i = 0; i < l; i++) {
        scanf("%d", a+i);
    }

    for (i = 0; i < l; i++) {
        printf("%d ", *a+i);
    }

    printf("%c", g);

    printf("ADDR(g): %p, VALUE(g): %d \n", &g, g);
    printf("ADDR(l): %p, VALUE(l): %d \n", &l, l);
    printf("ADDR(i): %p, VALUE(i): %d \n", &i, i);
    printf("ADDR(a): %p, VALUE(a): %p \n", &a, a);

    free(a);
    return 0;
}

```

- Demonstrate that `a` is in stack by adding the two lines of code after `free(a)`, and run.

```

[fs-217-88:mm yingli$ gcc dynamicAllocate.c
[fs-217-88:mm yingli$ ./a.out
Enter the number of integers: 2
Enter integers: 1 2
1 2
ADDR(g): 0x109f4c038, VALUE(g): 10
ADDR(l): 0x7ffee5cb4a48, VALUE(l): 2
ADDR(i): 0x7ffee5cb4a44, VALUE(i): 2
ADDR(a): 0x7ffee5cb4a38, VALUE(a): 0x7fd5b2500000
2
fs-217-88:mm yingli$

```

```

#include <stdio.h>
#include <stdlib.h>

char g = 0xA; // '\n' in ASCII

int main () {
    int l, i, *a;

    printf("Enter the number of integers: ");
    scanf("%d", &l);
    a = (int*) malloc (l * sizeof(int));

    if (a == NULL) {
        printf("Error! memory not allocated");
        exit(0);
    }

    printf("Enter integers: ");
    for (i = 0; i < l; i++) {
        scanf("%d", a+i);
    }

    for (i = 0; i < l; i++) {
        printf("%d ", *a+i);
    }

    printf("%c", g);

    printf("ADDR(g): %p, VALUE(g): %d \n", &g, g);
    printf("ADDR(l): %p, VALUE(l): %d \n", &l, l);
    printf("ADDR(i): %p, VALUE(i): %d \n", &i, i);
    printf("ADDR(a): %p, VALUE(a): %p \n", &a, a);

    free(a);

    a = &l;
    printf("%d\n", *a);

    return 0;
}

```

- If `a` is in heap, after invoking `free(a)`, it should be invalid to assign a value to `a` or print its value. However, `a = &l` and `printf("%d\n", *a)` are valid statements here, which demonstrate that `a` is in stack instead of in heap.
 - Besides, comparing `a`'s address and `a`'s value, it's obvious that `a` is in stack and the memory referenced by the address stored in `a` is in heap.
- Now, we know that local variables use stack space, dynamically allocated variables ask space from heap, and statically allocated global variables and programs use static area.
- So, note that [use global variables with caution](#) (avoid using too much static area space)
- Each program has fixed size of stack, although the actual size of stack frame is dynamic.
- Each function call push a stack frame into the stack for information like return address, return value, arguments, local variables, copies of registers (e.g., pointer to the address of the function in the program)

- Therefore, a recursive function may need more than one stack frame, and deep recursive functions will need much more stack space.
- From the perspective of memory complexity and space complexity, recursive functions can be expensive.
- Depending on the compiler, [deep recursion may cause stack overflow](#). Even it won't, the program may be inefficient.
- Show fibonacci.c. see the difference between $n = 5$ and $n = 50$
- The code in fibonacci.c implements fibonacci sequence in two ways: recursion and loop.

```
#include <stdio.h>
#include "my_timing.h"

/* recursive method: double recursion, variable n grows exponentially */
unsigned long FibonacciR(unsigned n) {
    if (n > 2)
        return FibonacciR(n-1) + FibonacciR(n-2);
    else
        return 1;
}

/* without recursion */
unsigned long Fibonacci(unsigned n) {
    if (n <= 1) {
        return n;
    }

    unsigned long curr = 1;
    unsigned long prev = 1;
    for (int i = 2; i < n; ++i) {
        unsigned long temp = curr;
        curr += prev;
        prev = temp;
    }

    return curr;
}

int main () {
    double t1, t2, t3;
    unsigned d;
    while (scanf("%d", &d) != 0) {
        printf("Entered: %d\n", d);
        t1 = get_time_sec();
        unsigned i = FibonacciR(d);
        t2 = get_time_sec();
        unsigned j = Fibonacci(d);
        t3 = get_time_sec();
        printf("FibonacciR number is: %u, Time: %f sec\n", i, t2 - t1);
        printf("Fibonacci number is: %u, Time: %f sec\n", i, t3 - t2);
    }

    return 0;
}
```

- Note: recursive functions can be implemented using loops.
- Below are the my_timing.h and corresponding my_timing.c files used in fibonacci.c.

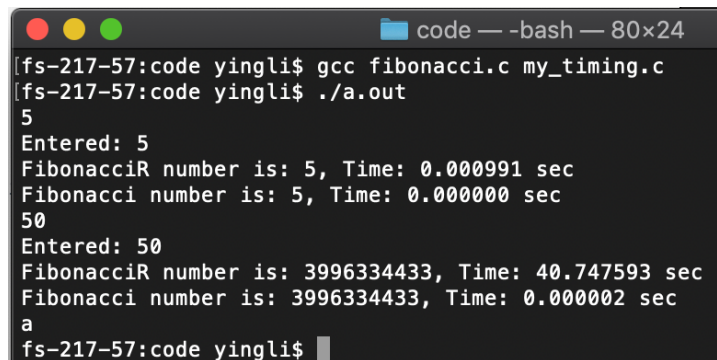
```
#ifndef MY_TIMING_H
#define MY_TIMING_H

// Return the time in seconds
// Note: it doesn't work to return a float - it must be a double.
double get_time_sec();

#endif

#include <sys/time.h> // for get_time_sec
#include <time.h>

// Return the time in seconds
// Note: it doesn't work to return a float - it must be a double.
double get_time_sec()
{
    struct timeval t;
    struct timezone tzp;
    gettimeofday(&t, &tzp);
    return t.tv_sec + t.tv_usec*1e-6;
} // end get_time_sec
```



```
code — -bash — 80x24
fs-217-57:code yingli$ gcc fibonacci.c my_timing.c
fs-217-57:code yingli$ ./a.out
5
Entered: 5
FibonacciR number is: 5, Time: 0.000991 sec
Fibonacci number is: 5, Time: 0.000000 sec
50
Entered: 50
FibonacciR number is: 3996334433, Time: 40.747593 sec
Fibonacci number is: 3996334433, Time: 0.000002 sec
a
fs-217-57:code yingli$
```

- The output of fibonacci.c demonstrated that the recursive implementation (FibonacciR) is more expensive than the loop implementation (Fibonacci), especially when the recursion is deep (after entering 50).

Stack Overflow

- Stack overflow happens when a program hits its stack space limit.
- When an **interpreted language hits the stack space limit** the interpreter generally **throws an exception and terminates**.
- **Compiled languages** depend upon the **operating system** to **determine when a stack overflow has occurred**.
- **Older operating systems without adequate memory management** would often **simply fail** as the stack grew over important code, causing an unrecoverable system-wide crash.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    int i;
    int bi = 0x05050505;
    int ai = 0x0A0B0C0D;
    unsigned char *ptr;

    ptr = (unsigned char*)&ptr;
    for (i=0; i<10000; i++) {
        printf( "%d: %p (%x)\n", i, (ptr+i), *(ptr+i));
    }

    return 0;
}
```

- **Modern operating systems** and memory systems **give each program its own virtual stack space**; **when** a program **overflows** its own stack of the **OS can isolate the program and terminate** it.
- Show task2.c. Segment fault, overflow the stack.

Heap Overflow

- Heap overflow happens when there is not available heap space that meets the required size.
- Heap is a consecutive space. Some areas are used, and others are available.
- **malloc** must obtain an **unused section** of the heap of the requested size and return the pointer to the beginning of that area.
- malloc needs to seek the available heap space that is large enough to the required memory size. If there is no large enough space, malloc returns null, and the heap overflow happens, **even if there are smaller available pieces of memory in the heap**.
- So, it is important to invoke free after done with the dynamically allocated memory. **free marks** the area of the heap that had been used by the **given pointer as unused**.

- Allocating memory takes time, especially when you allocate a large chunk of memory.
- Show memory.c example, run without comment out `free(other_array)`, run with `free(other_array)` commented out.

```

/**
 * memory.c
 * Learn about the layout of the heap.
 * Push the heap to its boundaries, and look at timing.
 */

#include <stdlib.h>
#include <stdio.h>
#include "my_timing.h"

int main() {
    int i;
    double t1, t2;

    // allocate space for n * subn ints
    unsigned long n = 100000;
    unsigned long subn = 10000000;
    unsigned long int other_n = n * subn;
    int **array;
    array = (int **)malloc( n * sizeof(int *) );

    t1 = get_time_sec();
    for (i = 0; i < n; i++) {
        array[i] = (int *)malloc(subn * sizeof(int) );
    }
    t2 = get_time_sec();
    printf( "Allocated %lu arrays of %lu ints in %f seconds\n",
           n, subn, t2-t1 );

    for (i=0; i < n; i++) {
        free( array[i] );
    }
    free(array);

    int num_times = 35;
    int *other_array = NULL;
    t1 = get_time_sec();
    for (i = 0; i < num_times; i++) {
        other_array = (int *)malloc( other_n * sizeof(int) );
        if (other_array == NULL)
            printf( "Failed to allocate large array at iteration %d\n", i );
        //free(other_array);
    }
    t2 = get_time_sec();
    if (other_array != NULL)
        printf( "Allocated one array of %lu ints %d times in %f seconds\n",
               other_n, num_times, t2-t1 );
}

```

```
$ ./a.out
Allocated 100000 arrays of 100000000 ints in 0.109065 seconds
a.out(30746,0x7fffc580c3c0) malloc: *** mach_vm_map(size=4000000000000) failed
(error code=3)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
Failed to allocate large array at iteration 34
```

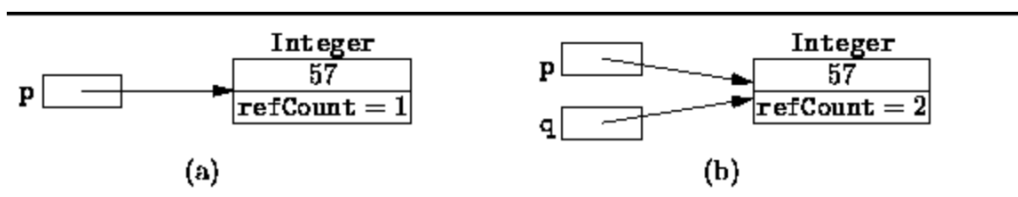
- The output indicates that allocating memory takes time, and allocating large pieces of memory takes more time.
- It also demonstrated that without invoking free appropriately, the program may run into heap overflow.
- When heap overflow happens or while the program is looking for available space, the computer will trigger **garbage collection**.

Garbage Collection

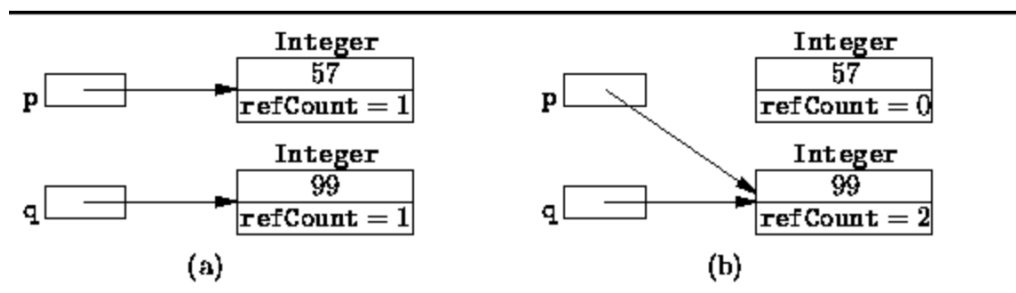
- **Garbage** is any block of **heap memory that cannot be accessed by the program**.
- It includes the memory that **was allocated but not free after use**, and the pieces of **memory that not large enough to meet the allocating requests**.
- Garbage collection needs to **identify garbage** and then **make garbage memory available**.
- A separate process running in the background to reclaim unused heap areas for later use by the program.
- How does a garbage collection system know when an object is no longer in use?
- When should garbage collection occur?
- There are many different strategies in use.
- Here are three general strategies: **reference counting**, **mark-sweep**, and **copy collection**.
- Specific garbage collection systems may be hybrids.
- All these strategies leverage linked list to manage memory. All **unused nodes are in a singly linked list**.

Reference Counting

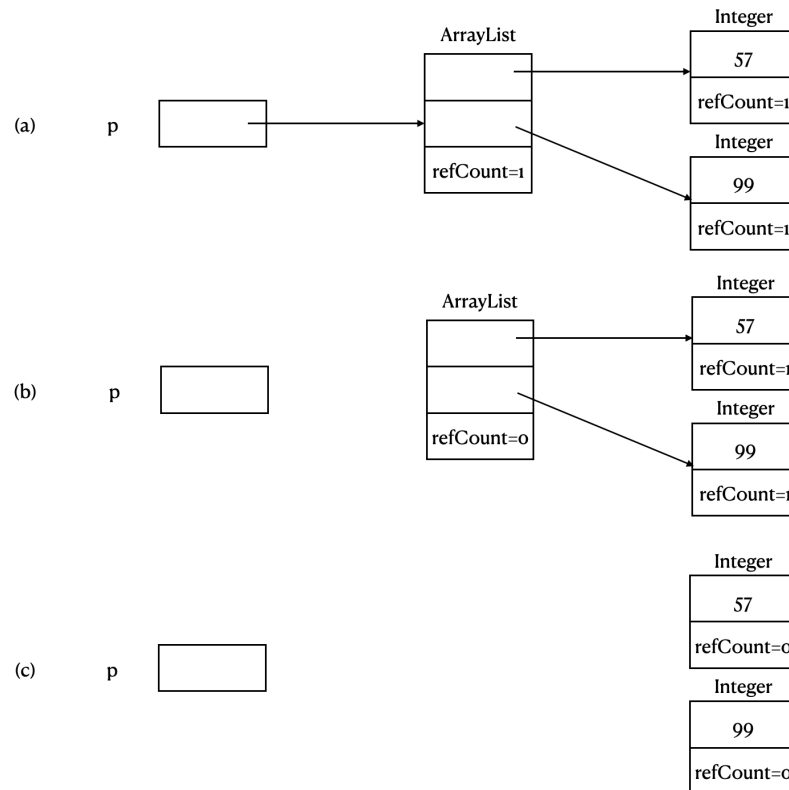
- **Each node has a field to count** the number of references to that node.
- **Assignments involving references must increment the counter** of the node specified by the r-value.
 - E.g. `Integer p = new Integer(57); Integer q = p;`



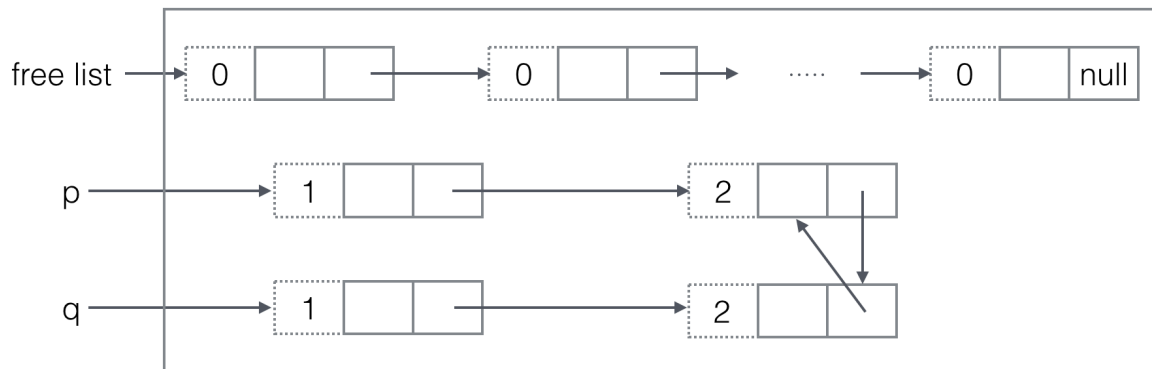
- **Sometimes**, when assigning, it also **needs to decrement the counter of another node**.
 - E.g. `Integer p = new Integer(57); Integer q = new Integer(99); p = q;`



- Example: `ArrayList<Integer> p = new ArrayList<Integer>(); p.add(new Integer(57)); p.add(new Integer(99));` What the memory for l looks like? What if I assign `p = null`? what the memory looks like?



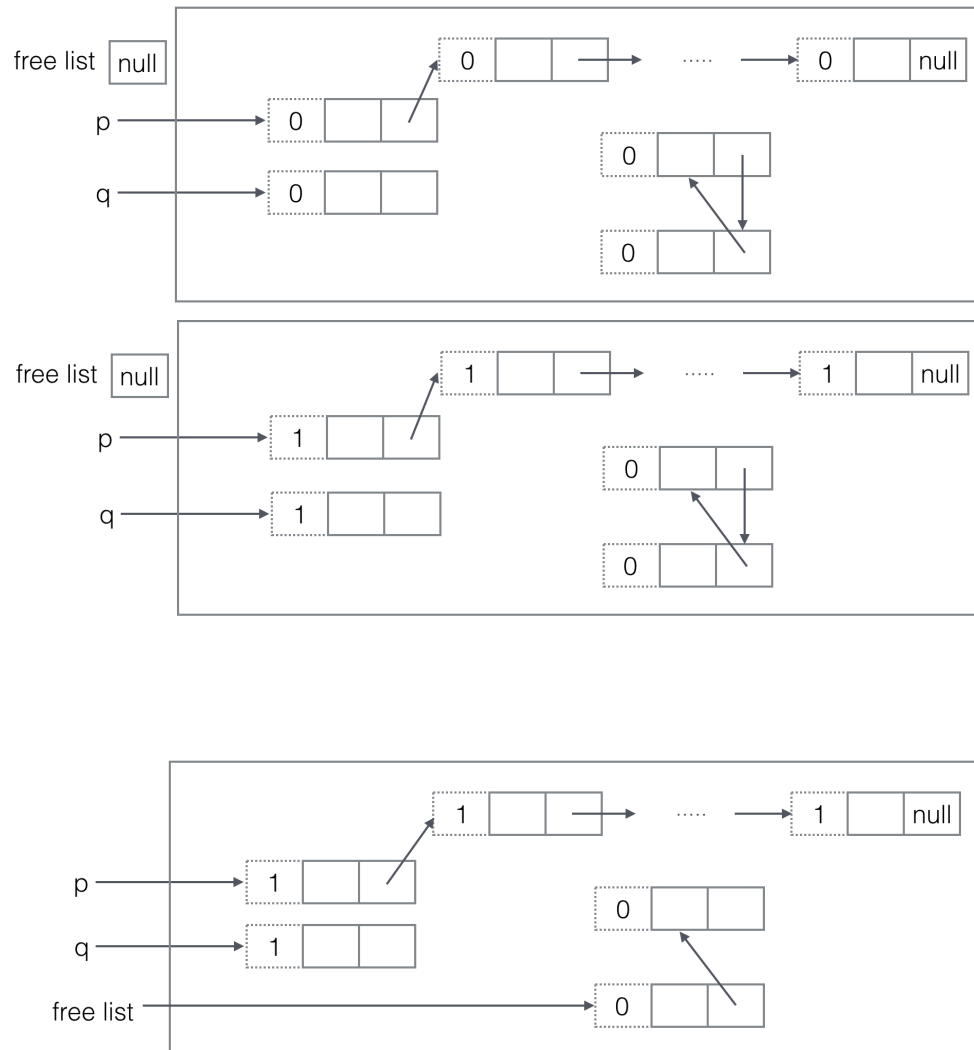
- Anytime the program **deletes a reference** to a node, **or the reference's lifetime ends**, the program has to **decrement the reference counter** of the node.
- If a **counter becomes zero**, because of either an assignment or a deletion, then the **program must travers any children it references** and **see if their counters are now also zero**. Any **nodes with zero counters** go back to the free list.



- REF: <http://www.brpreiss.com/books/opus5/html/page421.html>
- So far, reference counting looks like a good idea. However, it does not always work.
- Advantage: Does not require time in large chunks to clean up memory.
- Disadvantage: Cannot detect inaccessible **rings of nodes**

Mark-sweep

- **Mark-sweep** was the **first garbage collection** algorithm to be developed that is **able to reclaim cyclic data structure**.
- When using marking-sweep, **unreferenced memory blocks are not reclaimed immediately**. Instead, garbage is allowed to **accumulate until all variable memory has been exhausted**.
- Mark-sweep consists of **two phases**: the **mark phase** and the **sweep phase**.
- Instead of a counter, **each node has a single boolean that is initially zero, which mean unmarked**.
- The **mark phase** starts with each reference in the active symbol table, follow that reference through memory, setting each visited node's boolean marker to 1.
- **Once the reference traversal is complete, run through the entire heap and out any node who's marker is 0 back to the free list**. This is the sweep phase.



- Mark-sweep **trace out the memory blocks that are in use**. So it is **able to correctly identify and collect garbage** even in the presence of **reference cycles**.
- The **issue** of mark-sweep is that when the algorithm is triggered, the **execution of the program is suspended temporarily**. Once all unreferenced objects have been reclaimed, the normal execution of the program can resume.

Fragmentation

- **Fragmentation is a phenomenon** that occurs **in a long-running program** that has **undergone garbage collection several times**.
- The **problem** is that **memory blocks tend to become spread out** in the heap. **Marked nodes end up being separated by many, small unmarked nodes**.
- This **leads to** that it may become **impossible to allocate memory for a variable**. Although there may be sufficient unused memory, the **unused memory is not contiguous**. Since **variables typically occupy consecutive memory locations**, it is impossible to allocate storage.

- The **mark-sweep does not address fragmentation**. Even after reclaiming the storage, the heap may still be too fragmented to allocate the required amount of space.

Copy Collection

- Copy collection is also called **defragment collection**.
- When using copy collection, the **heap is divided into two separate regions**.
- At any point in time, all dynamically **allocated blocks reside in only one of the two regions**. We call this region **active region**. the other region is unoccupied, which is called **inactive region**.
- When the **memory in the active region is exhausted**, the **program is suspended and garbage collection is invoked**.
- Copy collection algorithm **copies** the memory block of all **active references from the active region to the inactive region**.
- **As** each memory block is **copied**, the **reference is updated to the new location**.
- **After the copying is completed**, the **active and inactive region exchange their roles**.
- **Since** the copy collection algorithm **copies only the memory block of active references**, the **garbage blocks are left behind**.
- The **memory space occupied by the garbage is reclaimed all at once when the active region becomes inactive**.
- **As** the copy collection algorithm **copies blocks from active region to inactive region**, it **stores these blocks in contiguous memory locations**. Therefore, this algorithm **automatically defragments the heap**.

