System Level Programming

Software College of SCU

Week09

Unit 6. Memory Layout and Allocation

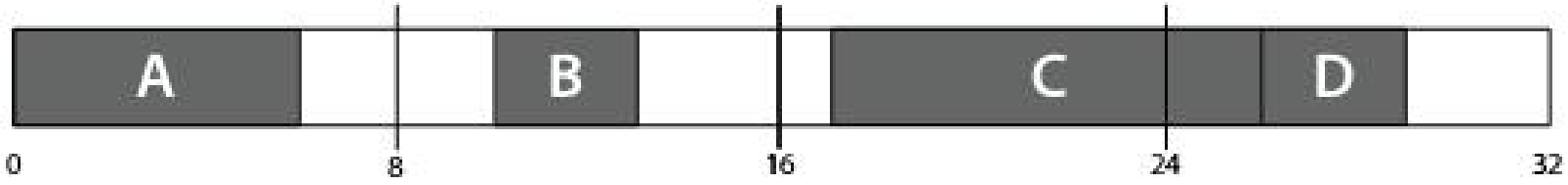
- 6.1 Several Uses of Memory
- 6.2 Memory Bugs
- 6.3 Garbage Collection

6.3 Garbage Collection

- 6.3.1 Introduction
- 6.3.2 Classical GC algorithms
- 6.3.3 Summary

6.3.1 Introduction (1/12)

- How do we keep track of memory allocation?
  - Use bitmap
    - an array of bits, one per allocation chunk



Memory regions

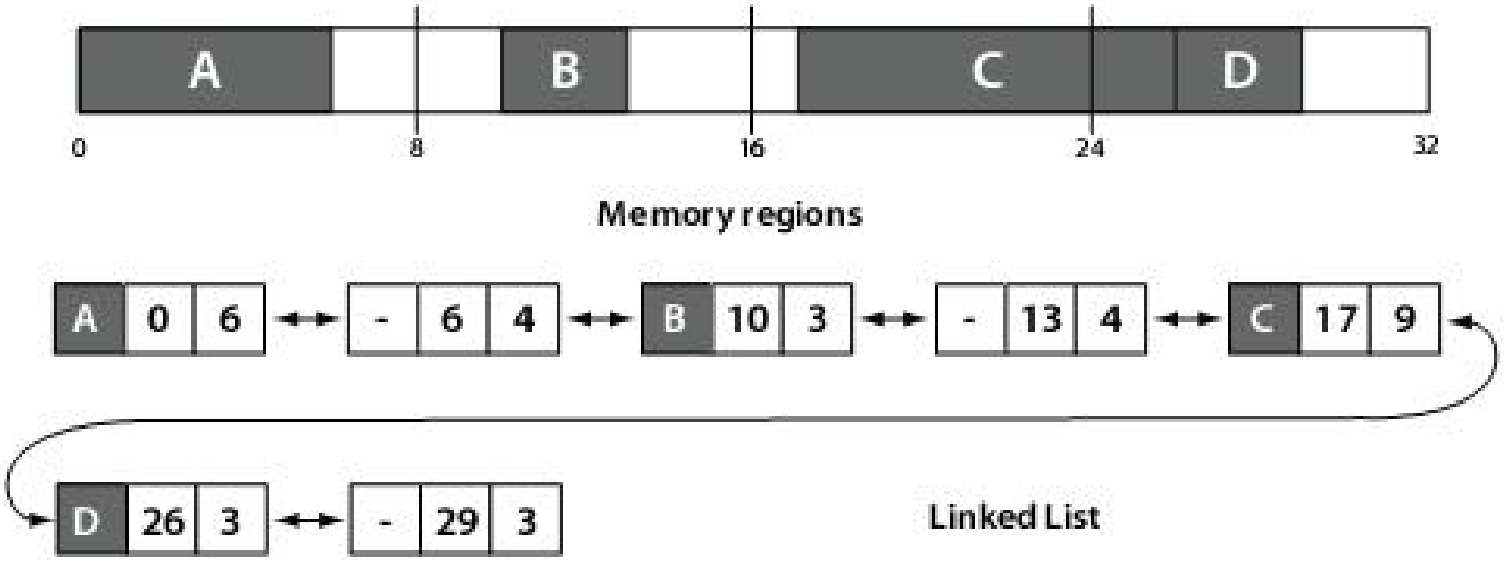
1111 1100 0011 1000 0111 1111 1111 1000

Bitmap

ure 7.1: A bitmap can store whether a chunk of memory is allocated or free.

6.3.1 Introduction (2/12)

- How do we keep track of memory allocation?
  - Use linked list
    - stores contiguous regions of free or allocated memory.



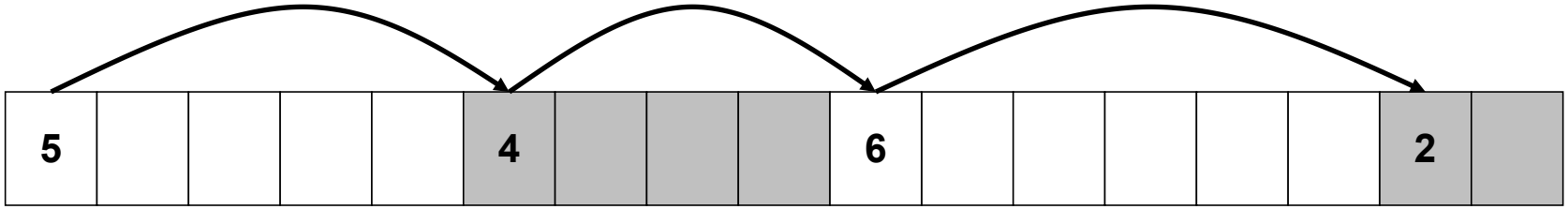
igure 7.2: A linked list can store allocated and unallocated regions.

6.3.1 Introduction (3/12)

- Dynamic Memory Management
  - Explicit Memory Management (EMM)
  - Automatic Memory Management (AMM)

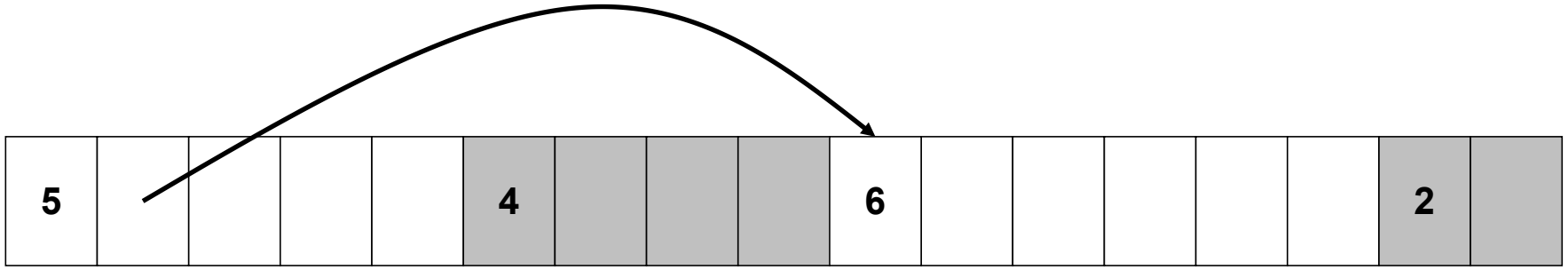
6.3.1 Introduction (4/12)

- EMM: Keeping Track of Free Blocks
  - Method 1: Implicit list using lengths -- links all blocks(隐式空闲链表)

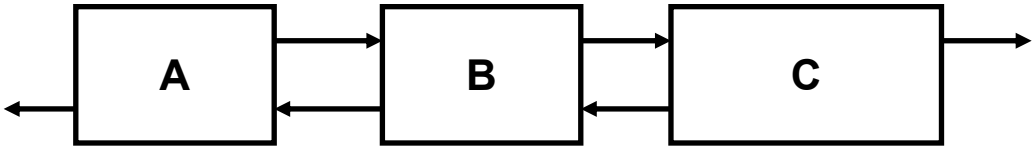


6.3.1 Introduction (5/12)

- EMM: Keeping Track of Free Blocks
  - Method 2: Explicit list among the free blocks using pointers within the free blocks



- Typically doubly linked



6.3.1 Introduction (6/12)

- Deallocation
  - Coalesce:To combine two or more nodes into one.

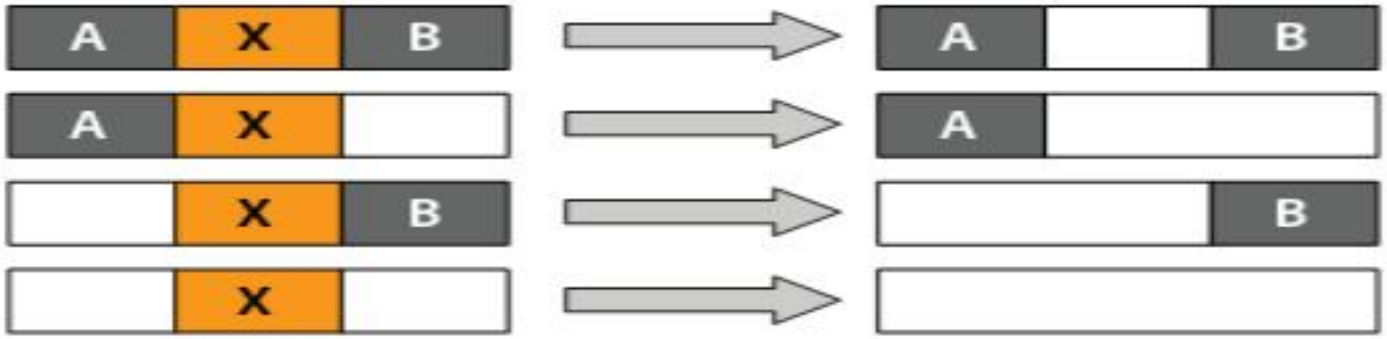
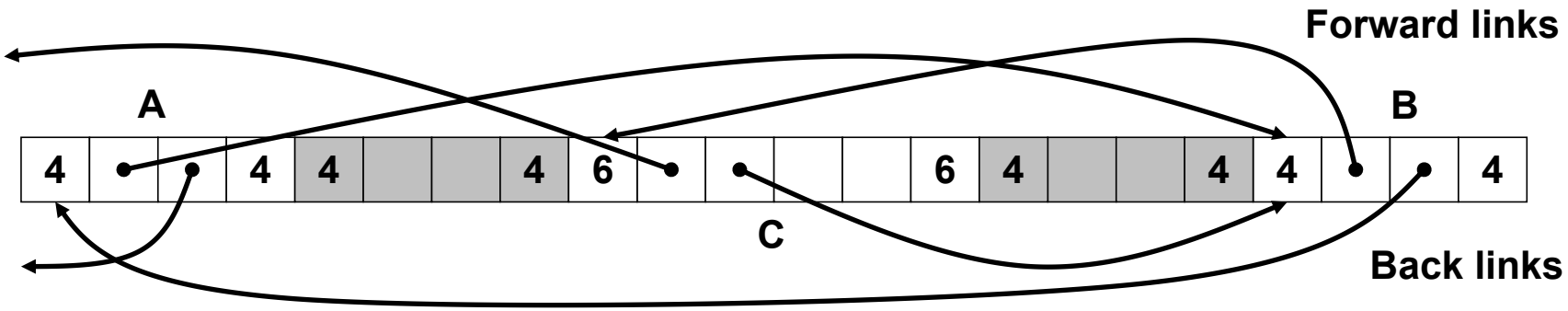


Figure 7.3: Coalescing free nodes on deallocation.

6.3.1 Introduction (7/12)

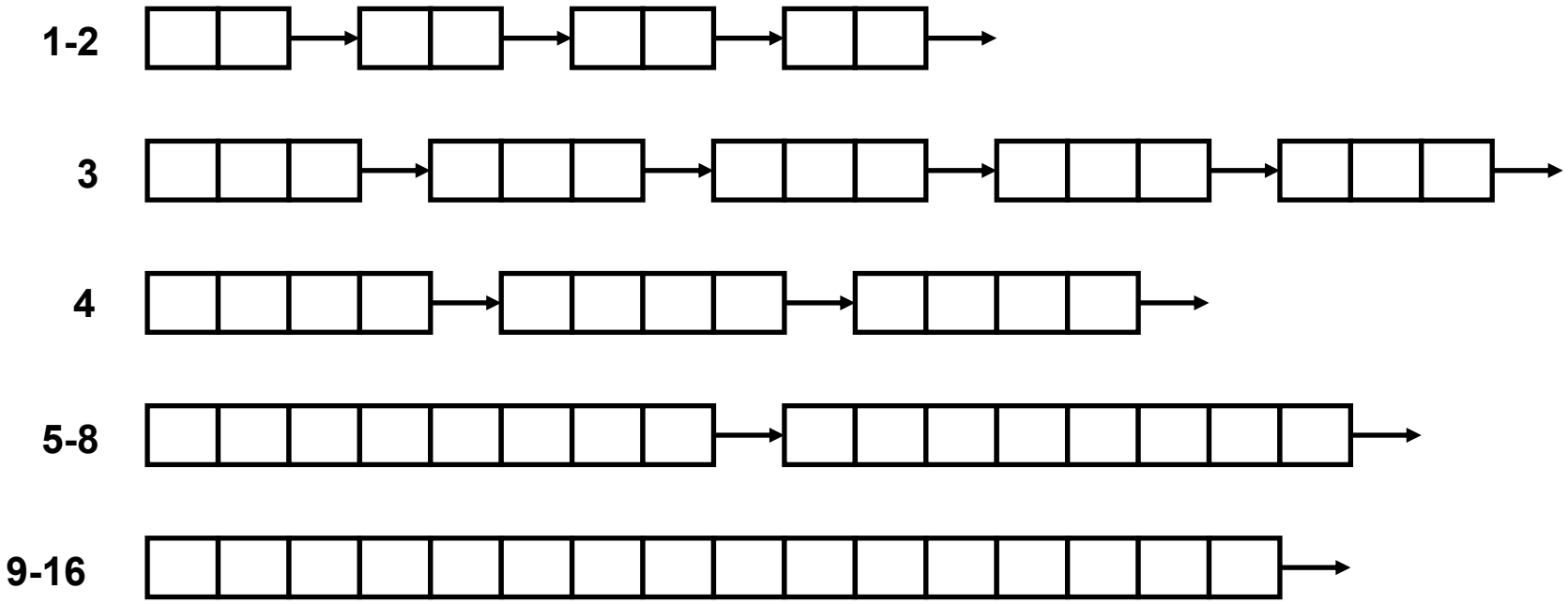
- EMM
  - Still need boundary tags(边界标记) for coalescing



  - It is important to realize that links are not necessarily in the same order as the blocks

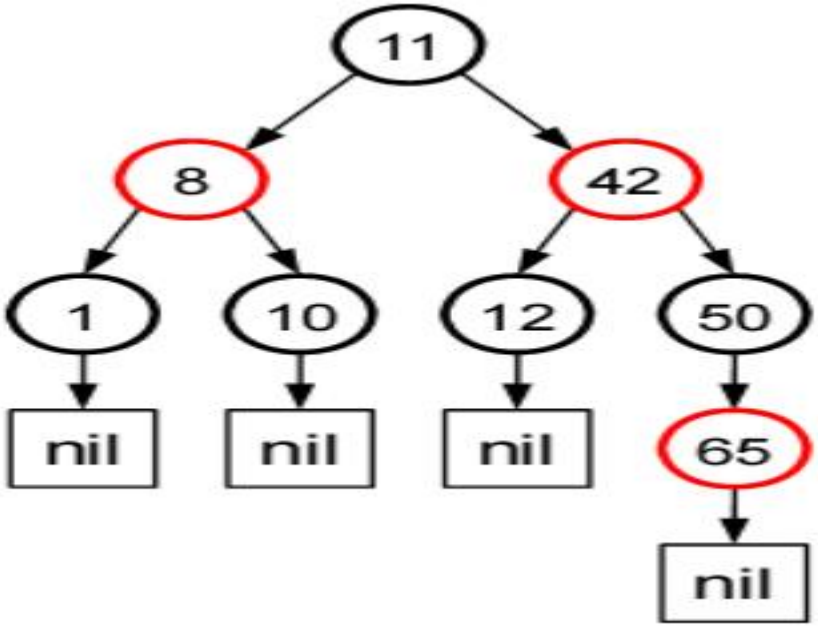
6.3.1 Introduction (8/12)

- EMM: Keeping Track of Free Blocks
  - Method 3: Segregated free lists (分离的空闲链表)
    - Different free lists for different size classes



6.3.1 Introduction (9/12)

- EMM: Keeping Track of Free Blocks
  - Method 4: Blocks sorted by size
    - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key



6.3.1 Introduction (10/12)

- Why Automatic Memory Management?
  - Storage management is still a hard problem in modern programming
  - C and C++ programs have many storage bugs
    - forgetting to free unused memory
    - dereferencing a dangling pointer
    - overwriting parts of a data structure by accident
    - and so on...
  - Storage bugs are hard to find
    - a bug can lead to a visible effect far away in time and program text from the source

6.3.1 Introduction (11/12)

- AMM: Garbage collection
  - automatic reclamation of heap-allocated storage
  - Implicit Memory Management -- application never has to free
  - Lazy processing: blocks are reorganized only if needed

```
void foo() {
  int *p = malloc(128);
  return; /* p block is now garbage */
}
```

6.3.1 Introduction (12/12)

- AMM: Garbage collection
  - Invented 1958 by John McCarthy for Lisp
  - Common in functional languages, scripting languages, and modern object oriented languages:
    - Lisp, Java, Perl, …

6.3 Garbage Collection

- 6.3.1 Introduction
- 6.3.2 Classical GC algorithms
- 6.3.3 Summary

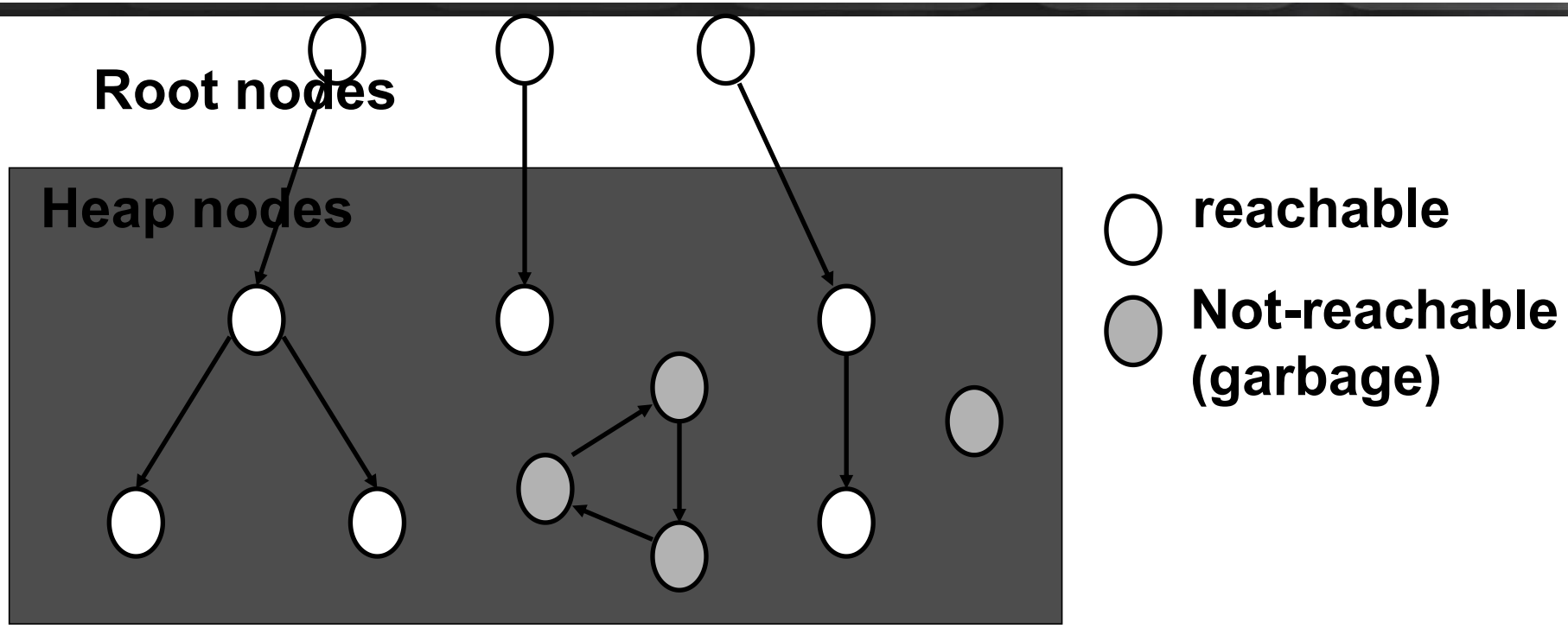
6.3.2 Classical GC algorithms

- 6.3.2.1 Basic concepts
- 6.3.2.2 Mark and sweep collection
- 6.3.2.3 Copying collection
- 6.3.2.4 Reference Counting
- 6.3.2.5 Generational garbage collection

6.3.2.1 Basic concepts (1/2)

- We view memory as a directed graph
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called root nodes (e.g. registers, locations on the stack, global variables)

6.3.2.1 Basic concepts (2/2)



- A node (block) is reachable if there is a path from any root to that node. Non-reachable nodes are garbage (never needed by the application)

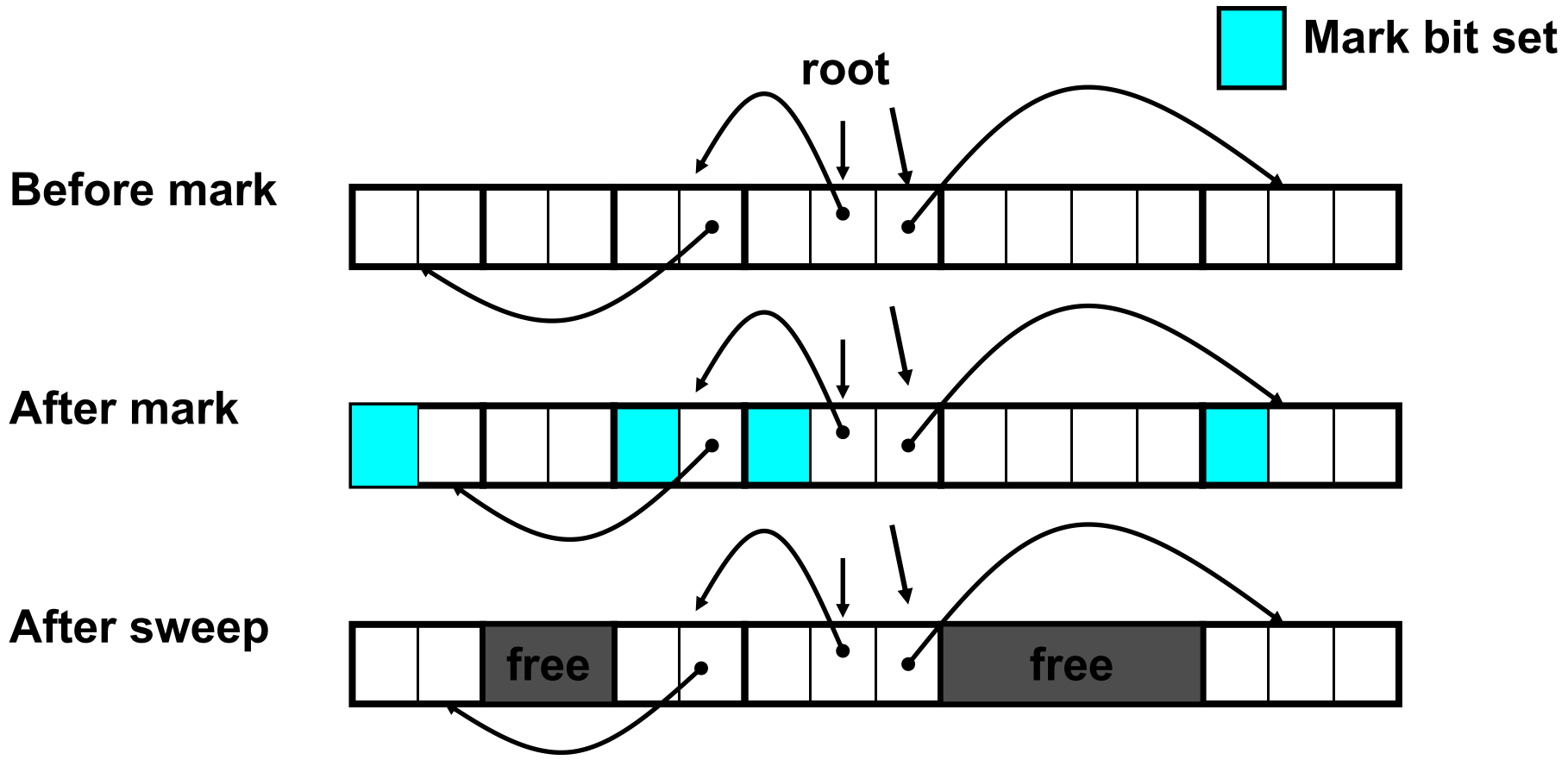
6.3.2 Classical GC algorithms

- 6.3.2.1 Basic concepts
- 6.3.2.2 Mark and sweep collection
- 6.3.2.3 Copying collection
- 6.3.2.4 Reference Counting
- 6.3.2.5 Generational garbage collection

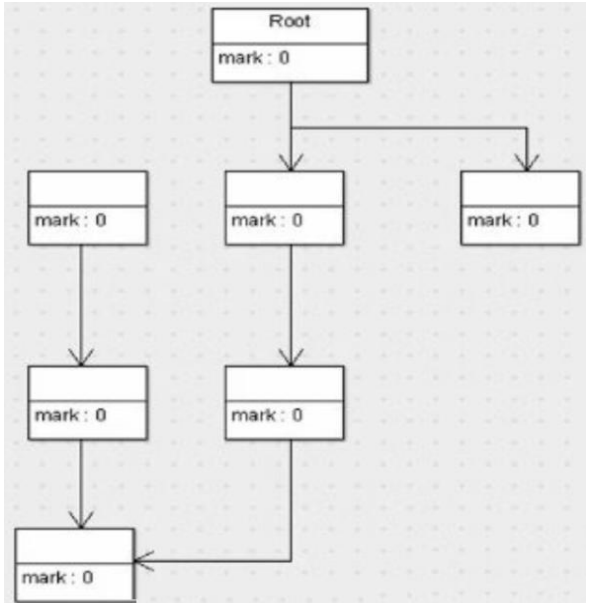
6.3.2.2 Mark and Sweep Collecting (1/10)

- Mark and Sweep Collecting 标记清除法(McCarthy, 1960)
  - Can build on top of malloc/free package
    - Allocate using malloc until you “run out of space”
  - When out of space:
    - Use extra mark bit in the head of each block
    - Mark: Start at roots and sets mark bit on all reachable memory
    - Sweep: Scan all blocks and free blocks that are not marked
  - Conservative/保守的 Garbage Collector
    - 有些是Garbage，但是可能没有标识出来

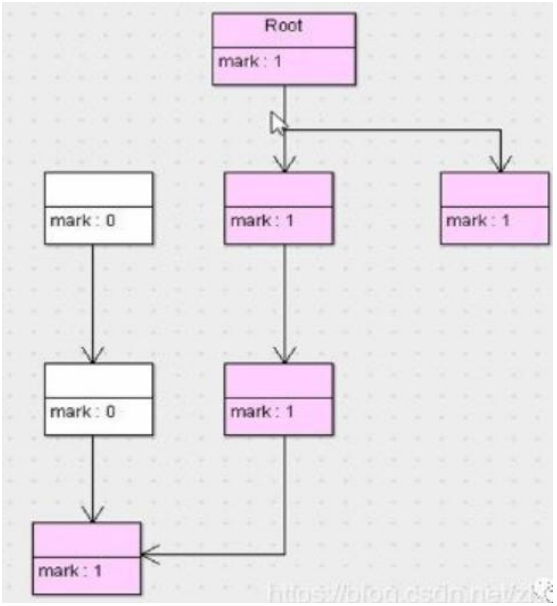
6.3.2.2 Mark and Sweep Collecting (2/10)



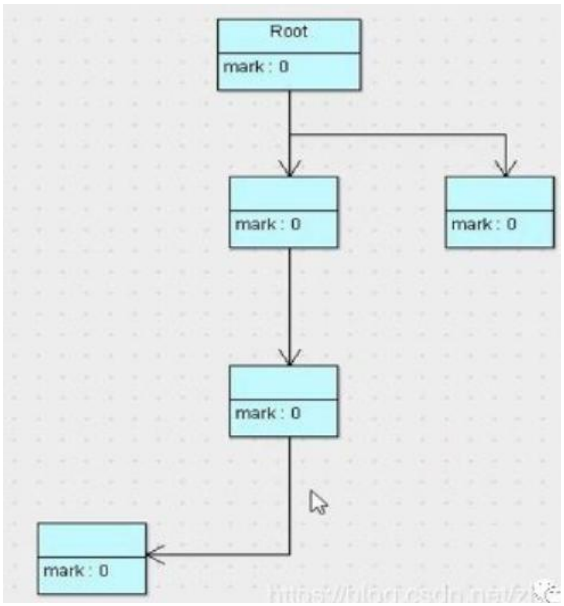
6.3.2.2 Mark and Sweep Collecting (3/10)



Before GC



GCing



After GC

6.3.2.2 Mark and Sweep Collecting (4/10)

- At the top level the algorithm in C like Pseudocode looks as follows:

```
void GC()
{
        HaltAllProcessing();
        ObjectCollection roots = GetRoots();
        for(int i = 0; i < roots.Count(); ++i)
                Mark(roots[i]);
        Sweep();
}
```

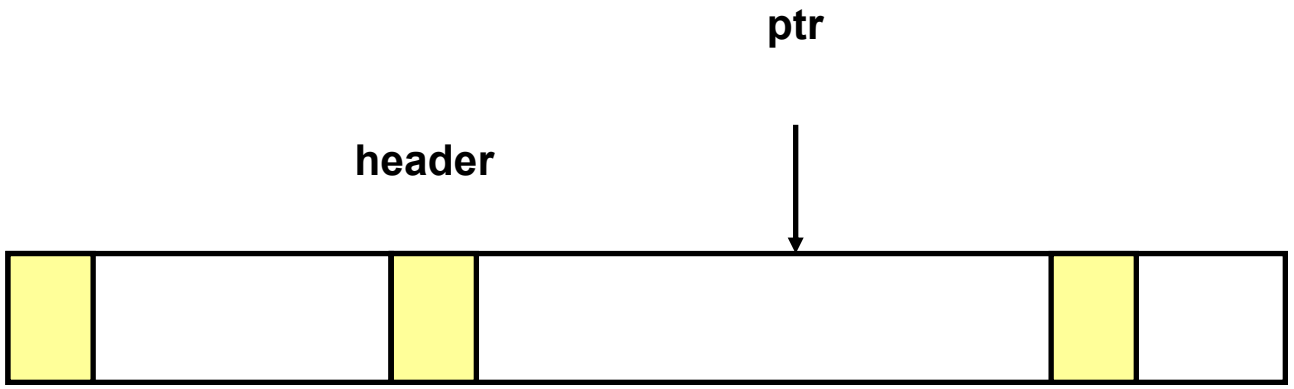
- It has three phases, enumeration of roots, marking all object references starting from the root and finally sweeping them.

6.3.2.2 Mark and Sweep Collecting (5/10)

- Root Enumeration
  - The root enumeration lists all references held in registers, global or static fields, local variables on the stack, function arguments on stack, etc…
  - The runtime system needs to provide ways for the GC to collect the list of roots.
    - E.g. in the case of .NET maintains the information on the active roots and provides APIs to the GC to enumerate them.

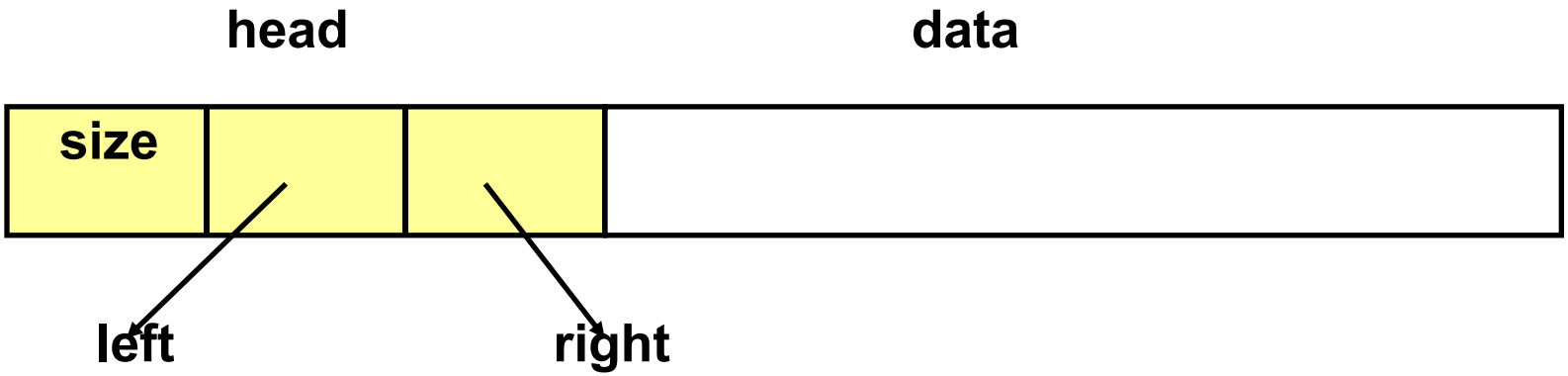
6.3.2.2 Mark and Sweep Collecting (6/10)

- A conservative Mark and Sweep collector for C programs
  - Is_ptr() determines if a word is a pointer by checking if it points to an allocated block of memory.
    - last 2 bits are non-zero => can't be a pointer
    - integer is not in allocated heap range => can't be a pointer
  - But, in C pointers can point to the middle of a block.



6.3.2.2 Mark and Sweep Collecting (7/10)

- So how do we find the beginning of the block?
  - Can use balanced tree to keep track of all allocated blocks where the key is the location
  - Balanced tree pointers can be stored in header (use two additional words)



6.3.2.2 Mark and Sweep Collecting (8/10)

Mark using depth-first traversal (深度优先遍历) of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return        // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // mark all children
     mark(p[i]);
   return;
}
```

Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

6.3.2.2 Mark and Sweep Collecting (9/10)

- Advantage
  - The advantage of mark and sweep is that you may never have to run it (it only runs when the heap is full).
  - The other advantage is that it finds and frees all unused memory blocks.

6.3.2.2 Mark and Sweep Collecting (10/10)

- Disadvantage
  - Because the Mark and Sweep cycle locks out other processing while it runs, the GC can make a noticable dent in a programs performace.
  - Fragmentation

6.3.2 Classical GC algorithms

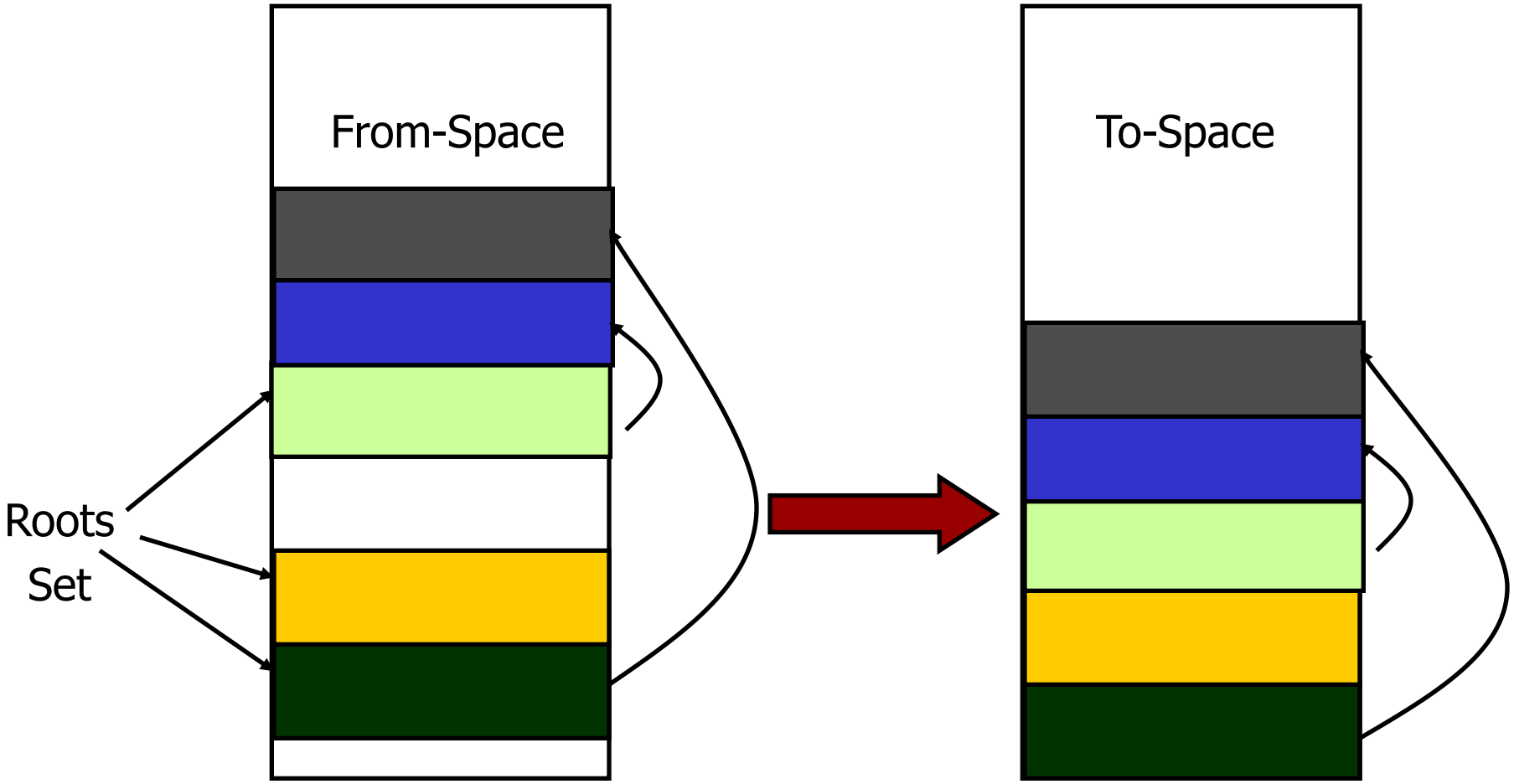
- 6.3.2.1 Basic concepts
- 6.3.2.2 Mark and sweep collection
- 6.3.2.3 Copying collection
- 6.3.2.4 Reference Counting
- 6.3.2.5 Generational garbage collection

6.3.2.3 Copying collection (1/5)

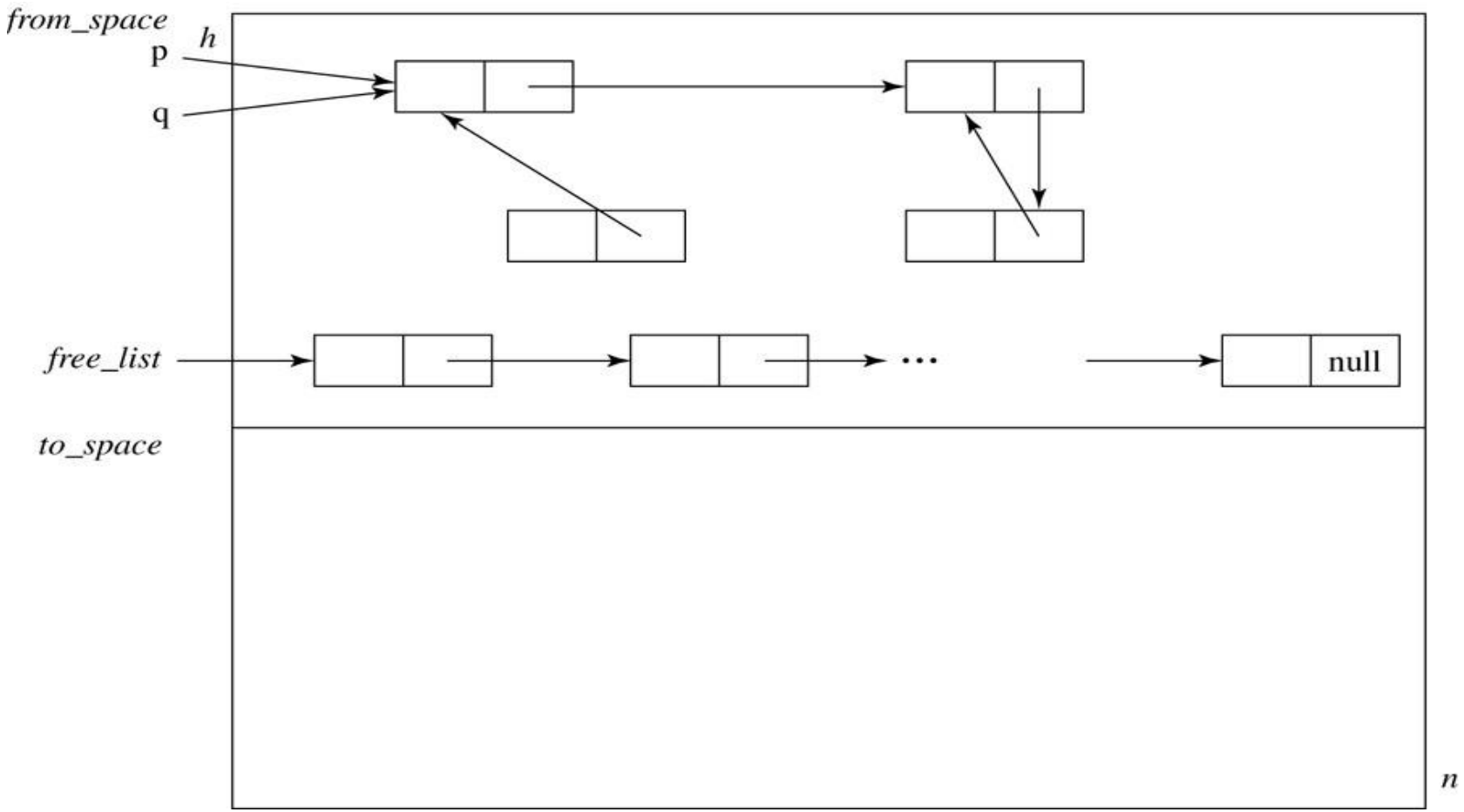
- Copying Collection 复制法
  - Basic idea: use 2 heaps
    - One used by program
    - The other unused until GC time
  - GC:
    - Start at the roots & traverse the reachable data
    - Copy reachable data from the active heap (from-space) to the other heap (to-space)
    - Dead objects are left behind in from space
    - Heaps switch roles

6.3.2.3 Copying collection (2/5)

- Copying Collection复制法

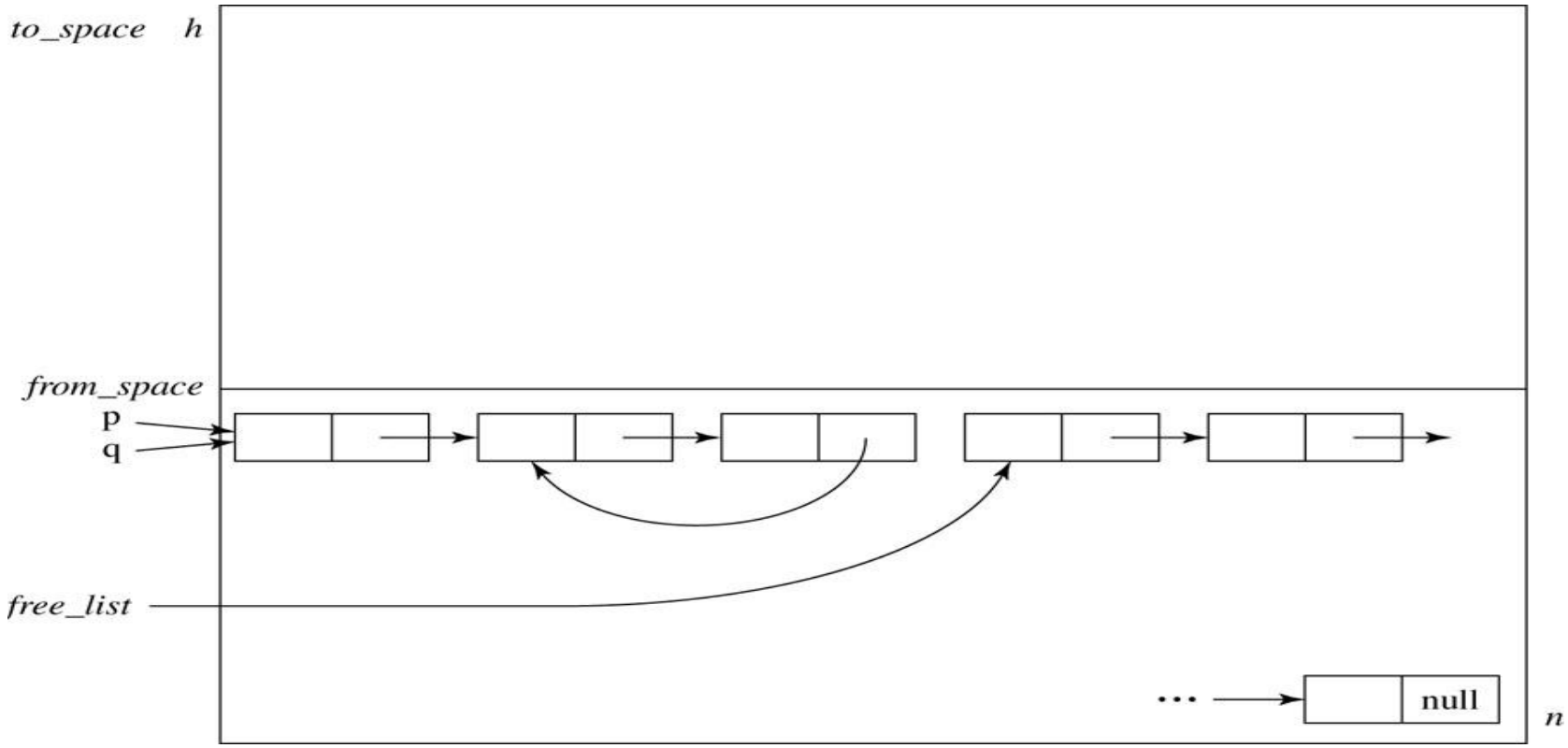


6.3.2.3 Copying collection (3/5)



Initial Heap Organization for Copy Collection

6.3.2.3 Copying collection (4/5)



Result of a Copy Collection Activation

6.3.2.3 Copying collection (5/5)

- The advantage of copy collection is that it is faster than mark and sweep.
  - Because it makes only a single pass through the heap.

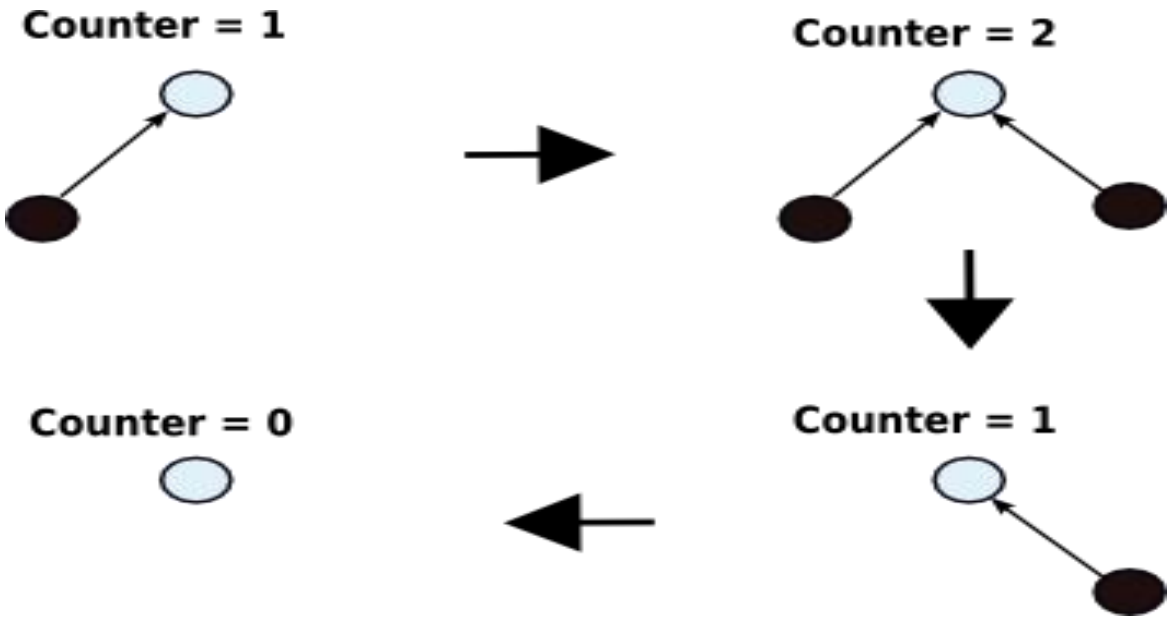
- The main disadvantage is that you can only every use half of your heap space.
- It also exhibits the unpredictable periodic interruptions that mark and sweep does.

6.3.2 Classical GC algorithms

- 6.3.2.1 Basic concepts
- 6.3.2.2 Mark and sweep collection
- 6.3.2.3 Copying collection
- 6.3.2.4 Reference Counting
- 6.3.2.5 Generational garbage collection

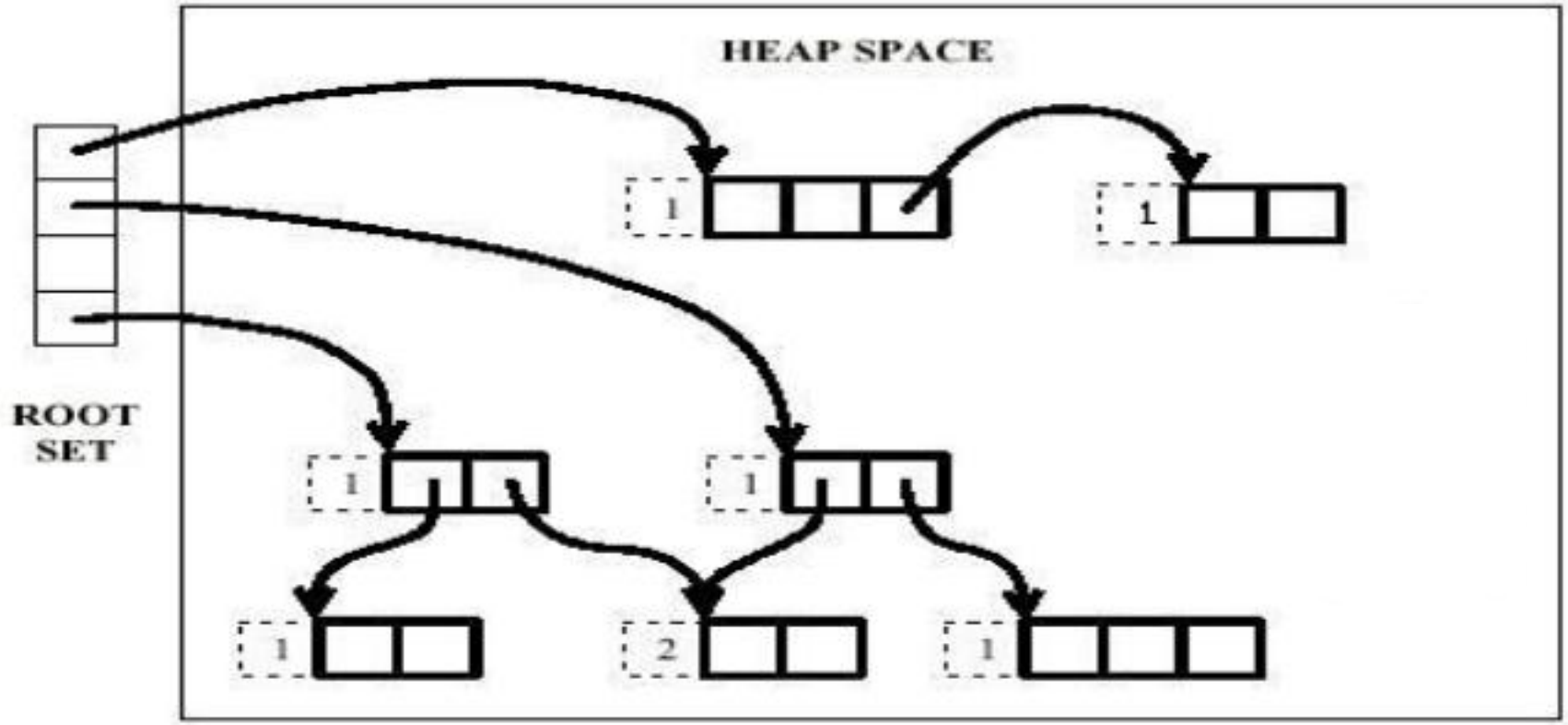
6.3.2.4 Reference Counting (1/5)

- Reference Counting 引用计数法
  - Keep track of the number of pointers to each object (the reference count).
  - When the reference count goes to 0, the object is unreachable garbage



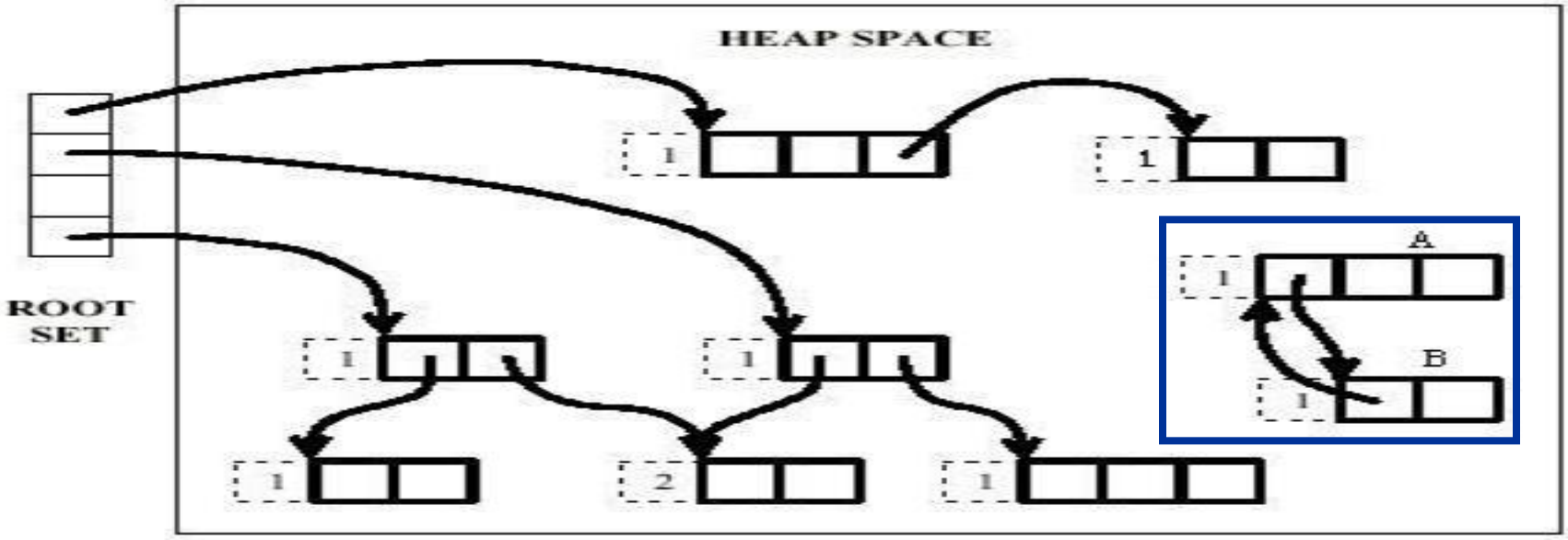
6.3.2.4 Reference Counting (2/5)

- Reference Counting 引用计数法



6.3.2.4 Reference Counting (3/5)

- Reference Counting - problems
  - Cost of reference counting
    - Too many ref-count increments and decrements
  - Fail to reclaim circular structures
    - Not always effective



6.3.2.4 Reference Counting (4/5)

- Reference counting has the advantage of being dynamic, the algorithm is performed whenever there is an assignment or other heap action.
- Disadvantages are:
  - Inability to detect inaccessible circular lists.
  - Overhead of keeping the counts (storage and time).
    - Every allocation and copying and function entry and exit has to have added code for adjusting the reference counts.

6.3.2.4 Reference Counting (5/5)

- There is no reference counting in Java.
- Python uses reference counting and offers cycle detection as well.

6.3.2 Classical GC algorithms

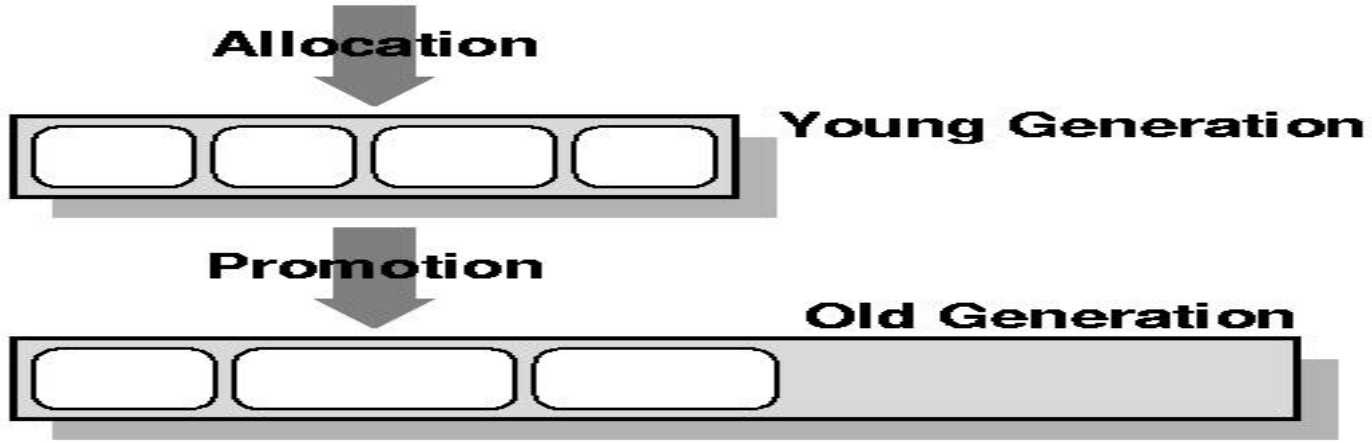
- 6.3.2.1 Basic concepts
- 6.3.2.2 Mark and sweep collection
- 6.3.2.3 Copying collection
- 6.3.2.4 Reference Counting
- 6.3.2.5 Generational garbage collection

6.3.2.5 Generational garbage collection (1/3)

- Generational GC 分代式垃圾回收法
  - Empirical(根据经验的)observation:
    - If an object has been reachable for a long time, it is likely to remain so
  - Empirical observation:
    - In many languages, most objects died young
  - Conclusion:
    - we save work by scanning the young objects frequently and the old objects infrequently
  - 达尔文的进化论：新生物种是最容易被淘汰的。

6.3.2.5 Generational garbage collection (2/3)

- Generational GC
  - Assign objects to different generations G0, G1,…
    - G0 contains young objects, most likely to be garbage
    - G0 scanned more often than G1
    - Common case is two generations (new, tenured)



6.3.2.5 Generational garbage collection (3/3)

- Generational GC
  - No need to traverse the entire reference tree
  - Older generations are garbage collected infrequently
  - The youngest generation can be collected in 5 to 50ms (ms stands for milliseconds, 1ms = 0.001 seconds).
  - The youngest generation, which is where most of the work is done, contains the highest proportion of garbage. By focusing the work here, the overall system is more efficient.

6.3 Garbage Collection

- 6.3.1 Introduction
- 6.3.2 Classical GC algorithms
- 6.3.3 Summary

6.3.3 Summary(1/4)

- 引用计数是解决显式内存分配问题的常用解决方案。实现赋值时递增和递减操作的代码通常是程序缓慢的原因之一。无论如何，引用计数也不是全面的解决方案，因为循环引用从不会被删除。

- 垃圾回收只会在内存变得紧张时才会运行。当内存尚且宽裕时，程序将全速运行，不会在释放内存上花费任何时间。

- 相对于过去的缓慢的垃圾回收程序，现代的垃圾回收程序要先进得多。分代、复制回收程序在很大程度上克服了早期的标记&清除算法的低效。

6.3.3 Summary(2/4)

- 现代垃圾回收程序进行堆紧缩。堆紧缩将减少程序引用的页的数量，这意味着内存访问命中率将更高，交换将更少。

- 采用垃圾回收的程序不会因为内存泄漏的累积而崩溃。采用 GC 的程序拥有更长期的稳定性。采用垃圾回收的程序 有更少的难以发现的指针错漏。 这是因为没有指向已经释放的内存的悬挂指针。因为没有显式的内存管理代码，也就不可能有相应的错漏。

- 采用垃圾回收的程序的开发和调试更快，因为不用开发、调试、测试或维护显式的释放代码。

- Automatic garbage collection can't be used with C and C++. Not true. It is possible to "plug in" a conservative garbage collector in your C or C++ program, giving fast and reliable automatic memory management.

6.3.3 Summary(3/4)

- 垃圾回收并非什么仙丹妙药。它有着以下不足：
  - 内存回收何时运行是不可预测的，所以程序可能意外暂停。
    - 运行内存回收的时间是没有上界的。尽管在实践中它的运行通常很快，但无法保证这一点。
    - 除了回收程序以外的所有线程在回收进行时都会停止运行。
  - 垃圾回收程序也许会留下一些本该回收的内存。在实践中，这不是什么大问题，因为显式内存回收程序通常会泄露一些内存，这致使它们最终耗尽所有内存，另一个理由就是显式内存回收程序通常会把内存放回自己的内部内存池中而不是把内存交还给操作系统。
  - 垃圾回收应该被实现为一个基本的操作系统内核服务。但是因为现实并非如此，就造成了采用垃圾回收的程序被迫带着它们的垃圾回收实现到处跑。尽管这个实现可以被做成一个共享 DLL，它也还是程序的一部分。

6.3.3 Summary (4/4)

- Reference
  - Garbage Collection in the Java HotSpot Virtual Machine, by Tony Printezis (See file: hotspot-gc.pdf)
  - Wikipedia on Garbage Collection (http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29)
  - https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/G1GettingStarted/index.html

Lab: Implementing a Conservative Garbage Collector

- 1. splay tree
- 2. about gc_malloc
- 3. conservative garbage collector

1. splay tree (1/10)

- Concept
  - A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again.
    - adjust tree in response to accesses to make common operations efficient
    - after access node is moved to root by *splaying*
- Performance
  - amortized such that 1 operations take O(lg n) where n is the number of nodes
- Amortized Analysis（均摊分析）

1. splay tree (2/10)

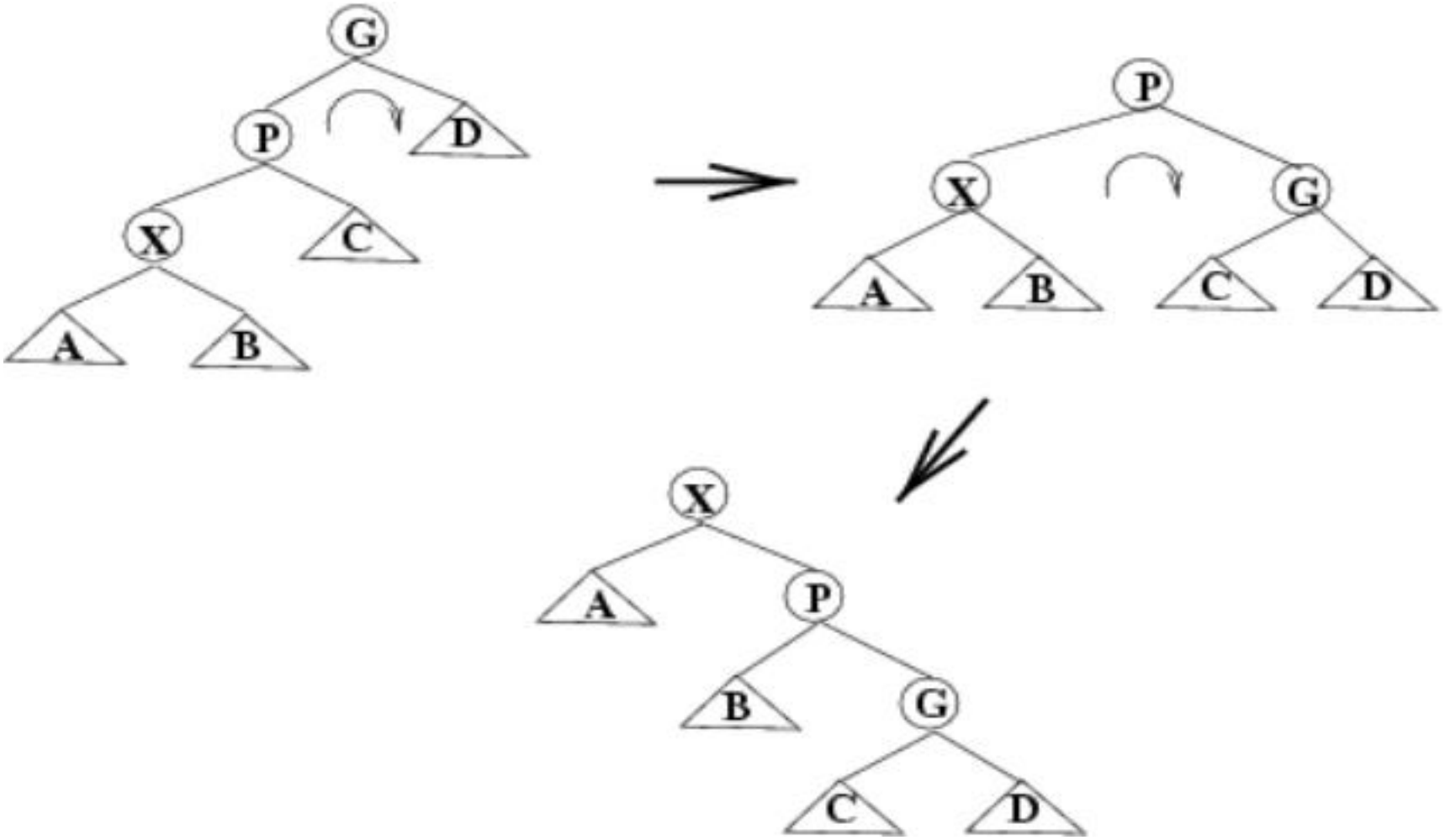
- Splay Operation
  - Traverse tree from node x to root, rotating along the way until x is the root
  - Each rotation
    - If x is root, do nothing.
    - If x has no grandparent, rotate x about its parent.
    - If x has a grandparent,
      - if x and its parent are both left children or both right children, rotate the parent about the grandparent, then rotate x about its parent
      - if x and its parent are opposite type children (one left and the other right), rotate x about its parent, then rotate x about its new parent (former grandparent)

1. splay tree (3/10)

- Node has no grandparent

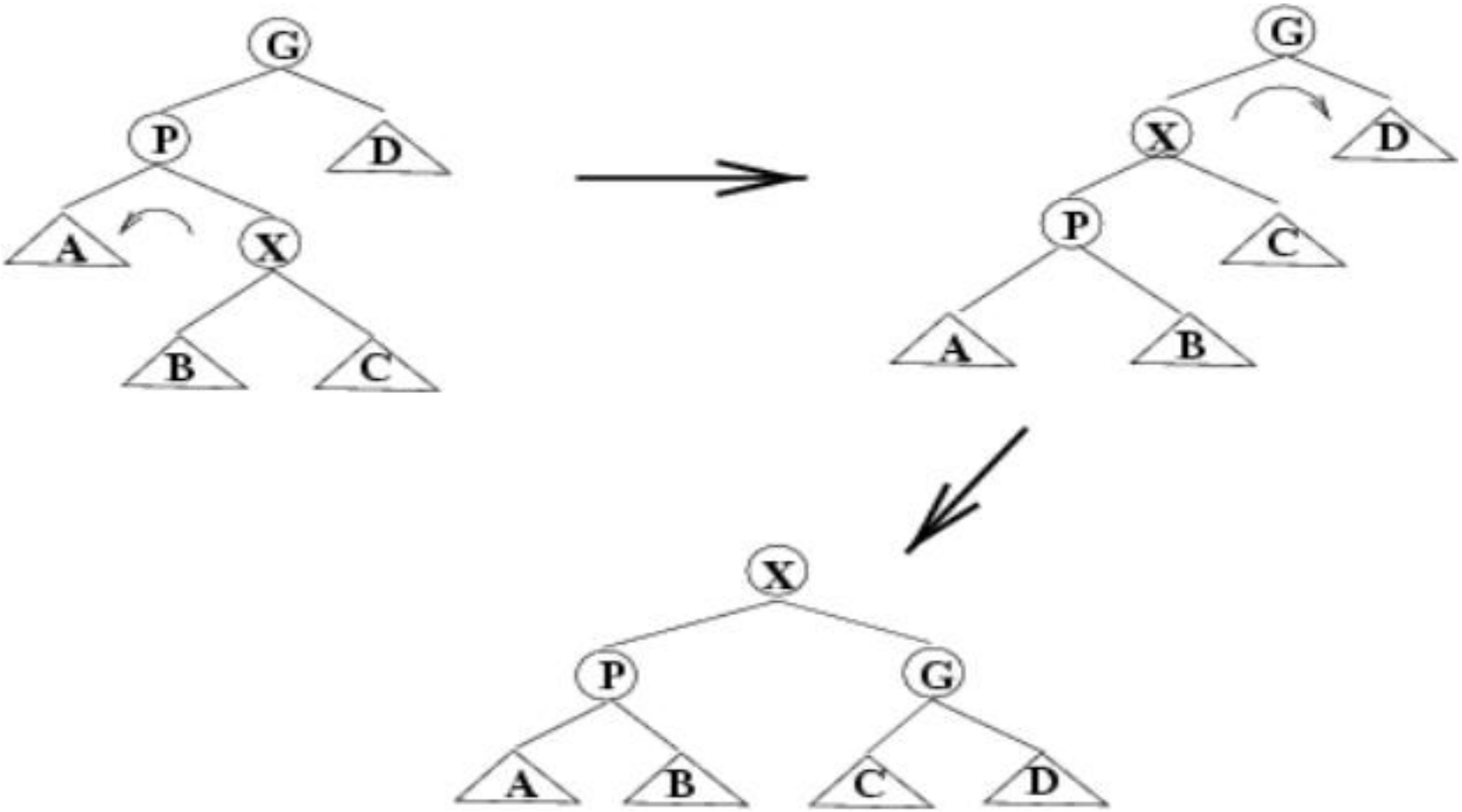
1. splay tree (4/10)

- Node and Parent are Same Side: Zig-Zig



1. splay tree (5/10)

- Node and Parent are Different Sides: Zig-Zag

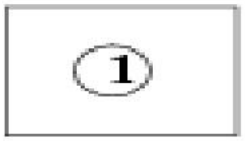


1. splay tree (6/10)

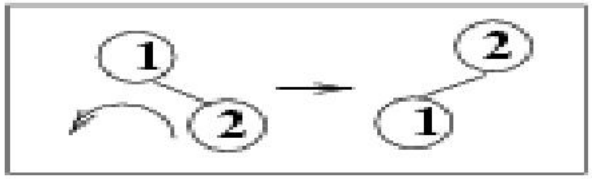
- Operations in Splay Trees
  - insert
    - first insert as in normal binary search tree
    - then splay inserted node
    - if there is a duplicate, the node holds the duplicate element is splayed
  - find
    - search for node
    - if found, splay to root; otherwise splay last node on path

1. splay tree (7/10)

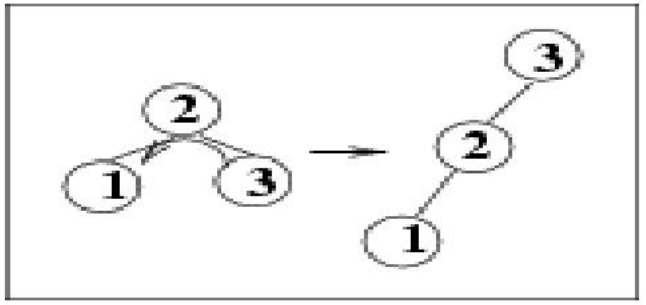
- Insertion in order into a Splay Tree



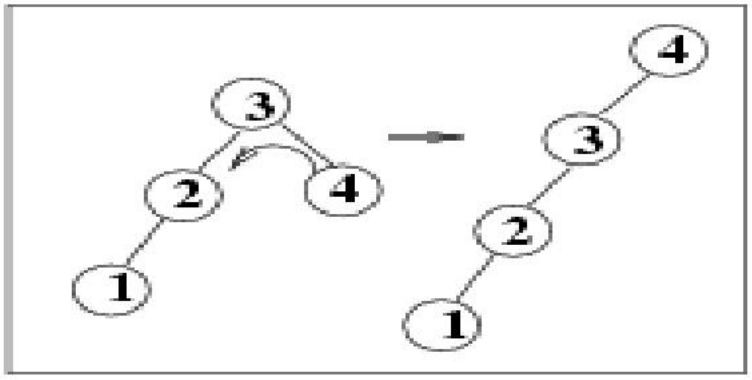
I



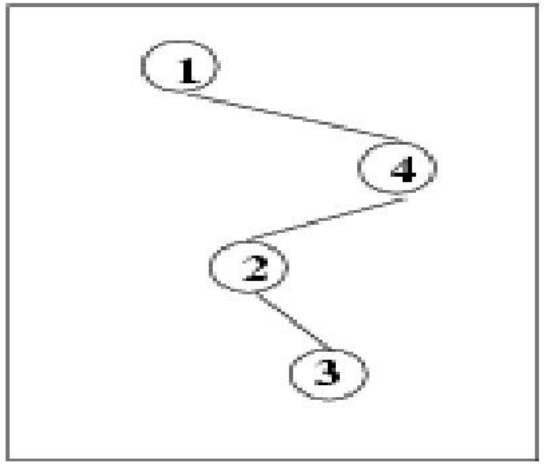
II



III



IV

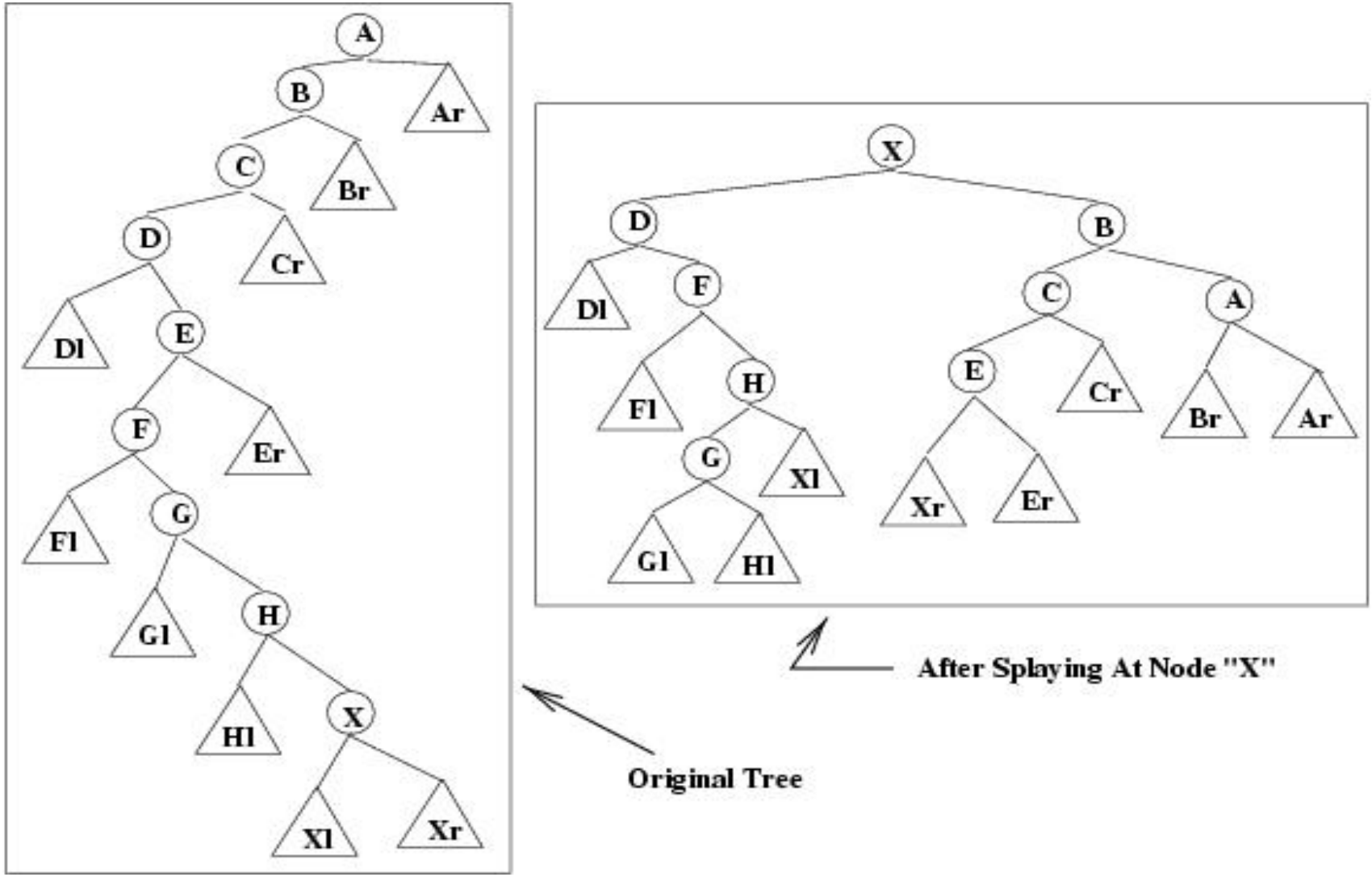


Access Node 1

1. splay tree (8/10)

- Operations on Splay Trees (cont.)
  - remove
    - splay selected element to root
    - disconnect left and right subtrees from root
    - do one of:
      - splay max item in T_L (then T_L has no right child)
      - splay min item in T_R (then T_R has no left child)
    - after deletion, we splay the parent of the removed node to the top of the tree

1. splay tree (9/10)

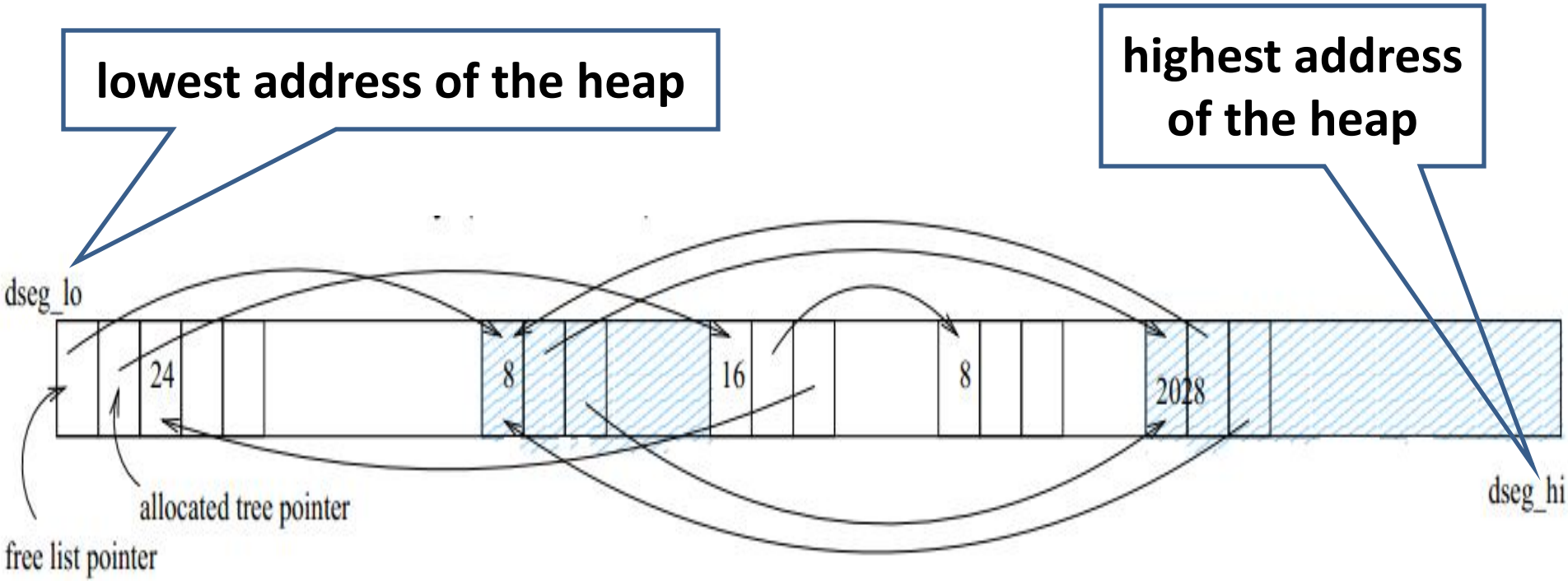


1. splay tree (10/10)

- Performance of Splay Trees
  - insert
    - regular bst insertion -- O(depth)
    - splay: O(1) for each rotation, O(depth) rotations

2. about gc_malloc (1/3)

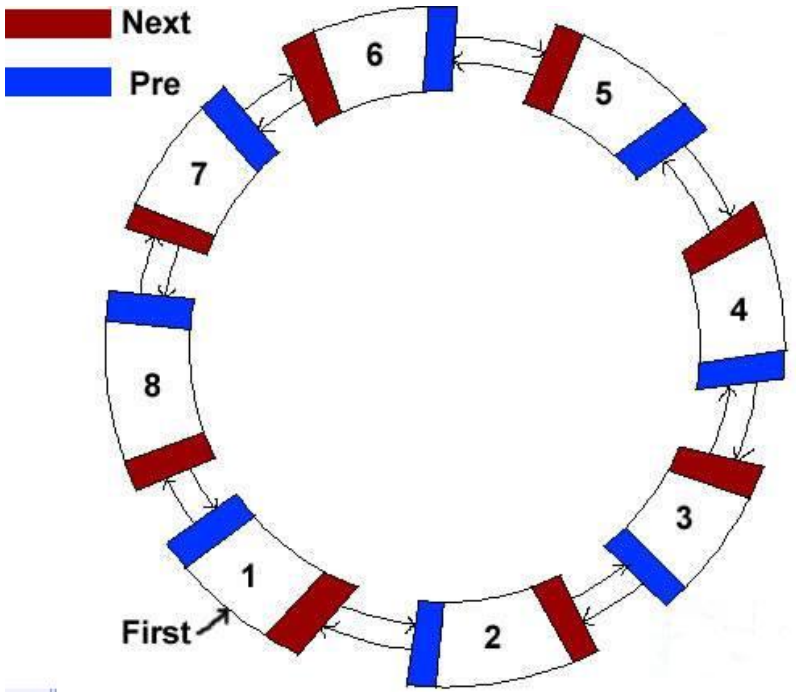
- Data structures in heap memory

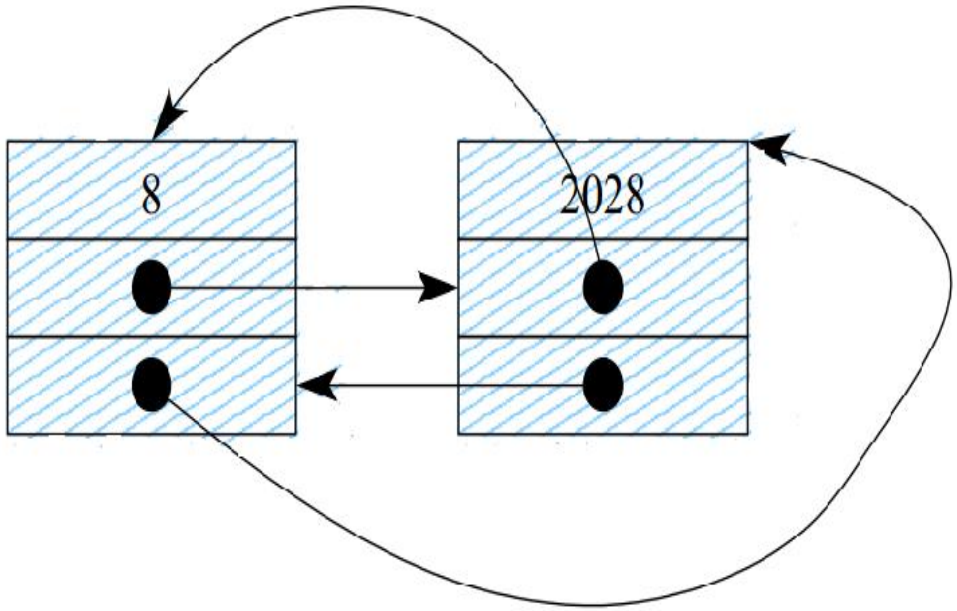


Freelist and Allocation Tree in Heap Memory

2. about gc_malloc (2/3)

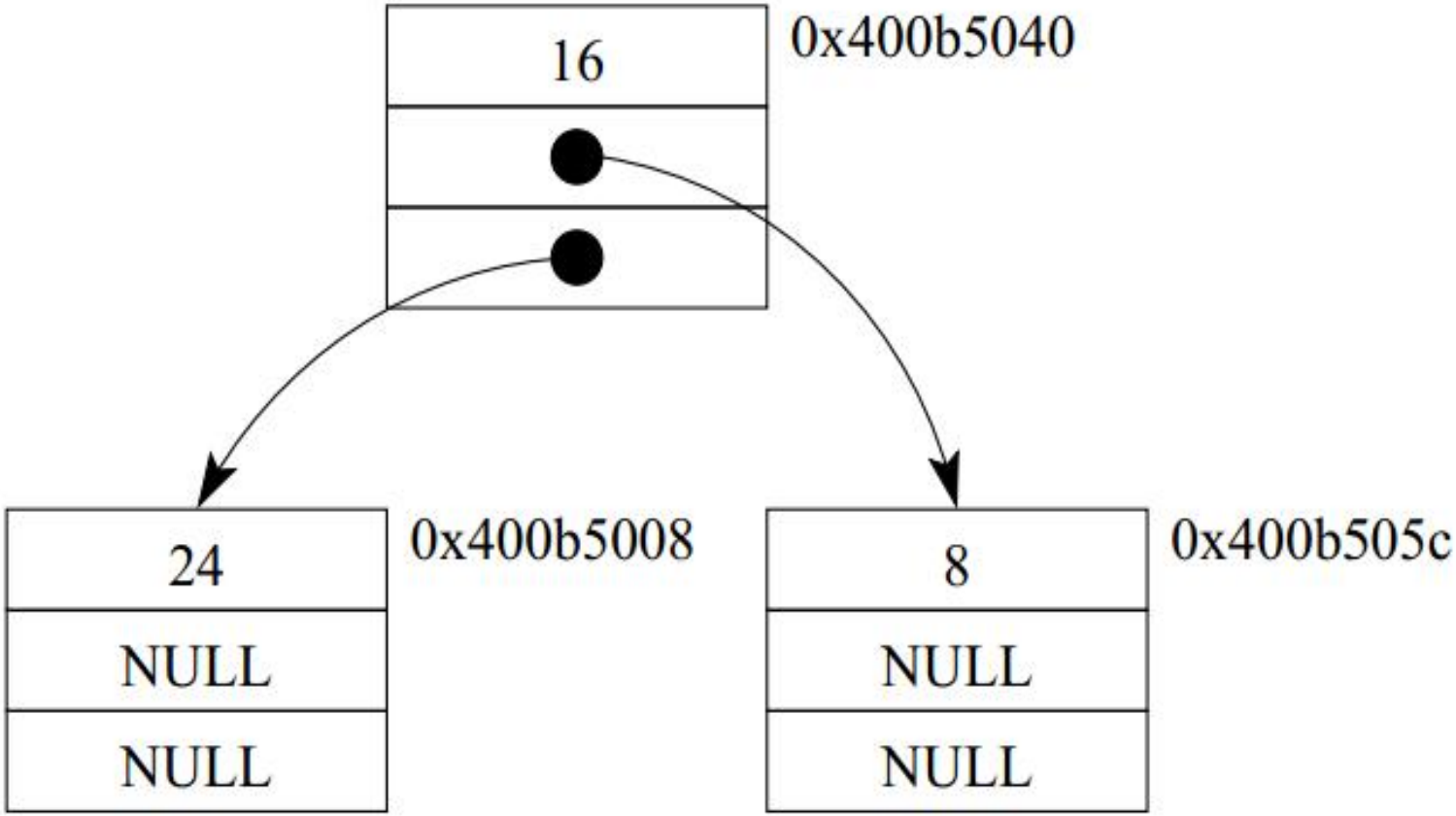
- Data structure of free list
  - Circular Doubly-Linked List





2. about gc_malloc (3/3)

- Data structure of allocation tree
  - Binary Search Tree



3. conservative garbage collector

- Algorithm

```
void GC()
{
        HaltAllProcessing();
        ObjectCollection roots = GetRoots();
        for(int i = 0; i < roots.Count(); ++i)
                Mark(roots[i]);
        Sweep();
}
```

4. GC Lab Implementation Details

- Document L3.pdf： Lab assignment
- lab_garbage_collector： VC6.0 files
- [Your task]
  - Fill malloc.c by realizing garbage_collect()

- Implementation Details Refers to:
  - gc-lab-reference-1.ppt