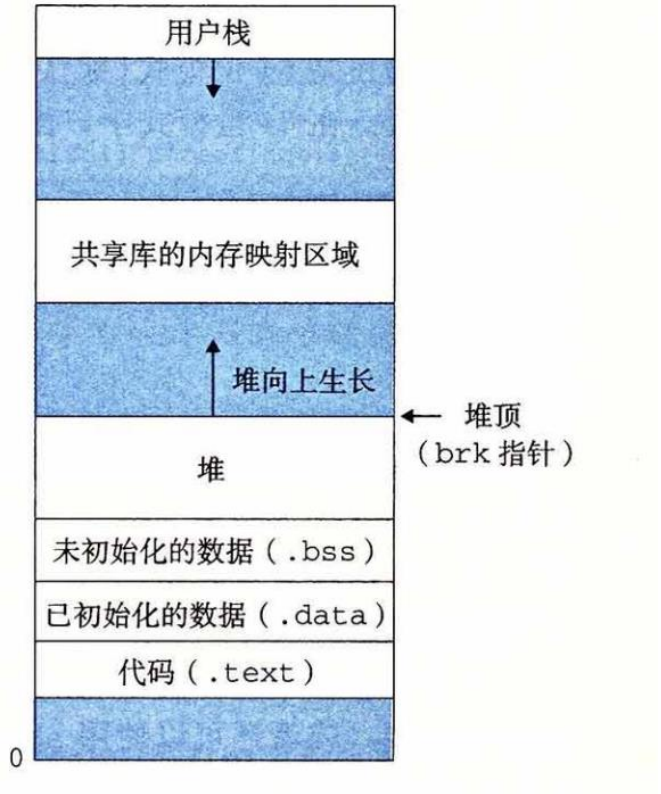
Dynamic Memory Allocation

2023.12.13

李之怡

动态内存分配

- 为什么：由于在编写程序过程中，可能不知道程序实际运行时某些数据结构的大小，需要额外请求虚拟内存
- 用**动态内存分配器**管理这部分虚拟内存
  - 称为**堆**
  - 堆是**内存块**的集合
  - 块：内部连续，分配/空闲
- 两种分配器：显式&隐式
  - 根据释放方式区分



显式内存分配

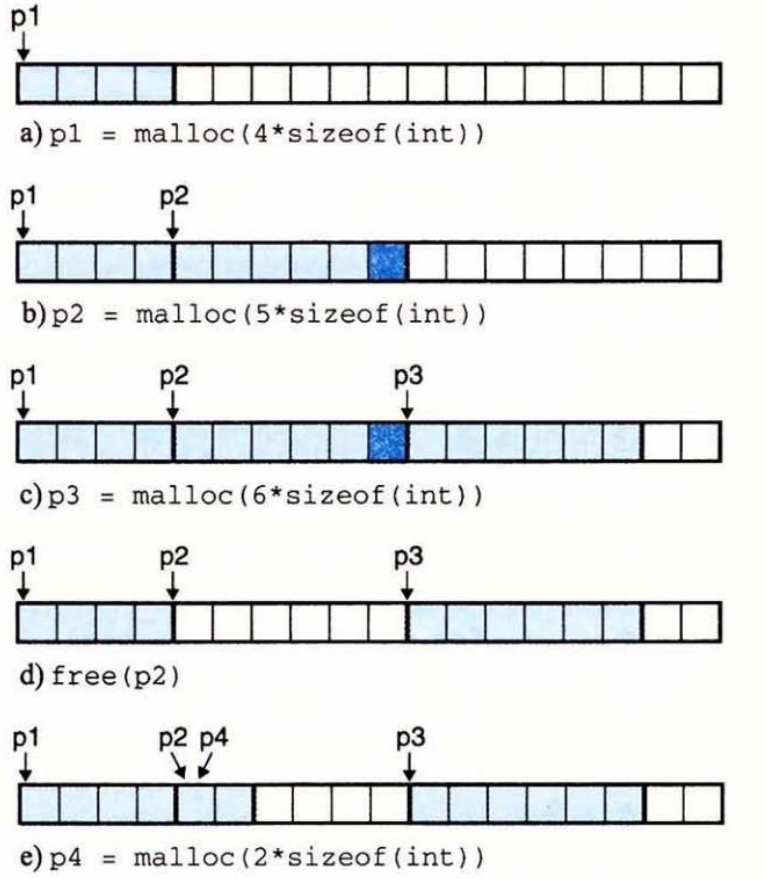
- C语言中的相关函数

```c
int *arr1, *arr2;
arr1 = (int *)malloc(3 * sizeof(int)); // 分配包含3个整数的内存块
arr1 = (int *)realloc(arr, 5 * sizeof(int)); // 重新分配为包含5个整数的内存块
arr2 = (int *)calloc(5, sizeof(int)); // 分配包含5个整数的内存块，并初始化为零
free(arr1); // 释放内存
free(arr2);

void *original_brk = sbrk(8); // 将堆扩展8个字节
void *current_brk = sbrk(0); // 获取当前堆顶位置
```

对齐与碎片

- 对齐：双字，字
- 碎片
  - 内部碎片：已分配块比有效载荷大时产生
  - 外部碎片：空闲块合计起来足够满足分配需求，但没有一个单独的空闲块足够大可以处理这个请求
  - 难以量化且不可预测，所以分配器会试图维持少量大空闲块，而不是大量小空闲块



评估分配器的效率

- 目标1：最大化吞吐率
- 吞吐率：单位时间内完成的请求数（分配请求+释放请求）
- 目标2：最大化内存利用率
- 峰值利用率：
  - 已分配块的有效载荷：应用程序请求一个p字节的**块荷(payload)** 。
  - 在请求R_k完成之后，**聚集有效载荷 (aggregate payload)** 表示为 P_k，为当前已分配的块的有效载荷之和。
  - H_k表示堆的当前的(单调非递减的)大小。

$$U_k = \frac{\max_{i \leqslant k} P_i}{H_k}$$

管理方式

- 空闲块组织：我们如何记录空闲块？
- 放置：我们如何选择一个合适的空闲块来放置一个新分配的块？
- 分割：在将一个新分配的块放置到某个空闲块之后，我们如何处理这个空闲块中的剩余部分？
- 合并：我们如何处理一个刚刚被释放的块？

隐式空间链表

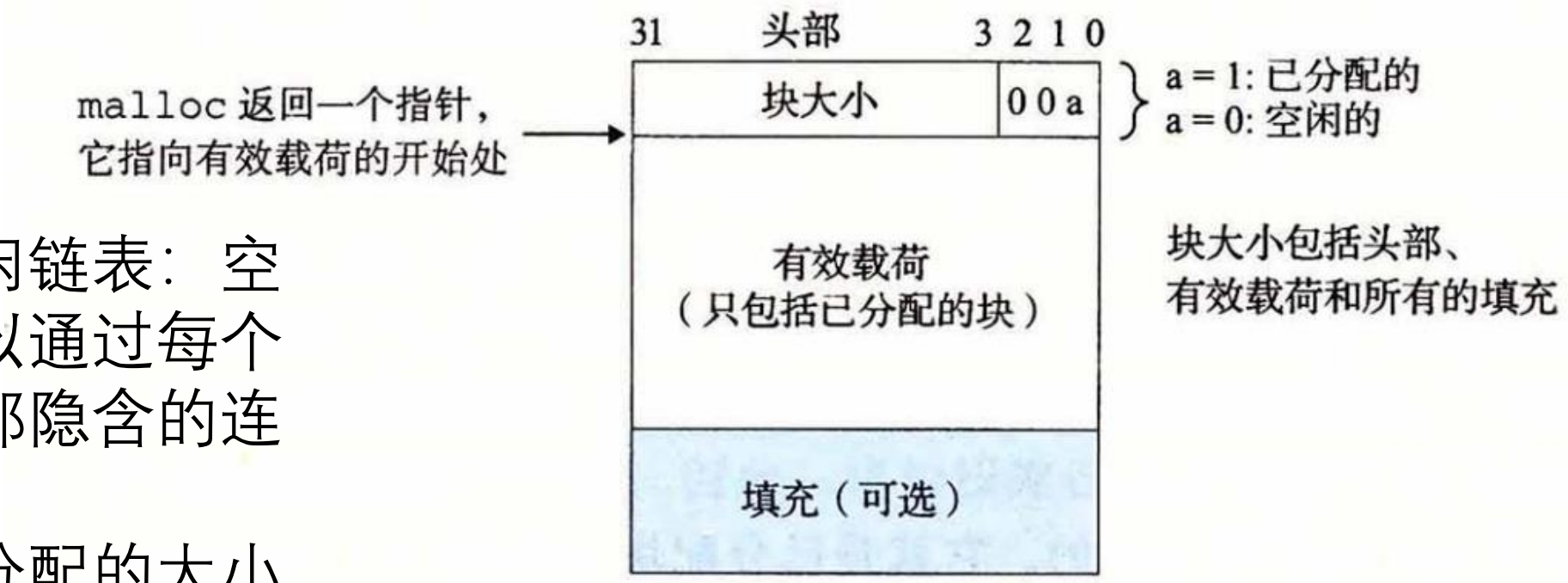


图 9-35　一个简单的堆块的格式

- 隐式空闲链表：空闲块可以通过每个块的头部隐含的连接起来
- 一个已分配的大小为24（0x18）字节的块，头部为 `0x00000018 | 0x1 = 0x00000019`

隐式空间链表

- 优点：简单
- 缺点：操作开销大，放置分配块时需要对空闲链表进行搜索
- 对最小块大小有要求：双字对齐的堆，最小的块也需要2个字，一个字做头。

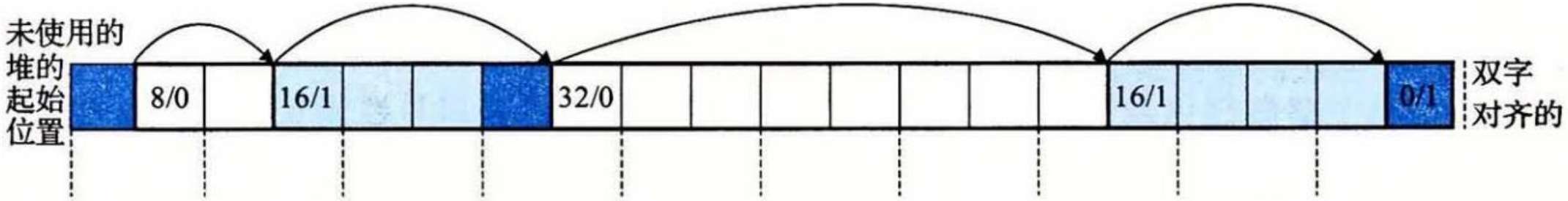


图 9-36　用隐式空闲链表来组织堆。阴影部分是已分配块。没有阴影的部分是空闲块。头部标记为(大小(字节)/已分配位)

分配

- 搜索空闲链表，寻找合适的空闲块
  - 首次适配：每次从头开始搜索，选择第一个合适的空闲块
  - 下一次适配：每次从上一次搜索结束的地方开始，选择第一个合适的空闲块
  - 最佳适配：检查每一个空闲块，选择大小最合适的空闲块
- 分割空闲块
  - 使用整个空闲块：内部碎片更多，但简单便捷;
  - 分割：将空闲块分割为已分配块和空闲块;
- 获取额外堆内存
  - 当分配器找不到合适空闲块时，可以合并空闲块，或请求额外的堆内存

释放

- 合并空闲块
  - 分配器释放一个已分配块时，可能有其他空闲块与之相邻，产生“假碎片”
  - 立即合并/推迟合并

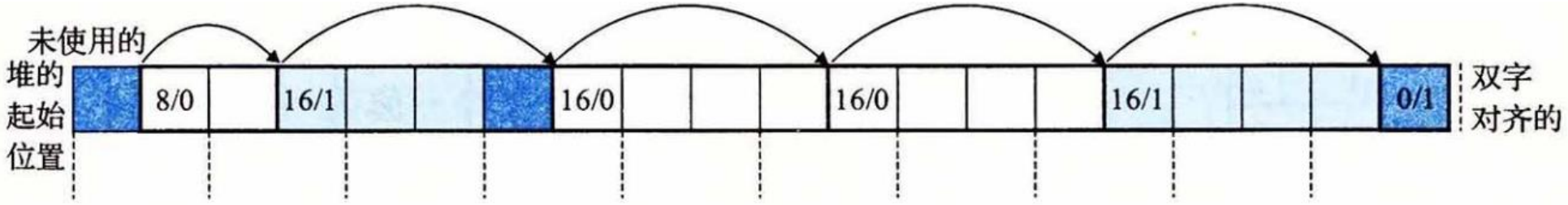


图 9-38　假碎片的示例。阴影部分是已分配块。没有阴影的部分是空闲块。头部标记为(大小(字节)/已分配位)

带边界标记的合并

- 边界标记：给每个块结尾处添加一个脚部，内容与头部相同；这样，分配器就可以通过检查脚部获得上一个块的起始位置与状态。
- 实现：使用当前块的头部、前面的块的脚部、后面的块的头部，更新三个块的头部中的信息

带边界标记的合并

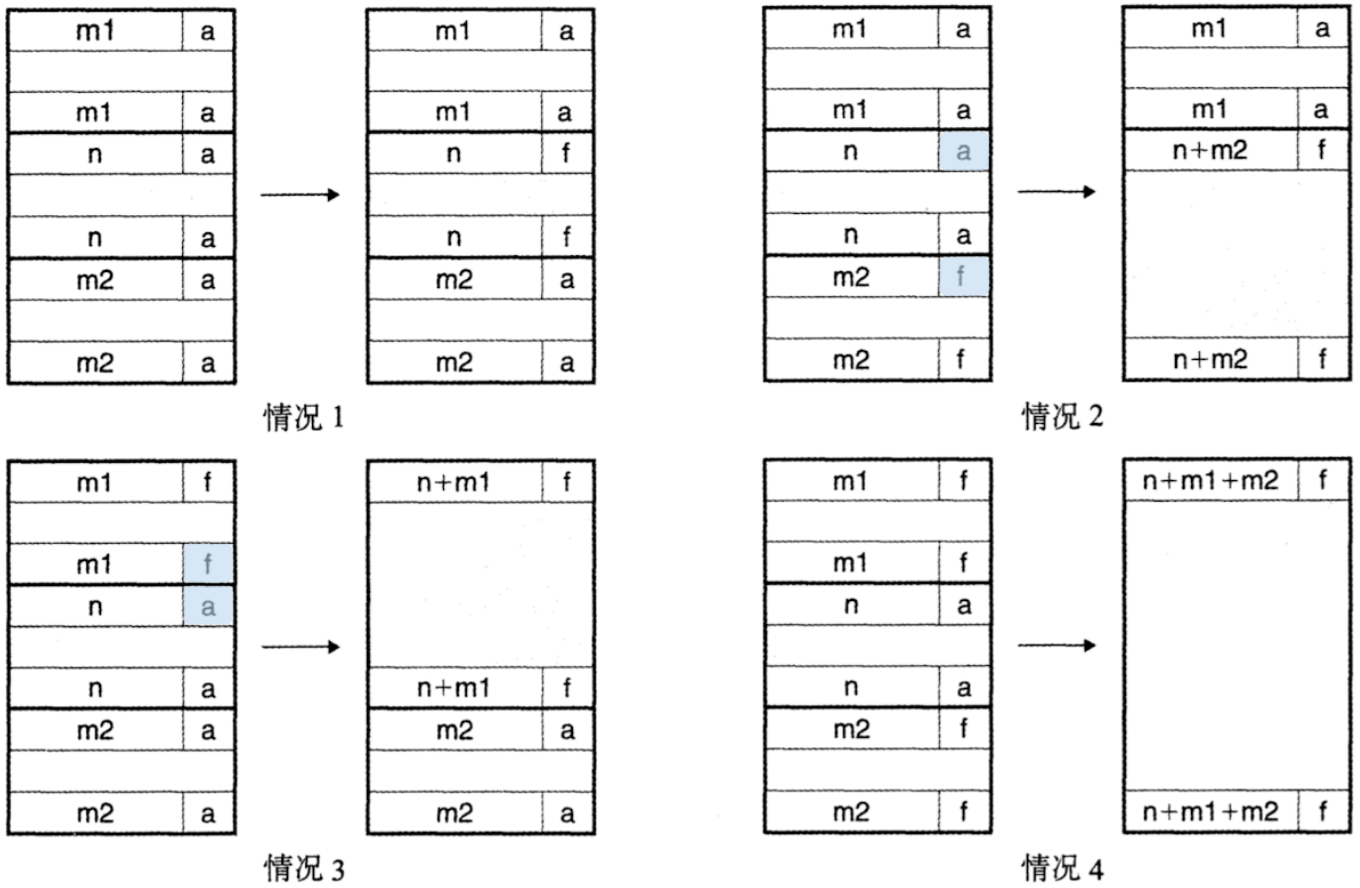


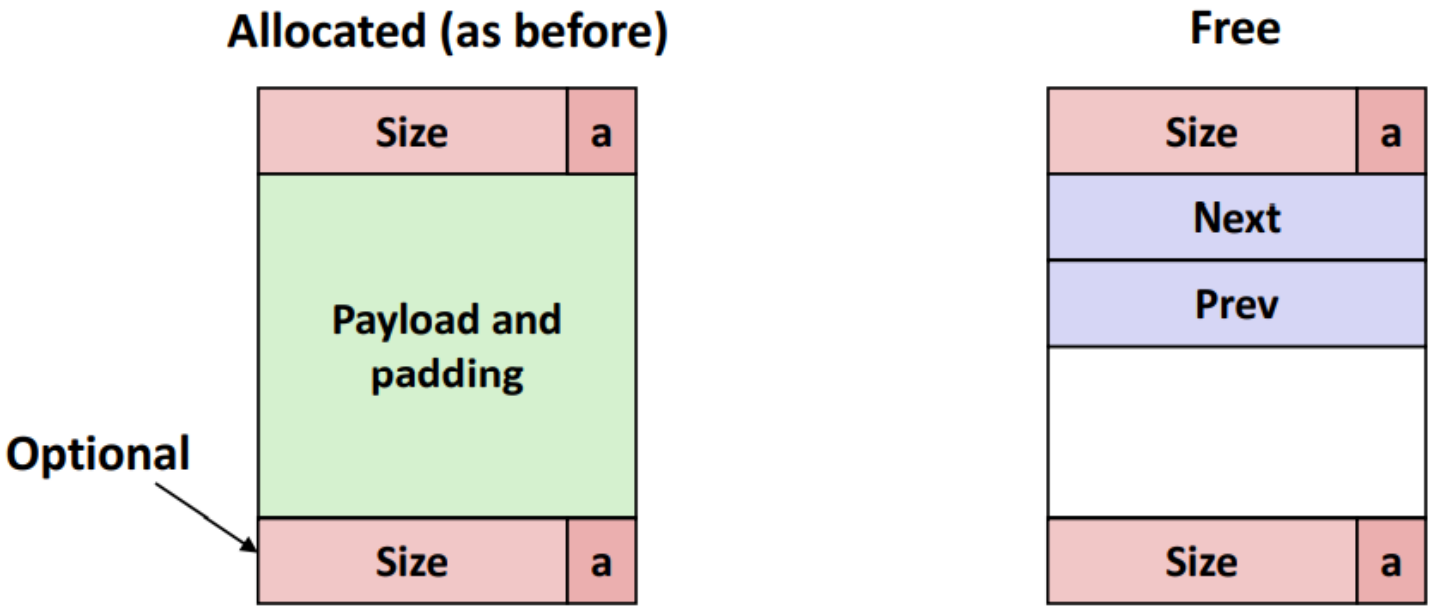
图 9-40　使用边界标记的合并(情况 1：前面的和后面块都已分配。情况 2：前面块已分配，后面块空闲。情况 3：前面块空闲，后面块已分配。情况 4：后面块和前面块都空闲)

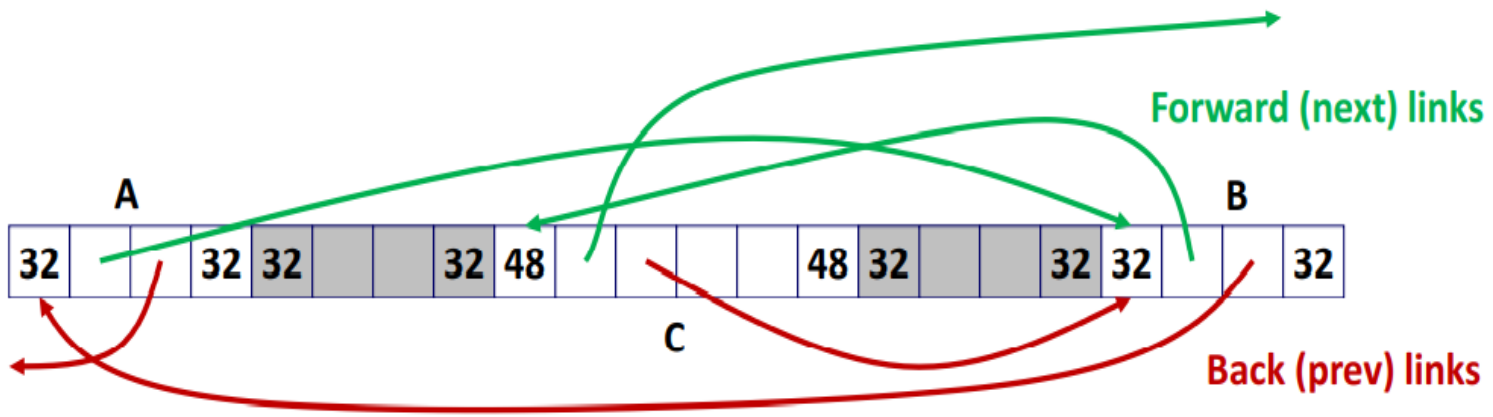
带边界标记的合并

- 缺陷：脚部消耗了大量的内存资源
- 优化：只有前面的块是空闲块时，才需要得知它的大小，才需要使用脚部
- 如果把前面块的分配/空闲位存放在当前块多余的低位中，那么已分配块就不需要脚部

显式空闲列表

- 在每个空闲块中都包含一个next和prev指针
- 指向的空闲块在空间上不一定是顺序的
- 分配时间从块总数的线性时间减少到空闲块数量的线性时间
- 缺点是空闲块需要足够大，以包含指针、头部和可能的尾部



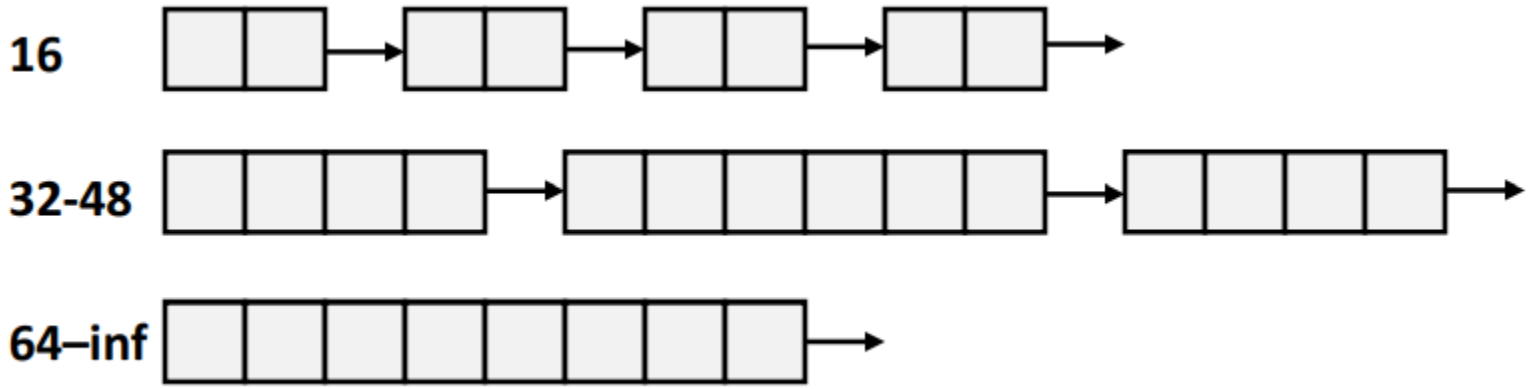


链表维护

- 释放一个块的时间取决于空闲链表中块的排序策略
- 不按地址顺序维护链表，分为后进先出（LIFO，将新释放的块放在链表头）和先进先出（FIFO，将新释放的块放在链表尾）；释放一个块可以在常数时间内完成，但是碎片较多
- 按照地址顺序来维护链表，链表中每个块的地址都小于其后继的地址；释放一个块需要线性时间，但是碎片较少

分离的空闲列表

维护多个空闲链表，将链表中所有可能的块大小分成一些大小类（size class）



通常将小的块分派到其自己的大小类中，将大块按照2的幂次分类

$$\{1\},\{2\},\{3\},\cdots,\{1023\},\{1024\},\{1025 \sim 2048\},\{2049 \sim 4096\},\{4097 \sim \infty\}$$

分离的空闲列表

简单分离存储

- 每个大小类的空闲链表只包含大小相等的块，每个块的大小为大小类中的最大值;
- 分配一个块时，检查相应空闲链表，若非空则分配第一块的全部（不会分割），若为空则请求额外内存片;
- 释放一个块时，将这个块插入到相应空闲链表头（不会合并）;
- 虽然操作都是常数时间，但容易造成内部和外部碎片
- 内存开销少，不需要头部脚部

分离的空闲列表

分离适配

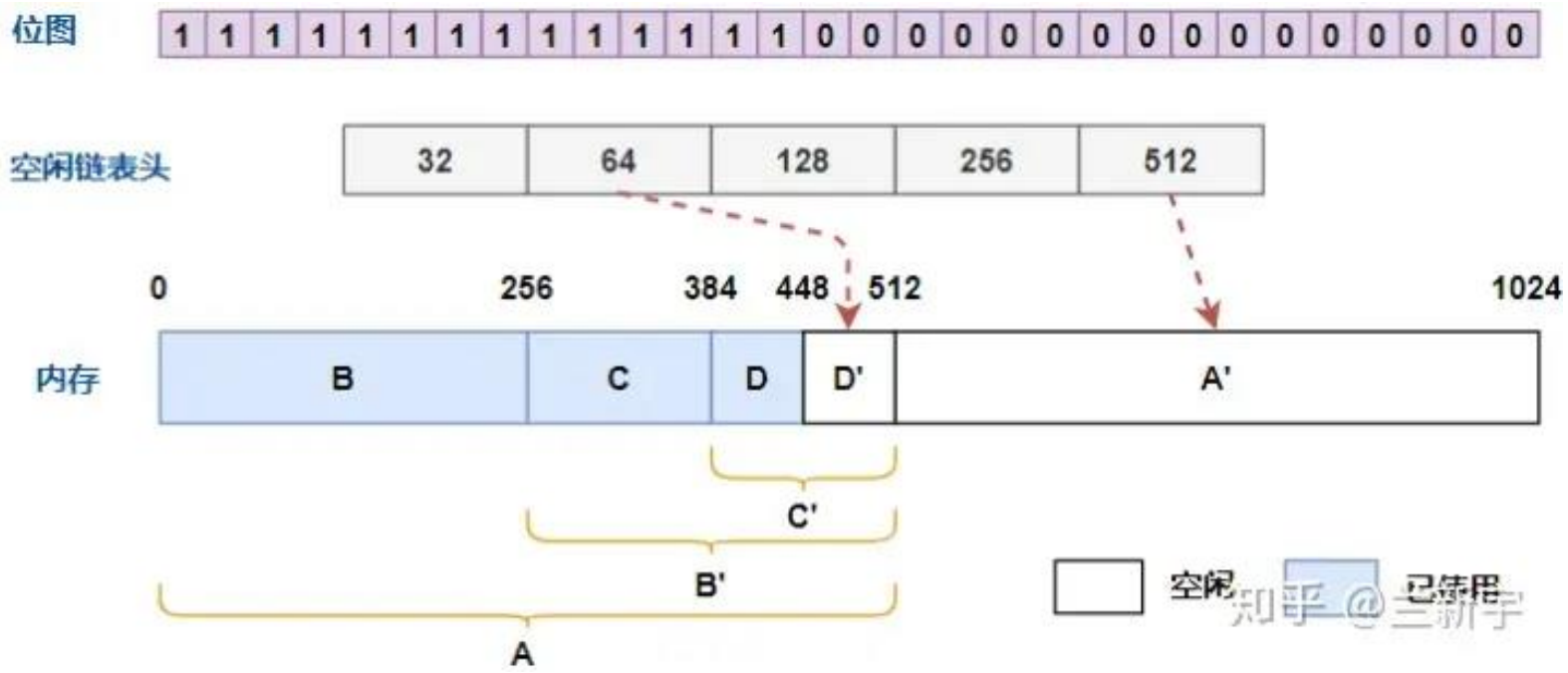
- 每个空闲链表和一个大小类相关联，组织成某种类型的显式或隐式链表；
- 分配一个块时，如果找到合适的块，就分割它并将剩余部分插入到合适的空闲链表中，如果没找到，就搜索下一个更大的大小类的空闲链表；
- 释放一个块时，执行合并再将结果放到相应的空闲链表中

伙伴系统

- 分离适配的一种特例，每个大小类都是2的幂次；
- 分配一个块时，如果块有剩余，则递归地二分这个块直到没有剩余，每个剩下的半块（伙伴）被放置在相应的空闲链表中；
- 释放一个块时，合并空闲的伙伴，遇到已分配的伙伴时停止合并

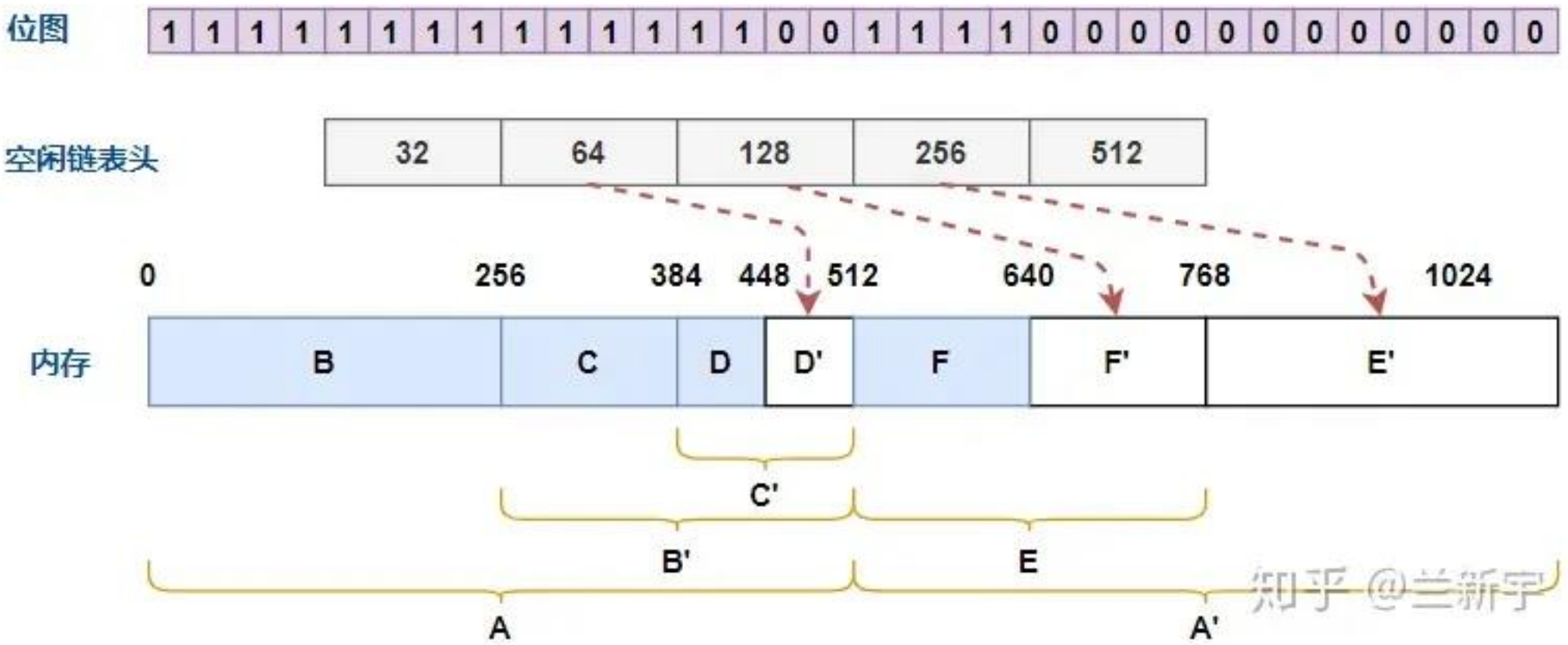
分离的空闲列表

伙伴系统
申请128字节:

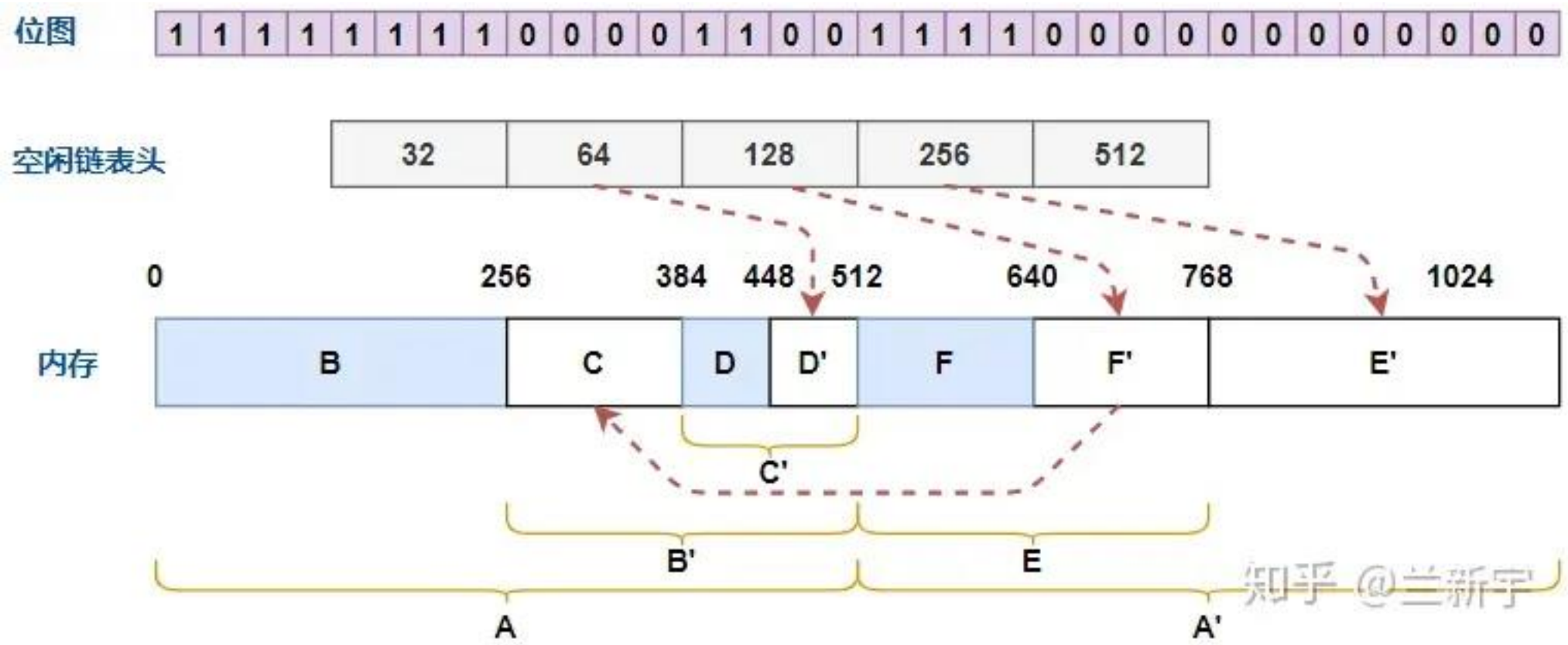


伙伴系统

申请128字节:

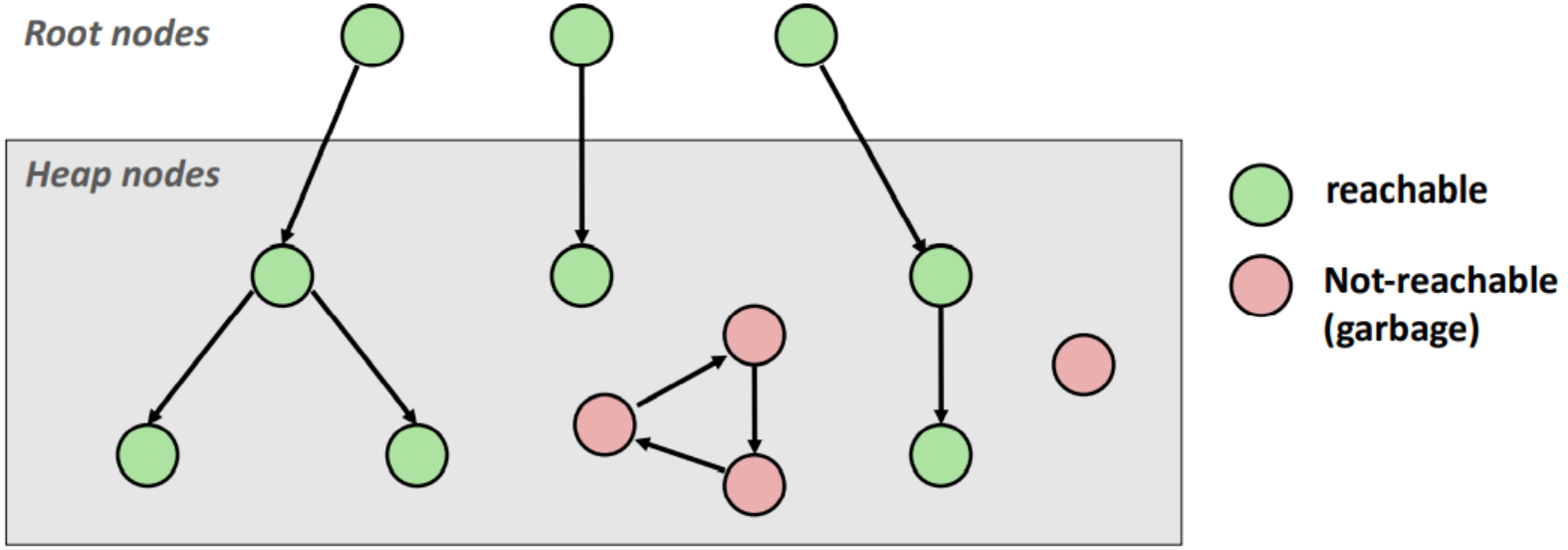


- 释放128字节的内存块C
- 释放64字节的内存块D



垃圾收集

在一个支持垃圾收集的系统中，应用不会显示地释放已分配块，垃圾收集器会定期识别垃圾块（程序不再需要的已分配块），并相应地调用free，将这些块释放。
垃圾收集器将内存视为一张有向可达图(reachability graph)，该图的节点被分成一组根节点(root node)和一组堆节点(heap node)。图中的不可达结点代表垃圾。



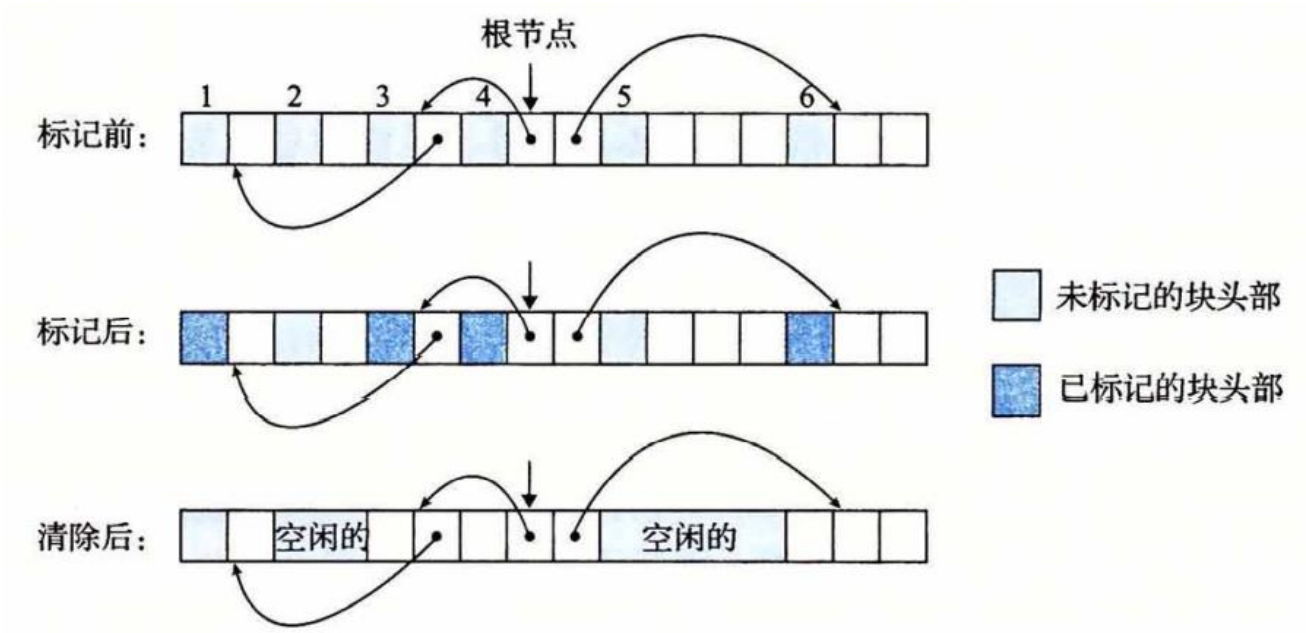
p→q意味着块p中的某个位置指向块q中的某个位置，可以是寄存器、栈里的变量，或者是虚存中读写数据区域里的全局变量

垃圾收集

Mark&Sweep垃圾收集器由标记阶段和清除阶段组成

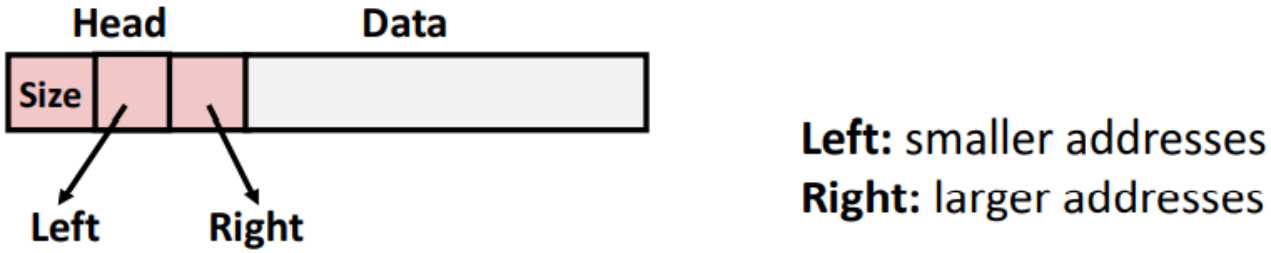
标记阶段：每个根节点调用一次mark函数，标记该根节点的所有未标记且可达的后继节点（块头部中空闲的低位中的一位作为标记位）；

清除阶段：调用一次sweep函数，释放所有未标记的已分配块



垃圾收集

C语言中，即使知道p是一个指针，但无法判断p是否指向一个已分配块的有效载荷中的某个位置。因此，可以将已分配块集合维护成一棵平衡二叉树，每个已分配块的头部里有left和right，指向其他已分配块的头部，在二分查找中，依赖size字段来判断p是否落在这个块中。



C和C++的收集器通常不能维持可达图的精确表示，被称为保守的垃圾收集器，其根本原因是C语言不会用任何类型信息来标记内存位置。

Example：某个可达的已分配块a中包含一个int，其值恰好对应于某个其他的已分配块b中的一个地址，收集器无法判断该数据是int而不是指针，分配器必须保守地将块b标记为可达，尽管块b事实上可能不可达。

常见问题

```
int val;

...

scanf("%d", val);
```

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

常见问题

```
char **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

```
int *search(int *p, int val) {

   while (p && *p != val)
      p += sizeof(int);

   return p;
}
```

```
char *p;

p = malloc(strlen(s));
strcpy(p,s);
```

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

常见问题

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```

```
int *foo () {
   int val;

   return &val;
}
```

```
x = malloc(N*sizeof(int));
  <manipulate x>
free(x);
   ...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

谢谢！