**1. 基本概念：存储空间的分配与管理**

动态存储管理-堆的管理

* **堆(heap)**：操作系统在内存中划出一块地址连续的大区域
  + 占用块：已分配给用户使用的一块地址连续的内存区域；
  + 空闲块：未曾分配的地址连续的内存区域

内存分配和回收方式

方法一：从高地址空闲块中进行分配，直到分配无法进行时，才回收所有用户不再使用的空闲块，重新组织一个大的空闲块来再分配。（中途不真正进行回收和重新组织）

方法二：每当新用户请求分配内存时，需查找整个内存区中所有空闲块，并从中找出一个合适的空闲块分配之；用户程序一旦运行结束，便将它所占的内存区释放成为空闲块

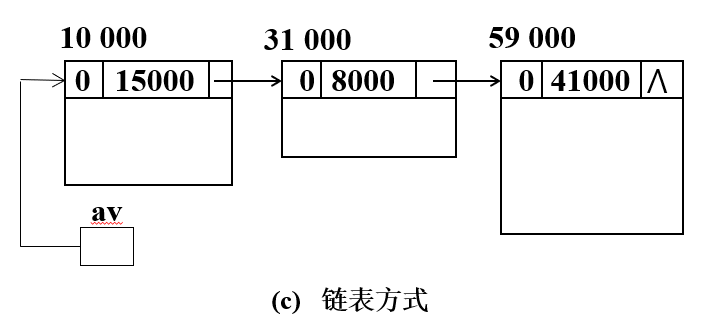
**2. 可利用空间表**

可利用空间表/存储池：包含所有可分配的空闲块，每个空闲块是链表中的一个结点

请求分配：删除结点

释放内存：插入结点

可利用空间表的**组织方式：**目录表、链表方式



**链式可利用空间表的分配和回收**

取决于结点(空闲块)的结构

方法一：**空闲块的大小相同**

分配：从表的首届点分配； 回收：将释放的空闲块插入表头

方法二：**空闲块大小只有几种规格**

建立若干个可利用空间链表，同一链表中的结点(块)大小都相同

分配：最接近该大小的某个链表的首结点； 回收：将所释放块插入到相应大小的链表表头

方法三：**请求分配的块大小不确定**

一开始链表中只有一个大小为整个堆的结点，随着分配和回收的进行，链表中的结点大小和个数动态变化

**链表结点中，增加一个表示结点大小的域(size)，以保存空闲块的大小**

几种方法如下：

**（1）首次拟合法 (First fit)**

分配：从表头指针开始查找可利用空间表，找到**第一个不小于n的空闲块，将用户所需的大小**分配给用户，剩下部分仍然是一个空闲块结点

回收：将释放的空闲块插在链表的表头

特点：分配时随机的；回收时仅需插入到表头；容易产生无法分配的内存碎片

**（2）最佳拟合法(Best fit)**

**分配**：找到一个**大小满足要求且最接近n**的空闲块，**将所需大小分配给用户**，剩下部分仍然是一个空闲块结点（需要重新插入该节点）

**回收**时：只要将释放的空闲块插入到可利用空间链表的**合适**位置（**按从小到大排列**）

特点：无论分配与回收，都需要查找空闲链表，最费时；容易产生无法分配的内存碎片；**适用于请求分配的内存块大小范围较广的系统**

**（3）最差拟合法(Worst fit)**

**分配**：找到一个大小**最大的**空闲块，将所需大小分配给用户，剩下部分仍然是空闲块结点（徐重新插入）

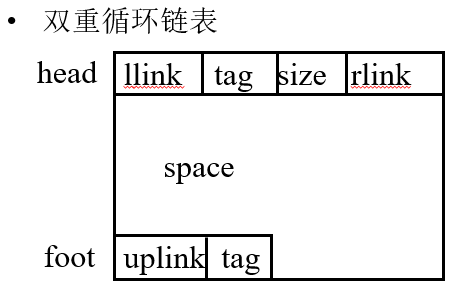
**回收**：只要将释放的空闲块插入到链表的**合适**位置（从大到小）

特点：**分配无需查找**，回收需要查找适当的位置；容易产生无法分配的内存碎片；**适用于请求分配的内存块的大小范围较窄的系统**

**以下的实现方法解决了内存碎片的问题：**

**3. 边界标识法 (Boundary Tag Method)**

* 操作系统中常用的动态存储管理方法
* 将所有的空闲块链接成一个**双重循环链表**
* 每个内存区域的**头部**和**底部**两个边界上分别设置标识，以标识该区域为占用块或空闲块



数据结构Space：（WORD\*）

头部域：llink指向前驱节点（WORD\*）

rlink指后继节点

tag 空闲0或占用1

size 空间大小

尾部域：uplink指向本结点的头部

* **分配约定**：
  + 选定适当**常量e**，设待分配空闲块、请求分配空间的大小分别为m、n
  + **当m-n≤e时：将整个空闲块分配给用户**；
  + **当m-n>e时：则只分配请求的大小n给用户**；
  + 尽量减少空闲块链表中出现小碎片(容量≤e) ，提高分配效率；减少对空闲块链表的维护工作量
  + 为了避免修改指针，约定**将高地址部分分配给用户**
* **查找约定：**
  + 每次需要查找空闲块时，从**上次刚分配结点的后继结点**开始查找空闲块
  + 作用：提高查找空闲块的速度，防止小容量结点聚集

分配算法：首次拟合法（但不保留小于等于e的剩余量）

回收算法：释放占用块时，检查刚释放的占用块的左、右紧邻是否为空闲块，以便使物理地址毗邻的空闲块合并成一个尽可能大的结点，根据左右空闲情况有四种不同的处理流程：

设释放块的头地址为p，则低地址相邻块尾部为p-1，高地址相邻块头部为p+p->size

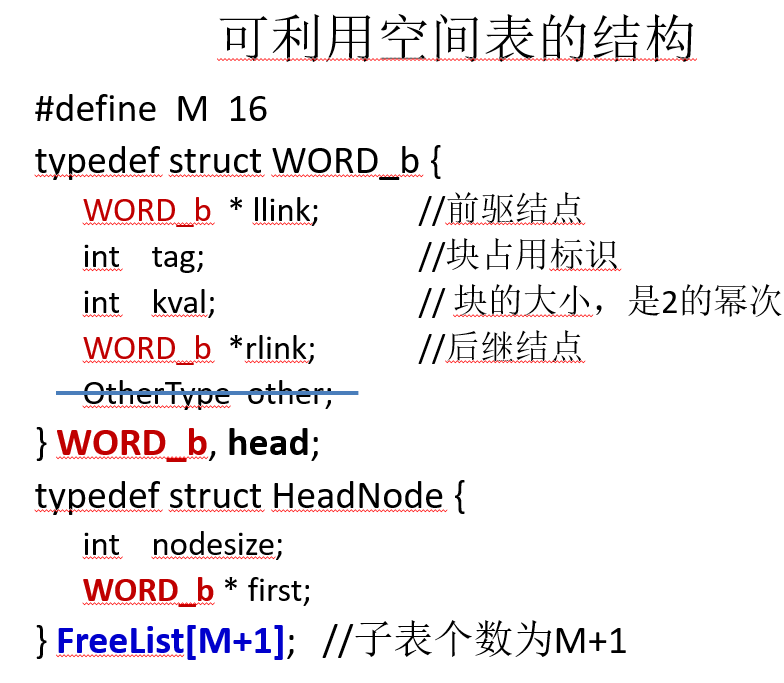
①左右均为占用块：直接插入空闲块，无需其他操作

②左空闲而右占用：和左边**合并**，改变左块的size域，重设左块的尾部指针

③左占用而右空闲：和右边合并，改变右块的size域，重设右块的头部指针

④左右均空闲：和左右合并，改变左块的size域，重设合并后的尾指针（删除右块结点，且要改变右结点后继结点的前驱指针）

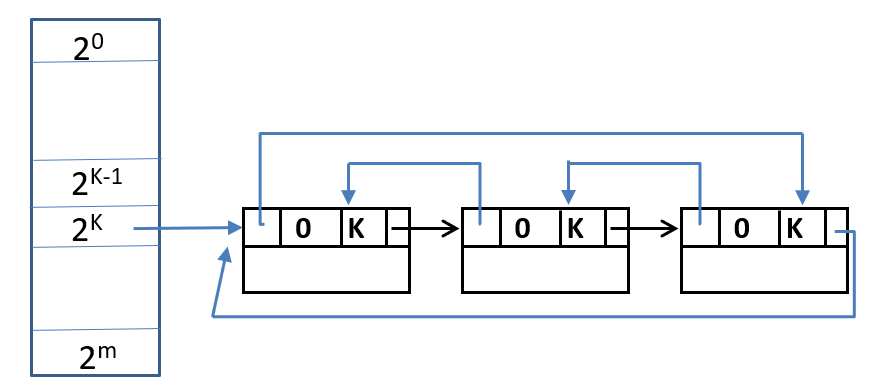
**4. 伙伴系统(Buddy System)**



与边界标识法类似，所不同是：无论占用块或空闲块，其大小均为**2的k次幂**

* 伙伴系统的可利用空间表：
  + 将所有大小相同的空闲块建于一张子表中，每个子表是一个**双重链表**，这样的链表可能有m+1个
  + 再将这m+1个表头指针用**向量结构/数组**组织成一个表

**分配算法：**

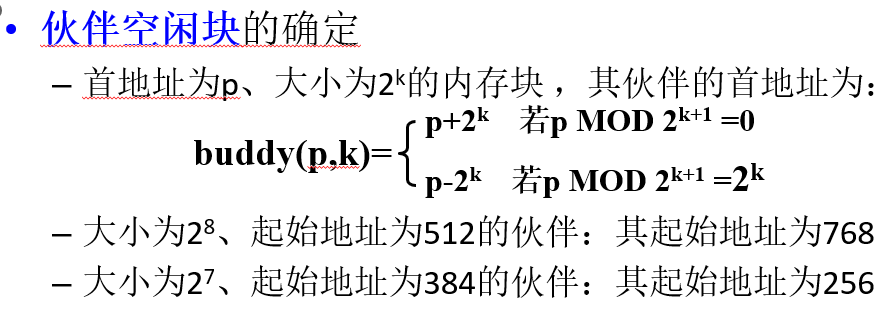


* + 若2k-1<n≤2k-1：第k+1个子表有空闲子表结点，则将子表中的任意一个结点分配之
  + 若**2k-2<n≤2k-1-1**：结点大小为2k-1的子表为空，则从结点大小为2k的子表中找到一个空闲结点，将其中一半分配给程序，剩余的一半插入到结点大小为2k-1的子表中

（若2k到2k+i都为空，则从2k+i+1取结点时，要依次给这些子表分配节点）

**回收算法：**

只有“互为伙伴”的两个子块均空闲时才合并；



* 设要回收的空闲块的首地址是p，其大小为2k：
* (1)判断其 “互为伙伴”的块是否空闲：
* 若不空闲，仅将要回收的空闲块直接插入到相应的子表中；否则转(2)；
* (2)按以下步骤进行空闲块的合并：
  + - 在相应子表中找到其伙伴并删除之；
    - 合并两个空闲块；
* (3)重复(2)，直到合并后的空闲块的伙伴不是空闲块为止
* 特点：算法**简单**；速度**快**；但由于只归并伙伴而容易产生碎片

**5. 无用单元收集/Garbage Collector(GC)**

很多由语言本身提供

无用单元：用户不再使用而系统没有回收的变量和结构

