Search 2

Zibin Zheng（郑子彬）

School of Data and Computer Science，SYSU

http://www.inpluslab.com

课程主页：http://inpluslab.sysu.edu.cn/dsa2016/

哈希(散列)查找

? 查找操作要完成什么任务？

待查值k ⟹ 确定k在存储结构中的位置

? 我们学过哪些查找技术？这些查找技术的共性？

顺序查找、折半查找、二叉排序树查找等。
这些查找技术都是通过一系列的给定值与关键码的比较，查找效率依赖于查找过程中进行的给定值与关键码的比较次数。

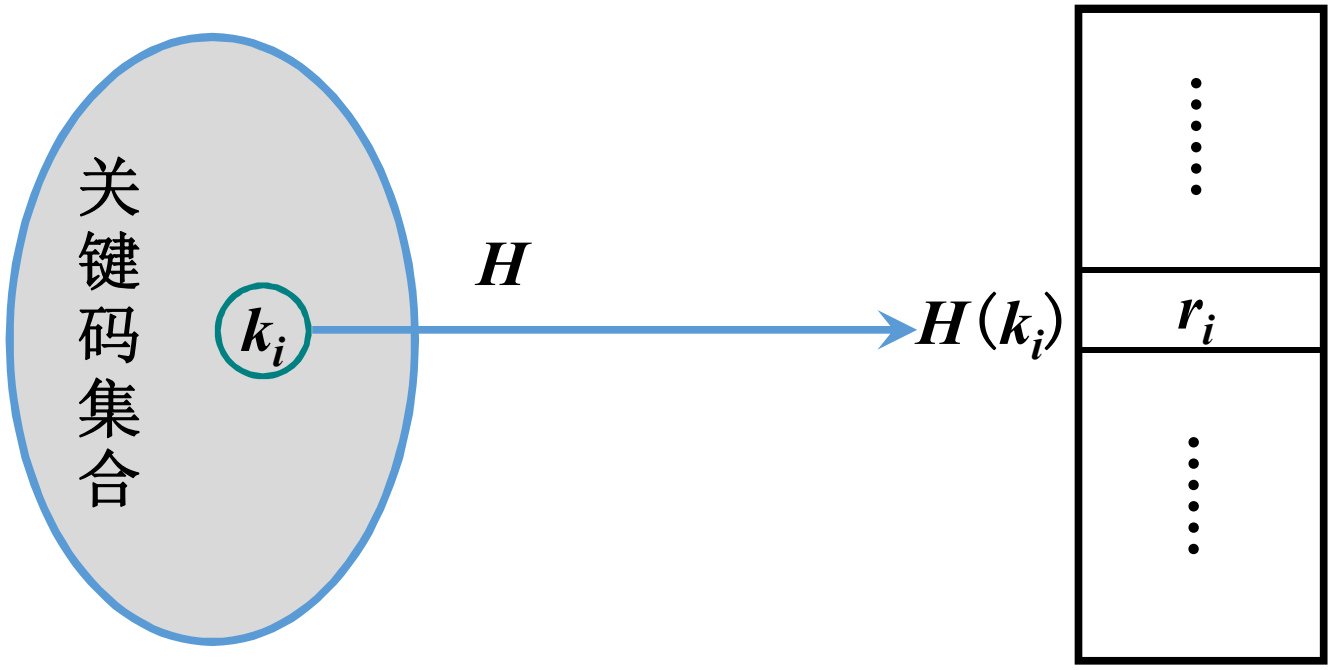
? 能否不用比较，通过关键码直接确定存储位置？

在存储位置和关键码之间建立一个确定的对应关系

散列表的查找技术

概 述

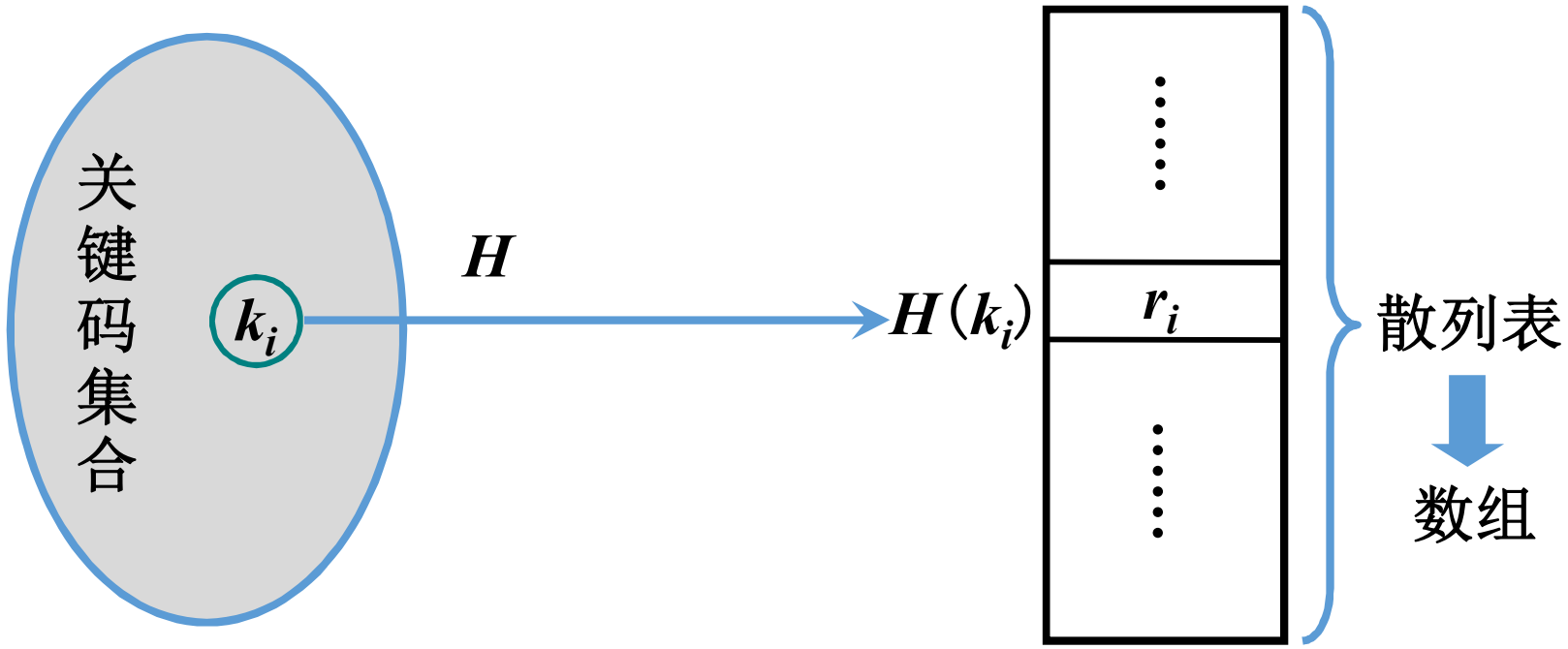
散列的基本思想：在记录的存储地址和它的关键码之间建立一个确定的对应关系。这样，不经过比较，一次读取就能得到所查元素的查找方法。



散列表的查找技术

概 述

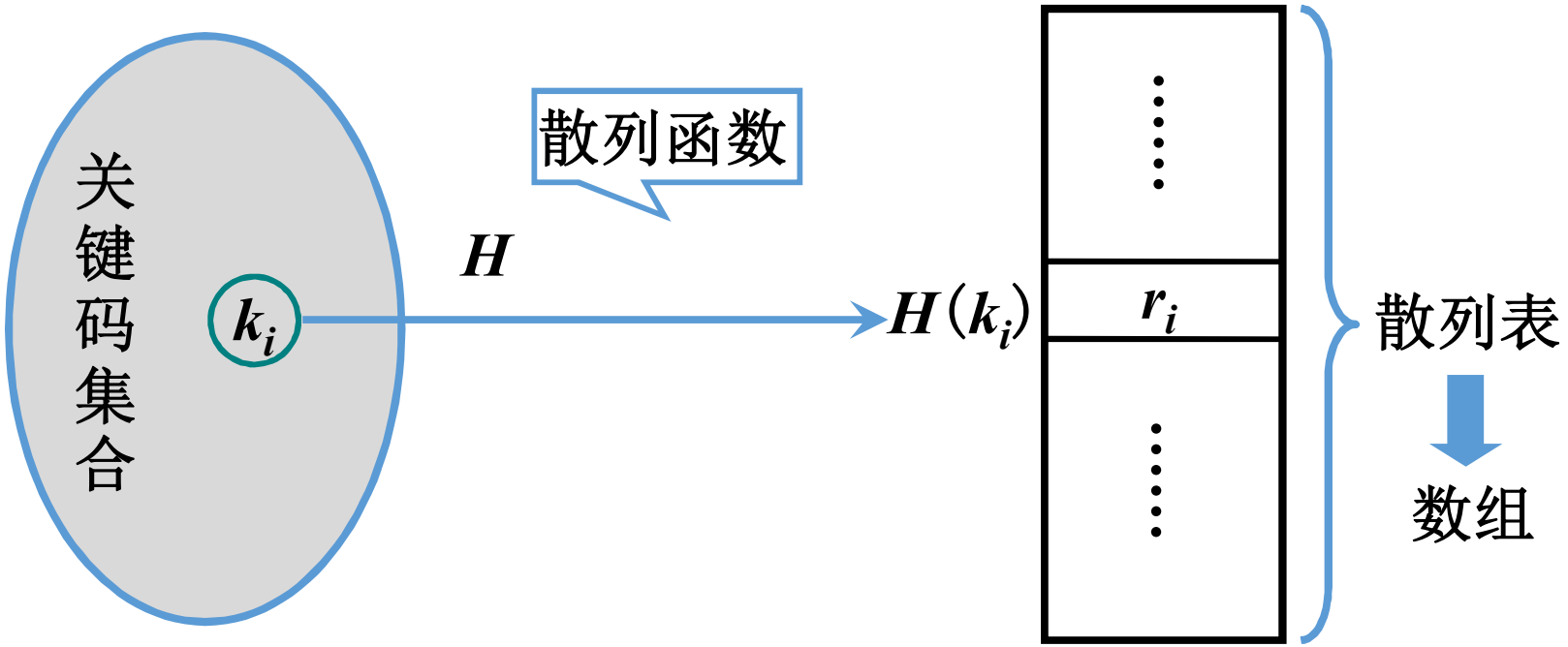
散列表：采用散列技术将记录存储在一块连续的存储空间中，这块连续的存储空间称为散列表。



散列表的查找技术

概 述

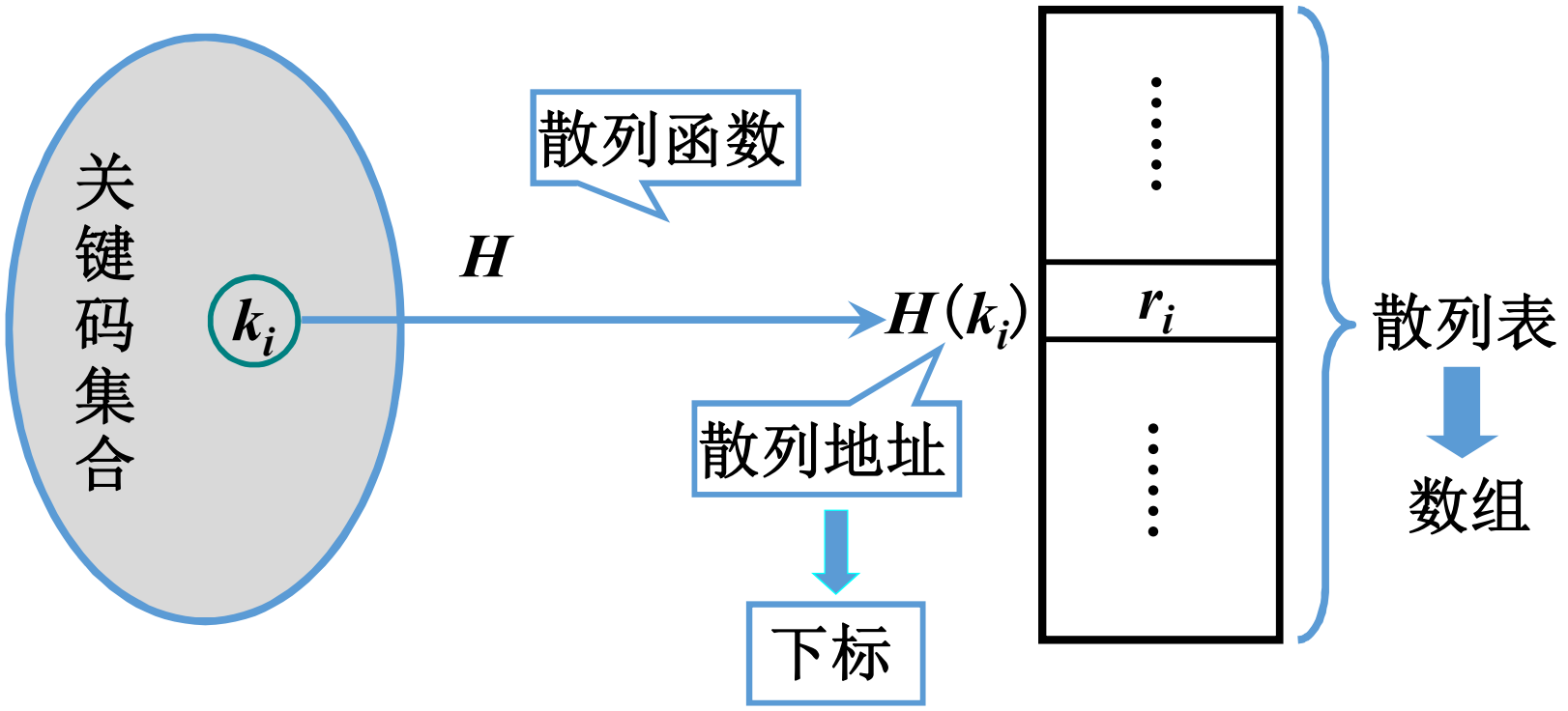
散列函数：将关键码映射为散列表中适当存储位置的函数。



散列表的查找技术

概 述

散列地址： 由散列函数所得的存储地址 。



散列表的查找技术

概 述

散列技术仅仅是一种查找技术吗？

散列既是一种查找技术，也是一种存储技术。

散列是一种完整的存储结构吗？

散列只是通过记录的关键码定位该记录，没有完整地表达记录之间的逻辑关系，所以，散列主要是面向查找的存储结构。

散列表的查找技术

概 述

 散列技术适合于哪种类型的查找？

散列技术一般不适用于允许多个记录有同样关键码的情况。散列方法也不适用于范围查找，换言之，在散列表中，我们不可能找到最大或最小关键码的记录，也不可能找到在某一范围内的记录。

散列技术最适合回答的问题是：如果有的话，哪个记录的关键码等于待查值。

散列表的查找技术

概 述

散列技术的关键问题：

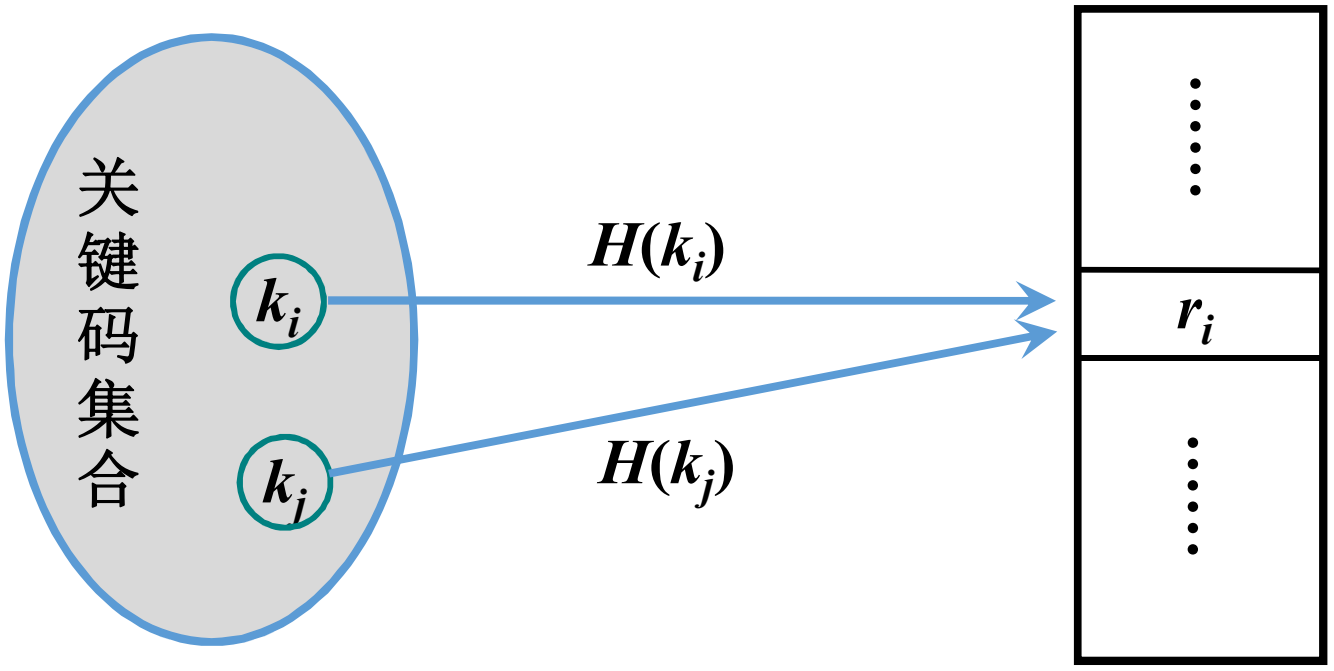
(1) 散列函数的设计。如何设计一个简单、均匀、存储利用率高的散列函数。

(2) 冲突的处理。如何采取合适的处理冲突方法来解决冲突。

散列表的查找技术

概 述

冲突：对于两个不同关键码$k_i \neq k_j$，有$H(k_i) = H(k_j)$，即两个不同的记录需要存放在同一个存储位置，k_i和k_j相对于H称做同义词。



散列表的查找技术

散列函数

设计散列函数一般应遵循以下原则：

(1) 计算简单。散列函数不应该有很大的计算量，否则会降低查找效率。

(2) 函数值即散列地址分布均匀。函数值要尽量均匀散布在地址空间，这样才能保证存储空间的有效利用并减少冲突。

散列表的查找技术

散列函数——直接定址法

散列函数是关键码的线性函数，即：

$$H(key) = a \times key + b \quad （a，b为常数）$$

例：关键码集合为{10, 30, 50, 70, 80, 90}，选取的散列函数为$H(key)=key/10$，则散列表为：

0	1	2	3	4	5	6	7	8	9
	10		30		50		70	80	90

适用情况？

事先知道关键码，关键码集合不是很大且连续性较好。

散列表的查找技术

散列函数——除留余数法

散列函数为：

$$H(key)=key \bmod p$$

如何选取合适的p，产生较少同义词？

例： $p=21=3\times7$

关键码	14	21	28	35	42	49	56
散列地址	14	0	7	14	0	7	14

散列表的查找技术

散列函数——除留余数法

一般情况下，选p为小于或等于表长（最好接近表长）的最小素数或不包含小于20质因子的合数。

适用情况？

除留余数法是一种最简单、也是最常用的构造散列函数的方法，并且不要求事先知道关键码的分布。

散列表的查找技术

散列函数——数字分析法

根据关键码在各个位上的分布情况，选取分布比较均匀的若干位组成散列地址。

例：关键码为8位十进制数，散列地址为2位十进制数

①	②	③	④	⑤	⑥	⑦	⑧	
8	1	3	4	6	5	3	2	
8	1	3	7	2	2	4	2	
8	1	3	8	7	4	2	2	
8	1	3	0	1	3	6	7	
8	1	3	2	2	8	1	7	
8	1	3	3	8	9	6	7	

散列表的查找技术

散列函数——数字分析法

适用情况：

能预先估计出全部关键码的每一位上各种数字出现的频度，不同的关键码集合需要重新分析。

散列表的查找技术

散列函数——平方取中法

对关键码平方后，按散列表大小，取中间的若干位作为散列地址（平方后截取）。

例：散列地址为2位，则关键码123的散列地址为：

$(1234)^2 = 1522756$

适用情况：

事先不知道关键码的分布且关键码的位数不是很大。

散列表的查找技术

散列函数——折叠法

将关键码从左到右分割成位数相等的几部分，将这几部分叠加求和，取后几位作为散列地址。

例：设关键码为**2 5 3 4 6 3 5 8 7 0 5**，散列地址为三位。

```
  2 5 3
  4 6 3
  5 8 7
+   0 5
-------
  1 3 0 8
```

移位叠加

```
  2 5 3
  3 6 4
  5 8 7
+   5 0
-------
  1 2 5 4
```

间界叠加

适用情况：

关键码位数很多，事先不知道关键码的分布。

散列表的查找技术

随机数法

取关键字的随机函数值作哈希地址，即H(key) = random(key)

当散列表中关键字长度不等时，该方法比较合适。

选取哈希函数，考虑以下因素

- 计算哈希函数所需时间；
- 关键字的长度；
- 哈希表长度（哈希地址范围）；
- 关键字分布情况；
- 记录的查找频率。

散列表的查找技术

处理冲突的方法——开放定址法

由关键码得到的散列地址一旦产生了冲突，就去寻找下一个空的散列地址，并将记录存入。

如何寻找下一个空的散列地址？

（1）线性探测法

（2）二次探测法

（3）随机探测法

散列表的查找技术

线性探测法

当发生冲突时，从冲突位置的下一个位置起，依次寻找空的散列地址。

对于键值key，设$H(key)=d$，闭散列表的长度为m，则发生冲突时，寻找下一个散列地址的公式为：

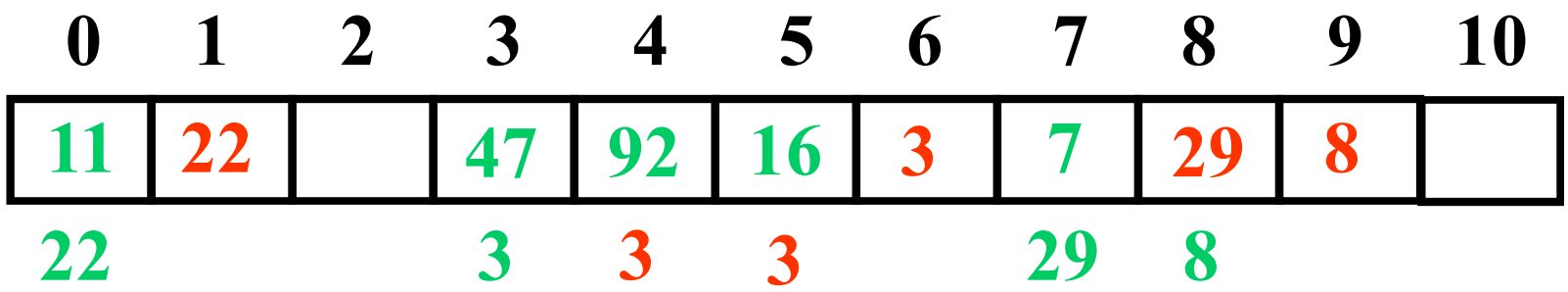
$$H_i=(H(key)+d_i)\ \%\ m \quad (d_i=1，2，\ldots，m-1)$$

用开放定址法处理冲突得到的散列表叫闭散列表。

散列表的查找技术

线性探测法

例：关键码集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列表表长为11，散列函数为$H(key)=key \bmod 11$，用线性探测法处理冲突，则散列表为：



堆积：在处理冲突的过程中出现的非同义词之间对同一个散列地址争夺的现象。

散列表的查找技术

在线性探测法构造的散列表中查找算法——伪代码

1. 计算散列地址j;
2. 若ht[j]等于k，则查找成功，返回记录在散列表中的下标；
   否则执行第3步；
3. 若ht[j]为空或整个散列表探测一遍，则查找失败，转4；
   否则，j指向下一单元，转2；
4. 若整个散列表探测一遍，则表满，抛出溢出异常；
   否则，将待查值插入；

散列表的查找技术

在线性探测法构造的散列表中查找算法——C++描述

```
int HashSearch1(int ht[ ], int m, int k)
{
   j = H(k);                          //计算散列地址
   if (ht[j] == k) return j;          //没有发生冲突，比较一次查找成功
   else if (ht[j] == Empty) {ht[j] = k; return 0; }   //查找不成功，插入
   i = (j + 1) % m;                   //设置探测的起始下标
   while (ht[i] != Empty && i != j)
   {
      if (ht[i] == k) return i;       //发生冲突，比较若干次查找成功
      else i = (i + 1) % m;           //向后探测一个位置
   }
   if (i == j) throw "溢出";
   else {ht[i] = k; return 0; }       //查找不成功，插入
}
```

散列表的查找技术

二次探测法

当发生冲突时，寻找下一个散列地址的公式为：

$$H_i=(H(key)+d_i)\%\ m$$

（$d_i=1^2$，-1^2，2^2，-2^2，…，q^2，$-q^2$且$q\leqslant m/2$）

散列表的查找技术

二次探测法

例：关键码集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列表表长为11，散列函数为$H(key)=key \bmod 11$，用二次探测法处理冲突，则散列表为：

0	1	2	3	4	5	6	7	8	9	10
11	22	3	47	92	16		7	29	8	
22			3	3			29	8		

散列表的查找技术

随机探测法

当发生冲突时，下一个散列地址的位移量是一个随机数列，即寻找下一个散列地址的公式为：

$$H_i=(H(\text{key})+d_i)\ \%\ m$$

（d_i是一个随机数列，i=1，2，……，m−1）

计算机中产生随机数的方法通常采用线性同余法，

$$\begin{cases} a_0 = d \\ a_n = (ba_{n-1}+c) \bmod m \qquad n = 1, 2, \cdots \end{cases}$$

其中，d称为随机种子。当b、c和m的值确定后，给定一个随机种子，产生确定的随机数序列。

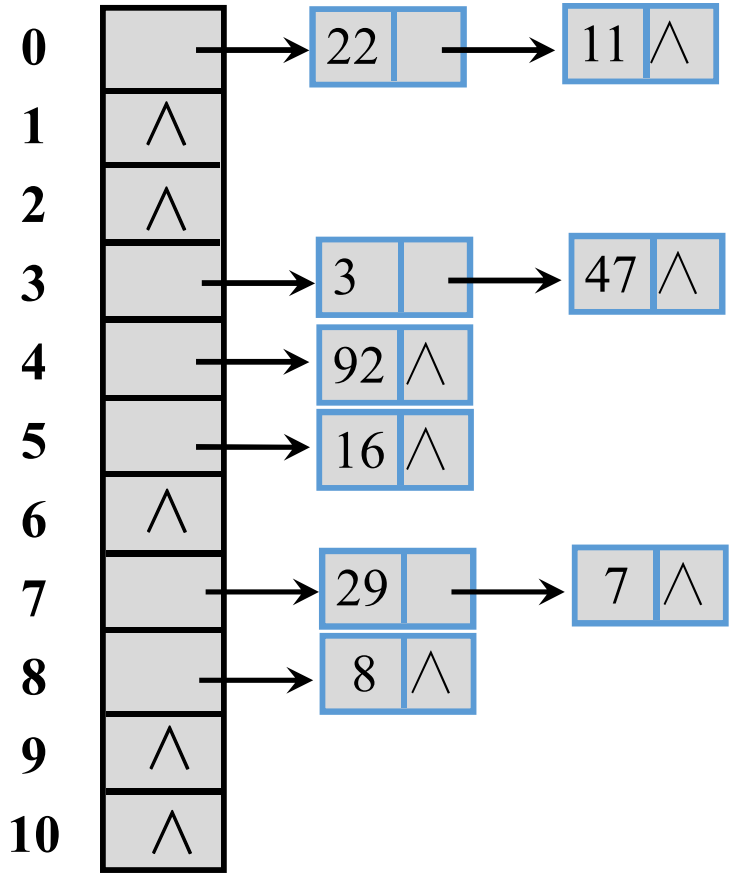
散列表的查找技术

处理冲突的方法——拉链法（链地址法）

- 基本思想：将所有散列地址相同的记录，即所有同义词记录存储在一个单链表中（称为同义词子表），在散列表中存储的是所有同义词子表的头指针。
- 用拉链法处理冲突构造的散列表叫做开散列表。
- 开散列表不会出现堆积现象。
- 设n个记录存储在长度为m的散列表中，则同义词子表的平均长度为n / m。

散列表的查找技术

例：关键码集合 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列函数为 $H(key)=key \bmod 11$，用拉链法处理冲突，构造的开散列表为：



散列表的查找技术

在拉链法构造的散列表查找算法——伪代码

1. 计算散列地址j；
2. 在第j个同义词子表中顺序查找；
3. 若查找成功，则返回结点的地址；

   否则，将待查记录插在第j个同义词子表的表头。

散列表的查找技术

在拉链法构造的散列表查找算法——C++描述

```
Node<int> *HashSearch2(Node<int> *ht[ ], int m, int k)
{
    j = H(k);
    p = ht[j];
    while (p != NULL && p->data != k)
        p = p->next;
    if (p->data == k) return p;
    else {
        q = new Node<int>; q->data = k;
        q->next = ht[j];
        ht[j] = q;
    }
}
```

散列表的查找技术

散列查找的性能分析

- 由于冲突的存在，产生冲突后的查找仍然是给定值与关键码进行比较的过程。
- 在查找过程中，关键码的比较次数取决于产生冲突的概率。影响冲突产生的因素有：
  - （1）散列函数是否均匀
  - （2）处理冲突的方法
  - （3）散列表的装载因子

$$\alpha = \frac{\text{表中填入的记录数}}{\text{散列表的长度}}$$

散列表的查找技术

几种处理冲突方法的平均查找长度

平均查找长度 / 处理冲突的方法	查找成功时	查找不成功时
线性探测法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
二次探测法	$-\frac{1}{\alpha}\ln(1+\alpha)$	$\frac{1}{1-\alpha}$
拉链法	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$

散列表的平均查找长度是装填因子 α 的函数，而不是查找集合中记录个数n的函数。在很多情况下，散列表的空间都比查找集合大，此时虽然浪费了一定的空间，但换来的是查找效率。

开散列表与闭散列表的比较

	堆积现象	结构开销	插入/删除	查找效率	估计容量
开散列表	不产生	有	效率高	效率高	不需要
闭散列表	产生	没有	效率低	效率低	需要

Case

- 在地址空间为0~13的散列区中，对以下关键字 {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec} 构造哈希表，设哈希函数为 H(key)=$\lfloor i/2 \rfloor$，其中i为key中首字母在字母表中的序号（A对应序号1）。
- （1）用线性探测再散列方法得到哈希表；
- （2）计算查找成功和查找失败时的平均查找长度。

Ad.	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Key	Apr	Aug	Dec	Feb		Jan	Mar	May	Jun	Jul	Sep	Oct	Nov	
M	1	2	1	1		1	1	2	4	5	2	5	6	

表中Ad.为地址号，M为地址计算次数

查找成功：AVL成功 = 总的地址计算次数/填入记录项数 = 31/12

查找失败：AVL失败 = [(5+4+3+2+1)+(9+8+7+6+5+4+3+2+1)]/14 = 60/14

动态存储管理

Overview

- 程序执行过程中，(数据)结构中的每一个数据元素都对应一定的存储空间，数据元素的访问都是通过对应的存储单元来进行的。
- 存储空间的分配与管理是由操作系统或编译程序负责实现的，是一个复杂而又重要的问题，现代的存储管理往往采用动态存储管理思想。
- 动态存储管理：如何根据“存储请求”分配内存空间？如何回收被释放的(或不再使用的)内存空间？

Overview

- 对于允许进行动态存储分配的程序设计语言，操作系统在内存中划出一块地址连续的大区域(称为堆)，由设计者在程序中利用语言提供的内存动态分配函数(如C的malloc()，calloc()，free()函数，C++的new，delete函数等)来实现对堆的使用。

1 两个基本概念

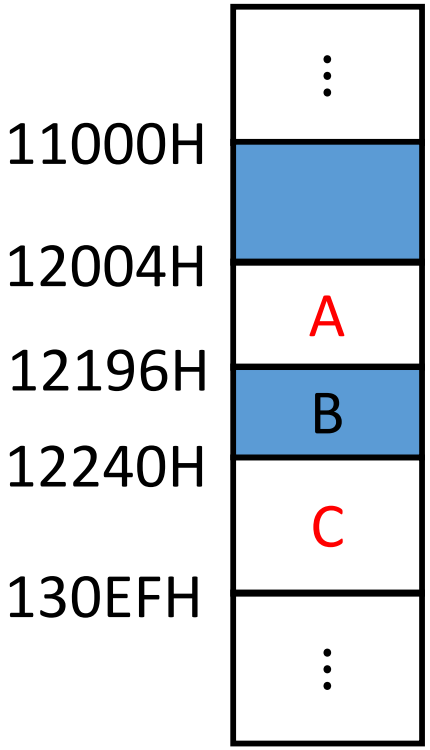
- 占用块：已分配给用户使用的一块地址连续的内存区域；
- 空闲块：未曾分配的地址连续的内存区域；

内存分配方式

- 当有用户程序进入系统请求分配内存时，系统有两种处理方式：
- (1) 系统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；
- (2) 用户程序一旦运行结束，便将它所占内存区释放成为空闲块，同时，每当新用户请求分配内存时，系统需要巡视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。
- 对于(2)的情况，系统需建立一张“可利用空间表” 。

内存分配方式

- 程序运行过程中，不断地对堆中的部分区域进行分配和释放，堆中会出现占用块和空闲块交错的状态，如下图所示。



堆的状态

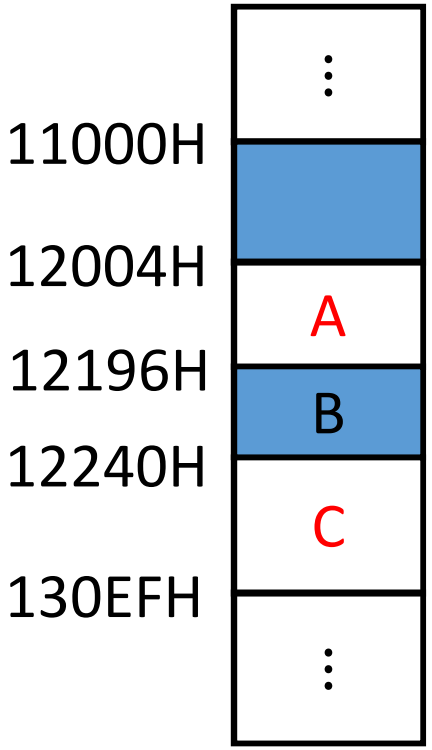
动态存储分配的基本问题

(1) 当某一时刻用户程序请求分配400个字节的存储空间，如何分配?

- 将块A分配给用户程序?
- 从大块C中划出一部分分配给用户程序?

(2) 当某一时刻分配B块的用户程序运行结束，B块要进行回收，如何回收?

- B块直接回收并成为一个独立的空闲块?
- B块回收并和前、后的空闲块A、C合并后形成一个更大的空闲块?



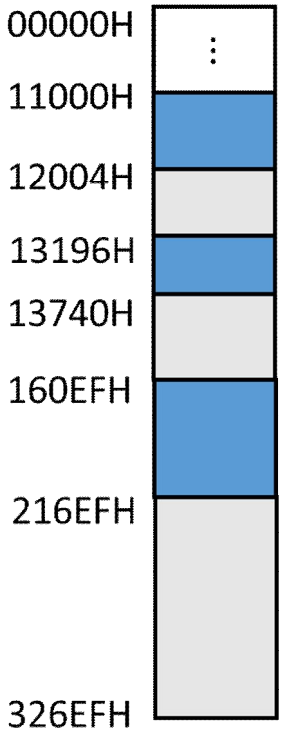
堆的状态

可利用空间表及分配方法

- 可利用空间表的方式：
  - 包含所有可分配的空闲块。
- 当用户请求分配时，系统从可利用空间表中删除一个结点分配之；
- 当用户释放其所占内存时，系统即回收并将它插入到可利用空间表中。
- 因此，可利用空间表亦称做“存储池”。

可利用空间表的组织

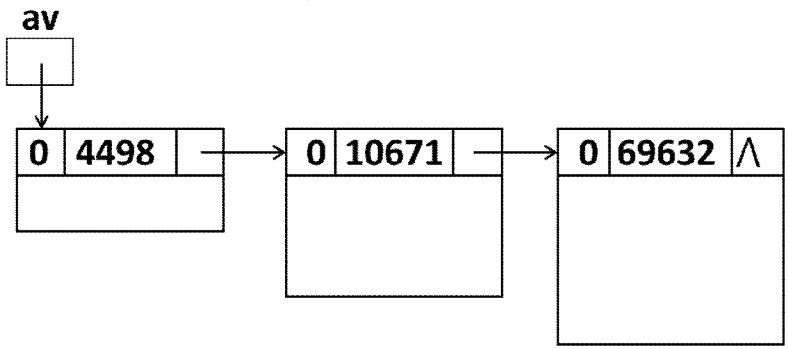
- 可用空间表的组织有两种方式：目录表方式和链表方式，如下图所示。动态存储管理中需要不断地进行空闲块的分配和释放，对目录表来说管理复杂，因此，可利用空间表通常以链表方式组织。



(a) 堆的状态

起始地址	空闲块大小	使用情况
12004H	4498	空闲
13740H	10671	空闲
216EFH	69632	空闲

(b) 目录表方式



(c) 链表方式

动态存储管理过程中的内存状态和空闲表结构

可利用空间表的组织

当可利用空间表以链表方式组织时，每个空闲块就是链表中的一个结点。

- 分配时：从链表中找到一个合适的结点加以分配，然后将该结点删除之；
- 回收时：将空闲块插入到链表中。

实际的动态存储管理实施时，具体的分配和释放的策略取决于结点(空闲块)的结构。

结点结构方式与分配策略

1 请求分配的块大小相同

将进行动态存储分配的整个内存区域(堆)按所需大小分割成若干大小相同的块，然后用指针链接成一个可利用空间表。

- 分配时：从表的首结点分配，然后删除该结点；
- 回收时：将释放的空闲块插入表头。

结点结构方式与分配策略

2 请求分配的块大小只有几种规格

根据统计概率事先对动态分配的堆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。

◆ 分配时：根据请求的大小，将最接近该大小的某个链表的首结点分配给用户。若剩余部分正好差不多是另一种规格大小，则将剩余部分插入到另一种规格的链表中，然后删除该结点；

◆ 回收时：只要将所释放的空闲块插入到相应大小的表头。

存在的问题：

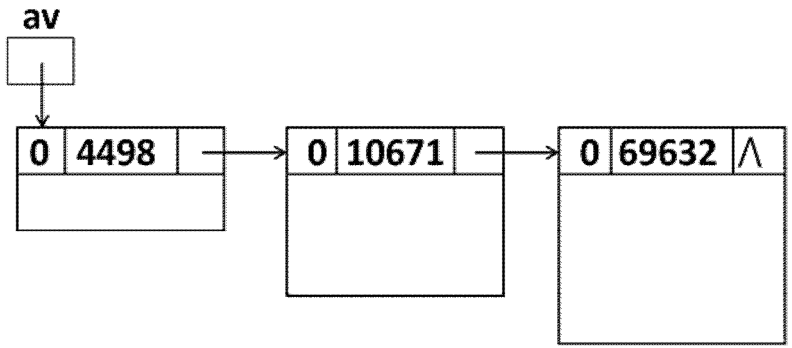
- 当请求分配的块空间大小比最大规格的结点还大时，分配不能进行。而实际上内存空间却可能存在比所需大小还要大的的连续空间，应该能够分配。

结点结构方式与分配策略

3 请求分配的块大小不确定

系统开始时，整个堆空间是一个空闲块，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。

由于链表中结点大小不同，结点中除标志域和链域之外，尚需有一个结点大小域(size)，以保存空闲块的大小，如图 。



(c) 链表方式

问题：若用户请求分配大小为n(kB)的内存，而链表中有一些规模不小于n的空闲块时，如何分配?有3种分配策略。

结点结构方式与分配策略

(1) 首次拟合法(First fit)

- 分配时：从表头指针开始查找可利用空间表，将找到的第一个不小于n的空闲块的部分(所需要大小)分配给用户，剩下部分仍然是一个空闲块结点；
- 回收时：将释放的空闲块插入在链表的表头。

特点：分配时随机的；回收时仅需插入到表头。

结点结构方式与分配策略

(2) 最佳拟合法(Best fit)

- 分配时：扫描整个可利用空间链表，找到一个大小满足要求且最接近n空闲块，将其中的一部分(所需要大小)分配给用户，剩下部分仍然是一个空闲块结点；
- 回收时：只要将释放的空闲块插入到链表的合适位置。

为了使分配时不需要扫描整个可利用空间链表，链表组织(块回收时)成按从小到大排序(升序) 。

优点：适用于请求分配的内存块大小范围较广的系统；

缺点：系统容易产生无法分配的内存碎片；无论分配与回收，都需要查找表，最费时；

结点结构方式与分配策略

(3) 最差拟合法(Worst fit)

- 分配时：扫描整个可利用空间链表，找到一个大小最大的空闲块，将其中的一部分(所需要大小)分配给用户，剩下部分仍然是一个空闲块结点；
- 回收时：只要将释放的空闲块插入到链表的合适位置。

为了使分配时不需要扫描整个可利用空间链表，链表组织(块回收时)成按从大到小排序(降序) 。

特点：适用于请求分配的内存块的大小范围较窄的系统；分配无需查找，回收需要查找适当的位置。

边界标识法

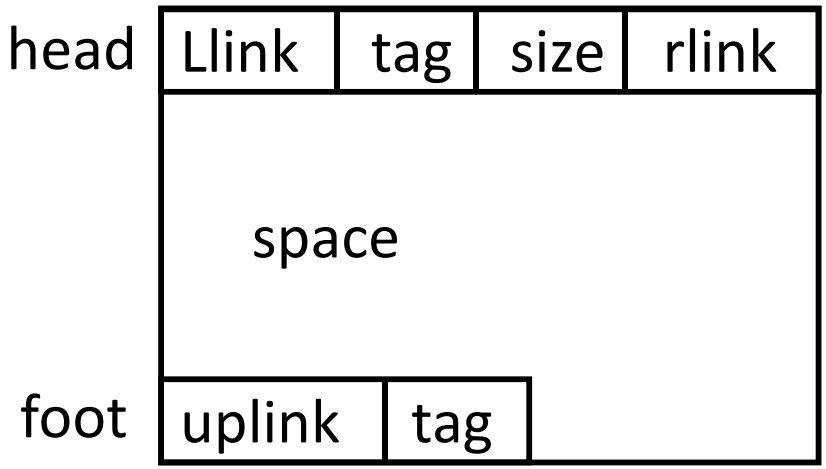
- 边界标识法(Boundary Tag Method)是操作系统中一种常用的进行动态分配的存储管理方法。
- 系统将所有的空闲块链接成一个双重循环链表，分配可采用几种方法(前述) 。

系统的特点

- 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，在回收块时易于判别在物理位置上与其相邻的内存区域是否为空闲块，以便于将所有地址连续的空闲存储区合并成一个尽可能大的空闲块。

可利用空闲表结点结构

```
typedef  struct word
    {  Union
        {  struct word *llink;
           struct word *uplink;
        };
        int tag;
        int size;
        struct word  *rlink;
        OtherType other;
    }WORD, head, foot, *Space;
#define FootLoc(p) p+p->size-1
```



分配算法

分配算法比较简单，可采用前述三种方法中的任一种进行分配。设采用首次拟合法，为了使系统更有效地运行，在边界标识法中还做了两条约定：

① 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；

作用：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。

② 空闲块链表中的结点数可能很多，为了提高查找空闲块的速度和防止小容量结点密集，每次查找时从不同的结点开始——上次刚分配结点的后继结点开始。

分配算法实现

```
Space AllocBoundTag( Space  *pav, int n )
    {  p = pav ;
       for  ( ; p &&p->size<n && p ->rlink!=pav; p=p->rlink ) % 查找不少于n的空闲块
          if ( !p || p->size<n ) return NULL ;
          else
             {  f=FootLoc( p ) ; Pav=p->rlink ;  % p指向找到的空闲块
               if  ( p->size–n<=e )              % 整块分配，不保留 <=e 的剩余量
              {  if  ( pav==p )  pav=NULL ;
                    else                        % 在表中删除分配的结点
                      {  pav->llink=p->link ;
                         p->llink->rlink=pav ;  }
                    p->tag=f->tag=1 ;
                 }
               else    % 分配该块的后n个字
                 {  f->tag=1 ; p->size-=n ; f= FootLoc( p ) ;
                    f->tag= 0 ; f->uplink=p ; p=f+1;
                    p->tag=1 ; p->size=n ;
                 }
             return p ;  % 返回分配块的首地址
          }
    }
```

回收算法

- 当用户释放占用块，系统需立即回收以备新的请求产生时进行再分配。关键的是使物理地址毗邻的空闲块合并成一个尽可能大的结点，则需检查刚释放的占用块的左、右紧邻是否为空闲块。
- 假设所释放的块的头地址为p，则与其低地址紧邻的块的底部地址为p-1；与其高地址紧邻的块的头地址为p+p->size，它们中的标志域就表明了两个相邻块的使用状况：
  - 若(p-1)->tag=0 ：则左邻块为空闲块；
  - 若(p+p->size)->tag=0 ：则右邻块为空闲块；

回收算法

- 回收算法需要考虑的4种情况：

(1) 释放块的左、右邻块均为占用块

将被释放块简单地插入到空闲块链表中即可。

```
p->tag=0 ; FootLoc(p)->uplink=p ;
FootLoc(p)->tag=0 ;
if ( !pav ) pav=p->llink=p->rrlink=p ;
else
    { q=pav->llink ; P->rlink=pav ;
      p->llink=q ; q->rlink=pav->llink=p ;
      Pav=p ;
    }
```

(2) 释放块的左邻块空闲而右邻块为占用

和左邻块合并成一个大的空闲块结点，改变左邻块的size域及重新设置(合并后)结点的底部。

```
n=p->size ; s=(p-1)->uplink ; s->size+=n;
f=p+n–1 ; f->uplink=s ; f->tag=0 ;
```

(3) 释放块的左邻占用而右邻空闲

和右邻块合并成一个大的空闲块结点，改变右邻块的size域及重新设置(合并后)结点的头部。

```
t=p+p->size ; p->tag=0 ; q=t->llink ; p->llink=q ;
q->rlink=p ; q1=t->rlink ; p->rlink=q1 ;
q1->llink=p ; p->size+=t->size ; FootLoc(t)-
>uplink=p ;
```

(4) 释放块的左、右邻块均为空闲块

和左、右邻块合并成一个大的空闲块结点，改变左邻块的size域及重新设置(合并后)结点的底部。

```
n=p->size ; s=(p-1)->uplink ; t=p+p->size ;
s->size+=n+t->size ;  q=t->llink ; q1=t->rlink ;
q->rlink=q1 ; q1->llink=q ;
FootLoc(t)->uplink=s;
```

伙伴系统

- 伙伴系统是一种非顺序内存管理方法，不是以顺序片段来分配内存，是把内存分为两个部分，只要有可能，这两部分就可以合并在一起; 且这两部分从来不是自由的，程序可以使用伙伴系统中的一部分或者两部分都不使用。
- 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。

可利用空间表的结构

- 为了再分配时查找方便起见，我们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，将这m+1个表头指针用向量结构组织成一个表，这就是伙伴系统的可利用空间表。
- 可利用空间表的数据类型描述如下：

```
#define  M  16
typedef struct WORD_b
    {  WORD_b  * llink;    /*  前驱结点  */
       int    tag;      /*  使用标识  */
       int    kval;     /*  块的大小,是2的幂次  */
       WORD_b  *rlink;    /*  后继结点  */
       OtherType    other;
    } WORD_b, head;
typedef  struct HeadNode
    {  int    nodesize;
       WORD_b * first;
    }FreeList[M+1];
```

分配算法

当程序提出大小为n的内存分配请求时，首先在可利用表中查找大小与n相匹配的子表.

1 算法思想

- 若存在$2^{k-1}<n\leq 2^k-1$的空闲子表结点：则将子表中的任意一个结点分配之；
- 若不存在$2^{k-1}<n\leq 2^k-1$的空闲子表结点：则从结点大小为2^k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2^{k-1}的子表中。

2 说明

在进行大小为n($2^{k-i-1}<n\leq 2^{k-i}-1$，i=1,2,...,k-1) 的内存分配请求时，若所有小于2^k的子表均为空(没有空闲结点)，则同样需要从大小为2^k的子表中找到一个空闲结点，将其中2^{k-i}一小部分分配给用户，而将剩余部分分割成若干个结点分别插入对应的子表。

回收算法

- 当程序释放所占用的块时，系统将该新的空闲块插入到可利用空闲表中，需要考虑合并成大块问题。在伙伴系统中，只有“互为伙伴”的两个子块均空闲时才合并；即使有两个相邻且大小相同的空闲块，如果不是“互为伙伴”（从同一个大块中分裂出来的)也不合并。

1 伙伴空闲块的确定

设p是大小为2^k的空闲块的首地址，且p MOD 2^{k+1} =0，则首地址为p和p+2^k的两个空闲块“互为伙伴”。

首地址为p大小为2^k的内存块的，其伙伴的首地址为：

$$buddy(p,k)=\begin{cases} p+2^k & \text{若} p \text{ MOD } 2^{k+1}=0 \\ P-2^k & \text{若} p \text{ MOD } 2^{k+1}=2^k \end{cases}$$

回收算法

2 回收算法

设要回收的空闲块的首地址是p，其大小为2^k的，算法思想是：

(1) 判断其 “互为伙伴” 的两个空闲块是否为空：

若不为空，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。

系统的特点：算法简单；速度快；但容易产生碎片。

伙伴系统例子

- 伙伴系统的宗旨就是用最小的内存块来满足内核的对于内存的请求。
- 在最初，只有一个块，也就是整个内存，假如为1M大小，而允许的最小块为64K，那么当我们申请一块200K大小的内存时，就要先将1M的块分裂成两等分，各为512K，这两分之间的关系就称为伙伴，
- 然后再将第一个512K的内存块分裂成两等分，各位256K，将第一个256K的内存块分配给内存，这样就是一个分配的过程。
- 下面我们结合示意图来了解伙伴系统分配和回收内存块的过程。

伙伴系统例子

Step	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
1	2^4															
2.1	2^3								2^3							
2.2	2^2				2^2				2^3							
2.3	2^1		2^1		2^2				2^3							
2.4	2^0	2^0	2^1		2^2				2^3							
2.5	A: 2^0	2^0	2^1		2^2				2^3							

1 初始化时，系统拥有1M的连续内存，允许的最小的内存块为64K，图中白色的部分为空闲的内存块，着色的代表分配出去了得内存块。

2 程序A申请一块大小为34K的内存，对应的order为0，即2^0=1个最小内存块

2.1 系统中不存在order 0(64K)的内存块，因此order 4(1M)的内存块分裂成两个order 3的内存块(512K)；

2.2 仍然没有order 0的内存块，因此order 3的内存块分裂成两个order 2的内存块(256K)；

2.3 仍然没有order 0的内存块，因此order 2的内存块分裂成两个order 1的内存块(128K)；

2.4 仍然没有order 0的内存块，因此order 1的内存块分裂成两个order 0的内存块(64K)；

2.5 找到了order 0的内存块，将其中的一个分配给程序A，现在伙伴系统的内存为一个order 0的内存块，一个order1的内存块，一个order 2的内存块以及一个order 3的内存块。

伙伴系统例子

Step	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
3	A: 2^0	2^0	B: 2^1		2^2				2^3							
4	A: 2^0	C: 2^0	B: 2^1		2^2				2^3							
5.1	A: 2^0	C: 2^0	B: 2^1		2^1		2^1		2^3							
5.2	A: 2^0	C: 2^0	B: 2^1		D: 2^1		2^1		2^3							
6	A: 2^0	C: 2^0	2^1		D: 2^1		2^1		2^3							

3 程序B申请一块大小为66K的内存，对应的order为1，即2^1=2个最小内存块，由于系统中正好存在一个order 1的内存块，所以直接用来分配；

4 程序C申请一块大小为35K的内存，对应的order为0，同样由于系统中正好存在一个order 0的内存块，直接用来分配；

5 程序D申请一块大小为67K的内存，对应的order为1；

5.1 系统中不存在order 1的内存块，于是将order 2的内存块分裂成两块order 1的内存块；

5.2 找到order 1的内存块，进行分配；

6 程序B释放了它申请的内存，即一个order 1的内存块；

伙伴系统例子

Step	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
7.1	A: 2^0	C: 2^0	2^1		2^1		2^1		2^2							
7.2	A: 2^0	C: 2^0	2^1		2^2				2^3							
8	2^0	C: 2^0	2^1		2^2				2^3							
9.1	2^0	2^0	2^1		2^2				2^3							
9.2	2^1		2^1		2^2				2^3							
9.3	2^2				2^2				2^3							
9.4	2^3								2^3							
9.5	2^4															

7 程序D释放了它申请的内存

7.1 一个order 1的内存块回收到内存当中；

7.2 由于该内存块的伙伴也是空闲的，因此两个order 1的内存块合并成一个order 2的内存块；

8 程序A释放了它申请的内存，即一个order 0的内存块

9 程序C释放了它申请的内存

9.1 一个order 0的内存块被释放；

9.2 两个order 0伙伴块都是空闲的，进行合并，生成一个order 1的内存块；

9.3 两个order 1伙伴块都是空闲的，进行合并，生成一个order 2的内存块；

9.4 两个order 2伙伴块都是空闲的，进行合并，生成一个order 3的内存块；

9.5 两个order 3伙伴块都是空闲的，进行合并，生成一个order 4的内存块。