Hash Indexing

Hash-based Indexing:

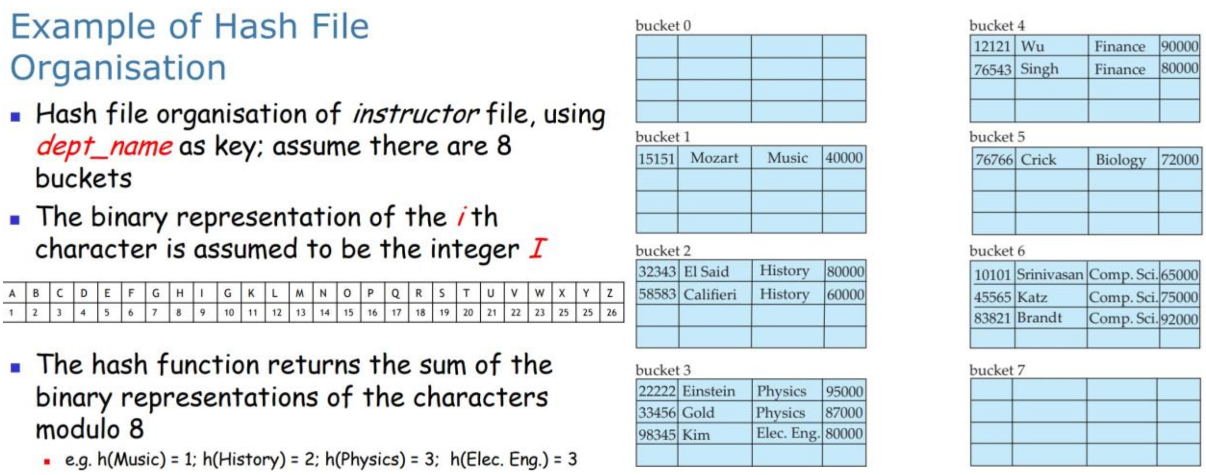
Static Hashing [静态哈希]:

A bucket is a unit of storage containing one or more records (a bucket is typically a disk block).
[桶是包含一条或多条记录的存储单元(桶通常是一个磁盘块)]
Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B, Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B.
[哈希函数 h 是一个从所有搜索键值 K 的集合到所有桶地址 B 的集合的函数，哈希函数 h 是一个从所有搜索键值 K 的集合到所有桶地址 B 的集合的函数]
Records with different search-key values may be mapped to the same bucket; thus, entire bucket has to be searched sequentially to locate a record.
[具有不同搜索键值的记录可以映射到同一个桶；因此，必须依次搜索整个桶来定位一条记录]

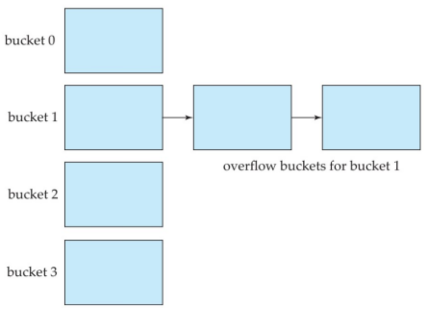
Good or bad case:

Worst hash function maps all search-key values to the same bucket, an ideal hash function is uniform and random.
i.e. If we have N buckets, numbered 0 to N-1, a hash function h of the following form works well in practice:

h(value) = (a*value + b) mod N



i.e. key(music) = mod ((13+21+19+9+3), 8) = 1
There comes a question: bucket overflow can occur because of insufficient [不足] buckets or skew [倾斜] in distribution of records (multiple records have same search-key value, chosen hash function produces non-uniform distribution of key values).
The overflow can be reduced but can't be eliminated. usually, it's handled by using overflow buckets (Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list).



For index-structure creation, hash indices are always secondary indices😕
The disadvantage of static hashing:
(1) if initial number of buckets is too small, with the file grows, there will happens a lot of overflow events.
(2) if initial number of buckets is too big, in normal case, it will wasted so many space.
(3) if we periodic re-organization of the file with a new hash function, it's too expensive and unstable.
Sooo……here's dynamic hashing coming!😋

Dynamic Hashing [动态哈希]:

Allows the hash function to be modified dynamically.
[允许动态修改哈希函数]

Extendable hashing – one form of dynamic hashing:

At any time use only a prefix of the hash function to index into a table of bucket addresses.
[在任何时候，只使用哈希函数的前缀索引到桶地址表中]
Let the length of the prefix be i bits, bucket address table size = 2^i (initially i = 0), value of i grows and shrinks as the size of the database grows and shrinks.
[设前缀长度为 i 位，则桶地址表大小等于 2 的 i 次方(初始 i 等于 0)，i 的值随着数据库大小的增长收缩而增长收缩]
Multiple entries in the bucket address table may point to the same bucket (n:1), thus, actual number of buckets is < 2^i (the number of buckets also changes dynamically due to coalescing and splitting of buckets).
[桶地址表中的多条引表项可能指向同一个桶(n:1)，因此实际桶数小于 2^i(桶数也会因桶的合并和分裂而动态变化)]

Example of Binary Representation:

1 =	1 =	1
10 =	2+0 =	2
11 =	2+1 =	3
100 =	4+0+0 =	4
101 =	4+0+1 =	5
110 =	4+2+0 =	6
111 =	4+2+1 =	7
1000 =	8+0+0+0 =	8
1001 =	8+0+0+1 =	9
1010 =	8+0+2+0 =	10
1011 =	8+0+2+1 =	11
1100 =	8+4+0+0 =	12
1101 =	8+4+0+1 =	13
1110 =	8+4+2+0 =	14
1111 =	8+4+2+1 =	15
10000 =	16+0+0+0+0 =	16
10001 =	16+0+0+0+1 =	17
10010 =	16+0+0+2+0 =	18
10011 =	16+0+0+2+1 =	19
10100 =	16+0+4+0+0 =	20
10101 =	16+0+4+0+1 =	21
10110 =	16+0+4+2+0 =	22
10111 =	16+0+4+2+1 =	23
11000 =	16+8+0+0+0 =	24
11001 =	16+8+0+0+1 =	25
11010 =	16+8+0+2+0 =	26
11011 =	16+8+0+2+1 =	27
11100 =	16+8+4+0+0 =	28
11101 =	16+8+4+0+1 =	29
11110 =	16+8+4+2+0 =	30
11111 =	16+8+4+2+1 =	31

R: i=3 G: i=4 B: i=5

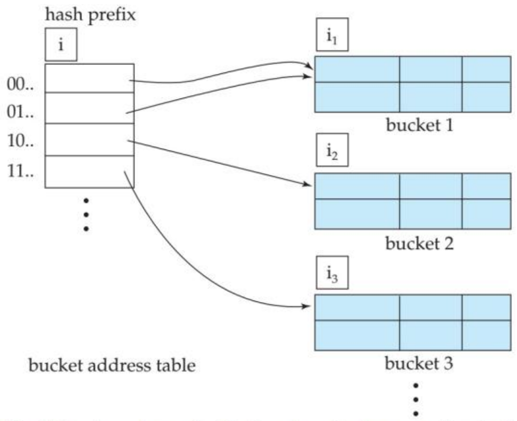
General Extendable Hash Structure:

Let the length of the prefix be i bits (write it on the top of the bucket-address-table)

Each bucket j stores a value i j (write it on the top of the bucket)

All the entries in the bucket-address-table that point to the same bucket have the same hash values on the first ij bits. The number of bucket-address-table entries that point to bucket j is: 2^(i-ij)

i.e.



In this structure $i = 2$, $i_2 = i_3 = i$, whereas $i_1 = i - 1$

▲Queries:

To locate the bucket containing search-key Kj:

(1) Compute h(Kj) = X

(2) Use the first i high order bits of X as a displacement into bucket address table, and follow the pointer to appropriate bucket

▲Insertion:

To insert a record with search-key value Kj:

follow same procedure as look-up and locate the bucket, say j (定位)

if have empty room in bucket j:

    [1] insert record in the bucket

else:

    if i > ij (more than one pointer to bucket j):

        [1] allocate a new bucket z and set ij = iz = (ij + 1)

        [2] Update the second half of the bucket address table entries originally pointing to j change to point to z

        [3] remove each record in bucket j and re-insert (possibly in j or z)

        [4] re-compute new bucket for kj and insert record in the bucket (further splitting is required if the bucket is still full)

    elif i reaches some limit or too many splits have happened in this insertion:

        [1] create an overflow bucket

    else:

        [1] increment i and double the size of the bucket address table

        [2] replace each entry in the table by two entries that point to the same bucket

        [3] re-compute new bucket address table entry for kj

        [4] now i > ij so use the first case above

▲Deletion:

To delete a key value:

locate it in its bucket and remove it (定位)

if after deletion the bucket becomes empty:

[1] remove it (with appropriate updates to the bucket address table)

else:

[1] coalescing[合并] buckets (can coalesce only with a “buddy” bucket having same value of ij or same ij –1 prefix)

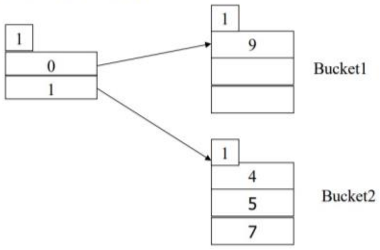
[2] when it is necessary, decreasing bucket address table size is also possible

(it’s very expensive and only if buckets become much smaller than the size of the table)

Example of insertion: <u>insertion of 13</u>

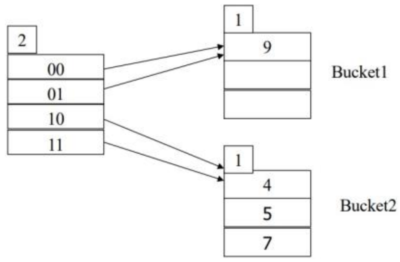
(Step 1)

- Consider the extendable hashing with hash function h(x) = x mod 8 and a bucket can hold three records. Draw the hash index after inserting 13. Initial hash index shown below.



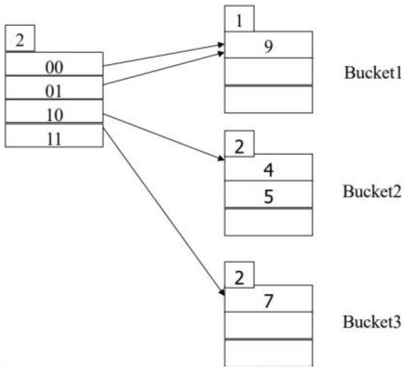
(Step 2)

- The intermediate hash index is shown below after doubling the size of the bucket address table.
- Search key 13 should be inserted to Bucket 2.



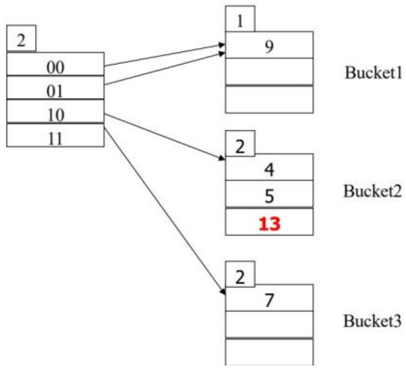
(Step 3)

- Increment i_j for Bucket 2 and 3.
- Reinsert search keys 4, 5 and 7.



(Step 4)

- Re-insert search key 13.



<u>Extendable Hashing</u> vs. Other Schemes

Advantage☺:

Hash performance does not degrade with growth of file (minimal space overhead)

[不会随着文件增多性能下降]

Disadvantage☹:

Extra level of indirection to find desired record

[需要多加一级迂回查找到数据]

Bucket address table may itself become very big (larger than memory), can’t allocate very large contiguous areas on disk either, solution: B+-tree structure to locate desired record in bucket address table

[桶地址表可能会变很大，由于不能在磁盘上分配很大一块连续区域，解决方案是使用 B+树结构]

Changing size of bucket address table is an expensive operation

[更改桶地址表的大小是一项开销很大的操作]

<u>Comparison</u> of Ordered Indexing and Hashing:

The choice depends on:

[1]Cost of periodic re-organisation

[2]Relative frequency of insertions and deletions

[3]Is it desirable to optimise average access time at the expense of worst-case access time?

[4]Expected type of queries

i.e. Hash-indices are extensively used in-memory but not used much on disk.

```
if "select A1, A2, ... An from r where Ai = c":
    choose hashing indices
if "select A1, A2, ... An from r where Ai >=c2 and Ai <=c1":
    choose ordered indices
```

Lecture 2b is about advance indices which is not required to test……