Hashing: Lecture III

References: Chapters 11.6 - 11.7

Hashing with Buckets

This is a variation of hashed files in which more than one record/key is stored per hash address.

bucket = block of records corresponding to one address in the hash table.

The hash function gives the **Bucket Address**.

Example: for a bucket holding 3 records, if we insert the following keys:

key	**Home Address**
KING	33
LAND	33
MARX	33
NUTT	33

Keys 'KING', 'LAND' and 'MARX' will be placed in their home address and key 'NUTT' will be an overflow record.

Effects of Buckets on Performance

We should slightly change some formulas:

$$\texttt{packing density} = \frac{r}{b \cdot N}$$

We will compare the following two alternatives:

1. Storing 750 data records into a hashed file with 1,000 addresses, each holding 1 record.
2. Storing 750 data records into a hashed file with 500 bucket addresses, each bucket holding 2 records.

- In both cases the packing density is 0.75 or 75%.
- In the first case r/N=0.75.
  In the second case r/N=1.50.

Estimating the probabilities as defined in last lecture:

	p(0)	**p(1)**	**p(2)**	**p(3)**	**p(4)**
1) r/N=0.75 (b=1)	0.472	0.354	0.133	0.033	0.006
2) r/N=1.50 (b=2)	0.223	0.335	0.251	0.126	0.047

Calculating the number of overflow records in each case:

1. b=1 (r/N=0.75):
   Number of overflow records =
   $= N \cdot [1 \cdot p(2) + 2 \cdot p(3) + 3 \cdot p(4) + \ldots]$
   $= r - N\,[p(1) + p(2) + p(3) + \ldots]$ (formula derived last class)
   $= r - N \cdot (1 - p(0))$
   $= 750 - 1000 \cdot (1 - 0.472) = 750 - 528 = 222.$
   This is about 29.6% overflow.

2. b=2 (r/N=1.5):
   Number of overflow records =
   $= N \cdot [1 \cdot p(3) + 2 \cdot p(4) + 3 \cdot p(5) + \ldots]$
   $= r - N \cdot p(1) - 2 \cdot N \cdot [p(2) + p(3) + \ldots]$ (similar formula for b=2)
   $= r - N \cdot [p(1) + 2[1 - p(0) - p(1)]$
   $= r - N \cdot [2 - 2 \cdot p(0) - p(1)]$
   $= 750 - 500 \cdot [2 - 2 \cdot (0.223) - 0.335] = 140.5 \cong 140.$
   This is about 18.7% overflow.

Refer to table 11.4 page 495 of the book to see the percentage of collisions for different packing densities and different bucket sizes.

For the previous example, the data is:

	Bucket Size				
Packing Density %	**1**	**2**	**5**	**10**	**100**
75%	29.6%	18.7%	8.6%	4.0%	0.0%

Implementation Issues:

1. Bucket structure
   A Bucket should contain a counter that keeps track of the number of records stored in it. Empty slots in a bucket may be marked '//.../'.

   Ex: Bucket of size 3 holding 2 records:

2	JONES	/////////...//	ARNSWORTH

2. Initializing a file for hashing:
   - Decide on the **Logical Size** (number of available addresses) and on the number of buckets per address.
   - Create a file of empty buckets before storing records. An empty bucket will look like:

0	/////////...//	/////////...//	/////////...//

3. Loading a hash file:
   When inserting a key, remember to:
   - Wrap around when searching for available bucket.
   - Be careful with infinite loops when hash file is full.

Making Deletions

Deletions in a hashed file have to be made with care.

Example:

Record	ADAMS	JONES	MORRIS	SMITH
Home Address	5	6	6	5

Hashed File using Progressive Overflow:

⋮	⋮
4	//////////
5	ADAMS
6	JONES
7	MORRIS
8	SMITH
⋮	⋮

- Delete 'MORRIS'

If 'MORRIS' is simply erased, a search for 'SMITH' would be unsuccessful:

⋮	⋮	
4	//////////	← empty slot
5	ADAMS	
6	JONES	
7	//////////	← empty slot (WRONG: can't find 'SMITH' !!!)
8	SMITH	
⋮	⋮	

Search for 'SMITH' would go to home address (position 5) and when reached 7 it would conclude 'SMITH' is not in the file; which is wrong!!

Idea: use **TOMBSTONES**, i.e. replace deleted records with a marker indicating that a record once lived there:

⋮	⋮	
4	//////////	
5	ADAMS	
6	JONES	
7	######	← tombstone (CORRECT: will find 'SMITH')
8	SMITH	
⋮	⋮	

A search must continue when it finds a tombstone, but can stop whenever an empty slot is found. A search for 'SMITH' will continue when if finds the tombstone in position 7 of the above table.

Note: Only insert a **tombstone** when the next record is occupied or is a tombstone. If the next record is an empty slot, we may mark the deleted record as empty. Why ?

Insertions should be modified to work with **tombstones**: if either an empty slot or a tombstone is reached, place the new record there.

Effects of Deletions and Additions on Performance

The presence of too many tombstones increases search length.

Solutions to the problem of deteriorating average search lengths:

1. Deletion algorithm may try to move records that follow a tombstone backwards towards its home address.
2. Complete reorganization: re-hashing.
3. Use a different type of collision resolution technique.