Hashing

- Data organization in main memory or disk
  - sequential, indexed sequential, binary trees, …
  - the location of a record depends on other keys
  - unnecessary key comparisons to find a key

- The goal of hashing is to retrieve a record with a single key comparison
  - the location of a record is computed using its key only
  - good for random accesses (usually 1-3 comparisons)
  - slow for range queries
Hash Table

- **Hash Function**: transforms keys to array indices

\[ h(key) : \text{Hash Function} \]

\[ \begin{array}{c|c}
\text{index} & \text{data} \\
\hline
\end{array} \]

\[ m \]

- \( n \)
<table>
<thead>
<tr>
<th>position</th>
<th>key</th>
<th>record</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4967000</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8421002</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>395</td>
<td></td>
<td></td>
</tr>
<tr>
<td>396</td>
<td>4618396</td>
<td></td>
</tr>
<tr>
<td>397</td>
<td>4957397</td>
<td></td>
</tr>
<tr>
<td>398</td>
<td></td>
<td></td>
</tr>
<tr>
<td>399</td>
<td>1286399</td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>990</td>
<td>0000990</td>
<td></td>
</tr>
<tr>
<td>991</td>
<td>0000991</td>
<td></td>
</tr>
<tr>
<td>992</td>
<td>1200992</td>
<td></td>
</tr>
<tr>
<td>993</td>
<td>0047993</td>
<td></td>
</tr>
<tr>
<td>994</td>
<td></td>
<td></td>
</tr>
<tr>
<td>995</td>
<td>9846995</td>
<td></td>
</tr>
<tr>
<td>996</td>
<td>4618996</td>
<td></td>
</tr>
<tr>
<td>997</td>
<td>4967997</td>
<td></td>
</tr>
<tr>
<td>998</td>
<td></td>
<td></td>
</tr>
<tr>
<td>999</td>
<td>0001999</td>
<td></td>
</tr>
</tbody>
</table>
Properties of Good Hash Functions

1. Two records cannot occupy the same location
   - *hash collision:* \( \exists k_i \neq k_j, i \neq j : h(k_i) = h(k_j) \)
2. uniform: distributes keys evenly in hash space
3. perfect: \( \exists k_i \neq k_j, i \neq j : h(k_i) \neq h(k_j) \)
4. order preserving: \( \exists k_i \leq k_j, i \neq j : h(k_i) \leq h(k_j) \)

\[ \nabla \] Difficult to find a hash function with these properties
   - property 1 is the most essential to good hashing
   - most functions are no better than \( h(key) = key \mod m \)
Collision Resolution Techniques

1. **Open Addressing** (rehashing): compute new position to store the key in the same table
   - no extra space, stores at most m records
   i. **linear probing**
   ii. **double hashing**

2. **Separate Chaining**: lists of keys mapped to the same position
   - uses extra space
   - can store more than m records (size of hash table)
Open Addressing

• Compute a new address to store the key
  – if it is occupied, compute a new address (rehashing)
  – if this is occupied too, compute a new address again (and so on) until an empty position is found
  – primary hash function: $h(\text{key})$
  – rehash function: $\text{rh}(i)=\text{rh}(h(\text{key}))$
  – hash sequence: $(h_0, h_1, h_2 \ldots) = (h(\text{key}), \text{rh}(h(\text{key})), \text{rh}(\text{rh}(h(\text{key})))), \ldots$

• Retrieval: to find a key follow the same hash sequence
Problem 1: Locate Empty Positions

- No empty position can be found
  i. the table is full
    • check the number of empty positions
  ii. the hash function fails to find an empty position although the table is not full!!
    • $h(key) = key \mod 1000$
    • $rh(i) = (i + 200) \mod 1000 \iff$ checks only 5 positions on a table of 1000 positions
    • use $rh(key)$ that checks the entire table
    • $rh(i) = (i+c) \mod 1000$ where GCD(c,m) = 1
Problem 2: Primary Clustering

- Keys that hash into different addresses compete with each other in successive rehashes
  - \( h(key) = key \mod 100 \)
  - \( rh(i) = (i+1) \mod 100 \)
  - keys: 990, 991, 992, 993, 994 \(\rightarrow\) 94
Problem 3: Secondary Clustering

- Different keys which hash to the same hash value have the same rehash sequence
  - \( h(key) = key \mod 10 \)
  - \( rh(i,j) = (i + j) \mod 10 \)

i. key 23:
   \( h(23) = 3 \)
   \( rh = 4, 6, 9, 3, \ldots \)

ii. key 13:
   \( h(13) = 3 \)
   \( rh = 4, 6, 9, 3, \ldots \)
(i) Linear Probing

- Store the key into the next free position
  - $h_0 = h(key)$ usually $h_0 = key \mod m$
  - $h_i = (h_{i-1} + 1) \mod m$, $i \geq 1$

$S = \{22, 35, 301, 99, 102, 452\}$
Observation 1

- Different insertion sequences $\rightarrow$ different hash sequences
  - $S_1 = \{11,3,27,99,8,50,7,22,12,31,33,40,53\} \rightarrow 28$ probes
  - $S_2 = \{53,40,33,31,12,22,77,50,8,99,27,3,11\} \rightarrow 30$ probes

\[ H(key) = \text{key mod 13} \]
Observation 2

• **Deletions are not easy:**
  – \( h(\text{key}) = \text{key mod 10} \)
  – \( rh(i) = (i+1) \text{ mod 10} \)

• **Action:** delete(65) and search(5)

• **Problem:** search will stop at the empty position and will not find 5

• **Solution:**
  – mark position as deleted rather than empty
  – the marked position can be reused
Observation 3

- **Linear probing** tends to create long sequences of occupied positions
  - the longer a sequence is, the longer it tends to become
  - $P$: probability to use a position in the cluster

\[ P = \frac{B + 1}{m} \]
Observation 4

• Linear probing suffers from both primary and secondary clustering

• Solution: *double hashing*
  – uses two hash functions \( h_1, h_2 \) and a
  – rehashing function \( rh \)
Double Hashing

• Two hash functions and a rehashing function
  – primary hash function \( h_1(key) = key \ mod \ m \)
  – secondary hash function \( h_2(key) \)
  – rehashing function: \( rh(key) = (i + h_2(key)) \ mod \ m \)

• \( h_2(m, key) \) is some function of \( m, key \)
  – helps \( rh \) in computing random positions in the hash table
  – \( h_2 \) is computed once for each key!
Example of Double Hashing

i. hash function:
   - \( h_1(key) = key \mod m \)
   
\[
h_2(key) = \begin{cases} 
m \div 2 & q = 0 \\ 
q & q \neq 0
\end{cases}
\]

   - \( q = (key \div m) \mod m \)

ii. rehash function:
   - \( rh(i, key) = (i + h_2(key)) \mod m \)
Example (continued)

A.  \( m = 10, \ key = 23 \)

\[ h_1(23) = 3, \ h_2(23) = 2 \]

\( rh: \ 5, \ 7, \ 9, \ 1, \ldots \)

B.  \( m = 10, \ key = 13 \)

\[ h_1(key) = 3, \ h_2(23) = 1 \]

\( rh: \ 2, \ 3, \ 4, \ 5, \ldots \)
Performance of Open Addressing

• Distinguish between
  – successful and
  – unsuccessful search

• Assume a series of probes to random positions
  – independent events
  – load factor: $\lambda = \frac{n}{m}$
  – $\lambda$: probability to probe an occupied position
  – each position has the same probability $P=\frac{1}{m}$
Unsuccessful Search

- The hash sequence is exhausted
  - let \( u \) be the expected number of probes
  - \( u \) equals the expected length of the hash sequence
  - \( P(k) \): probability to search \( k \) positions in the hash sequence
\[ u = \sum_{k \geq 1} kP(k) = \]

\[ P(1) + \]

\[ P(2) + P(2) + \]

\[ P(3) + P(3) + P(3) + \]

\[ \cdots \]

\[ P(k) + P(k) + \cdots + P(k) + \]

\[ \cdots \]

\[ \underbrace{P(\geq 1\ \text{probes}) + P(\geq 2\ \text{probes}) + \cdots} \]
\[ u = \sum_{k \geq 1} P(\geq k \text{ probes}) = \]

\[ \sum_{k \geq 1} P(\text{first } k - 1 \text{ positions occupied}) = \]

\[ \sum_{k \geq 1} \lambda^{k-1} \leq \sum_{k \geq 1} \lambda^k \Rightarrow \text{independent events} \]

\[ u = \frac{1}{1 - \lambda} \]

\( u \) increases with \( \lambda \) \( \Rightarrow \) performance drops as \( \lambda \) increases
Successful Search

- The hash sequence is not exhausted
  - the number of probes to find a key equals the number of probes $s$ at the time the key was inserted plus 1
  - $\lambda$ was less at that time
  - consider all values of $\lambda$

$$s = \int_{0}^{\lambda} \frac{1}{\lambda} (u + 1) dx = 1 + \frac{1}{\lambda} \ln\left(\frac{1}{1 - \lambda}\right)$$

$u$: equivalent to successful search

increases with $\lambda$
Performance

• The performance drops as $\lambda$ increases
  – the higher the value of $\lambda$ is, the higher the probability of collisions

• Unsuccessful search is more expensive than successful search
  – unsuccessful search exhausts the hash sequence
## Experimental Results

<table>
<thead>
<tr>
<th>LOAD FACTOR</th>
<th>SUCCESSFUL</th>
<th></th>
<th>UNSUCCESSFUL</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LINEAR</td>
<td>i + bkey</td>
<td>DOUBLE</td>
<td>LINEAR</td>
<td>i + bkey</td>
</tr>
<tr>
<td>25%</td>
<td>1.17</td>
<td>1.16</td>
<td>1.15</td>
<td>1.39</td>
<td>1.37</td>
</tr>
<tr>
<td>50%</td>
<td>1.50</td>
<td>1.44</td>
<td>1.39</td>
<td>2.50</td>
<td>2.19</td>
</tr>
<tr>
<td>75%</td>
<td>2.50</td>
<td>2.01</td>
<td>1.85</td>
<td>8.50</td>
<td>4.64</td>
</tr>
<tr>
<td>90%</td>
<td>5.50</td>
<td>2.85</td>
<td>2.56</td>
<td>50.50</td>
<td>11.40</td>
</tr>
<tr>
<td>95%</td>
<td>10.50</td>
<td>3.52</td>
<td>3.15</td>
<td>200.50</td>
<td>22.04</td>
</tr>
</tbody>
</table>
Performance on Full Table

<table>
<thead>
<tr>
<th>TABLE SIZE (m)</th>
<th>SUCCESSFUL</th>
<th>UNSUCCESSFUL</th>
<th>LOG$_2$m</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LINEAR</td>
<td>$i + b_{key}$</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>100</td>
<td>6.60</td>
<td>4.62</td>
<td>4.12</td>
</tr>
<tr>
<td>500</td>
<td>14.35</td>
<td>6.22</td>
<td>5.72</td>
</tr>
<tr>
<td>1000</td>
<td>20.15</td>
<td>6.91</td>
<td>6.41</td>
</tr>
<tr>
<td>5000</td>
<td>44.64</td>
<td>8.52</td>
<td>8.02</td>
</tr>
<tr>
<td>10000</td>
<td>63.00</td>
<td>9.21</td>
<td>8.71</td>
</tr>
</tbody>
</table>
Separate Chaining

• Keys hashing to the same hash value are stored in separate lists
  – one list per hash position
  – can store more than $m$ records
  – easy to implement
  – the keys in each list can be ordered
$h(key) = key \mod m$
Performance of Separate Chaining

- Depends on the average chain size
  - insertions are independent events
  - let $P(c,n,m)$: probability that a position has been selected $c$ times after $n$ insertions
  - $P(c,n,m)$ is also the probability that the chain has length $c$
  - $P(c,n,m)$ follows the binomial distribution

$$P(c,n,m) = \binom{n}{c} p^c q^{n-c}$$

- $p=1/m$: success case
- $q=1-p$: failure case
\[ P(c, n, m) = \binom{n}{c} \left( \frac{1}{m} \right)^c \left( 1 - \frac{1}{m} \right)^{n-c} = \]

\[
\frac{1}{c!} \frac{n}{m} \frac{n-c+1}{m} \left( 1 - \frac{1}{m} \right)^{-c} \left( 1 - \frac{1}{m} \right)^n
\]

\[
\frac{n-c+1}{m} \rightarrow \lambda
\]

\[
n, m \rightarrow \infty \quad \Rightarrow \quad \left( 1 - \frac{1}{m} \right)^{-c} \rightarrow 1\quad \Rightarrow \quad P(c,n,m) = \left( \frac{1}{c!} \lambda^c e^{-\lambda} \right)
\]

Poison
Unsuccessful Search

• The entire chain is searched
  – the average number of comparisons equals its average length $u$

$$u = \sum_{c \geq 0} c P(c, l) = \sum_{c \geq 0} c \frac{\lambda}{c!} e^{-\lambda} = \lambda$$
Successful Search

• Not the whole chain is searched
  – the average number of comparisons equals the length $s$ of the chain at time the key was inserted plus 1
  – the performance at the time a key was inserted equals that of unsuccessful search!

\[
s = \int_{0}^{\lambda} \frac{1}{\lambda} (u + 1) \, dx = \frac{1}{\lambda} \int_{0}^{\lambda} (x + 1) \, dx = 1 + \frac{\lambda}{2}
\]
Performance

• The performance drops with the length of the chains
  – worst case: all keys are stored in a single chain
  – worst case performance: $O(N)$
  – unsuccessful search performs better than successful search!! WHY ?
  – no problem with deletions!!
Coalesced Hashing

- The hash sequence is implemented as a linked list within the hash table
  - no rehash function
  - the next hash position is the next available position in linked list
  - extra space for the list

\[
h(key) = \text{key mod 10}
\]

<table>
<thead>
<tr>
<th>keys: 19, 29, 49, 59</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
initialization

<table>
<thead>
<tr>
<th>0</th>
<th>nilkey</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>nilkey</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>5</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>7</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>9</td>
<td>nilkey</td>
<td>-1</td>
</tr>
</tbody>
</table>

initially: $avail = 9$

$h(\text{key}) = \text{key mod 10}$

keys: 14, 29, 34, 28, 42, 39, 84, 38

<table>
<thead>
<tr>
<th>0</th>
<th>nilkey</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>nilkey</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>38</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>84</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>39</td>
<td>-1</td>
</tr>
<tr>
<td>7</td>
<td>28</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>34</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>29</td>
<td>6</td>
</tr>
</tbody>
</table>

linked list
Performance of Coalesced Hashing

- **Unsuccessful search**
  \[ \frac{1}{4} e^{2\lambda} - \frac{\lambda}{2} + 0.75 \text{ probes/search} \]

- **Successful search**
  \[ \frac{e^{2\lambda} - 1}{8\lambda} + \frac{\lambda}{4} + 0.75 \text{ probes/search} \]
Hashing on the Disk

- Use **“disk pages”** or **“buckets”** to store records
  - several records fit within one bucket
  - first retrieve the appropriate page into the main memory
  - then searching for the appropriate record within the bucket comes for free
• page size $b$: maximum number of records in page
• space utilization $u$: measure of the use of space

$$u = \frac{\text{# stored records}}{\text{# pages} \cdot b}$$
Collisions

• Keys that hash to the same hash position are stored within the same page

• If the page is full, collisions cause
  
    i.  *page splits*: split pages content between the old and a new page

    ii. *overflows*: list of overflow pages

• The larger the page size the less the overflows
Access Time

• Goal: find key in one disk access
  – access time ~ number of page accesses
  – many overflows cause more disk accesses
  – large $u$ means good space utilization but many overflows
Problems

• File expansion (dynamic growth)
  – file size adapts to space requirements
• Searching with overflows \(\Rightarrow\) bad performance
• Non-uniform distribution of keys:
  – many keys map to the same addresses
• Categories of Methods
  – pseudo-dynamic: open addressing, chaining, …
  – dynamic: dynamic hashing, extendible hashing, linear hashing, spiral storage…
Dynamic Hashing Schemes

• Dynamic growth without total reorganization
  – typically 1-3 disk accesses to access a key
  – access time and space utilization are a typical trade-off
  – space utilization between 50-100%
    • typically 69%
  – not always easy implementation
Dynamic Schemes With Index

- At least two disk accesses
  - one to access the index and
  - one to access the data (overflows cause more disk accesses)
  - if we keep index in main memory → one disk access less
  - problem: the index may become too large

\[
\begin{array}{c}
\text{index} \\
\text{data pages}
\end{array}
\]

- Methods
  - Dynamic hashing (Larson 1978)
  - Extendible hashing (Fagin et.al. 1979)
Dynamic Schemes Without Index

- Ideally, less space and less disk accesses
  - at least one disk access
  - overflows are allowed \(\rightarrow\) more disk accesses

- Methods
  - Linear Hashing (Litwin 1980)
  - Linear Hashing with Partial Expansions (Larson 1980)
  - Spiral Storage (Martin 1979)
Hash Functions

• Support shrinking or growing hash file
  – shrinking or growing address space
  – the hash function adapts to these changes
  – hash functions using first (last) bits of key
    – key = $b_{n-1}b_{n-2}...b_i b_{i-1}...b_2b_1b_0$
    – $h_i(key) = b_{i-1}...b_2b_1b_0$ supports $2^i$ addresses
    – $h_i$: one more bit than $h_{i-1}$ to address larger files

$$h_i(key) = \begin{cases} h_{i-1}(key) \\ h_{i-1}(key) + 2^i \end{cases}$$
Dynamic Hashing (Larson 1978)

- Two level index
  - *primary index* $h_1(key)$: accesses a hash table
  - *secondary index* $h_2(key)$: accesses a binary tree
• **Primary index is fixed**
  
  – $h_1(key) = \text{key mod } m$

• **Dynamic behavior on secondary index**
  
  – $h_2(key)$ uses $i$ bits of key
  
  – the bit sequence of $h_2$ denotes which path on the binary tree of the secondary index to follow in order to access the data page
  
  – scan $h_2$ from left to right
    
    • bit 1: follow right path
    
    • bit 0: follow left path
\( h_1(\text{key}) = \text{key mod 6} \)
\( h_2(\text{key}) = 11 \ldots \leftarrow \text{depth of binary tree} = 2 \)
Insertions

• Initially fixed size primary index and no data

  - insert record: store in new page on $h_1$ location
  - if page is full, allocate one extra page and split contents of old page between old and new page
  - use one extra bit in $h_2$ for addressing
E.G.M. Petrakis

Hashing

1. h₁=0, h₂=any
2. h₁=3, h₂=any

index
storage

b

h₁=0, h₂=0
h₁=0, h₂=1
h₁=3, h₂=any
h₁=3, h₂=any
h₁=0, h₂=0
h₁=0, h₂=10
h₁=0, h₂=11
h₁=3, h₂=0
h₁=3, h₂=1

E.G.M. Petrakis

Hashing

49
Deletions

• Find record to be deleted using $h_1$ and $h_2$
• Delete record and
• Check “siblink” page:
  – less than $b$ records in both pages?
  – if yes merge the two pages and delete one empty page
  – shrink secondary index by one level and reduce $h_2$ by one bit
Extendible Hashing (Fagin et.al. 1979)

• Dynamic hashing without index
  – primary hashing is omitted
  – hash function similar to secondary hash function and all binary trees at same level
  – the index shrinks and grows according to file size
  – data pages attached to the index
dynamic hashing

dynamic hashing with all binary trees at same level

address bits

extendible hashing
Insertions

- Initially 1 index and 1 data page
  - 0 address bits
  - insert records in this data page
• Page 0 overflows:
  – 1 more key bit for addressing, 1 extra page
  – index doubles!!
  – split contents of previous page into 2 pages according to next bit of key
  – global depth $d$: # index bits $\Rightarrow 2^d$ index size
  – local depth $l$: max # bits for record addressing

\[
\begin{array}{c}
\text{d} \\
0 \\
1 \\
\end{array} 
\xrightarrow{1} 
\begin{array}{c}
\text{l} \\
1 \\
\end{array}
\xrightarrow{1}
\begin{array}{c}
\text{l} \\
1 \\
\end{array}
\]

$d$: global depth = 1

$l$: local depth = 1
• **Page 0 overflows:**

   - $l \leq d$
   - Contains records with same 1st bit of key
   - Contains records with same 2 bits of key

• **Page 1 overflows:**

   - 1 more key bit for addressing
   - $2^{d-1}$: number of pointers to page
• Page 100 overflows:
  – no need to double index
  – page 100 splits into two (1 new page)
  – local depth $l$ is increased by 1

\[
2^{d-l} + 1
\]
Insertion Algorithm

• If \( l < d \) split overflowed page (1 extra page)
• If \( l = d \) double index, split page and
  
  – \( d \) is increased by 1 \( \rightarrow \) 1 more bit for addressing
  
  – update pointers with either way:
    a) if \( d \) prefix bits are used for addressing
       \[ d = d + 1; \]
       for (i=2\(^d\)-1, i>=0,i--) index[i]=index[i/2];
    a) if \( d \) suffix bits are used
       for (i=0; i <= 2\(^d\)-1; i++) index[i]=index[i]+2\(^d\);
Deletion Algorithm

• Find and delete record
• Check siblink page
• If less than $b$ records in both pages
  – merge pages and free empty page
  – decrease local depth $l$ by 1 (records in merged page have 1 less common bit)
  – if $l < d$ everywhere $\Rightarrow$ reduce index (half size)
  – update pointers
delete with merging

l < d

E.G.M. Petrakis
Hashing
Observations

- A page splits and there are more than $b$ keys with same next bit
  - take one more bit for addressing (increase $l$)
  - if $d=l$ the index is doubled again!!
- Hashing might fail for non-uniform distributions of keys (e.g., multiple keys with same value)
  - if the distribution of keys is known, transform it to uniform
- Dynamic hashing performs better for non-uniform distributions (affected locally)
• Storage utilization:

\[ u = \frac{b}{2b} = 50\% \]

After splitting

– in general \( 50\% < u < 100\% \)

– on the average \( u \sim \ln 2 \sim 69\% \) (no overflows)
• Overflows – storage utilization:
  - allow overflows in order to achieve higher $u$ and to avoid page doubling (if $d=l$)

  $u=\frac{2b}{3b}\approx 66\%$ after splitting
  - for smaller overflow pages (e.g., $b/2$) $u = \frac{(b+b/2)}{2b} \approx 75\%$
  - double index only if the overflow overflows!!
Performance of Extendible Hashing

• For $n$: records and page size $b$
  – expected size of index (Flajolet)
  \[
  \frac{l}{b \log 2} n \left(1 + \frac{1}{b}\right) \approx \frac{3.92}{b} n \left(1 + \frac{1}{b}\right)
  \]
  – 1 disk access/retrieval when index in main mem
  – 2 disk accesses when index is stored on disk
  – overflows increase number of disk accesses
Linear Hashing (Litwin 1980)

- Dynamic hashing scheme without index
- Indices refer to directly page addresses
- Overflows are allowed
- The file grows one page at a time
- The page which splits is not always the one which overflowed
- The pages split in a predetermined order
• Initially **n** empty pages
  – **p** points to the page that splits

![Diagram 1](image1)

• **Overflows** are allowed

![Diagram 2](image2)
• A page splits whenever the “splitting criterion” is satisfied
  – a new page is added at the end of the file
  – pointer \( p \) points to the next page
  – split contents of old page between old and new page based on key values
- b = b_{page} = 4, b_{overflow} = 1
- initially n = 5 pages
- hash function h_0 = k \mod n
- splitting criterion u > A\%
- alternatively split when overflow overflows, etc.
- Page 5 is added at end of file
- The contents of 0 are split between 0 and 5 based on hash function $h_1$
- $p$ points to next page
Hash Functions for Linear Hashing

• Initially as simple as \( h_0 = key \mod n \)
• As pages are added, \( h_0 \) alone becomes insufficient
• The file will eventually double its size
• In that case use \( h_1 = key \mod 2n \)
• In the meantime
  – use \( h_0 \) for pages not yet split
  – use \( h_1 \) for pages that have already split
• Split contents of page pointed to by \( p \) based on \( h_1 \)
Hash functions (continued)

• When the file has doubled its size, \( h_0 \) is no longer needed
  – set \( h_0 \leftarrow h_1 \) and continue (e.g., \( h_0 = k \mod 10 \))
• The file will eventually double its size again
• Deletions cause merging of pages whenever a merging criterion is satisfied
  – merging criterion e.g., \( u < B\% \)
Hash functions for linear hashing

• Initially $n$ pages and $0 \leq h_0(k) \leq n$
• Series of hash functions

\[
h_{i+1}(k) = \begin{cases} 
  h_i(k) \\
  h_i(k) + n2^i 
\end{cases}
\]

• Selection of hash function:
  
  if $h_i(k) \geq p$ then use $h_i(k)$
  
  else use $h_{i+1}(k)$
Linear Hashing with Partial Expansions (Larson 1980)

- **Problem with Linear Hashing**: pages to right of p delay to split
  - large chains of overflows on the rightmost pages
- **Solution**: do not wait that much to split a page
  - \( k \) partial expansions: take pages in groups of \( k \)
  - all \( k \) pages of a group split together
  - the file grows at lower rates
• Two partial expansions on a file with $2n$ pages
  – initially: $n$ groups with $k=2$ pages each
  – groups: $(0, n) (1, 1+n) \ldots (i, i+n) \ldots (n-1, 2n-1)$

  – all pages of the same group split together: after the first split, pages 0 and 2n split together, some records go to page 2n (new page)
• **1\textsuperscript{st} expansion**: after \( n \) splits, all pages are split
  – at the end of 1\textsuperscript{st} expansion the file has 3\( n \) pages
  – the file grows at lower rate
  – the file has size 1.5 times larger

  \[ \begin{array}{cccc}
    0 & n & 2n & 3n \\
  \end{array} \]

  – after 1\textsuperscript{st} expansion take pages in groups of 3
  – groups: \((j, j+n, j+2n), 0 \leq j \leq n\)

  \[ \begin{array}{cccc}
    0 & n & 2n & 3n \\
  \end{array} \]
• 2\textsuperscript{nd} expansion: after \( n \) splits the file has size 4n
  – repeat the same process having initially 4n pages
  – 2n groups

2 pointers to pages of the same group
Relative file size vs disk access/retrieval for Linear Hashing with 2 partial expansions:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Linear Hashing</th>
<th>Linear Hashing 2 part. Exp.</th>
<th>Linear Hashing 3 part. Exp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>retrieval</td>
<td>1.17</td>
<td>1.12</td>
<td>1.09</td>
</tr>
<tr>
<td>insertion</td>
<td>3.57</td>
<td>3.21</td>
<td>3.31</td>
</tr>
<tr>
<td>deletion</td>
<td>4.04</td>
<td>3.53</td>
<td>3.56</td>
</tr>
</tbody>
</table>

Parameters:
- $b = 5$
- $b' = 5$
- $u = 0.85$
Dynamic Hashing Schemes

• Very good performance on membership, insert, delete operations
• Suitable for both main memory and disk
• **Critical parameter**: space utilization $u$
  – large $u \rightarrow$ more overflows, bad performance
  – small $u \rightarrow$ less overflows, better performance
• Suitable for direct access queries (random accesses) but not for range queries