CS 350 Algorithms and Complexity

Fall 2015

Lecture 12: Space & Time Tradeoffs.
Part 2: Hashing & B-Trees

Andrew P. Black

Department of Computer Science
Portland State University
Space-for-time tradeoffs

- Two varieties of space-for-time algorithms:
  - input enhancement: preprocess the input (all or part) to store some info to be used later in solving the problem
    - counting sorts
    - string searching algorithms
  - prestructuring — preprocess the input to make accessing its elements easier
    - hashing
    - indexing schemes (e.g., B-trees)
Hashing

- A very efficient method for implementing a dictionary, i.e., a mapping from keys to values with the operations:
  - `find(k)` — also called `at(k)`, `lookup(k)`, `[k]`
  - `insert(k, v)` — also called `at(k) put(v)`, `put(k, v)`, `[k] := v`
  - `delete(k)`

- Based on representation-change and space-for-time tradeoff ideas

- Important applications:
  - symbol tables
  - set (values are not relevant)
  - databases (extendible hashing)
Hash tables and hash functions

- The idea of hashing is to map \( n \) keys into an array of size \( m \), called the hash table, by using a predefined function, called the hash function, \( h: \text{key} \rightarrow \text{index in the hash table} \).

- Example: student records, key = StudentID.
- Hash function: \( h(K) = K \mod m \) where \( m \) is some integer (typically, prime).

- Generally, a hash function should:
  - be easy to compute
  - distribute keys about evenly throughout the hash table
Question:

If \( m = 1000 \),
\[
h(k) = k \mod m,
\]
where is record with StudentId = 990159265 stored?

Numeric answer:
Collisions

- If \( h(k_1) = h(k_2) \), there is a collision

- Good hash functions result in fewer collisions than bad ones, but:
  - some collisions should be expected (birthday paradox)

- Two principle hashing schemes handle collisions differently:
  - Open hashing
    - each cell is a header of “chain” — a list of all keys hashed to it
  - Closed hashing
    - one key per cell
    - in case of collision, finds another cell in the hash table itself by
      - linear probing: use next free bucket, or
      - double hashing: use second hash function to compute increment
What Levitin didn’t tell you

знав

Hash functions usually designed in two parts:

1. the hash \( Object \rightarrow [0..2^{32}) \)
2. the address calculation

\[ [0..2^{32}) \rightarrow [0..hashTableSize) \]
A Real Hash Function:

function robertJenkins32bit5shiftHash(n) {
    var answer = n & 0xFFFFFFFF;
    answer = 0x479AB41D + answer + (answer << 8);
    answer = answer & 0xFFFFFFFF;
    answer = (0xE4AA10CE ⊻ answer) ⊻ (answer >> 5);
    answer = 0x09942F0A6 + answer - (answer << 14);
    answer = (0x5AEDD67D ⊻ answer) ⊻ (answer >> 3);
    answer = 0x17BEA992 + answer + (answer << 7);
    return (answer & 0xFFFFFFFF);
}

Returns 32 “scrambled” bits
Open hashing (Separate chaining)

- Keys are stored in linked lists *outside* a hash table whose elements serve as the lists’ headers.
- Example: A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

\[ h(k) = \text{sum of key’s letters’ positions in the alphabet mod 13} \]

<table>
<thead>
<tr>
<th>key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(key)</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AND</td>
<td>MONEY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FOOL</td>
<td>HIS</td>
<td>ARE</td>
<td>PARTED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOON</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Open hashing (Separate chaining)

- Keys are stored in linked lists *outside* a hash table whose elements serve as the lists’ headers.
- Example: A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

\[ h(k) = \text{sum of key’s letters’ positions in the alphabet } \mod 13 \]

<table>
<thead>
<tr>
<th>key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(key)</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Search for KID
Open hashing (Separate chaining)

- Keys are stored in linked lists outside a hash table whose elements serve as the lists’ headers.
- Example: A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

\[ h(k) = \text{sum of key’s letters’ positions in the alphabet mod 13} \]

<table>
<thead>
<tr>
<th>key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(key)</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Search for KID
\[ h(\text{KID}) = 11 \]
Open hashing (cont.)

- The ratio $\alpha = n/m$ is called the load factor.
- If hash function distributes keys uniformly, average length of linked list will be $\alpha$.
- Average number of key comparisons in successful, $S$, and unsuccessful searches, $U$: $S \approx 1 + \alpha/2$, $U = \alpha$.
- $\alpha$ is typically kept small (ideally, about 1).
- Open hashing still works if $n > m$ but it’s much less efficient.
Efficiencies (open hashing)

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$S \approx 1 + \alpha/2$</th>
<th>$U = \alpha$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.33</td>
<td>1.17</td>
<td>0.33</td>
</tr>
<tr>
<td>0.50</td>
<td>1.25</td>
<td>0.50</td>
</tr>
<tr>
<td>0.75</td>
<td>1.38</td>
<td>0.75</td>
</tr>
<tr>
<td>0.90</td>
<td>1.45</td>
<td>0.90</td>
</tr>
<tr>
<td>0.95</td>
<td>1.48</td>
<td>0.95</td>
</tr>
<tr>
<td>1.00</td>
<td>1.50</td>
<td>1.00</td>
</tr>
<tr>
<td>1.50</td>
<td>1.75</td>
<td>1.50</td>
</tr>
<tr>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
</tbody>
</table>
Mnemonic

This method is called *open hashing* because the hash table is *open*:

- the stored keys and values are not *in* the table, but “hang” *out* of it ...
- on separate chains (hence the alternative name)
Question:

- Suppose that you have 1000 records in memory, and you index them in a hash table with $\alpha = 1$. How many words of additional memory will be needed if you use an open hash table with separate chaining?

  Each pointer occupies 1 word. *Additional* means beyond the table and the records.)

- Numeric answer:
Closed hashing (Open addressing)

- All keys are stored *inside* the hash table.

<table>
<thead>
<tr>
<th>Key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(K)</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7 8 9 10 11 12
Closed hashing (Open addressing)

✧ All keys are stored *inside* the hash table.

<table>
<thead>
<tr>
<th>Key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(K)</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7 8 9 10 11 12

A
Closed hashing (Open addressing)

✧ All keys are stored *inside* the hash table.

<table>
<thead>
<tr>
<th>Key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(K)</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7 8 9 10 11 12

A

A

FOOL
Closed hashing (Open addressing)

✧ All keys are stored *inside* the hash table.

<table>
<thead>
<tr>
<th>Key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(K)</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FOOL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td>AND</td>
<td></td>
<td></td>
<td></td>
<td>FOOL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Closed hashing (Open addressing)

✦ All keys are stored inside the hash table.

<table>
<thead>
<tr>
<th>Key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(K)</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FOOL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AND</td>
<td></td>
<td></td>
<td></td>
<td>FOOL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AND</td>
<td></td>
<td></td>
<td></td>
<td>FOOL</td>
<td></td>
<td>HIS</td>
<td></td>
</tr>
</tbody>
</table>
Closed hashing (Open addressing)

- All keys are stored inside the hash table.

<table>
<thead>
<tr>
<th>Key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(K)</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

```
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
---|---|---|---|---|---|---|---|---|---|----|----|----|
A          |   |   |   |   |   |   |   |   |   |    |    |    |
A          |   |   |   |   |   |   |   |   |   |    |    |    |
A          |   |   |   |   |   |   |   |   |   |    |    |    |
A          |   |   |   |   |   |   |   |   |   |    |    |    |
A          |   |   |   |   |   |   |   |   |   |    |    |    |
A          |   |   |   |   |   |   |   |   |   |    |    |    |
```
Closed hashing (Open addressing)

- All keys are stored *inside* the hash table.

<table>
<thead>
<tr>
<th>Key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(K)</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

```
0   1   2   3   4   5   6   7   8   9  10  11  12
A
A
A
A
A
A
A
A
A
A

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FOOL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FOOL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FOOL</td>
<td>HIS</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FOOL</td>
<td>HIS</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FOOL</td>
<td>HIS</td>
<td>ARE</td>
</tr>
</tbody>
</table>
```
Closed hashing (Open addressing)

- All keys are stored *inside* the hash table.

<table>
<thead>
<tr>
<th>Key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(K)</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Closed hashing (Open addressing)

- All keys are stored *inside* the hash table.

<table>
<thead>
<tr>
<th>Key</th>
<th>A</th>
<th>FOOL</th>
<th>AND</th>
<th>HIS</th>
<th>MONEY</th>
<th>ARE</th>
<th>SOON</th>
<th>PARTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(K)</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>FOOL</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>FOOL</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>AND</td>
<td></td>
<td>A</td>
<td>FOOL</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td>AND</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>FOOL</td>
<td></td>
<td>A</td>
<td>AND</td>
<td>FOOL</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>AND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>AND</td>
<td></td>
<td>A</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>AND</td>
<td>MONEY</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>AND</td>
<td></td>
<td>A</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>AND</td>
<td>MONEY</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>AND</td>
<td></td>
<td>A</td>
<td>AND</td>
<td></td>
</tr>
</tbody>
</table>
Closed hashing (cont.)

- Does not work if \( n > m \)
- Avoids pointer chains
- Deletions are not straightforward
- Number of probes to find/insert/delete a key depends on load factor \( \alpha = \frac{n}{m} \) and on collision resolution strategy.

For linear probing:

\[
S \approx \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)}\right) \quad \text{and} \quad U \approx \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2}\right)
\]
Efficiencies (closed hashing)

✧ As the table fills (α approaches 1), number of probes in linear probing increases dramatically:

\[
S \approx \frac{1}{2}(1+1/(1-\alpha))
\]

\[
U \approx \frac{1}{2}(1+1/(1-\alpha)^2)
\]

<table>
<thead>
<tr>
<th>α</th>
<th>Successful</th>
<th>Unsuccessful</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.33</td>
<td>1.25</td>
<td>1.625</td>
</tr>
<tr>
<td>0.50</td>
<td>1.5</td>
<td>2.5</td>
</tr>
<tr>
<td>0.75</td>
<td>2.5</td>
<td>8.5</td>
</tr>
<tr>
<td>0.90</td>
<td>5.5</td>
<td>50.5</td>
</tr>
<tr>
<td>0.95</td>
<td>10.5</td>
<td>200.5</td>
</tr>
<tr>
<td>0.99</td>
<td>50.5</td>
<td>5000.5</td>
</tr>
</tbody>
</table>
Question:

✧ Suppose that you have 1000 records in memory, and you index them in a hash table with $\alpha = 1/2$. How many words of additional memory will be needed if you use a closed hash table?

Each pointer occupies 1 word. Additional means beyond the table and the records.)

✧ Numeric answer:
Question:

For the input:

30  20  56  75  31  19

and hash function $h(k) = k \mod 11$

1. Construct an open hash table of size 11

2. What is the largest number of key comparisons in a successful search in this table?
Question:

For the input:

30 20 56 75 31 19

30 mod 11 = 8

and hash function $h(k) = k \mod 11$

1. Construct an open hash table of size 11

2. What is the largest number of key comparisons in a successful search in this table?
Question:

For the input:

30 20 56 75 31 19

30 mod 11 = 8

and hash function $h(k) = k \mod 11$

1. Construct an open hash table of size 11

2. What is the largest number of key comparisons in a successful search in this table?
Question:

For the input:

30  20  56  75  31  19

and hash function \( h(k) = k \mod 11 \)

1. Construct a closed hash table of size 11

2. What is the largest number of key comparisons in a successful search in this table?
Question:

1. Construct a closed hash table of size 11

   30  20  56  75  31  19

   30 mod 11 = 8

   and hash function $h(k) = k \mod 11$

2. What is the largest number of key comparisons in a successful search in this table?
Question:

Suppose that you have 1000 records in memory, and you have 3000 words of memory to use for indexing them. What’s the best way to use that memory if most searches are expected to be successful? (Assume that each pointer occupies 1 word)

A. Open Hash table with separate chaining
B. Closed Hash table with open addressing
C. Binary search tree
## Problem

- Fill in the following table with the average-case efficiencies for 5 implementations of Dictionary

<table>
<thead>
<tr>
<th></th>
<th>array</th>
<th>sorted array</th>
<th>Binary search tree</th>
<th>Open Hash w/separate chaining</th>
<th>Closed Hash w/linear probing</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insertion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>deletion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2–3 trees

- 2–3 trees are perfectly-balanced search trees where each node has either:
  - 2 children (and 1 key), or
  - 3 children (and 2 keys).

Details:

- Lyn Turbak on 2–3 Trees
- Visualization of 2–3 trees
B+ Trees (slides based on those of Miriam Sy)

- The B+ Tree index structure is the most widely-used of several index structures that maintain their efficiency despite insertion and deletion of data.
- Balanced tree: every path from the root of the tree to a leaf is of the same length.
- All data are in leaves.
  - Hence: B+ tree
B Trees & B+ Trees

**B Trees:**
- Multi-way trees
- Dynamic growth
- Data stored with keys throughout the tree pages

**B+ Trees:**
- Contains features from B Trees
- Dynamic growth
- Separates index and data pages
- Internal nodes contain just keys
- All data are at the leaves
- Leaves can be linked sequentially
B Tree Order

The “order” of a B tree is the:

A. way the purchasing department buys it
B. maximum number of children any node can have
C. minimum number of children any node can have
D. number of keys in each index node
E. height of the tree
Fill Factor
Fill Factor

B+ Trees use a “fill factor” to control growth: fill factor = fraction of keys filled
Fill Factor

- B+ Trees use a “fill factor” to control growth: fill factor = fraction of keys filled.

- What is the minimum fill factor for a B+ Tree?
  A. 33.3%
  B. 50%
  C. 75%
  D. None of the above
B+ Tree Statistics
B+ Tree Statistics

* Each node has between \( \lceil \frac{m}{2} \rceil \) and \( m \) children
B+ Tree Statistics

- Each node* has between \( \lceil m/2 \rceil \) and \( m \) children

* Except the root
B+ Tree Statistics

✦ Each node* has between $\lceil m/2 \rceil$ and $m$ children

✦ 50% fill factor is the minimum for a B+ Tree.

* Except the root
B+ Tree Statistics

✧ Each node* has between \( \lceil m/2 \rceil \) and \( m \) children

✧ 50% fill factor is the minimum for a B+ Tree.

✧ For tree of order 5, these guidelines must be met:

* Except the root
B+ Tree Statistics

✦ Each node* has between \( \lceil m/2 \rceil \) and \( m \) children

✦ 50% fill factor is the minimum for a B+ Tree.

✦ For tree of order 5, these guidelines must be met:

\[
\begin{array}{|c|}
\hline
m, \text{ max pointers/page} & 5 \\
\hline
\end{array}
\]

* Except the root
B+ Tree Statistics

✦ Each node* has between ⌈m/2⌉ and m children

✦ 50% fill factor is the minimum for a B+ Tree.

✦ For tree of order 5, these guidelines must be met:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>m, max pointers/page</td>
<td>5</td>
</tr>
<tr>
<td>n = m−1, max keys/page</td>
<td>4</td>
</tr>
</tbody>
</table>

* Except the root
B+ Tree Statistics

- Each node* has between \(\lceil m/2 \rceil\) and \(m\) children
- 50% fill factor is the minimum for a B+ Tree.
- For tree of order 5, these guidelines must be met:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(m), max pointers/page</td>
<td>5</td>
</tr>
<tr>
<td>(n = m-1), max keys/page</td>
<td>4</td>
</tr>
<tr>
<td>(\lceil m/2 \rceil), min pointers/page</td>
<td>3</td>
</tr>
</tbody>
</table>

* Except the root
**B+ Tree Statistics**

✧ Each node* has between \( \lceil m/2 \rceil \) and \( m \) children.

✧ 50% fill factor is the minimum for a B+ Tree.

✧ For tree of order 5, these guidelines must be met:

<table>
<thead>
<tr>
<th>m, max pointers/page</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = m - 1 ), max keys/page</td>
<td>4</td>
</tr>
<tr>
<td>( \lceil m/2 \rceil ), min pointers/page</td>
<td>3</td>
</tr>
<tr>
<td>min keys/page</td>
<td>2</td>
</tr>
</tbody>
</table>

* Except the root
B+ Tree Statistics

- Each node* has between \( \lceil m/2 \rceil \) and \( m \) children
- 50% fill factor is the minimum for a B+ Tree.
- For tree of order 5, these guidelines must be met:

<table>
<thead>
<tr>
<th>( m, \text{ max pointers/page} )</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = m - 1, \text{ max keys/page} )</td>
<td>4</td>
</tr>
<tr>
<td>( \lceil m/2 \rceil, \text{ min pointers/page} )</td>
<td>3</td>
</tr>
<tr>
<td>( \text{min keys/page} )</td>
<td>2</td>
</tr>
<tr>
<td>( \text{minimum fill factor} )</td>
<td>50%</td>
</tr>
</tbody>
</table>

* Except the root
B+ Trees

✦ B+ Tree with $m=5$

✦ *Note:* B+ Tree in commercial database might have $m=2048$
Inserting Records

- The key determines a record’s placement in a B+ Tree
- The leaf pages are maintained in sequential order. A *doubly-linked list* (usually not shown) connects each leaf page with its sibling page(s).
Inserting Records

✦ The key determines a record’s placement in a B+ Tree

✦ The leaf pages are maintained in sequential order. A *doubly-linked list* (usually not shown) connects each leaf page with its sibling page(s).

✦ Why?
## Insertion Algorithm

<table>
<thead>
<tr>
<th>Leaf Page Full?</th>
<th>Index Page Full?</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Place the record in sorted position in the appropriate leaf page</td>
</tr>
</tbody>
</table>
| Yes             | No              | 1. Split the leaf page  
|                 |                 | 2. Copy middle key into the index page in sorted order.  
|                 |                 | 3. Left leaf page contains records with keys < the middle key.  
<p>|                 |                 | 4. Right leaf page contains records with keys ≥ than the middle key. |</p>
<table>
<thead>
<tr>
<th>Leaf Page Full?</th>
<th>Index Page Full?</th>
<th>Action</th>
</tr>
</thead>
</table>
| Yes            | Yes             | 1. Split the leaf page  
2. Records with keys < middle key go to the left leaf page  
3. Records with keys ≥ middle key go to the right leaf page.  
4. Split the index page  
5. Keys < middle key go to the left index page.  
6. Keys > middle key go to the right index page.  
7. The middle key goes to the next (higher) level. If the next level index page is full, continue splitting the index pages. |
1st Case: leaf page & index page not full

✧ Original Tree:

✧ Insert 28:
1st Case: leaf page & index page not full

✦ Original Tree:

✦ Insert 28:
2nd Case: leaf page Full, index page Not Full

- Insert a record with a key value of 70.
2nd Case: leaf page Full, index page Not Full

-$\uparrow$ Insert a record with a key value of 70.

-$\uparrow$ This record should go in the leaf page with 50, 55, 60, and 65.

-$\uparrow$ This page is full, so we need to **split** it as

<table>
<thead>
<tr>
<th>old leaf page</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 55 60 65</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>new left leaf page</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>new right leaf page</th>
</tr>
</thead>
<tbody>
<tr>
<td>60 65 70</td>
</tr>
</tbody>
</table>
Now add the new leaf to the index:

- The middle key (60) is copied into the index page between 50 and 75.
Now add the new leaf to the index:

- The middle key (60) is copied into the index page between 50 and 75.
Now add the new leaf to the index:

- The middle key (60) is copied into the index page between 50 and 75.
Now add the new leaf to the index:

- The middle key (60) is copied into the index page between 50 and 75.
3rd Case: leaf page & index page both full

To prior example, we need to add 95:
3rd Case: leaf page & index page both full

- We need to add 95 to leaf with

  \[
  \begin{array}{cccc}
  75 & 80 & 85 & 90 \\
  \end{array}
  \]

  which is full. So we need to split it into two pages:

  Left leaf page \hspace{2cm} Right leaf page

  \[
  \begin{array}{cccc}
  75 & 80 \\
  \end{array} \hspace{2cm} \begin{array}{cccc}
  85 & 90 & 95 \\
  \end{array}
  \]

- The middle key, 85, rises to the index page.
3rd Case: leaf page & index page both full

The index page is also full so we must split it:

**New root index page**

25 50

Left index page

Right index page

75 85
Before...

and after insertion of 95:
Rotation

- B+ Trees can incorporate rotation to reduce the number of page splits.
- A rotation occurs when a leaf page is full, but one of its siblings pages is not full. Rather than splitting the leaf page, we move a record to its sibling, adjusting indices as necessary.
- The left sibling is usually checked first, then the right sibling.
Rotation

For example, consider again the B+ Tree before the addition of 70. This record belongs in the leaf node containing 50 55 60 65. Notice how this node is full but its left sibling is not.

Using rotation, we move the record with the lowest key to its sibling. We must also modify the index page:
Before rotation:

After rotation:
Deletion Algorithm

Like the insertion algorithm, there are three cases to consider when deleting a record:

<table>
<thead>
<tr>
<th>Leaf Page Below Fill Factor</th>
<th>Index Page Below Fill Factor</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.</td>
</tr>
<tr>
<td>Leaf Page Below Fill Factor?</td>
<td>Index Page Below Fill Factor?</td>
<td>Action</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>1. Combine the leaf page and its sibling.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Change the index page to reflect the deletion of the leaf.</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>1. Combine the leaf page and its sibling.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Adjust the index page to reflect the other change.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Combine the index page with its sibling.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. Continue combining index pages until you reach a page with the correct fill factor, or you reach the root.</td>
</tr>
</tbody>
</table>
Deletion

To refresh our memories, here’s our B+ Tree right after the insertion of 95:
Deletion (1st case)

- Delete record with Key 70
Deletion (2nd case)

✧ Delete 25 from the B+ Tree

✧ Because 25 appears in the index page, when we delete 25, we must replace it with another key (here, with 28)
Deletion (2nd case)

- Delete 25 from the B+ Tree
- Because 25 appears in the index page, when we delete 25, we must replace it with another key (here, with 28)
Deletion (3rd case)

- Deleting 60 from the B+ Tree
- The leaf page containing 60 (60 65) will be below the fill factor after the deletion. Thus, we must combine leaf pages.
- The recombination removes one key from the index page. Hence, it will also fall below the fill factor. Thus, we must combine index pages.
- 60 appears as the only key in the root index page. Obviously, it will be removed with the deletion.
Deletion (3rd case)

- B+ Tree after deletion of 60
B+ Trees in Practice

- Typical Order: 200 to 1000
- Typical Fill Factor: 67%
- Typical Capacities:
  - Height 4: $133^4 = 312,900,700$ records
  - Height 3: $133^3 = 2,352,637$ records
B+ Trees in Practice

✧ Fill factor for newly created tree is a design parameter

✧ Suppose I create a new B+ tree with a fill factor of 100%

A. Index occupies minimum number of nodes
B. Adding a new entry will *always* require recursive splitting
C. Both of the above
D. None of the above
B⁺ Trees in Practice

✧ Fill factor for newly created tree is a design parameter

✧ Suppose I create a new B⁺ tree with a fill factor of 50%

A. Index occupies more nodes
B. Adding a new entry is unlikely to require recursive splitting
C. Both of the above
D. None of the above
Example: IBM Red Brick Warehouse 6.3

Each index node is 8172 bytes and corresponds to one 8 kB file system block minus 20 bytes overhead, and key size is the size in bytes of the key itself, plus 6 bytes of address.

Example

Assume a key size of 10 bytes and a table with 50 000 rows. To calculate the number of 8 kbyte blocks required to store the index for various fill factors, use the formulas in Estimating the size of indexes. The results are:

<table>
<thead>
<tr>
<th>Fill factor</th>
<th>keys per node</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>81</td>
<td>627</td>
</tr>
<tr>
<td>50%</td>
<td>409</td>
<td>124</td>
</tr>
<tr>
<td>100%</td>
<td>817</td>
<td>63</td>
</tr>
</tbody>
</table>

If this table is to be loaded once and you anticipate no additions, use a fill factor of 100%.
Concluding Remarks

❖ Tree structured indexes are ideal for range-searches, also good for equality searches

❖ $B^+$ Tree is a dynamic structure
  ❖ Insertions and deletions leave tree height-balanced
  ❖ High fanout means depth is often just 3 or 4
  ❖ Root can be kept in main memory
  ❖ Almost always better than maintaining a sorted file
Problem

- Find the minimum order of the B$^+$ tree that guarantees that no more than 3 disk reads will be necessary to access any record in a file of 100 million records.
- Assume that the root is in main memory.
Problem

✧ Find the minimum order of the B⁺ tree that guarantees that no more than 3 disk reads will be necessary to access any record in a file of 100 million records.
✧ Assume that the root is in main memory.
✧ Hint: look at inequality 7.7
Problem

✧ Find the minimum order of the B⁺ tree that guarantees that no more than 3 disk reads will be necessary to access any record in a file of 100 million records.
✧ Assume that the root is in main memory.
✧ Hint: look at inequality 7.7

Numeric Answer:
Problem

✧ Find the minimum order of the $B^+$ tree that guarantees that no more than 3 disk reads will be necessary to access any record in a file of 100 million records.

✧ Assume that the root is in main memory.

✧ Hint: look at inequality 7.7

$$h \leq \lceil \log_{\lceil m/2 \rceil} \left( \frac{n + 1}{4} \right) \rceil + 1.$$ 

Numeric Answer:
Problem

Draw the B-tree obtained after inserting 30 and then 31 in the B-tree in Figure 7.8. Assume that a leaf cannot contain more than three items.
Solution

Inserting 31 will require the leaf’s split and then its parent’s split:
Outline an algorithm for finding the largest key in a B⁺-tree.
Problem

Why are “recursive splits” bad news in a real database index?
Problem

✧ Levitin mentions that recursive splits can be avoided by “splitting on the way down”.

✧ Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18
Problem

Levitin mentions that recursive splits can be avoided by “splitting on the way down”.

Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18
Problem

- Levitin mentions that recursive splits can be avoided by “splitting on the way down”.
- Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18
Problem

Levitin mentions that recursive splits can be avoided by “splitting on the way down”.

Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18
Problem

✧ Levitin mentions that recursive splits can be avoided by “splitting on the way down”.

✧ Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18
Problem

✧ Levitin mentions that recursive splits can be avoided by “splitting on the way down”.

✧ Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18
Problem

✧ Levitin mentions that recursive splits can be avoided by “splitting on the way down”.

✧ Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18

b. The principal advantage of splitting full nodes (4-nodes with 3 keys) on the way down during insertion of a new key lies in the fact that if the appropriate leaf turns out to be full, its split will never cause a chain reaction of splits because the leaf’s parent will always have a room for an extra key. (If the parent is full before the insertion, it is split before the leaf is reached.) This is not the case for the insertion algorithm employed for 2-3 trees (see Section 6.3).

The disadvantage of splitting full nodes on the way down lies in the fact that it can lead to a taller tree than necessary. For the list of part (a), for example, the tree before the last one had a room for key 18 in the leaf containing key 15 and therefore didn’t require a split executed by the top-down insertion.
Problem

✦ Levitin mentions that recursive splits can be avoided by “splitting on the way down”.

✦ Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18

b. The principal advantage of splitting full nodes (4-nodes with 3 keys) on a way down during insertion of a new key lies in the fact that if the appropriate leaf turns out to be full, its split will never cause a chain reaction of splits because the leaf’s parent will always have a room for an extra key. (If the parent is full before the insertion, it is split before the leaf is reached.) This is not the case for the insertion algorithm employed for 2-3 trees (see Section 6.3).

The disadvantage of splitting full nodes on the way down lies in the fact that it can lead to a taller tree than necessary. For the list of part (a), for example, the tree before the last one had a room for key 18 in the leaf containing key 15 and therefore didn’t require a split executed by the top-down insertion.
Problem

Levitin mentions that recursive splits can be avoided by “splitting on the way down”.

Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18
Problem

Levitin mentions that recursive splits can be avoided by “splitting on the way down”.

Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18
Problem

✧ Levitin mentions that recursive splits can be avoided by “splitting on the way down”.

✧ Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18
Problem

Levitin mentions that recursive splits can be avoided by “splitting on the way down”.

Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18
Problem

- Levitin mentions that recursive splits can be avoided by “splitting on the way down”.
- Construct a B tree of order 4 by inserting (in order) into an empty tree

10, 6, 15, 31, 20, 27, 50, 44, 18
Problem

- Levitin mentions that recursive splits can be avoided by “splitting on the way down”
- What are the advantages and disadvantages of this variation (called a top-down B tree)?

![Diagram of a top-down B tree]

![Diagram of a 2-3 tree]

b. The principal advantage of splitting full nodes (4-nodes with 3 keys) on the way down during insertion of a new key lies in the fact that if the appropriate leaf turns out to be full, its split will never cause a chain reaction of splits because the leaf’s parent will always have a room for an extra key. (If the parent is full before the insertion, it is split before the leaf is reached.) This is not the case for the insertion algorithm employed for 2-3 trees (see Section 6.3).

The disadvantage of splitting full nodes on the way down lies in the fact that it can lead to a taller tree than necessary. For the list of part (a), for example, the tree before the last one had a room for key 18 in the leaf containing key 15 and therefore didn’t require a split executed by the top-down insertion.