3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context

Premature optimization

“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%.”

Don Knuth

Symbol table implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log N</td>
<td>N</td>
<td>log N</td>
<td>N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
</tr>
<tr>
<td>red-black BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

Q. Can we do better?
A. Yes, but with different access to the data.

Hashing: basic plan

Save items in a key-indexed array (index is a function of the key).

Hash function. Method for computing array index from key.

Issues.
- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

Classic space-time tradeoff.
- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context

Hash Tables: quiz 1

What's the best way to hash a 10-digit phone number to a value between 0 and 999?

A. First 3 digits
B. Last 3 digits
C. Either A or B
D. Neither A nor B
E. I don't know.
Java’s hash code conventions

All Java classes inherit a method `hashCode()`, which returns a 32-bit int.

**Requirement.** If `x.equals(y)`, then `(x.hashCode()) == y.hashCode()`.

**Highly desirable.** If `x.equals(y)`, then `(x.hashCode()) != y.hashCode()`.

**Default implementation.** Memory address of `x`.

**Legal (but poor) implementation.** Always return 17.

**Customized implementations.** Integer, Double, String, File, URL, Date, ...

**User-defined types.** Users are on their own.

Implementing hash code: integers, booleans, and doubles

Java library implementations

```
public final class Integer {
    private final int value;

    public int hashCode() {
        return value;
    }
}
```

```
public final class Double {
    private final double value;

    public int hashCode() {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >> 32));
    }
}
```

Warning: -0.0 and +0.0 have different hash codes

Are these magic constants??

Implementing hash code: strings

```
public final class String {
    private final char[] s;

    public int hashCode() {
        int hash = 0;
        for (int i = 0; i < length(); i++)
            hash = hash * 31 + s[i];
        return hash;
    }
}
```

Java library implementation

```
char Unicode

... ... ...
'a' 97
'b' 98
'c' 99
...
```

```
Recall:
```
```
h = s[0] · 31^L-1 + s[1] · 31^{L-2} + ... + s[L-1] · 31^0
```

**Peek ahead:** modular hashing

Convert hash code to array index by taking remainder mod array length

Q. What could go wrong if the length of array is 31 (or a multiple of 31)?
   A. Only the last character of the string affects the array index

Q. Is this hash function better or worse than Java's?
   A. Worse, because it is much more likely that the array length will have a common factor with 30 than with 31.

Horner's method: only $L$ multiplications/additions to hash string of length $L$.

```
String s = "call";
s.hashCode(); 3045982 = 99·31^1 + 97·31^0 + 108·31^1 + 108·31^0
```

```
= 108 + 31·(108 + 31·(97 + 31·(99))
```

```
Implementing hash code: strings

Performance optimization.
- Cache the hash value in an instance variable.
- Return cached value.

```java
class String {
    private final int hash;
    private final char[] sj;

    public int hashCode() {
        if (h != 0) return h;
        h = s.length;
        for (int i = 0; i < length(); i++)
            h = h * 31 + sj[i];
        return h;
    }
}
```

Q. What if `hashCode()` of string is 0?  `hashCode()` of "pizzilung sandwiches" is 0

Implementing hash code: user-defined types

```java
public final class Transaction implements Comparable<Transaction> {
    private final String who;
    private final Date when;
    private final double howmuch;

    public Transaction(String who, Date when, double howmuch) {
        // as before 
    }

    public boolean equals(Object y) {
        // as before 
    }

    public int hashCode() {
        int hash = 17;
        hash = 31 * hash + who.hashCode();
        hash = 31 * hash + when.hashCode();
        hash = 31 * hash + (Double) howmuch.hashCode();
        return hash;
    }
}
```

`equals()` method for user-defined types.

Hash code design

"Standard" recipe for user-defined types.
- Combine each significant field using the 31x + y rule.
- If field is a primitive type, use wrapper type `hashCode()`.
- If field is `null`, use 0.
- If field is a reference type, use `hashCode()`.
- If field is an array, apply to each entry.

In practice. Recipe above works reasonably well; used in Java libraries.
In theory. Keys are bitstrings: "universal" family of hash functions exist.

Basic rule. Need to use the whole key to compute hash code; consult an expert for state-of-the-art hash codes.

Hash tables: quiz 2

Which of the following is an effective way to map a hashable key to an integer between 0 and M-1?

A. `private int hash(Key key) { return key.hashCode() % M; }`

B. `private int hash(Key key) { return Math.abs(key.hashCode()) % M; }`

C. Both A and B.
D. Neither A nor B.
E. I don't know.
### Modular hashing

**Hash code.** An int between $-2^{31}$ and $2^{31} - 1$.

**Hash function.** An int between 0 and $M - 1$ (for use as array index).

```java
private int hash(Key key)
    { return key.hashCode() % M; }
```

1-in-a-billion bug

```java
private int hash(Key key)
    { return Math.abs(key.hashCode()) % M; }
```

**correct**

```java
private int hash(Key key)
    { return (key.hashCode() & 0x7fffffff) % M; }
```

### Uniform hashing assumption

**Uniform hashing assumption.** Each key is equally likely to hash to an integer between 0 and $M - 1$.

**Bins and balls.** Throw balls uniformly at random into $M$ bins.

**Birthday problem.** Expect two balls in the same bin after $\sim \sqrt{\pi M / 2}$ tosses.

### Hash value frequencies for words in Tale of Two Cities (M = 97)

Java's String data uniformly distribute the keys of Tale of Two Cities.

### 3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context

[Algorithms](http://algs4.cs.princeton.edu)
Collisions

Collision. Two distinct keys hashing to same index.

Birthday problem ⇒ can’t avoid collisions.

unless you have a ridiculous (quadratic) amount of memory

Collisions

Collisions

Collisions

Collisions

Separate-chaining symbol table

Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]

- Hash: map key to integer $i$ between 0 and $M - 1$.
- Insert: put at front of $i$th chain (if not already in chain).
- Search: sequential search in $i$th chain.

Separate-chaining symbol table

Separate-chaining symbol table

Separate-chaining symbol table: Java implementation

public class SeparateChainingHashST<Key, Value> {
  private int M = 9; // number of chains
  private Node[] st = new Node[M]; // array of chains
  
  private static class Node {
    private Object key; // no generic array creation
    private Object val; // declare key and value of type Object
    private Node next;
    ...}
  
  private int hash(Key key) {
    return (key.hashCode() & 0xffffffff) % M; }
  
  public Value get(Key key) {
    int i = hash(key);
    for (Node x = st[i]; x != null; x = x.next)
      if (key.equals(x.key)) return (Value) x.val;
    return null;
  }
}
Separate-chaining symbol table: Java implementation

```java
public class SeparateChainingHashTable<Key, Value> {
    private int M = 97; // number of chains
    private Node[] st = new Node[M]; // array of chains

    private static class Node {
        private Object key;
        private Object val;
        private Node next;
        ...
    }

    private int hash(Key key) {
        return (key.hashCode() & 0xffffffff) % M;
    }

    public void put(Key key, Value val) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)
            if (key.equals(x.key)) { x.val = val; return; }
        st[i] = new Node(key, val, st[i]);
    }
}
```

Analysis of separate chaining

**Proposition.** Under uniform hashing assumption, the number of keys in each list is close to $N/M$.

**Consequence.** Number of probes for search/insert is proportional to $N/M$.

- $M$ too large $\Rightarrow$ too many empty chains.
- $M$ too small $\Rightarrow$ chains too long.
- Typical choice: $M = \frac{1}{2} N \Rightarrow$ constant-time ops.

Q. When to resize?

Resizing in a separate-chaining hash table

**Goal.** Average length of list $N/M = \text{constant}$.

- Double size of array $M$ when $N/M \geq 8$;
  - halve size of array $M$ when $N/M \leq 2$.
- Note: need to rehash all keys when resizing.

Deletion in a separate-chaining hash table

**Q.** How to delete a key (and its associated value)?

**A.** Easy: need to consider only chain containing key.
## Symbol Table Implementations: Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Ordered Access</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>log N</td>
</tr>
<tr>
<td>red-black BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>separate chaining</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption

## Collision Resolution: Linear Probing

**Linear probing.** [Amdahl–Boehme–Rochester–Samuel, IBM 1953]

- Maintain keys and values in two parallel arrays.
- When a new key collides, find next empty slot, and put it there.

### Linear-Probing Hash Table (M = 16, N = 10)

<table>
<thead>
<tr>
<th>keys[]</th>
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(K, 14)</td>
<td>K</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>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash(K) = 7</td>
<td>14</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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| vals[]  | 11 | 10 | 9 | 5 | 6 | 12 | 13 | 4 | 8 |

## 3.4 Hash Tables

**Hash.** Map key to integer \( i \) between 0 and \( M - 1 \).

- **Insert.** Put at table index \( i \) if free; if not try \( i + 1, i + 2 \), etc.
- **Search.** Search table index \( i \); if occupied but no match, try \( i + 1, i + 2 \), etc.

**Note.** Array size \( M \) must be greater than number of key-value pairs \( N \).

<table>
<thead>
<tr>
<th>keys[]</th>
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(K, 14)</td>
<td>K</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>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash(K) = 7</td>
<td>14</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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| vals[]  | 11 | 10 | 9 | 5 | 6 | 12 | 13 | 4 | 8 |

\( M = 16 \)
Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M - 1$.
Insert. Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

Linear-probing hash table

<table>
<thead>
<tr>
<th>keys[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M - 1$.
Insert. Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

Linear-probing hash table

<table>
<thead>
<tr>
<th>keys[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M - 1$.
Insert. Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

Linear-probing hash table

<table>
<thead>
<tr>
<th>keys[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M - 1$.
Insert. Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

Linear-probing hash table

<table>
<thead>
<tr>
<th>keys[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

insert S
hash(S) = 6

Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M - 1$.
Insert. Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

Linear-probing hash table

<table>
<thead>
<tr>
<th>keys[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Linear-probing hash table demo:  insert

Hash. Map key to integer \( i \) between 0 and \( M - 1 \).
Insert. Put at table index \( i \) if free; if not try \( i + 1, i + 2 \), etc.

```
linear-probing hash 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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
keys[] |      | S |      |      |      |      |      |      |      |      |      |      |      |      |      |
```

Linear-probing hash table demo:  insert

Hash. Map key to integer \( i \) between 0 and \( M - 1 \).
Insert. Put at table index \( i \) if free; if not try \( i + 1, i + 2 \), etc.

```
insert E
hash(E) = 10
```

```
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
keys[] |      | S |      |      |      |      |      |      |      |      |      |      |      |      |      |
```

Linear-probing hash table demo:  insert

Hash. Map key to integer \( i \) between 0 and \( M - 1 \).
Insert. Put at table index \( i \) if free; if not try \( i + 1, i + 2 \), etc.

```
insert E
hash(E) = 10
```

```
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
keys[] |      | S |      |      |      |      |      |      |      |      |      |      |      |      |      |
```
**Linear-probing hash table demo: insert**

**Hash.** Map key to integer $i$ between 0 and $M - 1$.

**Insert.** Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

**Linear-probing hash 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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<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>S</td>
<td></td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

**Linear-probing hash table demo: insert**

**Hash.** Map key to integer $i$ between 0 and $M - 1$.

**Insert.** Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

**Linear-probing hash 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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<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>S</td>
<td></td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

**Linear-probing hash table demo: insert**

**Hash.** Map key to integer $i$ between 0 and $M - 1$.

**Insert.** Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

**Linear-probing hash 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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<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>S</td>
<td></td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

**Linear-probing hash table demo: insert**

**Hash.** Map key to integer $i$ between 0 and $M - 1$.

**Insert.** Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

**Linear-probing hash 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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<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></td>
<td></td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>
Linear-probing hash table demo: insert

**Hash.** Map key to integer $i$ between 0 and $M - 1$.
**Insert.** Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

---

**linear-probing hash table**

<table>
<thead>
<tr>
<th></th>
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td></td>
<td></td>
<td>A</td>
<td>S</td>
<td>E</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>

---

Linear-probing hash table demo: insert

**Hash.** Map key to integer $i$ between 0 and $M - 1$.
**Insert.** Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

---

**insert R**
**hash(R) = 14**

<table>
<thead>
<tr>
<th></th>
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>S</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Linear-probing hash table demo: insert

**Hash.** Map key to integer $i$ between 0 and $M - 1$.
**Insert.** Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

---

**insert R**
**hash(R) = 14**

<table>
<thead>
<tr>
<th></th>
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>S</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Linear-probing hash table demo: insert

**Hash.** Map key to integer $i$ between 0 and $M - 1$.
**Insert.** Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

---

**insert R**
**hash(R) = 14**

<table>
<thead>
<tr>
<th></th>
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>S</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Linear-probing hash table demo: insert

**Hash.** Map key to integer $i$ between 0 and $M - 1$.
**Insert.** Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.
**Linear-probing hash table demo: insert**

**Hash**: Map key to integer $i$ between 0 and $M - 1$.

**Insert**: Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

```
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>S</td>
<td>E</td>
<td>R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Linear-probing hash table demo: insert**

```
insert C
hash(C) = 5
```

```
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>S</td>
<td>E</td>
<td>R</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>
```

**Linear-probing hash table demo: insert**

```
insert C
hash(C) = 5
```

```
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>E</td>
<td>R</td>
<td></td>
</tr>
</tbody>
</table>
```
Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and M – 1.
Insert. Put at table index i if free; if not try i + 1, i + 2, etc.

linear-probing hash table

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

keys[]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>A</th>
<th>C</th>
<th>S</th>
<th></th>
<th></th>
<th></th>
<th>E</th>
<th></th>
<th>R</th>
</tr>
</thead>
</table>

Linear-probing hash table demo: insert

Hash. Map key to integer i between 0 and M – 1.
Insert. Put at table index i if free; if not try i + 1, i + 2, etc.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

keys[]

|   |   |   |   |   |   |   |   | A | C | S |   |   |   | E |   | R |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

insert H
hash(H) = 4

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

keys[]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>A</th>
<th>C</th>
<th>S</th>
<th></th>
<th></th>
<th></th>
<th>E</th>
<th></th>
<th>R</th>
</tr>
</thead>
</table>

H
Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M - 1$.
Insert. Put at table index $i$ if free; if not try $i + 1$, $i + 2$, etc.

```
insert H
hash(H) = 4
```

```
keys[]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
H A C S E R
```

Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M - 1$.
Insert. Put at table index $i$ if free; if not try $i + 1$, $i + 2$, etc.

```
insert H
hash(H) = 4
```

```
keys[]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
H A C S E R
```
Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M-1$.
Insert. Put at table index $i$ if free; if not try $i+1$, $i+2$, etc.

```
insert X
hash(X) = 15
keys[]
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```
keys[]
```

```
A C S H E R X
```

Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M-1$.
Insert. Put at table index $i$ if free; if not try $i+1$, $i+2$, etc.

```
insert X
hash(X) = 15
keys[]
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```
keys[]
```

```
A C S H E R X
```

Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M-1$.
Insert. Put at table index $i$ if free; if not try $i+1$, $i+2$, etc.

```
insert X
hash(X) = 15
keys[]
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```
keys[]
```

```
A C S H E R X
```

Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M-1$.
Insert. Put at table index $i$ if free; if not try $i+1$, $i+2$, etc.

```
linear-probing hash table
```
Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M - 1$.
Insert. Put at table index $i$ if free; if not try $i + 1$, $i + 2$, etc.

```
insert M
hash(M) = 1
```

```
keys[] = [M, A, C, S, H, E, R, X]
```

Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M - 1$.
Insert. Put at table index $i$ if free; if not try $i + 1$, $i + 2$, etc.

```
insert M
hash(M) = 1
```

```
keys[] = [M, A, C, S, H, E, R, X]
```

Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M - 1$.
Insert. Put at table index $i$ if free; if not try $i + 1$, $i + 2$, etc.

```
insert M
hash(M) = 1
```

```
keys[] = [M, A, C, S, H, E, R, X]
```

Linear-probing hash table demo: insert

Hash. Map key to integer $i$ between 0 and $M - 1$.
Insert. Put at table index $i$ if free; if not try $i + 1$, $i + 2$, etc.

```
insert M
hash(M) = 1
```

```
keys[] = [M, A, C, S, H, E, R, X]
```
**Linear-probing hash table demo: insert**

**Hash.** Map key to integer \( i \) between 0 and \( M \) – 1.

**Insert.** Put at table index \( i \) if free; if not try \( i + 1, i + 2, \) etc.

```
insert P
hash(P) = 14
```

```
keys[]
```

<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
keys[]
```

<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
p
```

```
p
```

```
keys[]
```

<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>M</td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
p
```

```
p
```

```
keys[]
```

<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>M</td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
p
```
Linear-probing hash table demo: insert

Hash. Map key to integer \( i \) between 0 and \( M - 1 \).
Insert. Put at table index \( i \) if free; if not try \( i + 1 \), \( i + 2 \), etc.

linear-probing hash table

<table>
<thead>
<tr>
<th>keys[]</th>
<th>P</th>
<th>M</th>
<th>A</th>
<th>C</th>
<th>S</th>
<th>H</th>
<th>E</th>
<th>R</th>
<th>X</th>
</tr>
</thead>
</table>

Linear-probing hash table demo: insert

Hash. Map key to integer \( i \) between 0 and \( M - 1 \).
Insert. Put at table index \( i \) if free; if not try \( i + 1 \), \( i + 2 \), etc.

insert L
hash(L) = 6

<table>
<thead>
<tr>
<th>keys[]</th>
<th>P</th>
<th>M</th>
<th>A</th>
<th>C</th>
<th>S</th>
<th>H</th>
<th>E</th>
<th>R</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Linear-probing hash table demo: insert

**Hash.** Map key to integer $i$ between 0 and $M - 1$.
**Insert.** Put at table index $i$ if free; if not try $i + 1, i + 2$, etc.

```plaintext
insert L
hash(L) = 6
```

<table>
<thead>
<tr>
<th>keys[]</th>
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>M</td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L</td>
<td>R</td>
<td>X</td>
</tr>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

### Linear-probing hash table demo: search

**Hash.** Map key to integer $i$ between 0 and $M - 1$.
**Search.** Search table index $i$; if occupied but no match, try $i + 1, i + 2$, etc.

```plaintext
search L
hash(L) = 6
```

<table>
<thead>
<tr>
<th>keys[]</th>
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>M</td>
<td></td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td>X</td>
</tr>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Linear-probing hash table demo: search

**Hash.** Map key to integer \( i \) between 0 and \( M - 1 \).

**Search.** Search table index \( i \); if occupied but no match, try \( i + 1 \), \( i + 2 \), etc.

```
search E
hash(E) = 10
```

```
keys[] = P M A C S H L E R X
```

Linear-probing hash table demo: search

**Hash.** Map key to integer \( i \) between 0 and \( M - 1 \).

**Search.** Search table index \( i \); if occupied but no match, try \( i + 1 \), \( i + 2 \), etc.

```
search E
hash(E) = 10
```

```
keys[] = P M A C S H L E R X
```

Linear-probing hash table demo: search

**Hash.** Map key to integer \( i \) between 0 and \( M - 1 \).

**Search.** Search table index \( i \); if occupied but no match, try \( i + 1 \), \( i + 2 \), etc.

```
linear-probing hash table
```

```
keys[] = P M A C S H L E R X
```

Linear-probing hash table demo: search

**Hash.** Map key to integer \( i \) between 0 and \( M - 1 \).

**Search.** Search table index \( i \); if occupied but no match, try \( i + 1 \), \( i + 2 \), etc.

```
search L
hash(L) = 6
```

```
keys[] = P M A C S H L E R X
```
Linear-probing hash table demo: search

Hash. Map key to integer $i$ between 0 and $M - 1$.
Search. Search table index $i$; if occupied but no match, try $i + 1$, $i + 2$, etc.

search L
hash(L) = 6

<table>
<thead>
<tr>
<th>keys[]</th>
<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td></td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>

search hit
(return corresponding value)
**Linear-probing hash table demo: search**

**Hash.** Map key to integer $i$ between 0 and $M-1$.

**Search.** Search table index $i$; if occupied but no match, try $i+1, i+2$, etc.

```
search K
hash(K) = 5

keys[] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
   P M A C S H L E           R X

search K
hash(K) = 5

keys[] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
   P M A C S H L E           R X
```

**Linear-probing hash table demo: search**

**Hash.** Map key to integer $i$ between 0 and $M-1$.

**Search.** Search table index $i$; if occupied but no match, try $i+1, i+2$, etc.

```
search K
hash(K) = 5

keys[] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
   P M A C S H L E           R X

search K
hash(K) = 5

keys[] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
   P M A C S H L E           R X
```
Linear-probing hash table demo: search

**Hash.** Map key to integer \( i \) between 0 and \( M - 1 \).

**Search.** Search table index \( i \); if occupied but no match, try \( i + 1, i + 2 \), etc.

```java
public class LinearProbingHashST<Key, Value> {  
    private int M = 30001;  
    private Value[] vals = (Value[]) new Object[M];  
    private Key[] keys = (Key[]) new Object[M];  
    private int hash(Key key) { /* as before */ }  
    private void put(Key key, Value val) { /* next slide */ }  
    public Value get(Key key) {  
        for (int i = hash(key); keys[i] != null; i = (i + 1) % M)  
            if (key.equals(keys[i]))  
                return vals[i];  
        return null;  
    }  
}
```

Linear-probing symbol table: Java implementation

```java
public class LinearProbingHashST<Key, Value> {  
    private int M = 30001;  
    private Value[] vals = (Value[]) new Object[M];  
    private Key[] keys = (Key[]) new Object[M];  
    private int hash(Key key) { /* as before */ }  
    private void put(Key key, Value val) { /* prev slide */ }  
    public Value get(Key key) {  
        int i;  
        for (i = hash(key); keys[i] != null; i = (i + 1) % M)  
            if (keys[i].equals(key))  
                break;  
        keys[i] = key;  
        vals[i] = val;  
    }  
}
```
Clustering

**Cluster.** A contiguous block of items.

**Observation.** New keys likely to hash into middle of big clusters.

### Analysis of linear probing

**Proposition.** Under uniform hashing assumption, the average $\#$ of probes in a linear probing hash table of size $M$ that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right)$$

- search hit

$$\sim \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right)$$

- search miss / insert

**Parameters.**

- $\alpha$ too small $\Rightarrow$ too many empty array entries.
- $\alpha$ too large $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N / M \sim \frac{1}{2}$

Q. When to resize?

### Resizing in a linear-probing hash table

**Goal.** Average length of list $N / M \leq \frac{1}{2}$.

- Double size of array $M$ when $N / M \geq \frac{1}{2}$.
- Halve size of array $M$ when $N / M \leq \frac{1}{2}$.
- Need to rehash all keys when resizing.

#### before resizing

<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>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td>E</td>
<td>S</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vals[]</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### after resizing

<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td>A</td>
<td>S</td>
<td>E</td>
<td>R</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>vals[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Deletion in a linear-probing hash table

Q. How to delete a key (and its associated value)?

A. Requires some care: can’t just delete array entries.

#### before deleting 5

<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td>R</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vals[]</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### after deleting 5

<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>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td>R</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vals[]</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>4</td>
<td>5</td>
<td>11</td>
<td>12</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Ordered Ops</th>
<th>Key Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>red-black BST</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td>separate chaining</td>
<td>$N$</td>
<td>$N$</td>
<td>$1^*$</td>
<td>$1^*$</td>
</tr>
<tr>
<td>-hash to two positions, reduces length of longest chain to $\sim \log \log N$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linear probing</td>
<td>$N$</td>
<td>$N$</td>
<td>$1^*$</td>
<td>$1^*$</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption

Separate chaining vs. linear probing

Separate chaining.
- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

Linear probing.
- Less wasted space.
- Better cache performance.

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. [separate-chaining variant]
- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\sim \log \log N$.

Double hashing. [linear-probing variant]
- Use linear probing, but skip a variable amount, not just 1 each time.
- Effectively eliminates clustering.
- Can allow table to become nearly full.
- More difficult to implement delete.

Cuckoo hashing. [linear-probing variant]
- Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur).
- Constant worst-case time for search.

Hash tables vs. balanced search trees

Hash tables.
- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for String (e.g., cached hash code).

Balanced search trees.
- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement compareTo() correctly than equals() and hashCode().

Java system includes both.
- Red-Black BSTs: java.util.TreeMap, java.util.TreeSet.
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context

War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?
A. Obvious situations: aircraft control, nuclear reactor, pacemaker, HFT, ...
A. Surprising situations: denial-of-service attacks.

Real-world exploits. [Crosby–Wallach 2003]
- Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem.
- Perl 5.8.0: insert carefully chosen strings into associative array.
- Linux 2.4.20 kernel: save files with carefully chosen names.

Algorithmic complexity attack on Java

Goal. Find family of strings with the same hashCode().
Solution. The base-31 hash code is part of Java’s String API.

<table>
<thead>
<tr>
<th>key</th>
<th>hashCode()</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Aa”</td>
<td>2112</td>
</tr>
<tr>
<td>“Bb”</td>
<td>2112</td>
</tr>
<tr>
<td>“AA”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“AA”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“AA”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“AAA”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“AAA”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“AAA”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“XXX”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“XXX”</td>
<td>-540425984</td>
</tr>
<tr>
<td>“XXX”</td>
<td>-540425984</td>
</tr>
</tbody>
</table>

2^N strings of length 2N that hash to same value!

Diversion: one-way hash functions

One-way hash function. Hard to find a key that will hash to a desired value (or two keys that hash to same value).

Ex. MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160, ....

Applications. Crypto, message digests, passwords, Bitcoin, ....

Caveat. Too expensive for use in ST implementations.