3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
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>search</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete</td>
<td>delete</td>
<td></td>
<td></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>$n$</td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\sqrt{n}$</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>hashing</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$1 \dagger$</td>
</tr>
</tbody>
</table>

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

\[ \dagger \text{ under suitable technical assumptions} \]
Hashing: basic plan

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

**Hash function.** Method for computing array index from key.

**Example.** Symbol table that maps US state names to capitals.

Assume that hash() takes in a string and outputs an integer between 0 and 100.

<table>
<thead>
<tr>
<th>String</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>“New Jersey”</td>
<td>4</td>
</tr>
<tr>
<td>“California”</td>
<td>1</td>
</tr>
<tr>
<td>“Texas”</td>
<td>4</td>
</tr>
</tbody>
</table>

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

All Java classes inherit a method `equals()`.

**Java requirements.** For any references `x`, `y` and `z`:

- **Reflexive:** `x.equals(x)` is true.
- **Symmetric:** `x.equals(y)` iff `y.equals(x)`.
- **Transitive:** if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- **Non-null:** `x.equals(null)` is false.

**Default implementation.** `(x == y)`

**Customized implementations.** `Integer`, `Double`, `String`, `java.net.URL`, ...

**User-defined implementations.** Some care needed.
Implementing equals for user-defined types

Exercise. What are 5 additions/modifications you need to make to this code?

```java
public class Date
{
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Date that)
    {
        if (this.day != that.day) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year) return false;
        return true;
    }
}
```
Implementing equals for user-defined types

Seems easy, but requires some care.

```java
public final class Date {
    private final int month;
    private final int day;
    private final int year;

    public boolean equals(Object y) {
        if (y == this) return true;
        if (y == null) return false;
        if (y.getClass() != this.getClass())
            return false;

        Date that = (Date) y;
        if (this.day != that.day ) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year ) return false;
        return true;
    }
}
```

typically unsafe to use equals() with inheritance (would violate symmetry)

must be Object.
optimization (for reference equality)
check for null
objects must be in the same class (religion: getClass() vs. instanceof)
cast is now guaranteed to succeed
check that all significant fields are the same
Equals design

“Standard” recipe for user-defined types.

- Optimization for reference equality.
- Check against null.
- Check that two objects are of the same type; cast.
- Compare each significant field:
  - if field is a primitive type, use ==
  - if field is an object, use equals() and apply rule recursively
  - if field is an array of primitives, use Arrays.equals()
  - if field is an array of objects, use Arrays.deepEquals()

Best practices.

- Do not use calculated fields that depend on other fields.
- Compare fields mostly likely to differ first.
- Make compareTo() consistent with equals().

\[
x.\text{equals}(y) \text{ if and only if } (x.\text{compareTo}(y) == 0)
\]
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Computing the hash function

**Idealistic goal.** Scramble the keys uniformly to produce a table index.
Computing the hash function

**Idealistic goal.** Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

**Ex 1.** Last 4 digits of Social Security number.
**Ex 2.** Last 4 digits of phone number.

**Practical challenge.** Need different approach for each key type.
Hash tables: quiz 1

Which is the last digit of your day of birth?

A. 0 or 1
B. 2 or 3
C. 4 or 5
D. 6 or 7
E. 8 or 9
Which is the last digit of your year of birth?

A. 0 or 1
B. 2 or 3
C. 4 or 5
D. 6 or 7
E. 8 or 9
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, `java.net.URL`, ...

**User-defined types.** Users are on their own.
Implementing hash code: integers, booleans, and doubles

Java library implementations

```java
public final class Integer {
    private final int value;
    ...
    
    public int hashCode() {
        return value;
    }
}

public final class Boolean {
    private final boolean value;
    ...
    
    public int hashCode() {
        if (value) return 1231;
        else return 1237;
    }
}

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
31x + y rule.

- Initialize hash to 1.
- Repeatedly multiply hash by 31 and add next integer in array.

```java
public class Arrays {
    ...

    public static int hashCode(int[] a) {
        if (a == null)
            return 0;  // special case for null

        int hash = 1;
        for (int i = 0; i < a.length; i++)
            hash = 31*hash + a[i];
        return hash;
    }
}
```
Implementing hash code: strings

Treat a string as an array of characters.

```java
public final class String {
    private final char[] s;
    
    public int hashCode() {
        int hash = 0; // Initialize to 0 rather than 1
        for (int i = 0; i < length(); i++)
            hash = s[i] + (31 * hash);
        return hash;
    }
}
```

Java library implementation

<table>
<thead>
<tr>
<th>char</th>
<th>Unicode</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>'a'</td>
<td>97</td>
</tr>
<tr>
<td>'b'</td>
<td>98</td>
</tr>
<tr>
<td>'c'</td>
<td>99</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Aside: string hash collisions in Java

2^n strings of length 2n that hash to same value!
public final class Transaction
{
  private final String who;
  private final Date when;
  private final double amount;

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

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

  ...

  public int hashCode()
  {
    int hash = 1;
    hash = 31*hash + who.hashCode();
    hash = 31*hash + when.hashCode();
    hash = 31*hash + ((Double) amount).hashCode();
    return hash;
  }
}
public final class Transaction
{
    private final String who;
    private final Date when;
    private final double amount;

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

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

    ...

    public int hashCode()
    { return Objects.hash(who, when, amount); }
}
Hash code design

“Standard” recipe for user-defined types.

- Combine each significant field using the $31x + y$ rule.
- Shortcut 1: use Objects.hash() for all fields (except arrays).
- Shortcut 2: use Arrays.hashCode() for primitive arrays.
- Shortcut 3: use Arrays.deepHashCode() for object arrays.

In practice. Recipe above works reasonably well; used in Java libraries.

In theory. Keys are bitstring; “universal” family of hash functions exist.

Basic rule. Need to use the whole key to compute hash code.
Which code maps hashable keys to integers 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.
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).

typically a prime or power of 2

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

bug

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

1-in-a-billion bug

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

correct

```java
private int hash(Key key) {
    if (m is a power of 2, can use
    key.hashCode() & (m-1))
    return key.hashCode() % m;
}
```
Uniform hashing assumption

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

Mathematical model: balls & bins. Toss $n$ balls uniformly at random into $m$ bins.

Bad news. Expect two balls in the same bin after $\sim \sqrt{\pi m / 2}$ tosses.

Birthday problem. In a random group of 23 or more people, more likely than not that two people will share the same birthday.
Uniform hashing assumption

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

Mathematical model: balls & bins. Toss $n$ balls uniformly at random into $m$ bins.

Good news: load balancing.

- When $n = m$, expect most loaded bin has $\sim \frac{\ln m}{\ln \ln m}$ balls.
- When $n >> m$, the number of balls in each bin is “likely” “close” to $n / m$.

Visual evidence.

Can be quantified and proved; see COS 340
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Collisions

**Collision.** Two distinct keys hashing to same index.

- hash("New Jersey") = 4
- hash("California") = 1
- hash("Texas") = 4

Can’t avoid collisions (recall birthday problem).

No index gets too many collisions (recall load balancing).

⇒ OK to do a linear search through all colliding keys.
Separate-chaining symbol table

Use an array of \( m \) linked lists.
- 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 hash table (\( m = 4 \))

\[
egin{array}{c}
\text{put}(L, 11) \\
\text{hash}(L) = 3
\end{array}
\]
Separate-chaining symbol table

Use an array of \( m \) linked lists.

- Hash: map key to integer \( i \) between 0 and \( m - 1 \).
- Insert: put at front of \( i^{\text{th}} \) chain (if not already in chain).
- Search: sequential search in \( i^{\text{th}} \) chain.

separate-chaining hash table (\( m = 4 \))

get(E)
\[ \text{hash}(E) = 1 \]
Separate-chaining symbol table: Java implementation

```java
public class SeparateChainingHashST<Key, Value> {
    private int m = 128; // 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() & 0x7fffffff) % 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;
    }
}
```
public class SeparateChainingHashST<Key, Value> {
    private int m = 128; // 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() & 0x7fffffff) % 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

**Recall.** Under uniform hashing assumption, length of each chain is approximately $n / m$ (load balancing in balls and bins model).

**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: $m \sim \frac{1}{4} n \Rightarrow$ constant time per operation.

![Hash value frequencies for words in Tale of Two Cities (m = 97)](image)
Resizing in a separate-chaining hash table

**Goal.** Average length of list \( n / m = \text{constant} \).
- Double length \( m \) of array when \( n / m \geq 8 \);
- halve length \( m \) of array when \( n / m \leq 2 \).
- Note: need to rehash all keys when resizing.

before resizing \((n/m = 8)\)

![Diagram showing hash table before resizing with \( n/m = 8 \)]

after resizing \((n/m = 4)\)

![Diagram showing hash table after resizing with \( n/m = 4 \)]

x. hashCode() does not change;
but hash(x) typically does
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.

before deleting C

after deleting C

st[]
0
1
2
3
K → I
P → N → L
J → F → C → B
O → M
K → I
P → N → L
J → F → B
O → M
## 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</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</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></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>$1^\dagger$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$^\dagger$ under uniform hashing assumption
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Collision resolution: alternate approach

Open addressing.

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

**Note.** If the array is full, the search doesn’t terminate.

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>P</td>
<td>M</td>
<td></td>
<td></td>
<td>A</td>
<td>C</td>
<td></td>
<td>H</td>
<td>L</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{put}(K, 14) \quad \text{hash}(K) = 7 \quad \text{K}
\]

<table>
<thead>
<tr>
<th>vals[]</th>
<th>00</th>
<th>10</th>
<th></th>
<th>09</th>
<th>05</th>
<th>06</th>
<th>12</th>
<th>13</th>
<th></th>
<th></th>
<th>04</th>
<th>08</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td></td>
<td></td>
<td>09</td>
<td>05</td>
<td>06</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
<td>04</td>
<td>08</td>
</tr>
</tbody>
</table>
Linear-probing hash table summary

**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 length $m$ must be greater than number of key–value pairs $n$.

<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>L</th>
<th>E</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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>1</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>2</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>3</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>4</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>5</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>6</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>7</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>8</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>9</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>10</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>11</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>12</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>13</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>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>
</tr>
<tr>
<td>15</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>

$m = 16$
public class LinearProbingHashST<Key, Value> {
    private int m = 32768;
    private Value[] vals = (Value[]) new Object[m];
    private Key[] keys = (Key[]) new Object[m];

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

    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
public class LinearProbingHashST<Key, Value>
{
    private int m = 32768;
    private Value[] vals = (Value[]) new Object[m];
    private Key[] keys = (Key[]) new Object[m];

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

    private Value get(Key key) { /* prev slide */   }

    public void put(Key key, Value val)
    {
        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;
    }
}
Under the uniform hashing assumption, where is the next key most likely to be added in this linear-probing hash table?

A. Index 7
B. Index 14
C. Index 4 or index 14
D. All open indices are equally likely
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 at most

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

search hit

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

search miss / insert

Proof. Out of scope; see:

Parameters.

- $m$ too large $\Rightarrow$ too many empty array entries.
- $m$ too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = n / m \sim \frac{1}{2}$. # probes for search hit is about $3/2$
  # probes for search miss is about $5/2$
Resizing in a linear-probing hash table

**Goal.** Fullness of array (“load factor”) $n / m \leq \frac{1}{2}$.

- Double length of array $m$ when $n / m \geq \frac{1}{2}$.
- Halve length of array $m$ when $n / m \leq \frac{1}{8}$.
- Need to rehash all keys when resizing.

**before resizing**

<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>
</tr>
</thead>
<tbody>
<tr>
<td>keys[]</td>
<td>E</td>
<td>S</td>
<td></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></td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**after resizing**

<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></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>vals[]</td>
<td>2</td>
<td>0</td>
<td></td>
<td>1</td>
<td>3</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>
Deletion in a linear-probing hash table

Q. How to delete a key (and its associated value)?
A. Requires some care: can’t simply delete array entries.

before deleting S

<table>
<thead>
<tr>
<th>keys[]</th>
<th>keys after deleting S?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</td>
<td></td>
</tr>
<tr>
<td>P M A C S H L E R X</td>
<td></td>
</tr>
<tr>
<td>vals[]</td>
<td>vals after deleting S</td>
</tr>
<tr>
<td>10 9 8 4 0 5 11 12</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><strong>Search</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential search (unordered list)</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>equals()</td>
</tr>
<tr>
<td>Binary search (ordered array)</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>Red–black BST</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\sqrt{n}$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>Separate chaining</td>
<td>$n$</td>
<td>$n$</td>
<td>$1^\dagger$</td>
<td>equals()</td>
</tr>
<tr>
<td>Linear probing</td>
<td>$n$</td>
<td>$n$</td>
<td>$1^\dagger$</td>
<td>equals()</td>
</tr>
</tbody>
</table>

$^\dagger$ under uniform hashing assumption
Separate chaining vs. linear probing

Separate chaining.
- Performance degrades gracefully as number of keys increases.
- Clustering less sensitive to poorly-designed hash function.
  - Potentially fewer probes.

Linear probing.
- Less wasted space.
- Better cache performance (locality).

```
keys[]   | P | M | A | C | S | H | L | E | R | X 
vals[]   | 10| 9 | 8 | 4 | 0 | 5 | 11| 12| 3 | 7
```
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).

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

Java system includes both.

Separate chaining vs. linear probing.