Searching

Figure 8.1. Perceptual change blindness in visual search, find five significant differences between these two images

Searching is the second fundamental operation we will study in this course. As with sorting, efficient searching is a critical foundation in computer science. We review $O(n)$ linear search and $O(\log n)$ binary search, then discuss more sophisticated approaches. Two of these techniques, trees and hashing, form the basis for searching very large data collections that must remain on disk.

8.1 Linear Search

The simplest search takes a collection of $n$ records and scans through them from start to end, looking for a record with a target key $k_t$. 
Chapter 8. Searching

Best case performance—when the target is the first record—is $O(1)$. Worst case performance—when the target is the last record or the target is not in the collection—is $O(n)$. On average, we assume we must search about $n/2$ records to find a target contained in the collection, which also runs in $O(n)$ time.

The main purposes of linear search are twofold. First, since it is very simple to implement, we sometimes use linear search when $n$ is small or when searching is rare. Second, linear search represents a hard upper bound on search performance. If a search algorithm requires $O(n)$ time (or more), we’d often be better off using a simple linear search.

8.2 Binary Search

If a collection is maintained in sorted order, we can perform a binary search.

```plaintext
binary_search(k, arr, l, r)
Input:  k, target key; arr, sorted array to search; l, left endpoint; r, right endpoint
n = r-l+1
if n ≤ 0 then
    return -1  // Searching empty range
end

c = 1 + ⌊n / 2⌋
if k == arr[c] then
    return c  // Target record found
else if k < arr[c] then
    return binary_search(k, arr, l, c-1)  // Search left half
else
    return binary_search(k, arr, c+1, r)  // Search right half
end
```

Calling `binary_search(k_t, arr, 0, n-1)` initiates a search. This compares the target key $k_t$ to the key at the center of the collection $k_c$. If $k_t = k_c$, the target record is found. Otherwise, sorted order tells us if $k_t < k_c$ than $k_t$ is left of the center record, otherwise $k_t > k_c$ and $k_t$ is right of the center record. Searching continues recursively until $k_t$ is found, or until the collection is exhausted.

Binary search discards half the collection ($n/2$ records) on its first comparison, then half the remaining collection ($n/4$ records) on its next comparison, and so on. Any operation that halves the size of the collection on each step runs in $O(\log n)$ time.

8.3 Binary Search Tree

If we choose to implement binary search, we must decide what type of data structure to use to manage the sorted collection. One possibility is a sorted array. As shown above, this provides $O(\log n)$ search performance. Unfortunately, maintaining the collection is not as fast. Inserting a new record requires $O(\log n)$ time to find its correct position, but then requires $O(n)$ time to shift part of the collection to make space to hold the new record. Deletion similarly requires $O(n)$ time to fill the hole left by the old record.
There is the also the practical issue of choosing a good initial array size, and the need to allocate more space if the array overflows.

A common alternative is a binary search tree, or BST. A BST is a tree structure made up of nodes, each of which hold a record and references to two (possibly empty) child subtrees (Fig. 8.2). The subtrees are normally labelled left and right. Each node in the BST satisfies the following ordering properties.

1. All records in a node’s left subtree have keys smaller than the node’s key.
2. All records in a node’s right subtree have keys larger than the node’s key.

Given this ordering, performing a binary search with a BST is very simple.

```plaintext
bst_search(k, node)
  Input:  k, target key; node, node to search
  if node == null
    return null // Searching empty tree
  end
  if k == node.key
    return node // Target record found
  else if k < node.key
    return bst_search( k, node.left ) // Search left subtree
  else
    return bst_search( k, node.right ) // Search right subtree
  end
```

The logic applied here is identical to binary search, since BSTs are designed specifically to support this search strategy.

**Insertion.** To insert a record with key \( k_t \) into a BST, we search for \( k_t \) in the tree. When we reach an empty subtree, we insert a new node containing \( k_t \)’s record. Since insertion requires a search followed by a constant time operation, insertion performance is identical to search performance.

**Deletion.** To delete a record with key \( k_i \) from a BST, we search for \( k_i \) in the tree. If a node containing \( k_i \) is found, we remove it and correct the BST based on three possible configuration cases.

1. If the node has no children, nothing needs to be done (Fig. 8.2a).
2. If the node has one subtree, promote its subtree’s root (Fig. 8.2b).
3. If the node has two subtrees (Fig. 8.2c)
   a) Find the successor to \( k_t \)—the smallest value greater than \( k_t \)—in the right subtree by walking right once, then walking left as far as possible.
   b) Remove the successor from the tree, since it has an empty left subtree it must match Case 1 or Case 2 above.
   c) Promote the successor to the node’s position.
Figure 8.2. Deletion from a BST: (a) deleting J, a node with no subtrees; (b) deleting D, a node with one subtree; (c) deleting M, a node with two subtrees

Again, since deletion requires a search followed by a constant time operation, deletion performance is identical to search performance.

**Performance.** Search performance in a BST depends on its shape. Suppose the BST is balanced: for any node in the tree, the height of its left and right subtrees are about equal. For example, the left BST in Fig. 8.2a is roughly balanced, since the difference in left and right subtree heights is no more than 1 throughout the tree. A balanced BST with $n$ records has a height of about $\lg n$, producing best case search performance of $O(\lg n)$ time.

A fully unbalanced BST is one in which every internal node has one subtree empty. Here, the BST degenerates into a linked list of $n$ nodes, producing worst case search performance of $O(n)$. Unfortunately, the common situation of inserting records with keys in sorted or nearly sorted order produces this worst case.

### 8.4 Splay Tree

One way to address the worst case $O(n)$ performance of BSTs is to ensure the tree never becomes unbalanced. Variations like AVL trees and red-black trees enforce guarantees on the difference in subtree heights by applying rotations during insertion and deletion to maintain balance. Both AVL and red-black trees improve worst case performance to $O(\lg n)$. 

52
Another type of self-adjusting tree is the splay tree, proposed by Sleator and Tarjan at AT&T Bell Labs\(^1\). A splay tree’s method of adjusting is simple, compared to the complicated rotation cases needed for AVL and red-black trees. One disadvantage of splay trees is that they only guarantee \(O(\lg n)\) amortized performance. The average cost of a sequence of searches is \(O(\lg n)\), but a single search may take up to \(O(n)\).

Implementing a splay tree requires adding a splay operation. to walk a node \(N\) to the top of the tree. Three different types of splaying can occur, based on the relative positions of \(N\), \(N\)'s parent \(P\), and \(N\)'s grandparent \(G\) (if it exists).

1. **Root.** If \(P\) is the root of the tree, rotate \(N\) to replace \(P\) (Fig. 8.3a).
2. **Inline.** If \(N\) is left of \(P\) and \(P\) is left of \(G\), or vice-versa, rotate \(P\) to replace \(G\), then rotate \(N\) to replace \(P\) (Fig. 8.3b).
3. **Angle.** If \(N\) is right of \(P\) and \(P\) is left of \(G\), or vice-versa, rotate \(N\) to replace \(P\), then to replace \(G\) (Fig. 8.3c).

Searching a splay tree is identical to searching a BST, with any record found being splayed to the top of the tree. To insert a record, we first apply a BST insertion to

---

position the record, then splay it to the top of the tree. To delete a record, we apply a
BST deletion, then splay the deleted record’s parent to the top of the tree, if it exists.

**Performance.** Although it’s possible for a splay tree to have a height of $n$ (e.g., after
inserting $n$ elements in increasing order), over time the tree will self-adjust to a height
of $\lg n$, producing amortized $O(\lg n)$ performance.

Another important aspect of a splay tree is that commonly queried records will
move near the root of the tree, meaning they can be found very quickly. If a collection
has a non-uniform pattern of access, splay trees can perform better in absolute terms
than other types of self-adjusting search trees.

### 8.5 k-d Tree

A k-dimensional or k-d tree is a binary tree used to subdivide a collection of records
into ranges for $k$ different attributes in each record. The k-d tree was proposed by
Jon Louis Bentley in 1975 to support associative, or multiattribute, searches\(^2\). For
example, we could take a collection of weather reports and divide them by properties
like latitude, longitude, temperature, or precipitation. We could then make queries
like: “Return all records with temperature $< 0^\circ$ C and precipitation $> 4$cm.”

k-d trees are often used as a method of flexible secondary indexing, although there
is no reason why primary keys cannot participate as one of the $k$ dimensions.

A k-d tree’s structure is similar to a BST, except at each level of the tree we rotate
between the $k$ dimensions used to subdivide the tree’s records. For example, a 2-d
tree using attributes temperature and pressure would subdivide based on temperature
at the root node, subdivide based on pressure in the root node’s children, based again
on temperature in the children’s children, and so on.

#### 8.5.1 k-d Tree Index

Like any binary tree, each k-d tree node contains a key value $k_c$ and two subtrees:
left and right. Unlike a BST, however, records are normally not stored in the internal
nodes. Instead, the target key $k_t$ is used to choose which subtree to enter: the left
subtree if $k_t \leq k_c$, or the right subtree if $k_t > k_c$. Leaf nodes contain collections
of records, specifically all records that satisfy the conditions along the root-to-leaf path.

Suppose we wanted to use information about Snow White and the seven dwarfs to
build a k-d tree index. We will use the attributes height ($ht$) and weight ($wt$) as the two
dimensions to subdivide records in the tree.

The construction algorithm works identical to BST, except that we rotate between
the $k = 2$ dimensions as we walk through each level of the tree.

1. Sleepy is inserted into the root of the tree, which uses $ht$ as its subdivision
   attribute.

\(^2\)Multidimensional binary search trees used for associative searching. Bentley. *Communications of the
ACM* 18, 9, 509–517, 1975.
### Table 8.1.
Estimated heights (in inches) and weights (in pounds) of Snow White and each of the seven dwarfs

<table>
<thead>
<tr>
<th>Name</th>
<th>ht</th>
<th>wt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sleepy</td>
<td>36</td>
<td>48</td>
</tr>
<tr>
<td>Happy</td>
<td>34</td>
<td>52</td>
</tr>
<tr>
<td>Doc</td>
<td>38</td>
<td>51</td>
</tr>
<tr>
<td>Dopey</td>
<td>37</td>
<td>54</td>
</tr>
<tr>
<td>Grumpy</td>
<td>32</td>
<td>55</td>
</tr>
<tr>
<td>Sneezy</td>
<td>35</td>
<td>46</td>
</tr>
<tr>
<td>Bashful</td>
<td>33</td>
<td>50</td>
</tr>
<tr>
<td>Ms. White</td>
<td>65</td>
<td>98</td>
</tr>
</tbody>
</table>

**Figure 8.4.** A k-d tree split by $ht$ and $wt$, indexed using Snow White and the seven dwarfs: (a) the first three insertions, with $ht$ subdividing the root and $wt$ subdividing the second level; (b) adding a $ht$ subdivision node on the third level; (c) the final tree, with Snow White and the dwarfs inserted into the appropriate buckets.
Chapter 8. Searching

2. Happy and Doc are inserted as children of Sleepy. Since Happy’s \( ht = 34 \leq 36 \), Happy goes to the left of the root. Doc’s \( ht = 38 > 36 \), so he goes to the right of the root (Fig. 8.4a). Both Happy and Doc use \( wt \) as their subdivision attribute.

3. Dopey is inserted next. His \( ht = 37 \) puts him to the right of the root, and his \( wt = 51 \) puts him to the left of his parent (Fig. 8.4b).

4. The remaining dwarfs and Snow White are inserted using an identical approach.

Once the k-d tree index is complete, it acts as a method to locate records based on their \( ht \) and \( wt \) attributes. Buckets are placed at each null subtree, ready to hold additional entries as they are inserted. Fig. 8.4d shows the buckets containing the initial dwarfs and Snow White.

**Interpretation.** A k-d tree index subdivides the \( k \)-dimensional space of all possible records into subspaces over a continuous range of values for each dimension. Another way to visualize a k-d tree index is as a subdivision of \( k \)-dimensional space using \((k-1)\)-dimensional cutting planes that represent each entry in the index.

The height–weight index in Fig. 8.4 can be visualized this way. Since the index uses \( k = 2 \) dimensions, we subdivide a 2D plane using 1D lines into regions that represent each bucket in the tree (Fig. 8.5).

### 8.5.2 Search

To search for records that match attribute ranges in a k-d tree, we perform the following operations.

1. Identify all paths whose internal nodes satisfy the target attribute ranges. This may produce multiple paths.

2. Perform an in-memory search of each path’s bucket for records that match the target criteria.

Figure 8.5. A subdivision of the \( k = 2 \) dimensional plane into subspaces representing each bucket in the k-d tree

8.5.2 Search

To search for records that match attribute ranges in a k-d tree, we perform the following operations.

1. Identify all paths whose internal nodes satisfy the target attribute ranges. This may produce multiple paths.

2. Perform an in-memory search of each path’s bucket for records that match the target criteria.
For example, suppose we search for records with $ht \leq 36$ and $wt \leq 47$.

- at the root, branch left ($ht \leq 36$),
- at the next node, branch left again ($wt \leq 49$),
- at the next node, branch left and right ($ht \leq 35$ and $ht > 35$ both fall within the target range of $ht \leq 36$),
- along the right path we reach bucket 3, and
- along the left path, branch left ($wt \leq 50$), reaching bucket 1.

The search produces two paths that identify buckets 1 and 3 as potentially containing target records. Examining either Fig. 8.4c or Fig. 8.5 shows that

- Bucket 1: $ht \leq 35$ and $wt \leq 50$
- Bucket 3: $35 < ht \leq 36$ and $wt \leq 52$

Both buckets may include records with $ht \leq 36$ and $wt \leq 47$. Moreover, no other buckets in the table could contain these types of records.

### 8.5.3 Performance

It should be clear that a k-d tree’s index has a critical impact on its performance. Ideally, the index should subdivide data stored in the tree in a balanced way, for example, by placing all the buckets at the same level in the tree, and by storing about the same number of elements in each bucket. If the data is known a-priori, median elements can be used to construct the index\(^3\).

Our k-d tree is an example of an index that is designed for a certain class of individuals: those with $ht \leq 37$ and $wt \leq 55$. If we try to store a large number of records outside this range, they will all be forced into only one or two different buckets.

For dynamic trees, maintaining balance in the index is more complicated. Here, adaptive k-d trees can be used to try to adjust the index when buckets become too full or out of balance. A simple, although potentially inefficient, suggestion is to take all the records in an out-of-balance area of the tree, then re-partition them and reconstruct the affected region of the index\(^4\).

### 8.6 Hashing

A second major class of algorithms used for efficient searching are hash algorithms. A hash function converts a key $k_i$ into a numeric value $h$ on a fixed range $0 \ldots n - 1$. $h$ is used as a location or an address for $k_i$ within a hash table $A$ of size $n$. This is analogous to indexing on an array, since we can store and retrieve $k_i$ at $A[h]$. If the hash function runs in constant time, search, insertion and deletion are $O(1)$ operations.

---


Chapter 8. Searching

Unfortunately, the number of possible records $m \gg n$ is normally much larger than the number of hash values $n$. Given this, three important properties distinguish hashing from using $h$ to directly index into $A$.

1. The hash value for $k_i$ should appear random.
2. Hash values should be distributed uniformly over the range $0 \ldots n - 1$.
3. Two different keys $k_i$ and $k_j$ can hash to the same $h$, producing a collision.

8.6.1 Collisions

Collisions are a major issue, particularly if each location in a hash table can only hold one record. If two records both hash to the same location, what should we do?

One answer might be, “Choose a hash function that doesn’t produce collisions.” This is harder than it sounds, however. Suppose we’re storing credit card information, and we decide to use the credit card number as a key. For card numbers of the form 0000 0000 0000 0000, there are $m = 10^{16}$ possible numbers (10 quadrillion).

Clearly, it’s not possible to create an in-memory array of size $n = 10^{16}$. Of course, every possible card number isn’t being used, in part because the credit card companies haven’t issued $10^{16}$ cards, and in part because different parts of a credit card number are dependent in various ways\(^5\) (e.g., certain parts of the card number represent check-sums to ensure the card is valid, other parts define card type, bank number, and so on). Card numbers do span a reasonable part of the range from around $1 \times 10^{15}$ to $9 \times 10^{15}$, however, so an array is still not feasible.

Even if the total number and range of the keys is small, it’s still difficult to define a perfect hashing function with no collisions. For example, if we wanted to store $m = 4000$ keys in an array of size $n = 5000$, it’s estimated that only 1 in $10^{12000}$ functions will be perfect. Given this, a more tractable approach is to reduce the number of collisions, and to determine how to handle collisions when they occur.

8.6.2 Hash Functions

Here is a common fold-and-add hash function.

1. Convert $k_i$ to a numeric sequence.
2. Fold and add the numbers, checking for overflow.
3. Divide the result by a prime number, and return the remainder as $h$.

Consider $k_i = \text{Subramanian}$. We convert this into a numeric sequence by mapping each character to its ASCII code, then binding pairs of ASCII codes.

<table>
<thead>
<tr>
<th>S</th>
<th>u</th>
<th>b</th>
<th>r</th>
<th>a</th>
<th>m</th>
<th>a</th>
<th>n</th>
<th>i</th>
<th>a</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>85</td>
<td>117</td>
<td>98</td>
<td>114</td>
<td>97</td>
<td>109</td>
<td>97</td>
<td>110</td>
<td>105</td>
<td>97</td>
<td>110</td>
</tr>
</tbody>
</table>

Assume the largest character pair is zz with combined ASCII codes of 122122. To manage overflow during addition, we divide by prime number 125299 slightly larger than this maximum after each add, and keep the remainder.

\(^5\)http://www.mint.com/blog/trends/credit-card-code-01202011
8.7 Hash Value Distributions

\[ 85117 + 98114 = 193231 \mod 125299 = 67932 \]
\[ 67932 + 97109 = 165041 \mod 125299 = 39742 \]
\[ 35742 + 97110 = 136852 \mod 125299 = 11553 \] (8.1)
\[ 11553 + 10597 = 22150 \mod 125299 = 22150 \]
\[ 22150 + 110 = 22260 \mod 125299 = 22260 \]

We divide the result of 22260 by the size of the hash table, which itself should be prime. Here, we assume \( A \) has size \( n = 101 \), produce a final \( h \) of

\[ h = 22260 \mod 101 = 40 \] (8.2)

Other useful hash functions exist. For example, we could convert \( k_t \) to a numeric sequence, square the sequence, and use the middle digits modulo the hash table size for \( h \). Or, we could convert the numeric sequence to a different base, and use the converted value modulo the hash table size for \( h \).

8.7 Hash Value Distributions

Given a hash table size of \( n \) used to hold \( r \) records, what is the likelihood that

1. No key hashes to a particular address in the table.
2. One key hashes to a particular address.
3. Two keys has to a particular address.

and so on? Assume our hash function uniformly distributes its hash values. For any single key the probability it hashes to a given address is \( b \), and the probability that it doesn’t hash to that address (i.e., it hashes to some other address) is \( a \).

\[ b = \frac{1}{n}, \quad a = 1 - \frac{1}{n} \] (8.3)

Given \( a \) and \( b \), suppose we insert two keys into the hash table. We can compute individual cases, for example, the probability that the first key “hits” an address and the second key “misses”, or the probability that both keys hit.

\[ ba = \frac{1}{n} \left(1 - \frac{1}{n}\right) = \frac{1}{n} - \frac{1}{n^2} \]
\[ bb = \frac{1}{n} \frac{1}{n} = \frac{1}{n^2} \] (8.4)

What is the probability that \( x \) of \( r \) keys hash to a common address? First, we need to determine how many ways there are to arrange \( x \) hits in a sequence of \( r \) keys. This is the binomial coefficient, or choose probability \( r \) choose \( x \).

\[ C = \binom{r}{x} = \frac{r!}{x!(r-x)!} \] (8.5)
Given \( C \), the probability of \( x \) hits in \( r \) keys at a common address is

\[
C \cdot b^x \cdot a^{r-x} = C \left( \frac{1}{n} \right)^x \left( 1 - \frac{1}{n} \right)^{r-x}
\] (8.6)

Because of the \( r! \) in its equation, \( C \) is expensive to compute. Fortunately, the Poisson distribution \( \Pr(x) \) does a good job of estimating our probability.

\[
C \cdot b^x \cdot a^{r-x} \approx \Pr(x) = \frac{(r/n)^x \cdot e^{-(r/n)}}{x!}
\] (8.7)

Since \( x \) is normally small, the \( x! \) in the denominator is not an issue. Consider an extreme case, where we want to store \( r = 1000 \) keys in a hash table of size \( n = 1000 \). Here, \( r/n = 1 \). We can use this ratio to calculate \( \Pr(0) \), the probability an address is empty, \( \Pr(1) \), the probability one key hashes to an address, \( \Pr(2) \), the probability two keys hash to an address, and so on.

\[
\Pr(0) = \frac{1^0 \cdot e^{-1}}{0!} = 0.368
\]

\[
\Pr(1) = \frac{1^1 \cdot e^{-1}}{1!} = 0.368
\]

\[
\Pr(2) = \frac{1^2 \cdot e^{-1}}{2!} = 0.184
\] (8.8)

Based on these probabilities, and given our hash table size of \( n = 1000 \), we expect about \( n \cdot \Pr(0) = 1000 \cdot 0.368 = 368 \) entries that are empty, \( n \cdot \Pr(1) = 368 \) entries holding 1 key, \( n \cdot \Pr(2) = 184 \) entries that try to hold 2 keys, and so on.

### 8.8 Estimating Collisions

Consider our previous example with \( r = n = 1000 \). How many collisions do we expect to see in this situation? To answer this, we use the following hash table breakdown.

<table>
<thead>
<tr>
<th>368 entries 0 keys</th>
<th>368 entries 1 key</th>
<th>264 entries &gt; 1 keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>368 recs inserted</td>
<td>1000 - 368 = 632 recs inserted</td>
<td>264 recs accepted</td>
</tr>
<tr>
<td>632 - 264 = 368 recs collide</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( n \cdot \Pr(0) = 368 \) entries in the table hold no keys, and \( n \cdot \Pr(1) = 368 \) entries hold exactly 1 key. This means \( 1000 - n \cdot \Pr(0) - n \cdot \Pr(1) = 264 \) entries try to hold more than
8.9. Managing Collisions

Table 8.2 shows that, even for very low packing densities, some collisions will still occur. Because of this, we need ways to manage a collision when it happens. We look at two common approaches: progressive overflow and multi-record buckets.

### 8.9.1 Progressive Overflow

One simple way to handle a collision on insertion is to hash a record’s key, and if the resulting address \( h \) is already occupied, to walk forward through the table until an empty position is found.
Chapter 8. Searching

To delete a record, we find and remove it. We also mark its position as dirty to remember that, although this position is empty, it was previously occupied.

```plaintext
progressive_insert(rec, tbl, n)
Input:  rec, record to insert; tbl, hash table; n, table size
num = 0  // Number of insertion attempts
h = hash( rec.key )
while num < n do
  if tbl[ h ] is empty then
    tbl[ h ] = rec  // Store record
    break
  else
    h = ( h + 1 ) % n  // Try next table position
    num++
  end
end
```

```plaintext
progressive_delete(key, tbl, dirty, n)
Input:  key, key to remove; tbl, hash table; dirty, dirty entry table; n, table size
h = progressive_search( key, tbl, dirty, n )
if h != false then
  tbl[ h ] = empty  // Set table position empty
  dirty[ h ] = true  // Mark table position dirty
end
```

```plaintext
progressive_search(key, tbl, dirty, n)
Input:  key, key to find; tbl, hash table; dirty, dirty entry table; n, table size
num = 0  // Number of compare attempts
h = hash( key )
while num < n do
  if key == tbl[ h ].key then
    return tbl[ h ]  // Target record found
  else if tbl[ h ] is empty and !dirty[ h ] then
    return false  // Search failed
  else
    h = ( h + 1 ) % n  // Try next table position
    num++
  end
end
return false  // Search failed
```

To search for a record, we hash its key to get \( h \), then search from position \( h \) forward. If we find the record, the search succeeds. If we search the entire table without finding the record, the search fails. If we find an empty position whose dirty bit isn’t set, the search also fails.

Why does the search stop at empty positions that aren’t dirty, but jump over empty positions that are dirty? Suppose we insert three records A, B, and C that all hash to
8.9. Managing Collisions

the same position \( h \). A and B form a run in the table, a block of records that C must step over to insert itself (Fig. 8.6a).

Next, we delete B, then search for C. The run that forced C to position \( h + 2 \) is gone (Fig. 8.6b). The search algorithm wants to follow C’s insertion path to find it. If we stopped at any empty entry, we would fail to find C. Marking position \( h + 1 \) as dirty tells the search algorithm, “Although this position is empty, it may have been part of a run when C was inserted, so keep searching.”

Progressive overflow is simple to understand and implement, but it has a number of serious disadvantages.

1. The hash table can become full, and if it does, it’s very expensive to increase. Since the hash function divides by the table size \( n \), increasing \( n \) changes every key’s hash value. The means we must remove and re-insert every record if we resize the table.

2. Runs form as records are inserted, increasing the distance a record needs to walk from its initial hash position \( h \) during insertion.

3. Runs can merge with one another, forming very long super-runs.

Experimental analysis shows that, because of long run lengths, a table > 75% full deteriorates to \( O(n) \) linear search performance. Since deletion leaves dirty locations that a search must past over, if a table is ever > 75% full, searches will run in \( O(n) \) time regardless of the number of records the table currently holds.

8.9.2 Multi-Record Buckets

Another way to reduce collisions is to store more than one record in each hash table entry. For example, each entry could be implemented as an expandable array or a linked list—a bucket—capable of holding \( b > 1 \) records. Insertion and deletion work identical to a simple hash table, except that we no longer need to worry about exceeding the capacity of a table position.
To search for key \( k \) with hash value \( h \), we load the entire bucket \( A[h] \) and scan it using linear search, binary search, or whatever strategy we’ve implemented to try to find a target record.

Do buckets really reduce collisions? That is, for a table that can hold a fixed number of records, does reorganizing it to use buckets reduce the collision rate, compared to a simple hash table that holds one record per table entry?

If we use buckets, the packing density of \( A \) is now \( r/bn \), where \( n \) is the table size and \( b \) is the maximum number of entries each table position can hold. Suppose we try to insert \( r = 700 \) records into a simple hash table with \( n = 1000 \) entries. Table 8.2 reports a collision rate of 28.1% for a packing density of \( r/n = 700/1000 = 70\% \). Suppose we instead built a hash table with \( n = 500 \) entries, each of which can hold \( b = 2 \) records. The packing density \( r/bn = 700/2 \cdot 500 = 0.7 \) is the same 70%. What is its expected collision rate?

Using the Poisson equation (Eq. 8.7), we can compute the expected number of table entries that hold 0 keys, 1 key, 2 keys, and so on. Recall that Poisson uses \( r/n \). For the simple hash table, \( r/n = 700/1000 = 0.7 \), and for the hash table with buckets \( r/bn = 700/2 \cdot 500 = 1.4 \).

\[
\begin{align*}
\text{Pr}(0) &= \frac{0.7^0 e^{-0.7}}{0!} = 0.497 \\
\text{Pr}(1) &= \frac{0.7^1 e^{-0.7}}{1!} = 0.348 \\
\text{Pr}(2) &= \frac{0.7^2 e^{-0.7}}{2!} \approx 0.242
\end{align*}
\]

The equations and figure on the right represent the table with \( n = 500 \) buckets of size \( b = 2 \). The table has \( n \cdot \text{Pr}(0) = 124 \) entries that hold no keys, \( n \cdot \text{Pr}(1) = 172 \) entries that hold 1 key, and \( n \cdot \text{Pr}(2) = 121 \) entries that hold 2 keys. \( 500 - n \cdot \text{Pr}(0) - n \cdot \text{Pr}(1) - n \cdot \text{Pr}(2) = 83 \) entries try to hold more than 2 keys. \( 700 - 172 - (2 \cdot 121) = 286 \) keys hash to these positions, of which 166 are stored and 286 – 166 = 120 collide, for a collision rate of 17.1%.

So, by simply rearranging 1000 table entries into a two-bucket table, we can reduce the collision rate from 28.1% to 17.1%, or from 197 collisions to 120 collisions.
Using multi-record buckets still poses problems for efficiency. In particular, as $r >> n$ records are added to the table, the length of each bucket will become long, increasing the time needed for search, insertion—to check a bucket for duplicate keys—and deletion—to find the record to remove. We might be tempted to increase the size $n$ of the hash table, but this has the same problem that we saw with progressive overflow: changing $n$ changes the hash function, forcing us to remove and re-insert the table’s records if we resize it.