Searching

Organizing and retrieving information is at the heart of most computer applications

- Searching is a very frequently performed task
- Search process:
  - Abstract view: Determine if an element with a particular value is a member of a particular set
  - Common view: Try to find the record with a record collection that has a particular key value.
- Some of the techniques presented here require material from chapter 8.
  - Assigned reading: Chapter 8, section 8.3 and all of Chapter 9

Buffers and Buffer Pools (sec 8.3)

The general idea is to use a RAM buffer to hide latency.

- Caching or buffering: the act of storing in RAM a piece of data from a faster or slower device
  - allows the faster device to do something else while the slower device reads from or writes to the buffer
- Examples
  - CPU cache is a buffer for RAM
  - RAM is a buffer for disks of various types
  - Disk can buffer for tape
- Associated concepts:
  - Buffer pool: a set of multiple buffers
  - Page: a piece of memory large enough to fill a buffer

Buffer Pools

An example is virtual memory.

- A hard disk is used to simulate a very large RAM memory
- System RAM is the buffer pool
- A page is a block of memory (usually some multiple of 512 bytes)
  - Address space of a process can be broken into multiple pages
  - At any given time, some pages may be on disk and some in RAM
    - Requesting a memory address currently on disk causes a page fault
  - Two options for page fault:
    - Find an "empty" page in RAM and transfer the page from disk
    - No empty pages in RAM: follow a page replacement strategy
- Page replacement strategies:
  - FIFO
  - LFU
  - LRU

Searching

- Formal definition:
  - Suppose $k_1, k_2, \ldots, k_n$ are distinct keys
  - Given a collection $C$ of $n$ records of the form
    $$(k_1, I_1), (k_2, I_2), \ldots, (k_n, I_n)$$
    - $I_j$ is information associated with key $k_j$ for $1 \leq j \leq n$.
  - Search problem: given key value $K$, locate the record $(k_j, I_j)$ in $C$ such that $k_j = K$
    - successful search: record with $k_j = K$ is found
    - unsuccessful search: no record with $k_j = K$ is found
- Queries:
  - Exact-match query: search for a record whose key matches a specific key value
  - Range query: search for all records whose key values fall within a specified range
Searching Categorization

- Three general approaches
  - Sequential and list methods
    - Works well for sequences (duplicate keys allowed)
    - Appropriate for data stored in RAM
  - Direct access by key value (hashing)
    - Doesn’t work well for sequences
    - Works well for data on disk or in RAM
  - Tree indexing methods (chapter 10, not covered)

Searching Sorted Arrays

- Sequential search
  - $\Theta(n)$, average and worst case
  - Unacceptable for large data sets

- Binary search
  - $\Theta(\log n)$, average and worst case
  - Works only for previously sorted data

- Dictionary search
  - A “computed” binary search
  - Based on knowledge about key distribution
  - Also called interpolation search

Lists Ordered by Frequency

Instead of ordering by key value, a list may be ordered by frequency of access.

- Lists ordered by frequency: the expected frequency of occurrence determines ordering strategy
  - A sequential search is performed
    - Cost to access $i^{th}$ record is $i$
    - Order in decreasing order of probability: $p_i$ is the probability that record $i$ will be accessed
    - That is,
      $$p_1 \geq p_2 \geq \cdots \geq p_n$$
      (Note: $\sum_{i=1}^{n} p_i = 1$ must be true)
    - The cost to access each element is (position of element) $\times$ (probability of element)
    - Then the overall expected search cost is
      $$\bar{C}_n = 1p_1 + 2p_2 + \cdots + np_n$$

Lists Ordered by Frequency (cont.)

- Example: all records have equal probability
  - $p_i = \frac{1}{n}$
  - Then
    $$\bar{C}_n = 1 \times \frac{1}{n} + 2 \times \frac{1}{n} + \cdots + n \times \frac{1}{n}$$
    $$= \sum_{i=1}^{n} i/n = \frac{1}{n} \sum_{i=1}^{n} i$$
    $$= \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- Example: exponential frequency
  - Probabilities:
    $$p_i = \begin{cases} 
    1/2 & \text{if } 1 \leq i \leq n - 1 \\
    1/2^{n-1} & \text{if } i = n 
    \end{cases}$$
  - Thus,
    $$\bar{C}_n \approx \sum_{i=1}^{n} \frac{i}{2} \approx 2$$
The 80/20 Rule

Many real access patterns follow this rule of thumb:

- The 80/20 rule: 80
  - 80 and 20 are estimates (applications have their own values)
  - This behavior justifies caching techniques
  - When the rule applies, then reasonable search performance can be expected

- Example: Zipf distribution
  - A pattern followed by some naturally occurring distributions, including:
    - Distribution for frequency of word usage
    - Distribution for city populations
  - Related to the Harmonic series (chapter 2) as follows:
    - Zipf frequency for item \( i \) is \( 1/iH_n \)
    - (Here \( H_n = \sum_{i=1}^{n} 1/i \approx \log_2 n \))
    - Then
    \[
    C_n = \sum_{i=1}^{n} i/iH_n \\
    = n/H_n \\
    \approx n/\log_2 n
    \]

Self-Organizing Lists

This is why we studied the section on buffer pools.

- A self organizing list is a list that starts out unordered, but the access policy includes procedures to impose an order based on actual pattern of record access
  - Use rules called heuristics to determine how to reorder the list
  - The heuristics are similar to the buffer pool management strategies (buffer pools are like a form of self-organizing list)

- Heuristics:
  - Count: Count the frequency of access. When a record is found, increment its count and move it up if the count is greater than preceding record(s)
  - Move-to-front: when a record is found, move it to the front of the list
  - Transpose: when a record is found, swap it with the record ahead of it

CSC 375-Turner, Page 9

Self-Organizing Lists, Examples

- Initial list is A, B, C, D, E, F, G, H
- Access pattern is F D F G E G F A D F G E
  - Count heuristic:

  - Move-to-front heuristic:

  - Transpose heuristic:

CSC 375-Turner, Page 10

Self-Organizing Lists, Examples

- Application: text compression
  - Keep a table of words previously seen
  - Use the move-to-front heuristic
  - If a word is not yet seen, then send the word
  - If a word has been seen, then send its current table index

  - Example: The car on the left hit the car I left

  - becomes: The car on 3 left hit 3 5 1 5

  - Similar in spirit to Ziv-Lempel coding

CSC 375-Turner, Page 11

CSC 375-Turner, Page 12
**Searching in Sets**

Determining whether a value is a member of a set is a special case of searching for keys in a sequence of records.

- Any of the prior search methods can be used
- This problem allows us to speed up the process:
  - **Bit vector or bitmap** representation: use an array of \( n \) bits corresponding to \( n \) potential set members
    - \( i = 1 \) means that member \( i \) is present
    - \( i = 0 \) means that member \( i \) is not present
  - Application: document retrieval: find all documents in a set containing certain keywords
    - For each keyword, the system stores a bit vector (one bit for each document)
    - A '1' means that the document contains the keyword
    - Searching for three words is a logical AND of 3 bit vectors.

**Hashing**

A completely different approach in which search is by direct access based on the key value.

- **Hashing** is the process of accessing a record by mapping a key value to a position in a table.
- The mapping process requires a (normally \( \Theta(1) \)) mathematical function called the hash function, denoted by \( h \)
- The **Hash table** is an array that stores all of the records, denoted \( HT \)
- A record's position in the hash table is its slot
- The number of slots is denoted by \( M \), numbering is from 0 to \( M - 1 \)
- The mapping function \( h \) must work as follows:
  - For any value \( K \) in the key range, \( h(K) = i \), \( 0 \leq i < M \) such that \( key(HT(i)) = K \)

**Hashing (cont.)**

Hashing answers the specific question “what record, if any, has key value \( K \)?”

- Works well for sets (no duplicates)
- Not suitable for range queries
- Works well for in-memory and disk-based applications

**Example:**
- Store the \( n \) records with key values in the range 0 to \( n - 1 \)
- Hash function \( h(K) = K \)
- This is not a practical example (Why?)

**Example:**
- Store about 1000 records having keys in the range 0 to 16,383
- Impractical to keep a hash table with 16,383 slots
- We need a hash function that maps the key range to a smaller table

**Collisions**

- Given a hash function \( h(k) \) and keys \( k_1 \) and \( k_2 \):
  - If \( h(k_1) = h(k_2) = \beta \), then \( k_1 \) and \( k_2 \) have a collision at \( \beta \) under \( h \).
- Collisions are inevitable in most applications
  - Example: birthday sharing

- Minimizing collisions requires good hash functions
- Finding a record (or a place in which to insert) requires a two-step procedure:
  1. Compute table location \( h(k) \)
  2. Starting with slot \( h(k) \), search for the record containing key \( k \) (or an empty location where it may be inserted)
- The search procedure is the collision resolution technique. There are two major classes:
  - **Open hashing**, also called Separate chaining
  - **Closed hashing**, also called Open addressing
Hash Functions

- Requirement:
  - □ A hash function must compute a slot index within the hash table’s range; thus, it computes \((\text{some value} \mod M)\)

- Goals:
  - □ A practical hash function evenly distributes the records stored among the hash table slots
  - □ Ideally, the even distribution is to all slots with equal probability
    - ○ Success at this depends on the data’s distribution
  - □ It should also be fast (probably the easiest goal to accomplish)

- Two situations normally faced:
  - □ We know nothing about the incoming key distribution: attempt to evenly distribute the key range over the hash table, trying to avoid clustering
  - □ We know something about the incoming key distribution: use a distribution-dependent hash function.

Examples

- A Simple hash function:
  ```c
  int h (int x) {
    return (x % 16);
  }
  ```
  - □ The \mod 16 operation makes the function dependent on the lower 4 bits of the key

- Mid-square method: square the key value, taking the middle \(r\) bits from the result for a hash table having \(2^r\) slots

- Folding method: sum the ASCII values of all letters, taking the result \mod M:
  ```c
  int h(char *x) {
    int i = 0; int sum = 0;
    while (x[i] != NULL) {
      sum += (int) x[i];
      i++;
    }
    return (sum % M);
  }
  ```

Open Hashing

This is also called separate chaining.

- A collision resolution technique, in which:
  - □ The hash table is not an array of records; rather, it is an array of pointers
  - □ Each slot is treated as a bin so that collisions do not really occur
  - □ For a given record with key \(k\) and \(h(k) = \beta\):
    - ○ hash table slot \(\beta\) is the head of a linked list
    - ○ Insert into slot \(\beta\) becomes a linked list insert

CSC 375-Turner, Page 17

CSC 375-Turner, Page 18

CSC 375-Turner, Page 19

CSC 375-Turner, Page 20
**Closed Hashing**

This is also called **open addressing**

- All records are stored directly in the hash table
  - Each record $i$ has a **home position** defined by $h(k_i)$
    - If record $i$ is inserted and another record already occupies $i$'s home position, then another slot must be found to store $i$.
    - The search procedure to find a new slot is the **collision resolution policy**

---

**Bucket Hashing**

One implementation of closed hashing in which the extra list space is stored in the table.

- Divide the hash table into buckets
  - $M$ slots are divided into $B$ buckets, with $M \gg B$
  - Include overflow bucket with large capacity at end
  - Records hash to the first slot of the bucket, then fill it sequentially
  - Overflow is used if a given bucket is full
  - Search: check bucket then check overflow (using linear search in both)

---

**Collision Resolution Policies**

- **Goal**: find a free slot in the table

- **Search proceeds by following a probe sequence**: the series of slots visited during insert/search after a collision occurs
  - Whether inserting or searching, the probe sequence must be the same every time
  - Basic idea: follow probe sequence until one of the following is true:
    - record with key = $k$ is found
    - an empty slot is found (no record with key $k$ exists in the hash table)

- **Insert with Probing**:
  ```c
  void insert(item R) {
    int home, pos, i;
    home = h(key(R));
    if (Table[home] == EMPTY)
      Table[home] = R;
    else {
      for (i = 1; Table[pos] != EMPTY; i++) {
        pos = (home + probe(key(R), i)) % M;
        if (key(T[pos]) == key(R)) ERROR;
      }
      Table[pos] = R;
    }
  }
  ```

---

**Linear Probing**

From a given position, linear probing searches the next available slot in the table.

- **Probe function**:
  ```c
  int probe(int Key, int i) { return i; }
  ```

- If the end of the table is reached, it wraps around to the top (see code on previous page)
- At least one slot must always be empty in the table. Why?

- **Linear probing suffers from primary clustering**:
  - “Clusters” of occupied cells form
  - Any key hashing into a cluster requires several attempts to resolve the collision and then will add to the cluster
Primary Clustering

- Probabilities for which slot to use next are not the same
  - $h(k) = k \mod 11$
  - $1003 \mod 11 = 2$, $1924 \mod 11 = 10$, $3071 \mod 11 = 2$, $2071 \mod 11 = 3$, $4752 \mod 11 = 0$

Insert in the following order: 1003, 1924, 3071, 2071, 4752

CSC 375-Turner, Page 25

Better Linear Probing

- Use a constant $c$ to skip by, instead of going to the next slot on every probe
  - $\text{probe}(h(k), i) = h(k) + c \times i$
  - $M$ and $c$ should be relatively prime (Why?)

- Clustering can still exist
  - Example: $c = 3$, $h(k_1) = 3$, $h(k_2) = 9$
  - Probe sequences for $k_1$ and $k_2$ are linked together

CSC 375-Turner, Page 26

Pseudo Random Probing

An ideal probe function selects the next slot in the probe sequence at random

- Why can a real probe function not act randomly?

- Pseudo random probing:
  - Select a random permutation of the numbers from 1 to $M - 1$: $r_1, r_2, \ldots, r_{M-1}$
  - All searches and insertions use the same permutation:
    - $p(K, i) = \text{Perm}[i - 1]$
    - that is, the $i^{th}$ value in the probe sequence is $(h(k) + r_i) \mod M$

- Example:
  - $M = 101$
  - $r_1 = 2, r_2 = 5, r_3 = 32$
  - $h(k_1) = 30, h(k_2) = 28$
  - Probe sequence for $k_1$:
  - Probe sequence for $k_2$:

CSC 375-Turner, Page 27

Quadratic Probing

- The $i^{th}$ probe sequence function is $i^2$
- That is, the $i$th value in the probe sequence is $(h(k) + i^2) \mod M$

- Example:
  - $M = 101$
  - $h(k_1) = 23, h(k_2) = 24$
  - Probe sequence for $k_1$:
  - Probe sequence for $k_2$:

CSC 375-Turner, Page 28
Double Hashing

Prior probing methods can reduce or eliminate primary clustering.

- **Secondary clustering** occurs when two keys hash to the same slot, thus following the exact same probe sequence
- Desirable: the probe sequence is a function of both the key and the home position
- Double hashing adds a second hash function to the probe sequence:
  - $p(k, i) = i \times h_2(k)$ for $0 \leq i \leq M - 1$
  - Poor choice of $h_2(k)$ results in poor ("disastrous") performance
  - Make sure all cells can be probed by ensuring that all probe sequence constants are relatively prime to $M$
    - One method: make $M$ prime
    - Another method: set $M = 2^m$ and make $h_2$ return an odd value between 1 and $2^m$

Analysis of Closed Hashing

- Visualizing the expected performance of hashing based on load factor
- Load factor $\alpha = N/M$ where $N$ is the number of records stored

Rehashing

- Consequences of a hash table that is too full:
  - Running time for operations start to take too long
  - Insertions might fail for certain collision resolution strategies
- Solution: build a bigger table
  - Find a prime number at least twice as large as current value of $M$
  - Allocate a new hash table (array)
  - Scan through the old hash table, inserting all elements into the new hash table
  - Delete the old hash table
- Operation is expensive but occurs relatively infrequently
- Strategies:
  - Rehash when the table is half full
  - Rehash when an insertion fails
  - Rehash when the table reaches a certain load factor

Deletion

Deletion is tricky with hashing for the following reasons:

- Deleting a record must not hinder later searches (an empty slot means "stop the search")
- Positions should also not be made unusable due to deletions (avoid a "zombie slot")
- Solution:
  - Add a special mark in place of the deleted record
  - Mark is called the **tombstone**
  - Tombstones do not stop search but do add to average search time
  - Solutions to that added time:
    - Local reorganizations to try to shorten it
    - Periodically rehash the table