Announcements

- Homework 4 up on website
- Hw3 grades coming tomorrow
- Thanks for feedback on midterm evaluations
- I have old hw’s and midterms in my office; stop by after class or let’s set up a time for pickup
Review

- Midterm Solutions
- Huffman Coding Trees
  - Create forest of all characters weighted by frequency
  - Merge least weight trees until 1 tree left
- (Unfinished) proof sketch of optimality
Today’s Plan

- Finish Huffman optimality proof sketch
- Hash Tables ADT
  - Definition and Implementation
Huffman Details

- We can manage the forest with a priority queue:
  - buildHeap first,
  - find the least weight trees with 2 deleteMins,
  - after merging, insert back to heap.

- In practice, also have to store coding tree, but the payoff comes when we compress larger strings.
Induction: Suppose Huffman tree is optimal for \( N \) characters. What about \( N+1 \) characters?

Lemma 1: Optimal tree is full

Lemma 2: the 2 least frequent characters are at the deepest level in optimal tree

Lemma 3: Swapping characters at same depth doesn’t affect optimality
Optimality of Huffman

- Induction: Suppose Huffman tree is optimal for $N$ characters. What about $N+1$ characters?

- Lemma 1: Optimal tree is full

- Lemma 2: the 2 least frequent characters are at the deepest level in optimal tree

- Lemma 3: Swapping characters at same depth doesn’t affect optimality

- Lemma 4: An optimal tree exists where the least frequent characters are siblings at deepest level.
Optimality of Huffman

- number of bits of an encoding is \( B(T) = \sum_{i=1}^{N+1} F_i D_i \)
- \( F \) is the frequency of the character, \( D \) is the depth in the tree (the number of bits)
- Create new tree \( T^* \) by removing least frequent chars and replacing with a meta-character whose frequency is the frequency of both chars,
- meta-character is one level less deep
Optimality of Huffman

\[ B(T) = B(T^*) + F_1 + F_2 \]

Proof by contradiction: Assume there is a different tree \( T' \) that is better than \( T \)

\[
\begin{align*}
B(T') &< B(T) \\
B(T'^*) + F_1 + F_2 &< B(T^*) + F_1 + F_2 \\
B(T'^*) &< B(T^*)
\end{align*}
\]

That is a contradiction because \( T^* \) has \( N \) characters, which means Huffman is optimal via our inductive hypothesis.
Optimality of Huffman

- Assuming falseness of inductive step produced contradiction to inductive hypothesis

- Therefore, if Huffman codes are optimal for $N$ characters, they are also for $N+1$ characters

- Huffman is obviously optimal for 2 characters

- Huffman codes are optimal
Hash Table ADT

- **Search tree:**
  findMin, findMax, insert/delete, search

- **Priority Queue:**
  findMin (or max), insert/delete, no search

- **Hash Table:**
  insert/delete, search
Hash Table ADT

- **Search tree:**
  Stores complete order information

- **Priority Queue:**
  Stores incomplete order information

- **Hash Table:**
  Stores no order information
Hash Table ADT

- Insert or delete objects by **key**
- Search for objects by **key**
- **No** order information whatsoever
- Ideally $O(1)$ per operation
Implementation

- Suppose we have keys between 1 and K
- Create an array with maxKey entries
- Insert, delete, search are just array operations

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>K-3</th>
<th>K-2</th>
<th>K-1</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Obviously too expensive
Hash Functions

- A **hash function** maps any key to a valid array position
- Array positions range from 0 to N-1
- Key range possibly unlimited
Hash Functions

- For integer keys, \((\text{key mod N})\) is the simplest hash function

- In general, any function that maps from the space of keys to the space of array indices is valid

- but a good hash function spreads the data out evenly in the array;

- A good hash function avoids collisions
Collisions

- A **collision** is when two distinct keys map to the same array index

  - e.g., $h(x) = x \mod 5$
    - $h(7) = 2$, $h(12) = 2$

- Choose $h(x)$ to minimize collisions, but collisions are inevitable

- To implement a hash table, we must decide on collision resolution policy
Collision Resolution

Two basic strategies

- Strategy 1: Separate Chaining
- Strategy 2: Probing; lots of variants
Strategy 1: Separate Chaining

- Keep a list at each array entry
  - Insert(x): find h(x), add to list at h(x)
  - Delete(x): find h(x), search list at h(x) for x, delete
  - Search(x): find h(x), search list at h(x)

- We could use a BST or other ADT, but if h(x) is a good hash function, it won’t be worth the overhead
Separate Chaining
Average Case

- Load Factor $\lambda = \# \text{ objects} / \text{TableSize}$
- Average list length is $\lambda$
- Time to insert = constant, or constant + $\lambda$
- Time to search = constant + $\lambda$ or constant + $\lambda/2$
Strategy 1: Advantages and Disadvantages

- **Advantages:**
  - Simple idea
  - Removals are clean *

- **Disadvantages:**
  - Need 2\textsuperscript{nd} data structure, which causes extra overhead if the hash function is good
Strategy 2: Probing

- If \( h(x) \) is occupied, try \( h(x)+f(i) \mod N \) for \( i = 1 \) until an empty slot is found
- Many ways to choose a good \( f(i) \)
- Simplest method: Linear Probing
  - \( f(i) = i \)
Linear Probing Example

- $N = 5$
- $h(x) = x \mod 5$
- Insert 7
- Insert 12
- Insert 2
Primary Clustering

- If there are many collisions, blocks of occupied cells form: primary clustering
- Any hash value inside the cluster adds to the end of that cluster
- (a) it becomes more likely that the next hash value will collide with the cluster, and (b) collisions in the cluster get more expensive
Removals

• How do we delete when probing?

• Lazy-deletion: mark as deleted,
  • we can overwrite it if inserting,
  • but we know to keep looking if searching.
Quadratic Probing

- $f(i) = i^2$
- Avoids primary clustering
- Sometimes will never find an empty slot even if table isn’t full!
- Luckily, if load factor $\lambda \leq \frac{1}{2}$, guaranteed to find empty slot
Quadratic Probing Example

- $N = 7$
- $h(x) = x \mod 7$
- Insert 9
- Insert 16
- Insert 2
Double Hashing

- If $h_1(x)$ is occupied, probe according to
  $$f(i) = i \times h_2(x)$$

- 2\textsuperscript{nd} hash function must never map to 0

- Increments differently depending on the key
### Double Hashing Example

- **N = 7**
- **h1(x) = x mod 7,  h2(x) = 5-x mod 5**
- **Insert 9**
- **Insert 16**
- **Insert 2**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reading

- Homework 4
- Weiss Ch. 5