Notes on Problem Set 4

- Elementary Sorts

- Challenge:
  - Implement Shaker sort function

- Compare shell sort sequences
Question 1

Compare
  • Insertion sort
  • Selection sort
  • Bubble sort

On reverse ordered data
  • Selection is fastest
  • Followed by insertion (why not bubble sort?)
  • All do the same number of comparisons
void shaker_sort(Item * a, int start, int stop)
{
    int i;
    while (stop > start)
    {
        for (i = stop; i > start; i--)
            if (isLess(a[i], a[i - 1]))
                Exchange(a[i], a[i - 1]);
        start++;

        for (i = start; i < stop; i++)
            if (isLess(a[i + 1], a[i]))
                Exchange(a[i + 1], a[i]);
        stop--;
    }
}
void shaker_sort(Item * a, int start, int stop)
{
    int i, new_start, new_stop;
    while (stop > start)
    {
        for (new_start = i = stop; i > start; i--)
            if (isLess(a[i], a[i - 1]))
                { Exchange(a[i], a[i - 1]); new_start = i; }
        start = new_start;

        for (new_stop = i = start; i < stop; i++)
            if (isLess(a[i + 1], a[i]))
                { Exchange(a[i + 1], a[i]); new_stop = i; }
        stop = new_stop;
    }
}
Question 3

- Comparison of Shellsort sequences
  - Random data
    - Longer sequence about 1.5x faster
  - Ordered data
    - Shorter sequence about 3x faster
  - Reverse ordered data
    - Very similar performance
Problem Set 5

- Divide-and-Conquer Sorts
- Track stack usage
- Optimize function
  - Decide when to switch to brute-force methods
Question 1

- Stack usage should be about:
  - ~22 for basic quick sort
  - ~17 with median of three partitioning
  - ~8 with explicit stack

- Basic implementation and non-recursive version with explicit stack do the same comparisons, but in different orders
A simple stack counter...

```c
void quick_sort(Item * a, int start, int stop)
{
    if (stop <= start)
        return;

    if (max_depth < ++depth) max_depth = depth;
    int i = partition(a, start, stop);
    quick_sort(a, start, i - 1);
    quick_sort(a, i + 1, stop);
    depth--;
}
```
Question 2: Quick Sort Optimization ...
Question 2: Merge-Sort Optimization
Last Lecture ...

- Review Class

- Reminder:
  Exam on Wednesday, November 3

- Problem set 5:
  - Can collect from my office tomorrow.
Today

- Hashing
  - Uses extra memory to speed up searches.
  - Allocates memory for more than N elements.
  - Chaining
  - Linear Probing
  - Double Hashing

- Random number generation
Components of Hashing

- **Hash Function**
  - Generates table address for individual key

- **Collision-Resolution Strategy**
  - Deals with keys for which the Hash Function generates identical addresses
Desirable Hash Functions

Poor Hash Function:
Data is clustered

Good Hash Functions:
Data is Evenly Distributed
Conflict Resolution: Separate Chaining

- What to do with items where the hash function returns the same value?
- One option is to make each entry in the hash table an array or list …
  - Each entry corresponds to a "chain of items"
## Separate Chaining Example

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
</tr>
<tr>
<td>R</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>M</td>
<td>4</td>
</tr>
<tr>
<td>P</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>L</td>
<td>3</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>S</td>
<td>3</td>
</tr>
</tbody>
</table>

Chains:
- Chain 0: E → A
- Chain 1: G
- Chain 2: X → N → I → S
- Chain 3: L → P
- Chain 4: M → H → C → R
If we can guarantee that $M > N$

- In this case, $\alpha < 1$

Whenever there is a collision, search sequentially for the next empty slot
Linear Probing Example

<table>
<thead>
<tr>
<th>Item</th>
<th>Hash1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table after inserting element 1
Table after inserting element 2

Table after inserting all elements
Table index
## Number of Comparisons

<table>
<thead>
<tr>
<th>load factor ($\alpha$)</th>
<th>1/2</th>
<th>2/3</th>
<th>3/4</th>
<th>9/10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search Hit</strong></td>
<td>1.5</td>
<td>2.0</td>
<td>3.0</td>
<td>5.5</td>
</tr>
<tr>
<td><strong>Search Miss</strong></td>
<td>2.5</td>
<td>5.0</td>
<td>8.5</td>
<td>50.5</td>
</tr>
</tbody>
</table>

\[
Cost(Hit) = \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \quad \text{Cost(Miss)} = \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right)
\]
Conflict Resolution 2: Double Hashing

- Similar to linear hashing

- Guards against clustering by using hash function to generate increment for sequential searches
Double Hashing Example

<table>
<thead>
<tr>
<th>Item</th>
<th>Hash1</th>
<th>Hash2</th>
</tr>
</thead>
</table>

| Table after inserting element 1 | Table after inserting element 2 |

| Table after inserting all elements | Table index |
/* Searching for an item */
h = hash(item, M);
h2 = hash2(item, another_prime) + 1;
while (table[h] != item && table[h] != EMPTY)
    h = (h + h2) % M;
/* Search successful if table[h] != EMPTY */
/* Otherwise, item could be inserted at table[h] */
![Number of Comparisons](image)

<table>
<thead>
<tr>
<th>load factor ($\alpha$)</th>
<th>1/2</th>
<th>2/3</th>
<th>3/4</th>
<th>9/10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Hit</td>
<td>1.4</td>
<td>1.6</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td>Search Miss</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
<td>10</td>
</tr>
</tbody>
</table>

\[
\text{Cost(Hit)} = \frac{1}{\alpha} \ln \frac{1}{1 - \alpha} \quad \text{Cost(Miss)} = \frac{1}{1 - \alpha}
\]
Analysis of Double Hashing

- Performance similar to random hashing
  - Unique sequence of keys for each item

- Number of probes for a miss would be...

\[
1 + \frac{N}{M} + \left( \frac{N}{M} \right)^2 + \left( \frac{N}{M} \right)^3 \ldots = \frac{1}{1 - N/M} = \frac{1}{1 - \alpha}
\]
Analysis of Double Hashing

- Number of probes for a hit
  - The same as the cost of originally inserting the item
  - With N items, assume that each one is target with probability 1/N

\[
\frac{1}{N} \left( 1 + \frac{1}{1 - 1/M} + \frac{1}{1 - 2/M} + \frac{1}{1 - 3/M} + \ldots \right) = \\
\frac{1}{N} \left( 1 + \frac{M}{M - 1} + \frac{M}{M - 2} + \frac{M}{M - 3} + \ldots \right)
\]
Further Notes on Hashing

To ensure that search requires less than $t$ comparisons on average:

- $\alpha < (1 - 1/t)$ with linear hashing
- $\alpha < (1 - 1/sqrt(t))$ with double hashing

Dynamic hashing:

- Increase table size and rehash elements whenever $\alpha$ exceeds threshold (typically 0.5)
Cost Comparison

Cost of Searches with Double Hashing

Cost of Searches with Linear Probing
Summary

- Hashing
  - Chaining
  - Linear Probing
  - Double Hashing

- Cost of searches is nearly independent of N
  - Fast searches that don't require sorting
  - Not very effective for rank-based selection
Recommended Reading

- In Sedgewick’s “Algorithms in C”
  - Chapter 14
Random Numbers

- Time permitting, ...
- … discussion about random number generation will go here.