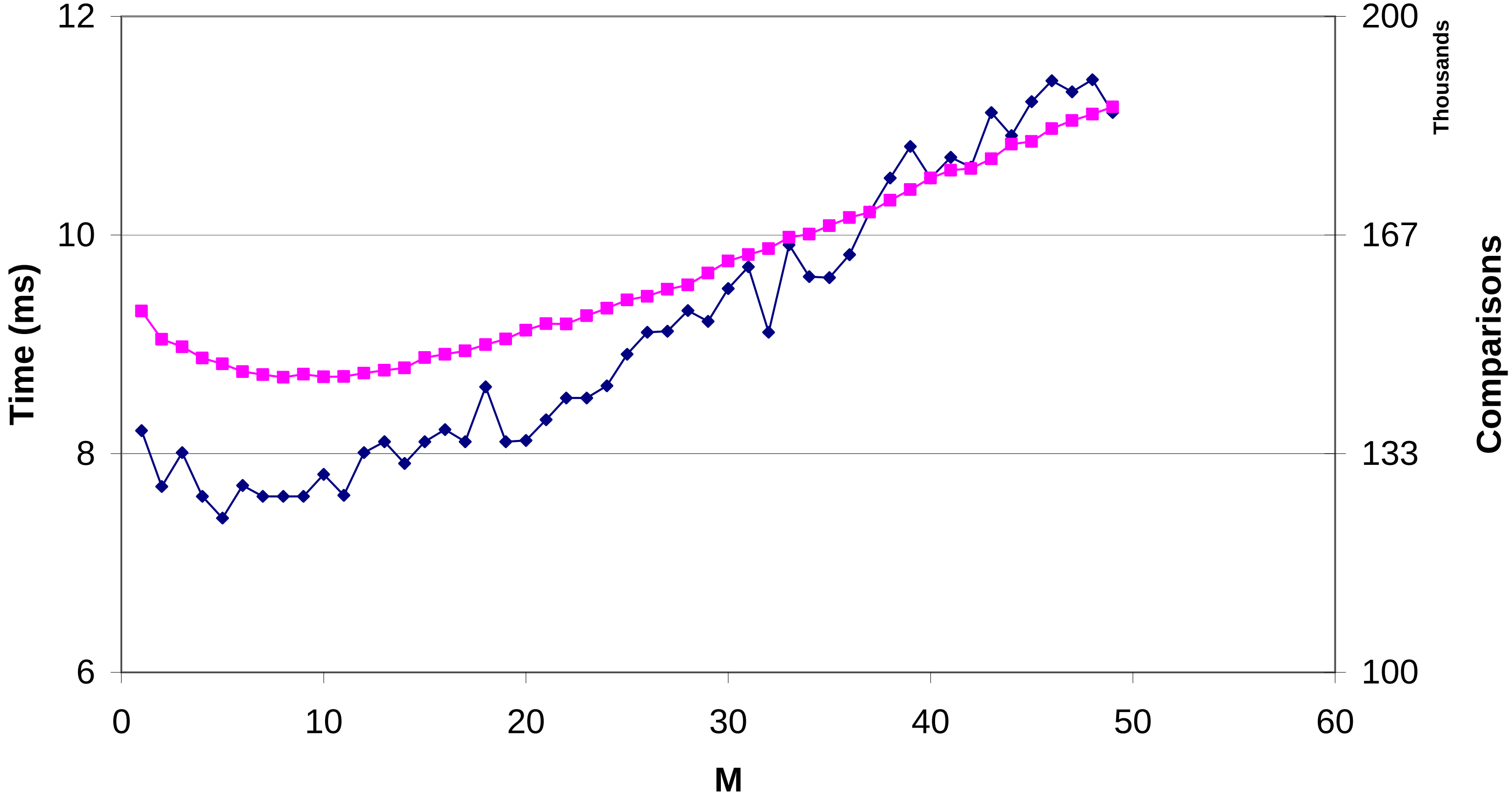
Hashing (continued...)

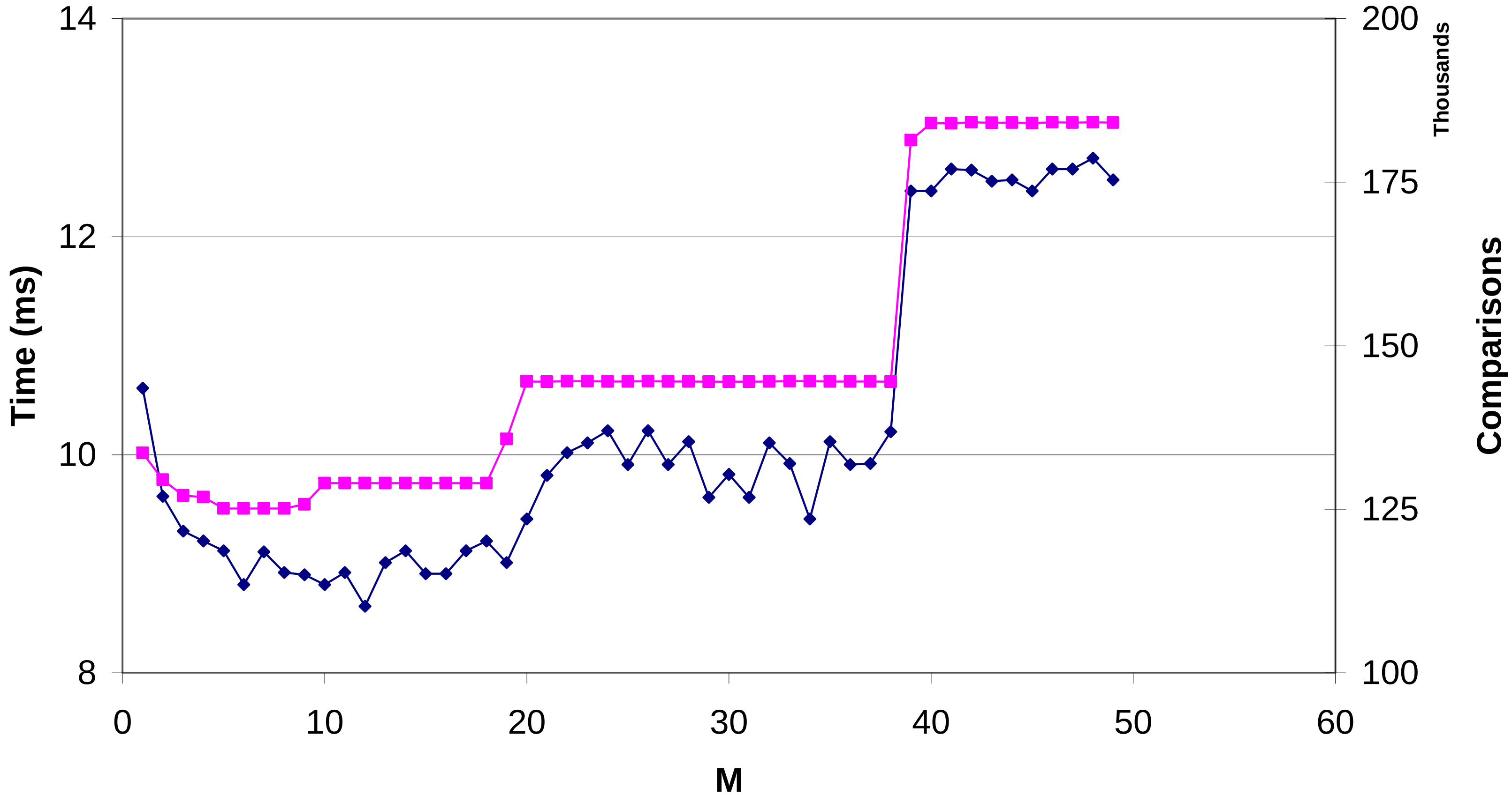
Biostatistics 615/815

Lecture 12

Homework 5, Question 1: Quick Sort Optimization …



Homework 5, Question 1: Merge-Sort Optimization



Last Lecture

- Introduction to hash tables
  - Desirable properties of hash functions
  - Using a chain of pointers to resolve collisions
- Fast way to organize data that does not rely on sorting
- Trades savings in computing time for additional memory use

Today

- More detailed consideration of hash tables
- Alternative conflict resolution strategies
  - Linear Probing
  - Double Hashing
- Managing the size of hash tables

Conflict Resolution 2: Linear Probing

- 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

- Linear probing effectively generates a series of locations to try for each item

- For example, we might specify that
  - For value A, try position 7, then 8, 9, 10 …
  - For value S, try position 3, then 4, 5, 6 …
  - For value E, try position 9, then 10, 11, 12 …

- If there are not many collisions (ie. the table is not very full)
  - Most items will be placed in the first location we try
  - Most items will be retrieved quickly

Linear Probing Example

A	S	E	R	C	H	I	N	G	X	M	P	**Item**
7	3	9	9	8	4	11	7	10	12	0	8	**Hash1**

							A						**Table after inserting element 1**
			S				A						**Table after inserting element 2**
			S				A		E				.
			S				A		E	R			.
			S				A	C	E	R			.
			S	H			A	C	E	R			.
			S	H			A	C	E	R	I		.
			S	H			A	C	E	R	I	N	.
G			S	H			A	C	E	R	I	N	.
G	X		S	H			A	C	E	R	I	N	.
G	X	M	S	H			A	C	E	R	I	N	.
G	X	M	S	H	P		A	C	E	R	I	N	**Table after inserting all elements**
0	1	2	3	4	5	6	7	8	9	10	11	12	**Table index**

Linear Probing: C fragments

```
/* Creating a hash table */
Item table[M];
for (i = 0; i < M; i++)
   table[M] = EMPTY;

/* Inserting or searching for an item */
h = hash(item, M);
while (table[h] != item && table[h] != EMPTY)
   h = (h + 1) % M;

/* Search successful if table[h] != EMPTY */
/* Otherwise, item could be inserted at table[h] */
if (table[h] == EMPTY)
   table[h] = item;
```

Cost Depends on Clustering…

- Consider two tables that are half full
  - In one, items occupy all the odd positions
  - In another, items occupy first M/2 positions
- Where do you expect searches to take longer?

Number of Comparisons

load factor (α)	1/2	2/3	3/4	9/10
Search Hit	1.5	2.0	3.0	5.5
Search Miss	2.5	5.0	8.5	50.5

$$Cost(\text{Hit}) = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad Cost(\text{Miss}) = \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

(These results from an analysis by Knuth, 1962, are actually quite tricky)

Notes on Linear Hashing

- Deleting elements is cumbersome
- Must rehash all other elements in cluster
- Or replace with "DELETED" element
  - Counted as mismatch in searches
  - Counted as empty slot for insert

Conflict Resolution 2: Double Hashing

- Similar to linear hashing
- Guards against clustering by using a second hash function to generate increment for sequential searches
- Very important to ensure table size is prime, or searches for empty slots could fail before table is full

Double Hashing Example

A	S	E	R	C	H	I	N	G	X	M	P	L	**Item**
7	3	9	9	8	4	11	7	10	12	0	8	6	**Hash1**
1	3	1	5	5	5	3	3	2	3	5	4	2	**Hash2**

							A						**Table after inserting element 1**
			S				A						**Table after inserting element 2**
			S				A		E				.
	R		S				A		E				.
	R		S				A	C	E				.
	R		S	H			A	C	E				.
	R		S	H			A	C	E		I		.
	R		S	H			A	C	E	N	I		.
	R		S	H			A	C	E	N	I	G	.
	R	X	S	H			A	C	E	N	I	G	.
M	R	X	S	H			A	C	E	N	I	G	.
M	R	X	S	H		P	A	C	E	N	I	G	.
M	R	X	S	H	L	P	A	C	E	N	I	G	. **Table after inserting all elements**
0	1	2	3	4	5	6	7	8	9	10	11	12	**Table index**

Double Hashing: C fragments

```
/* 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] */
if (table[h] == EMPTY)
  table[h] = item;
```

Number of Comparisons

load factor (α)	1/2	2/3	3/4	9/10
Search Hit	1.4	1.6	1.8	2.6
Search Miss	2.0	3.0	4.0	10

$$Cost(\text{Hit}) = \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \qquad Cost(\text{Miss}) = \frac{1}{1-\alpha}$$

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

$$\frac{1}{N}\left(1+\frac{1}{1-1/M}+\frac{1}{1-2/M}+\frac{1}{1-3/M}+\ldots\right)=$$

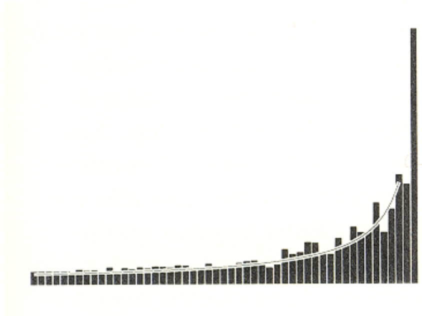
$$\frac{1}{N}\left(1+\frac{M}{M-1}+\frac{M}{M-2}+\frac{M}{M-3}+\ldots\right)$$

Further Notes on Hashing

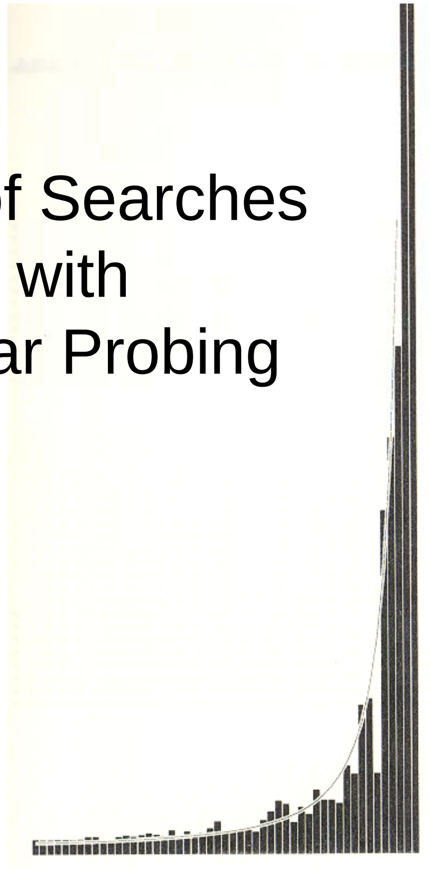
- To ensure that search requires less than *t* comparisons on average
  - $\alpha < (1 - 1/t)$ with double hashing
  - $\alpha < (1 - 1/\mathit{sqrt}(t))$ with linear hashing
- Dynamic hashing
  - Increase table size and rehash elements whenever α exceeds a threshold (e.g. 50%)

Cost Comparison

Cost of Searches
with
Double Hashing



Cost of Searches
with
Linear Probing



Quadratic Probing

- An intermediate strategy between linear probing and double hashing

- After the i^{th} collision, we check position $(h + c_1 i + c_2 i^2) \ mod\ M$

  - c_1 and c_2 are constants
  - $c_1 = c_2 = 0.5$ works well when M is prime

Dynamic Hashing

- Hash tables must balance:
  - Speed of inserting and retrieving elements
  - Usage of computer memory
- With dynamic hashing table is resized when it starts getting "full"
  - Avoid performance penalty for nearly full tables

Dynamic Hashing: C Fragment

```
/* Creating a hash table */
Item * table;
int    M = 2, N = 0;

table = malloc(sizeof(Item) * M);
for (i = 0; i < M; i++)
   table[M] = EMPTY;

/* Inserting or searching for an item */
h = hash(item, M);
while (table[h] != item && table[h] != EMPTY)
   h = (h + 1) % M;

/* Inserted new items into table */
if (table[h] == EMPTY)
   {
   table[h] = item;
   N++;
   }
```

Dynamic Hashing: C Fragment

```
/* Check if table is nearly full */
if (N >= M/2)
   {
   /* Allocate a new table */
   Item * newTable = malloc(sizeof(Item)) * M * 2;
   for (int i = 0; i < M * 2; i++)
       newTable[i] = EMPTY;

   /* Rehash all elements into the larger table */
   for (int i = 0; i < M; i++)
      if (table[i] != EMPTY)
         {
         h = hash(table[i], M * 2);
         while (newtable[h] != EMPTY)
            h = (h + 1) % (M * 2);
         newTable[h] = table[i];
         }

   /* Replace previous table */
   free(table);
   table = newTable;
   M *= 2;
   }
```

Is Dynamic Hashing Effective?

- The cost of resizing the table seems rather high …
- However, this only happens rarely …
  - Cost gets amortized over very many insertions
- Average cost per insertion is still O(1)!

Summary

- Hashing
  - Linear Probing
  - Double Hashing
  - Dynamic Hashing
- Cost of searches is nearly independent of N
  - Fast searches that don't require sorting
  - Not very effective if analysis requires ordered data

Recommended Reading

- Sedgewick, Chapter 14
- Peterson W. W. (1957) *IBM Journal of Research and Development* **1:**130-146
- Question to ponder: Does the order in which elements are inserted change the total cost of building hash table?