Probabilistic Analysis

Sequential Search

Input : Array A of n distinct integers.
Key K.
Output : p such that $A[p] = K$ or -1 if there is no such p.

```
function SeqSearch(A[ ],n,K)
for i ← 1 to n do
    if (A[i] = K) then
        return (i);
    end
end
/* Element x not found.                    */
return (−1);
```

Sequential Search: Expected Running Time

Expected running time =

$$\sum_{i=1}^{n} Pr(\mathsf{A}[i] = K)t(\mathsf{A}[i] = K) + Pr(K \notin \mathsf{A})t(K \notin \mathsf{A}).$$

$Pr(I)$ = probability that event I occurs.
$t(I)$ = running time given that event I occurs.

Expected Running Time

Expected/average running time $ET(n) = \sum_I Pr(I)t(I)$;

- $Pr(I)$ = probability of event I;
- $t(I)$ = running time given event I.

$Pr(I)$ depends on the input probability distribution (usually uniform).

Example

```
Func1(A, n)
/* A is an array of integers */
s ← 0;
k ← Random(n);
for i ← 1 to k do
    for j ← 1 to k do
        s ← s + A[i] * A[j];
    end
end
return (s);
```

`Random`(n) generates a random integer between 1 and n with uniform distribution (every integer between 1 and n is equally likely.)

Expectation

X is a random variable.
The expectation of X is:

$$E(X) = \sum_{I} Pr(X = I)\ I.$$

Linearity of expectation:

$$E(X_1 + X_2) = E(X_1) + E(X_2).$$

Conditional expectation:

$$E(X) = E(X \mid Y)\ Pr(Y) + E(X \mid \text{Not } Y)\ (1 - Pr(Y)).$$

Linearity of Expectation

X is a random variable.
The expectation of X is:

$$E(X) = \sum_I Pr(X = I)\ I.$$

Linearity of expectation:

$$E(X_1 + X_2) = E(X_1) + E(X_2).$$

$$E(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} E(X_i).$$

Linearity of Expectation

function `Func2`(A[ ],n)

/* A *is an array of* n *integers* */

$s \leftarrow 0$;

for $i \leftarrow 1$ **to** n **do**

  $k \leftarrow$ `Random`(n);

  for $j = 1$ **to** k^2 **do** $s \leftarrow s + j \times A[\lceil j/n \rceil]$;

end

return (s);

`Random`(n) generates a random integer between 1 and n with uniform distribution (every integer between 1 and n is equally likely.)

Conditional Expectation

X is a random variable.
The expectation of X is:

$$E(X) = \sum_I Pr(X = I)\, I.$$

Conditional expectation:

$$E(X) = E(X \mid Y)\, Pr(Y) + E(X \mid \text{Not } Y)\, (1 - Pr(Y)).$$

Conditional Expectation

function `Func3`(A[ ],n)
/* A *is an array of* n *integers* */
$k \leftarrow$ `Random`(n);
$s \leftarrow 0$;
if $(k = 1)$ **then**
  for $i \leftarrow n$ **to** n^2 **do** $s \leftarrow s + i \times A[\lceil i/n \rceil]$;
else
  for $i \leftarrow 1$ **to** $n\lfloor \log_2(n) \rfloor$ **do** $s \leftarrow s + i \times A[\lceil i/n \rceil]$;
end
return (s);

`Random`(n) generates a random integer between 1 and n with uniform distribution (every integer between 1 and n is equally likely.)

Conditional Expectation

```
function Func4(A[ ],n)
/* A is an array of n integers */
k ← Random(n);
s ← 0;
if (k ≤ √n) then
  | for i ← n to n^2 do  s ← s + i × A[⌈i/n⌉];
else
  | for i ← n to n⌊log_2(n)⌋ do  s ← s + i × A[⌈i/n⌉];
end
return (s);
```

Random(n) generates a random integer between 1 and n with uniform distribution (every integer between 1 and n is equally likely.)

Example

```
Func5(A, n)
/* A is an array of integers                                  */
if (n = 1) then return(0);
else
    s ← 0;
    for i ← 1 to ⌊n/2⌋ do
        s ← s + A[i] * A[n − i + 1];
    end
    k ← Random(n);
    if (k is even) then
        s ← s + Func5(A, n − 1);
    end
    return (s);
end
```

Example

```
Func6(A, n)
/* A is an array of integers                                  */
if (n ≤ 2) then  return(A[1]);
else
    k1 ← Random(n);
    k2 ← Random(n);
    if (k1 < k2) then
        return (A[n]);
    else
        s ← Func6 (A, n − 1) + Func6 (A, n − 2);
        return (s);
    end
end
```

Insertion into a Sorted Array

Input : Array A of n integers in sorted order.
($A[1] \leq A[2] \leq A[3] \ldots \leq A[n]$) Element x.

function `SortedInsert`(A[ ],n,x)

$A[n+1] \leftarrow x$;

$j \leftarrow n$;

while $(j > 0)$ **and** $(A[j] > A[j+1])$ **do**

  `Swap`($A[j], A[j+1]$);

  $j \leftarrow j - 1$;

end

Insertion Sort

Input : Array A of n elements.

Result : Array A containing a permutation of the input such that $A[1] \le A[2] \le A[3] \le \ldots \le A[n]$.

```
InsertionSort(A[ ],n)
for i ← 1 to n − 1 do
    /* insert A[i + 1] in A[1..i]                                  */
    /* maintains: A[1] ≤ A[2] ≤ A[3] ≤ . . . ≤ A[i]                */
    x ← A[i + 1];
    SortedInsert(A, i, x);
end
```

Insertion Sort (Version 2)

Input : Array A of n elements.

Result : Array A containing a permutation of the input such that $A[1] \leq A[2] \leq A[3] \leq \ldots \leq A[n]$.

Call to `SortedInsert` replaced by while loop.

```
InsertionSort(A[ ],n)
for i ← 2 to n do
    /* insert A[i] in A[1..(i − 1)]                          */
    /* maintains: A[1] ≤ A[2] ≤ A[3] ≤ ... ≤ A[i − 1]        */
    j ← i − 1;
    while (j > 0) and (A[j] > A[j + 1]) do
        Swap(A[j], A[j + 1]);
        j ← j − 1;
    end
end
```

Insertion Sort: Recursive Version

Input : Array A of n elements.

Result : Array A containing a permutation of the input such that $A[1] \leq A[2] \leq A[3] \leq \ldots \leq A[n]$.

```
InsertionSortRec(A[ ],n)
if (n > 1) then
    InsertionSort(A[ ],n − 1);
    /* Insert A[n] in A[1..(n − 1)]                */
    x ← A[n];
    SortedInsert(A, n − 1, x);
end
```

Insertion Sort: Recursive Version 2

Input : Array A of n elements.

Result : Array A containing a permutation of the input such that $A[1] \le A[2] \le A[3] \le \ldots \le A[n]$.

Call to `SortedInsert` replaced by while loop.

```
InsertionSortRec(A[ ],n)
if (n > 1) then
    InsertionSort(A[ ],n − 1);
    /* Insert A[n] in A[1..(n − 1)]                    */
    j ← n − 1;
    while (j > 0) and (A[j] > A[j + 1]) do
        Swap(A[j], A[j + 1]);
        j ← j − 1;
    end
end
```

Expectation Formula

Theorem. If X is a random variable taking only values $\{0, 1, 2, 3, \ldots\}$, then

$$E(X) = \sum_{i=1}^{\infty} Pr(X \geq i).$$

Proof. Let X_i be a random variable where $X_i = \begin{cases} 1 & \text{if } X \geq i, \\ 0 & \text{if } X < i. \end{cases}$

$$X = \sum_{i=1}^{\infty} X_i.$$

$$E(X_i) = \text{Prob}(X \geq i) \times 1 + \text{Prob}(X < i) \times 0 = \text{Prob}(X \geq i).$$

$$E(X) = E(\sum_{i=1}^{\infty} X_i) = \sum_{i=1}^{\infty} E(X_i) \qquad \text{by linearity of expectation}$$

$$= \sum_{i=1}^{\infty} \text{Prob}(X \geq i).$$

□

Columbus Casino

```
procedure ColumbusCasino()
c ← CoinFlip();
while (c = heads) do
    Print "I win";
    c ← CoinFlip();
end
Print "I quit";
```

Columbus Casino: Analysis

X = Number of heads.

Running time $= cX + c$.
(Last c term is the time for the last coin flip which is a tail.)

Expected running time $= E(cX + c) = cE(X) + c$.

Use formula $E(X) = \sum_{i=1}^{\infty} Pr(X \geq i)$.

$$E(X) = \sum_{i=1}^{\infty} Pr(X \geq i) = \sum_{i=1}^{\infty} (1/2)^i = 1.$$

Expected running time $= cE(X) + c = c + c = 2c$.

Example

```
function Func4(A[ ], n)
for i ← 1 to n do
    c ← CoinFlip();
    if (c == heads) then
        return (A[i]);
    end
end
return (A[n]);
```

QuickSort

Sort 2: Divide and Conquer

Input : Array A of at least j elements.
Integers i and j.

Result : A permutation of the i through j elements of A such that $A[i] \leq A[i+1] \leq A[i+2] \leq \ldots \leq A[j]$.

```
Sort2(A[ ],i,j)
if (i < j) then
    p ← Median(A, i, j);
    /* Partition A[i, ..., j] using p s.t. A[s] = p          */
    /* and A[i'] ≤ p ≤ A[j'] for i ≤ i' ≤ s ≤ j' ≤ j        */
    s ← Partition(A, i, j, p);
    Sort2(A[ ],i,s − 1);
    Sort2(A[ ],s + 1,j);
end
```

Partition

Input : Array A of at least j elements.
Integers i and j. Array element p.

Result : A permutation of array A such that $\mathsf{A}[s] = p$ and $\mathsf{A}[i'] \leq p \leq A[j']$ for $i \leq i' \leq s \leq j' \leq j$. Returns s.

```
Partition(A[ ], i, j, p)
low ← i;
high ← j;
while (low < high) do
    if (A[low] < p) then  low ← low + 1;
    else if (A[high] > p) then  high ← high − 1;
    else
        Swap(A[low], A[high]);
        if (A[low] = p and A[high] = p) then low ← low + 1;
    end
end
return (high);
```

QuickSort

Input : Array A of at least j elements.
Integers i and j.

Result : A permutation of the i through j elements of A such that $A[i] \leq A[i+1] \leq A[i+2] \leq \ldots \leq A[j]$.

```
QuickSort(A[ ],i,j)
if (i < j) then
    /* choose random element of A[ ]                                  */
    p ← RandomElement(A, i, j);
    /* Partition A[i, ..., j] using p s.t. A[s] = p                   */
    /* and A[i'] ≤ p ≤ A[j'] for i ≤ i' ≤ s ≤ j' ≤ j                  */
    s ← Partition(A, i, j, p);
    QuickSort(A[ ],i,s − 1);
    QuickSort(A[ ],s + 1,j);
end
```

QuickSort: Analysis

$ET(n)$ = expected running time of `QuickSort` on n values.
$ET(0) = 0$.

Assume array has no duplicates.
After partition, $A[s] = p$.
Let $m = s - i + 1$.
After partition, p is m'th element of $A[i], A[i+1], \ldots, A[j]$.

$$
\begin{aligned}
ET(n) &= \sum_{k=1}^{n} Pr(m = k) ET(m = k) \\
&= \sum_{k=1}^{n} \frac{1}{n} (ET(k-1) + ET(n-k) + cn).
\end{aligned}
$$

QuickSort: Upper Bounds

$ET(n)$ = expected running time of `QuickSort` on n values.

Assume array has no duplicates.

Let $m = s - i + 1$. (After partition, p is m'th element of $A[i], \ldots, A[j]$.)

$$\begin{aligned} ET(n) = & Pr(m < n/4)ET(m < n/4) + \\ & Pr(n/4 \leq m \leq 3n/4)ET(n/4 \leq m \leq 3n/4) + \\ & Pr(m > 3n/4)ET(m > 3n/4). \end{aligned}$$

$$\begin{aligned} Pr(m < n/4) &= 1/4. \\ Pr(m > 3n/4) &= 1/4. \\ Pr(n/4 \leq m \leq 3n/4) &= 1/2. \\ ET(m < n/4) &\leq ET(m = 1) = cn + ET(n-1). \\ ET(m > 3n/4) &\leq ET(m = n) = cn + ET(n-1). \\ ET(n/4 \leq m \leq 3n/4) &\leq ET(m = n/4) = cn + ET(n/4) + ET(3n/4). \end{aligned}$$

QuickSort: Upper Bounds

Let $m = s - i + 1$. (After partition, p is m'th element of $A[i], \ldots, A[j]$.)

$$\begin{aligned}
Pr(m < n/4) &= 1/4. \\
Pr(m > 3n/4) &= 1/4. \\
Pr(n/4 \leq m \leq 3n/4) &= 1/2. \\
ET(m < n/4) &\leq ET(m = 1) = cn + ET(n-1). \\
ET(m > 3n/4) &\leq ET(m = n) = cn + ET(n-1). \\
ET(n/4 \leq m \leq 3n/4) &\leq ET(m = n/4) = cn + ET(n/4) + ET(3n/4).
\end{aligned}$$

$$\begin{aligned}
ET(n) = \; & Pr(m < n/4) ET(m < n/4) + \\
& Pr(n/4 \leq m \leq 3n/4) ET(n/4 \leq m \leq 3n/4) + \\
& Pr(m > 3n/4) ET(m > 3n/4) \\
\leq \; & \frac{1}{4}\Big(cn + ET(n-1)\Big) + \frac{1}{2}\Big(cn + ET(n/4) + ET(3n/4)\Big) + \\
& \frac{1}{4}\Big(cn + ET(n-1)\Big).
\end{aligned}$$

QuickSort: Upper Bounds

$$\begin{aligned}
ET(n) &\leq \frac{1}{4}\Big(cn + ET(n-1)\Big) + \frac{1}{2}\Big(cn + ET(n/4) + ET(3n/4)\Big) + \\
&\quad \frac{1}{4}\Big(cn + ET(n-1)\Big) \\
&= cn + \frac{1}{2}ET(n-1) + \frac{1}{2}\Big(ET(n/4) + ET(3n/4)\Big) \\
&\leq cn + \frac{1}{2}ET(n) + \frac{1}{2}\Big(ET(n/4) + ET(3n/4)\Big). \\
\frac{1}{2}ET(n) &\leq cn + \frac{1}{2}\Big(ET(n/4) + ET(3n/4)\Big). \\
ET(n) &\leq 2cn + ET(n/4) + ET(3n/4) \\
&\leq c_2 n + ET(n/4) + ET(3n/4) \qquad \text{where } c_2 = 2c.
\end{aligned}$$

$\therefore ET(n) \in O(n \log_2(n))$.

QuickSort: Lower Bounds

In the best case, p is the median.

$$\begin{aligned} ET(n) &\geq cn + ET(n/2) + ET(n/2) \\ &= cn + 2ET(n/2). \end{aligned}$$

$$\therefore ET(n) \in \Omega(n \log_2(n)).$$

QuickSort: Version 2

Input : Array A of at least j elements.
Integers i and j.

Result : A permutation of the i through j elements of A such that $A[i] \leq A[i+1] \leq A[i+2] \leq \ldots \leq A[j]$.

```
QuickSort2(A[ ],i,j)
if (i < j) then
    p ← A[i];
    /* Partition A[i, ..., j] using p s.t. A[s] = p                  */
    /* and A[i'] ≤ p ≤ A[j'] for i ≤ i' ≤ s ≤ j' ≤ j                   */
    s ← Partition(A, i, j, p);
    QuickSort2(A[ ],i,s − 1);
    QuickSort2(A[ ],s + 1,j);
end
```

`QuickSort2`: Analysis

$ET(n)$ = expected running time of `QuickSort2` on n values.
$ET(0) = 0$.

Assume array has no duplicates and
<u>all permutations are equally likely</u>.

$$\begin{aligned} ET(n) &= \sum_{k=1}^{n} Pr(s = k) ET(s = k) \\ &= \sum_{k=1}^{n} \frac{1}{n} (ET(k-1) + ET(n-k) + cn). \end{aligned}$$

QuickSort: Version 3

Input : Array A of at least j elements.
Integers i and j.

Result : A permutation of the i through j elements of A such that $A[i] \leq A[i+1] \leq A[i+2] \leq \ldots \leq A[j]$.

QuickSort3(A[ ],i,j)

if $(i < j)$ **then**

  $p \leftarrow$ median element of $\{\mathsf{A}[i], \mathsf{A}[\lfloor (i+j)/2 \rfloor], \mathsf{A}[j]\}$;

  /* *Partition* $\mathsf{A}[i,\ldots,j]$ *using* p *s.t.* $\mathsf{A}[s] = p$ */

  /* *and* $\mathsf{A}[i'] \leq p \leq A[j']$ *for* $i \leq i' \leq s \leq j' \leq j$ */

  $s \leftarrow$ Partition(A, i, j, p);

  QuickSort3(A[ ],i,$s-1$);

  QuickSort3(A[ ],$s+1$,j);

end

Selection

Selection: Randomized Algorithm

Input : Array A of at least j elements.
Integers i, j and k.

Output : k'th element of A in sorted order.

```
RandomizedSelect(A[ ],i,j,k)
p ← RandomElement(A,i,j);
s ← Partition(A, i, j, p);
/* A[s] = p and A[i'] ≤ p ≤ A[j'] for i ≤ i' ≤ s ≤ j' ≤ j    */
m ← s − i + 1;          /* A[s] is the m'th element of A[i..j] */
if (m = k) then  x ← A[s];
else if (m > k) then  x ← RandomizedSelect(A, i, s − 1, k);
else /* m < k */
|   x ← RandomizedSelect(A, s + 1, j, k − m);
end
return (x);
```

`RandomizedSelect`: Analysis

$ET(n)$ = expected running time of `RandomizedSelect` on n values.

Assume array has no duplicates and all permutations are equally likely.

$$\begin{aligned} ET(n) &= \sum_{q=1}^{n} Pr(m = q) ET(m = q) \\ &= \sum_{q=1}^{n} \frac{1}{n} ET(m = q). \end{aligned}$$

$$ET(m = q) \quad \leq \quad \max(ET(m = q \text{ and } k < m), ET(m = q \text{ and } k > m)).$$

Therefore,

$$ET(n) = \frac{1}{n} \sum_{i=1}^{n} ET(m = q) \leq \frac{1}{n} \sum_{i=1}^{n} \max(ET(q-1), ET(n-q)).$$

`RandomizedSelect`: Upper Bounds

$ET(n)$ = expected running time of `RandomizedSelect` on n values.
Assume array has no duplicates.
$m = s - i + 1$. (After partition, p is m'th element of $A[i], \ldots, A[j]$.)

$$
\begin{aligned}
ET(n) = {} & Pr(m < n/4)ET(m < n/4) + \\
& Pr(n/4 \leq m \leq 3n/4)ET(n/4 \leq m \leq 3n/4) + \\
& Pr(m > 3n/4)ET(m > 3n/4).
\end{aligned}
$$

$$
\begin{aligned}
Pr(m < n/4) &= 1/4. \\
Pr(m > 3n/4) &= 1/4. \\
Pr(n/4 \leq m \leq 3n/4) &= 1/2. \\
ET(m < n/4) &\leq ET(m = 1) = cn + ET(n-1). \\
ET(m > 3n/4) &\leq ET(m = n) = cn + ET(n-1). \\
ET(n/4 \leq m \leq 3n/4) &\leq ET(m = n/4) = cn + \max(ET(n/4), ET(3n/4)) \\
&= cn + ET(3n/4).
\end{aligned}
$$

`RandomizedSelect`: Upper Bounds

$m = s - i + 1$. (After partition, p is m'th element of $A[i], \ldots, A[j]$.)

$$Pr(m < n/4) = 1/4.$$
$$Pr(m > 3n/4) = 1/4.$$
$$Pr(n/4 \leq m \leq 3n/4) = 1/2.$$
$$ET(m < n/4) \leq ET(m = 1) = cn + ET(n-1).$$
$$ET(m > 3n/4) \leq ET(m = n) = cn + ET(n-1).$$
$$ET(n/4 \leq m \leq 3n/4) \leq ET(m = n/4) = cn + ET(3n/4).$$

$$\begin{aligned} ET(n) = {} & Pr(m < n/4)ET(m < n/4) + \\ & Pr(n/4 \leq m \leq 3n/4)ET(n/4 \leq m \leq 3n/4) + \\ & Pr(m > 3n/4)ET(m > 3n/4) \\ \leq {} & \frac{1}{4}\Big(cn + ET(n-1)\Big) + \frac{1}{2}\Big(cn + ET(3n/4)\Big) + \\ & \frac{1}{4}\Big(cn + ET(n-1)\Big). \end{aligned}$$

`RandomizedSelect`: Upper Bounds

$$\begin{aligned}
ET(n) &\le \frac{1}{4}\Big(cn + ET(n-1)\Big) + \frac{1}{2}\Big(cn + ET(3n/4)\Big) + \\
&\quad \frac{1}{4}\Big(cn + ET(n-1)\Big) \\
&= cn + \frac{1}{2}ET(n-1) + \frac{1}{2}ET(3n/4) \\
&\le cn + \frac{1}{2}ET(n) + \frac{1}{2}ET(3n/4). \\
\frac{1}{2}ET(n) &\le cn + \frac{1}{2}ET(3n/4). \\
ET(n) &\le 2cn + ET(3n/4) \\
&\le c_2 n + ET(3n/4) \qquad \text{where } c_2 = 2c.
\end{aligned}$$

$\therefore ET(n) \in O(n)$.

`RandomizedSelect`: Lower Bounds

`RandomizedSelect` always executes **`Partition`** at least once.
`Partition` takes cn time. Therefore, $ET(n) \in \Omega(n)$.

Selection: Deterministic Algorithm

Input : Array A of at least j elements.
Integers i, j and k.
Output : k'th element of A in sorted order.

Select(A[ ],i,j,k)
$p \leftarrow$ ApproxMedian(A,i,j);
$s \leftarrow$ Partition(A, i, j, p);
/* $A[s] = p$ *and* $A[i'] \leq p \leq A[j']$ *for* $i \leq i' \leq s \leq j' \leq j$ */
$m \leftarrow s - i + 1$; /* $A[s]$ *is the* m*'th element of* $A[i..j]$ */
if $(m = k)$ **then** $x \leftarrow A[s]$;
else if $(m > k)$ **then** $x \leftarrow$ Select(A, i, $s-1$, k);
else /* $m < k$ */
| $x \leftarrow$ Select(A, $s+1$, j, $k-m$);
end
return (x);

ApproximateMedian

Input : Array A of at least j elements.
Integers i, j and k.

Output : Approximate median of $A[i], \ldots, A[j]$.

`ApproxMedian`($A[\,]$, i, j)

$n \leftarrow j - i + 1$;
Partition $A[i], \ldots, A[j]$ into $\lceil n/5 \rceil$ groups of 5;
for $i \leftarrow 1$ **to** $\lceil n/5 \rceil$ **do**
  | $B[i] \leftarrow$ median of i'th group;
end
$p \leftarrow$ `Select`(B, 1, $\lceil n/5 \rceil$, $\lfloor n/10 \rfloor$);
return (p);

Hashing

Dictionary

- `Dict.Init()`: Initialize the dictionary;
- `Dict.Insert`(Key K, Data D): Insert (key,data) in dictionary;
- **bool** `Dict.Member`(Key K):
  Return **true** if key K is in dictionary;
- Data `Dict.Retrieve`(Key K):
  Return data associated with key K.

Hashing

Hash table: $H[0], H[1], \ldots, H[m-1]$.
m = Size of hash table.

Hash functions: h : Key $\mathsf{K} \rightarrow \{0, 1, \ldots, m-1\}$.
Examples:

- $h(\mathsf{K}) = \mathsf{K} \bmod m$
- $h(\mathsf{K}) = (1771 * \mathsf{K}) \bmod m$

procedure `HashTable`.Insert(K,D)

$i \leftarrow h(K)$;
Insert(K,D) in $H[i]$;

Collisions

Hash table: $H[0], H[1], \ldots, H[m-1]$.
m = Size of hash table.
Hash function: h : Key $\mathsf{K} \rightarrow \{0, 1, \ldots, m-1\}$.

procedure `HashTable`.Insert(K,D)

1 $i \leftarrow h(K)$;

2 Insert(K,D) in $H[i]$;

Collisions: $h(K_1) = h(K_2)$ but $K_1 \neq K_2$.

Collisions

Hash table: $H[0], H[1], \ldots, H[m-1]$.

m = Size of hash table.

n = Number of elements in the table.

N = Size of the set containing all possible keys.

(e.g., 2^{32} or 2^{64} for 32-bit or 64-bit unsigned integers.)

(N could be infinite, e.g., keys are all possible strings.)

Typically,

$$N >> m > n.$$

Chained Hashing

$H[i]$ is a linked list.

```
procedure HashTable.Insert(K,D)
i ← h(K);
Add (K, D) to linked list H[i];
```

```
bool procedure HashTable.Member(K)
i ← h(K);
if (K is in linked list H[i]) then return (true);
else return (false);
```

```
Data procedure HashTable.Retrieve(K)
i ← h(K);
Retrieve (K, D) from linked list H[i];
return (D);
```

Expected Running time of `HashTable.Member`

m = size of the hash table

n = # elements in the hash table.

Expected running time of `HashTable.Member()` =
c*(Expected length of linked list $H[i]$ +1).

Expected length of linked list $H[i]$ =
Expected number of elements inserted in $H[i]$ =
"Average" number of elements inserted in $H[i] = n/m$.

Expected running time of `HashTable.Member()` $\in \Theta(n/m + 1)$.

Expected Running time of `HashTable.Member`

m = size of the hash table

n = # elements in the hash table.

$$\text{Let } X_j = \begin{cases} 1 & \text{if } h(K_j) = i, \\ 0 & \text{otherwise.} \end{cases}$$

Expected number of elements inserted in $H[i]$ =

$$E\left(\sum_{j=1}^{n} X_j\right) = \sum_{j=1}^{n} E(X_j)$$

$$= \sum_{j=1}^{n} \text{Prob}(h(K_j) = i) * 1 = \sum_{j=1}^{n} (1/m) = (n/m).$$

Expected time for `HashTable.Retrieve()` is $\Theta(1 + n/m)$.

Chained Hashing: Other operations

$H[i]$ is a linked list.

procedure `HashTable.Replace`(K,D)

/* *Replace data associated with key* K *by data* D */

$i \leftarrow h(K)$;

Find element e of linked list $H[i]$ with key K;

e.data $\leftarrow$ D;

procedure `HashTable.Add`(K,x)

/* *Add* x *to (numeric) data* D *associated with key* K */

$i \leftarrow h(K)$;

Find element e of linked list $H[i]$ with key K;

e.data $\leftarrow e$.data $+ x$;

Open Address Hashing

$H[i]$ contains a single key.
Hash function: $h(\text{Key } \mathsf{K}, \text{Integer } j)$.

```
procedure HashTable.Insert(K, D)
j ← 0;
repeat
    i ← h(K, j);
    if (H[i] is empty) then
        H[i] ← (K, D);
        return;
    else
        j ← j + 1;
    end
until (j = m);
error "hash table overflow";
```

Possible Rehashing Functions

$$h(\mathsf{K}, j) = (h(\mathsf{K}) + j) \bmod m;$$

$$h(\mathsf{K}, j) = (h(\mathsf{K}) + c_1 j + c_2 j^2) \bmod m;$$

$$h(\mathsf{K}, j) = (h_1(\mathsf{K}) + j * h_2(\mathsf{K})) \bmod m.$$

Open Address Hashing: Member

```
procedure HashTable.Member(K)
j ← 0;
repeat
    i ← h(K, j);
    if (H[i].key = K) then
        return (true);
    end
    j ← j + 1;
until (j = m) or (H[i] is empty);
return (false);
```

Open Address Hashing: Retrieval

```
procedure HashTable.Retrieve(K)
j ← 0;
repeat
    i ← h(K, j);
    if (H[i].key = K) then
        return (H[i].data);
    end
    j ← j + 1;
until (j = m) or (H[i] is empty);
return (∅);
```

Running Time Analysis: Insertion

```
procedure HashTable.Insert(K, D)
j ← 0;
repeat
    i ← h(K, j);
    if (H[i] is empty) then
        H[i] ← (K, D);
        return;
    else
        j ← j + 1;
    end
until (j = m);
error "hash table overflow";
```

Expected Running Time of `HashTable.Insert`

m = size of the hash table
n = # elements in the hash table.

Let $i_j = h(K, j)$.

$\text{Prob}(H[i_0] \text{ is not empty}) = \frac{n}{m}$.
$\text{Prob}(H[i_0] \text{ and } H[i_1] \text{ are not empty}) = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right) \leq \left(\frac{n}{m}\right)^2$.

$\text{Prob}(H[i_0], H[i_1], H[i_2] \text{ are not empty}) =$

$$\left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\left(\frac{n-2}{m-2}\right) \leq \left(\frac{n}{m}\right)^3.$$

$\text{Prob}(H[i_0], H[i_1], \ldots, H[i_k] \text{ are not empty}) =$

$$\left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\left(\frac{n-2}{m-2}\right)\cdots\left(\frac{n-k}{m-k}\right) \leq \left(\frac{n}{m}\right)^{k+1}.$$

Expected Running Time of `HashTable`.Insert

X = number of times loop 2-10 repeats.

Use formula $ET(X) = \sum_{k=1}^{\infty} Pr(X \geq k)$.

$$
\begin{aligned}
Pr(X \geq 1) &= 1 \\
Pr(X \geq k) &= \text{Prob}(H[i_0], H[i_1], \ldots, H[i_{k-2}] \text{ are not empty}) \\
&\leq (n/m)^{k-1}.
\end{aligned}
$$

Expected Running Time of HashTable.Insert

X = number of times loop 2-8 repeats.

Use formula $ET(X) = \sum_{k=1}^{\infty} Pr(X \geq k)$.

$$
\begin{aligned}
ET(n,m) = cET(X) &= \sum_{k=1}^{\infty} Pr(X \geq k) \\
&= c\sum_{k=1}^{n+1} Pr(X \geq k) \qquad \text{(since } Pr(X > n+1) \text{ is } 0\text{)} \\
&= c(Pr(X \geq 1) + \sum_{k=2}^{n+1} Pr(X \geq k)) \\
&= c(1 + \sum_{k=2}^{n+1} Pr(H[i_0], H[i_1], \ldots, H[i_{k-2}] \text{ are not empty})) \\
&\leq c(1 + (n/m) + (n/m)^2 + (n/m)^3 + \ldots + (n/m)^n) \\
&\leq \frac{c}{1-(n/m)}.
\end{aligned}
$$

Expected Running Time of `HashTable.Insert`

m = size of the hash table

n = # elements in the hash table.

$$ET(n,m) \le \frac{c}{1-(n/m)}.$$

If $n \le m/2$, then $(n/m) \le (1/2)$ and

$$ET(n,m) \le \frac{c}{1-(1/2)} = \frac{c}{1/2} = 2c.$$

Similar analysis for retrieval.

Open Address Hashing: Replace

```
bool procedure HashTable.Replace(K, D)
/* Replace data associated with key K by data D   */
/* Return false if key K not found                */
j ← 0;
repeat
    i ← h(K, j);
    if (H[i].key = K) then
        H[i].data ← D;
        return (true);
    end
    j ← j + 1;
until (j = m) or (H[i] is empty);
return (false);
```

Open Address Hashing: Add

```
bool procedure HashTable.Add(K, x)
/* Add x to (numeric) data D associated with key K      */
/* Return false if key K not found                      */
j ← 0;
repeat
    i ← h(K, j);
    if (H[i].key = K) then
        H[i].data ← H[i].data + x;
        return (true);
    end
    j ← j + 1;
until (j = m) or (H[i] is empty);
return (false);
```

Applications of Hashing

ContainsDuplicate

Return true if there is a duplicate element in an array.

```
ContainsDuplicate(A[ ],n)
HashTable.Init();
for i ← 1 to n do
    if (HashTable.Member(A[i])) then return true;
    else HashTable.Insert(A[i], true);
end
return false;
```