ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Special Topics on Algorithms

Fall 2023

Average Case Analysis

Vangelis Markakis

Ioannis Milis

Outline

- Introductory examples
  - FINDMAX
  - BINARY COUNTER INCREMENT
  - INSERTION SORT
- QUICKSORT
- BINARY SEARCH TREES
- HASHING

Outline

- Worst case examples may often not appear in practice
- Performing an average case analysis can be meaningful
- But: for such an analysis, we need an assumption on the input
  - input = random data according to some probability distribution
- Usually analysis is done by assuming a uniform distribution on all possible configurations of the input

Finding the MAX

```
Algorithm max(A[1..n])

Input: An array of n elements A[1..n]
Output: the position of the maximum element
max= A[1], position=1
for i=2 to n
     if A[i] > max  max=A[i], position=i  (*)
return position
```

Complexity:

- Number of steps
  Worst case = average case = O(n) (we have to execute the for loop)

- What about the commands inside the loop?

- Let T(n) = # of assignments = number of times (*) is executed
  Best case: T(n) = 1
  Worst case: T(n) = n
  Average case: ?

Finding the MAX (# assignments)

```
Algorithm max(A[1..n])

Input: An array of n elements A[1..n]
Output: the position of the maximum element
max= A[1], position=1
for i=2 to n
     if A[i] > max  max=A[i], position=i  (*)
return position
```

Average case analysis: Need a probabilistic assumption on the data

- There are n! possible orderings of n numbers: Natural to assume all orderings are equiprobable
  - true if each number has been picked independently from the uniform probability distribution
- Define a random variable for each iteration i, call it T_i
- $T_i = 1$, if assignment in the i^{th} iteration, 0 otherwise
- Pr [assignment in the i^{th} iteration] = Pr [$A[j] < A[i], \forall j<i$] = $1/i$
- Hence $Pr[T_i = 1] = 1/i$

Finding the MAX (# assignments)

```
Algorithm max(A[1..n])

Input: An array of n elements A[1..n]
Output: the position of the maximum element
max= A[1], position=1
for i=2 to n
    if A[i] > max  max=A[i], position=i  (*)
return position
```

Average case analysis: Need a probabilistic assumption on the data

- Pr [no assignment in the i^{th} iteration] = Pr [$\exists$ j<i : A[j] > A[i]] = (i-1)/I
- Expected value of T_i: $1\cdot \Pr[T_i = 1] + 0\cdot \Pr[T_i = 0]$

$$\mathbf{E}[T_i] = 1\frac{1}{i} + 0\frac{i-1}{i} = \frac{1}{i}$$

Finding the MAX (# assignments)

```
Algorithm max(A[1..n])

Input: An array of n elements A[1..n]
Output: the position of the maximum element
max= A[1], position=1
for i=2 to n
     if A[i] > max  max=A[i], position=i  (*)
return position
```

Average case analysis:

T(n) : total # of assignments $T(n) = \sum_{i=1}^{n} T_i$

$$\mathbf{E}[T(n)] = \mathbf{E}\left[\sum_{i=1}^{n} T_i\right] = \sum_{i=1}^{n} \mathbf{E}[T_i] = \sum_{i=1}^{n} \frac{1}{i} = H_n = O(\log n)$$

Linearity of Expectation

Incrementing a binary counter

Problem: Increment a binary counter by 1

Input: An array A of k bits, A[0], A[1], ..., A[k-1], representing the counter of value $x, \ 0 \leq x \leq n : x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$ (k= $\lfloor \log n \rfloor$ +1)

Output: Increase the counter by 1

```
INCREMENT(A);
i = 0;
while i < k and A[i] = 1 do
    A[i] = 0;
    i = i+1;
if i < k then A[i] = 1 else overflow
```

Complexity: We care for # of bit flips

- Best case: O(1), the LSB is 0 and only this is flipped
- Worst case: O(k), that is O(log n); all the bits are flipped
- Average case: ?

Incrementing a binary counter

The # of bit flips depends on the value of the counter

A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	**value**
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	2
0	0	0	0	0	0	1	1	3
0	0	0	0	0	1	0	0	4
0	0	0	0	0	1	0	1	5
0	0	0	0	0	1	1	0	6
0	0	0	0	0	1	1	1	7
0	0	0	0	1	0	0	0	8
0	0	0	0	1	0	0	1	9
0	0	0	0	1	0	1	0	10
0	0	0	0	1	0	1	1	11
0	0	0	0	1	1	0	0	12
0	0	0	0	1	1	0	1	13
0	0	0	0	1	1	1	0	14
0	0	0	0	1	1	1	1	15
0	0	0	1	0	0	0	0	16

Incrementing a binary counter

Assumption for average case analysis: All numbers with k bits equiprobable

Binary Number	Bit flips (x_i)	Probability (p_i)
......0	1	1/2
......01	2	1/4
....011	3	1/8
.	.	.
.	.	.
$0\underbrace{111\ldots111}_{i-1}$	i	$1/2^i$

Let
X = #bit flips

$$E(X)=\sum_{i=1}^{k} p_i x_i = \sum_{i=1}^{k} i\frac{1}{2^i} \le \sum_{i=0}^{\infty} i\frac{1}{2^i} = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2 = O(1) \quad !$$

InsertionSort

```
Algorithm SelectionSort (A[1..n])
A[0] := -∞ //only for technical convenience
for i:=2 to n do
    j := i;
    while A[j]<A[j-1] do
        swap (A[j], A[j-1]);
        j := j-1;
```

T(n) = # of comparisons

Best case: Array already sorted

1 comparison per iteration

$T(n) = n-1$

Worst case: Array sorted in reverse order

The i^{th} iteration requires i comparisons

$$T(n) = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 \sim O(n^2)$$

Average case: ?

InsertionSort

i^{th} iteration

Final position of `A[i]`	: i	i-1	...,	2	1
# of comparisons	: 1	2	...,	i-1	i
`Pr[A[i] goes to position j]`	: $\frac{1}{i}$	$\frac{1}{i}$	...,	$\frac{1}{i}$	$\frac{1}{i}$

- **Assumption for avg case analysis**: All permutations of the n numbers are equiprobable
- Expected number of comparisons in the i^{th} iteration =

$$T_i = \sum_{k=1}^{i} k \cdot \frac{1}{i} = \frac{i(i+1)}{2} \cdot \frac{1}{i} = \frac{i+1}{2}$$

InsertionSort

Summing over all iterations

Expected number of comparisons:

$$E[T(n)] = E\left[\sum_{i=2}^{n} T_i\right] = \sum_{i=2}^{n} E[T_i] =$$

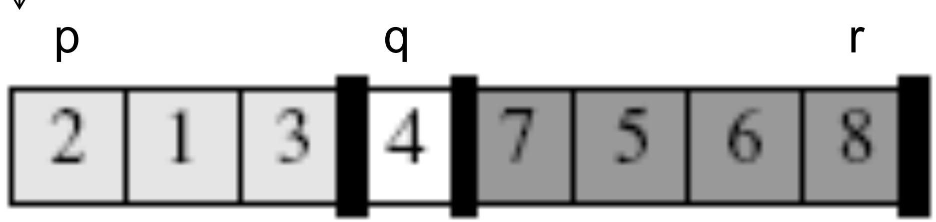
$$= \sum_{i=2}^{n} \frac{i+1}{2} = \frac{1}{2}\left(\frac{(n+1)(n+2)}{2} - 3\right)$$

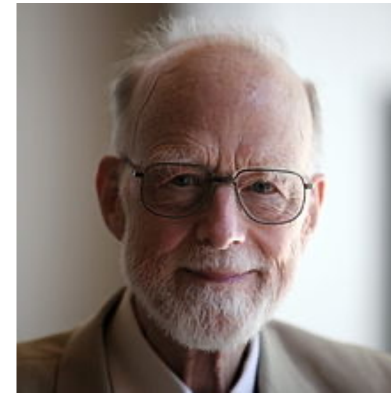
$$= \frac{n(n+1)}{4} + \frac{n-2}{2}$$

- Around $n^2/4$
- Almost half of the worst case, but again $O(n^2)$
- Here average case does not provide significant improvements

Quick Sort

```
QuickSort (A, p, r)
if p < r:
    select pivot x;
    q = Partition (A,p,r)
    //split A into A[p,q-1],A[q+1,r];
    // A[i] ≤ x, p ≤ i ≤ q-1
    // x ≤ A[i], q+1 ≤ i ≤ r
    // q is the final position of x
    QuickSort (A[p,q-1]);
    QuickSort (A[q+1,r]);
```





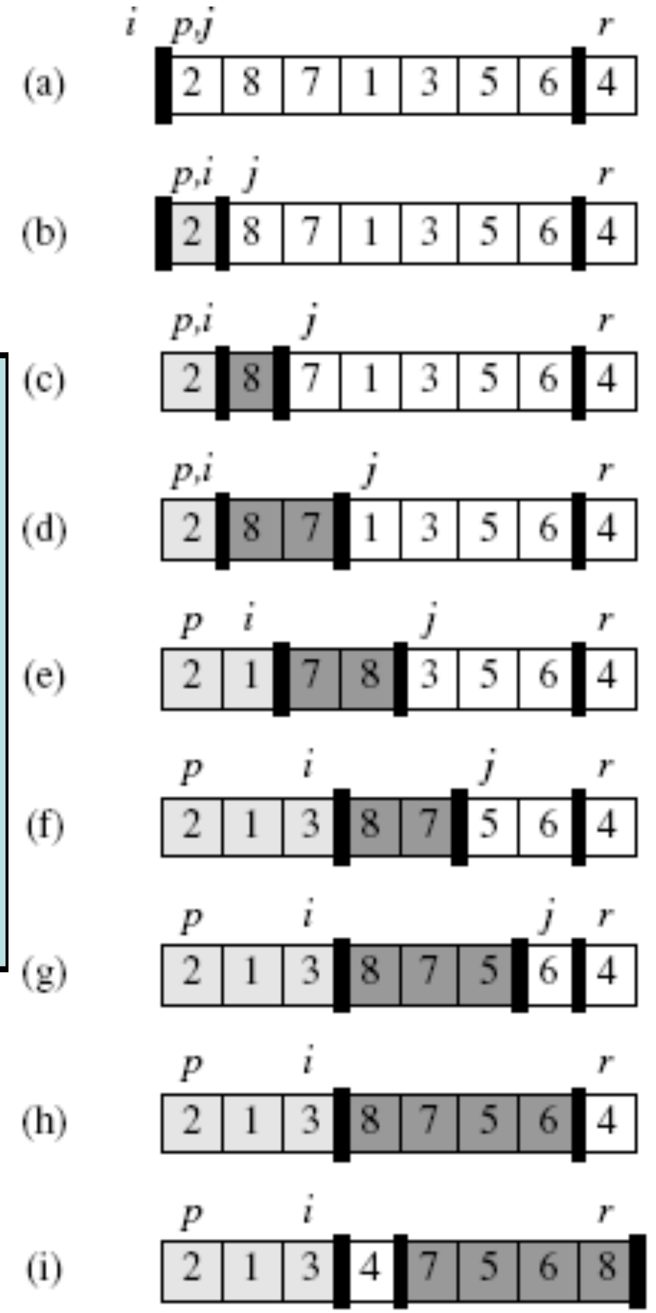
T. Hoare, 1960

R. Sedgewick
Ph.D. thesis, 1975

Quick Sort

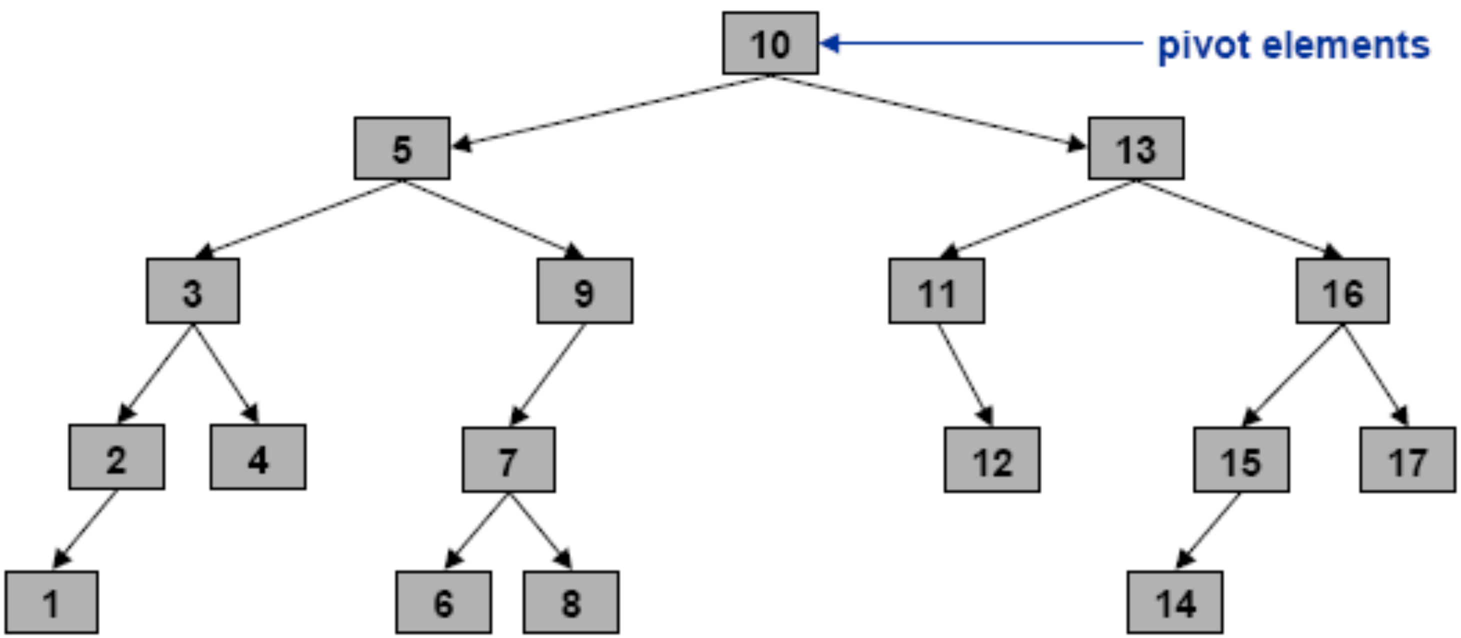
```
Partition (A, p, r)
x=A[r]
i=p-1
for j=p to r-1:
    if A[j]≤x: i=i+1
                swap(A[i],A[j])
swap(A[i+1],A[r])
q=i+1
return q
```

Complexity of Partition: O(n)
(n-1 iterations)



Quick Sort

7	6	12	3	11	8	2	1	15	13	17	5	16	14	9	4	10
7	6	4	3	9	8	2	1	5	10	17	15	16	14	11	12	13
1	2	4	3	5	8	6	7	9	10	12	11	13	14	15	17	16
1	2	3	4	5	8	6	7	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17



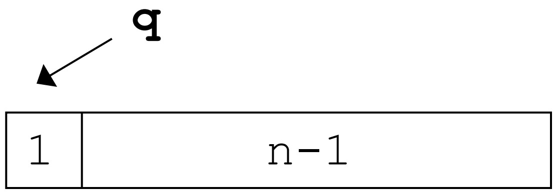
Quick Sort

```
QuickSort (A, p, r)
if p < r:
   select pivot x;
   q = Partition (A,p,r)
   //split A into A[p,q-1],A[q+1,r];
   // A[i] ≤ x, p ≤ i ≤ q-1
   // x ≤ A[i], q+1 ≤ i ≤ r
   // q is the final position of x
   QuickSort (A[p,q-1]);
   QuickSort (A[q+1,r]);
```

- Difficult to control the possible divisions into subproblems
  Partition (A,q,r): O(n), with n = r – p + 1
- Combining the solutions of the subproblems: easy,
  Nothing to do !
- For simplicity, suppose p=1, r=n

Complexity : $T(n) = T(q-1) + T(n-q) + O(n)$???

Quick Sort - Worst Case

- When we partition into  in every step

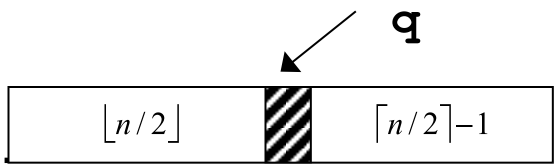
- Pivot is the min (or the max)

$$T(n) = T(n-1) + n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} = O(n^2)$$

If we choose as pivot = A[r], when does the worst case occur?

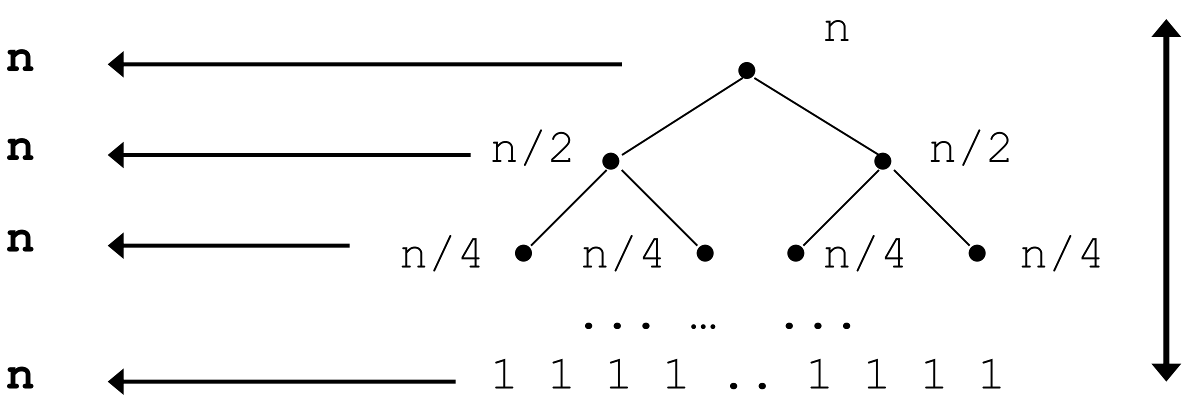
Quick Sort - Best Case

- Partition into
- Pivot is the median



in every step

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$$



log n
(depth of recursion)

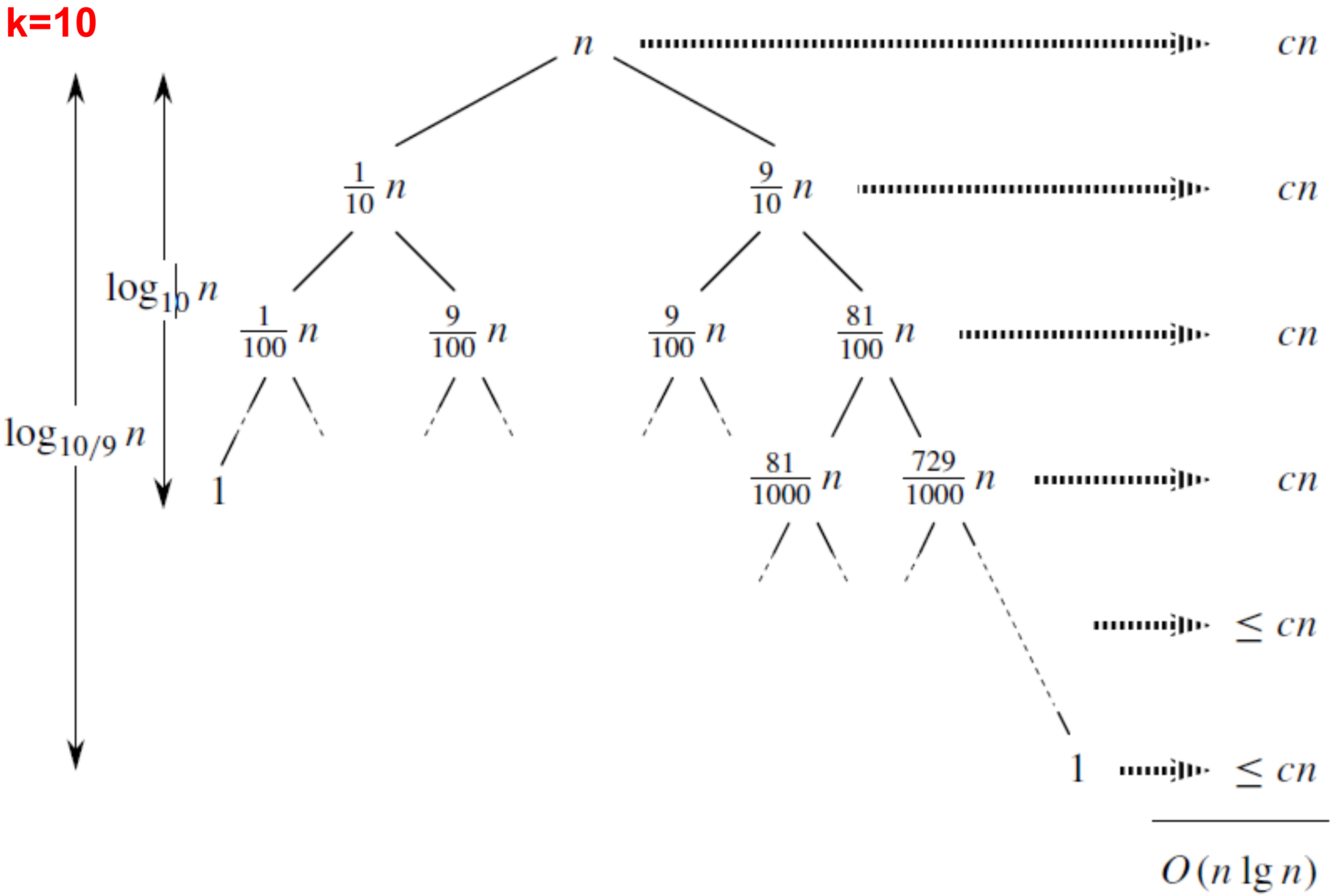
Quick Sort - Best Case

- Quicksort behaves well even if the partitioning at every step is quite unbalanced
- For example, suppose we partition into 90/10 proportions every time
- Or generally partition into $\le n\frac{k-1}{k}$ | $\ge \frac{n}{k}$ in every step for some constant k

$$T(n) = T\left(\frac{k-1}{k}n\right) + T\left(\frac{1}{k}n\right) + O(n)$$

- Depth of recursion = $\log_a n$, a = k/k-1 $\Rightarrow \log_a n = O(\log n)$
- $\Rightarrow$ T(n) = O(nlogn)
- True for any partitioning with constant k (independent of n)

Quick Sort - Best Case



Quick Sort - Average Case

Assumptions:

- All permutations of the n numbers are equiprobable
- All numbers of A[1..n] are distinct

Then, the pivot can end up in any position equiprobably

- **q:** final position of the pivot after running Partition
- Pr[Partition(A, p, r) = q] = 1/n for every q
- Complexity if pivot ends up at q: $T(q-1)+T(n-q)+(n-1)$

- Hence, expected complexity:

$$\mathrm{T(n)} = \sum_{q=1}^{n} \frac{1}{n}[T(q-1)+T(n-q)+(n-1)]$$

Quick Sort - Average Case

$$T(n) = \sum_{q=1}^{n} \frac{1}{n}[T(q-1) + T(n-q) + (n-1)]$$

$$= \frac{1}{n}\sum_{q=1}^{n}[T(q-1) + T(n-q)] + \frac{1}{n}\sum_{q=1}^{n}(n-1)$$

$$= \frac{1}{n}\sum_{q=1}^{n}[T(q-1) + T(n-q)] + \frac{n(n-1)}{n}$$

$$= \frac{2}{n}\sum_{q=1}^{n}T(q-1) + (n-1)$$

n-q: 0,1,2,..,n-2,n-1

q-1: n-1,n-2,…2,1,0

Quick Sort - Average Case

$$T(n) = \frac{2}{n}\sum_{q=1}^{n} T(q-1) + n - 1 \qquad (1)$$

(1) * n:

$$nT(n) = 2\sum_{q=1}^{n} T(q-1) + n(n-1) \qquad (2)$$

(2) for n-1:

$$(n-1)T(n-1) = 2\sum_{q=1}^{n-1} T(q-1) + (n-1)(n-2) \qquad (3)$$

---

(2) – (3):

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1) \Rightarrow$$

$$nT(n) = (n+1)T(n-1) + 2(n-1) \Rightarrow$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

Quick Sort - Average Case

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

$$\boxed{\text{Έστω} \quad \alpha_n = \frac{T(n)}{n+1}, \alpha_0 = 0}$$

$$\alpha_n = \alpha_{n-1} + \frac{2(n-1)}{n(n+1)} = \alpha_{n-2} + \frac{2(n-2)}{(n-1)n} + \frac{2(n-1)}{n(n+1)} = \ldots = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)}$$

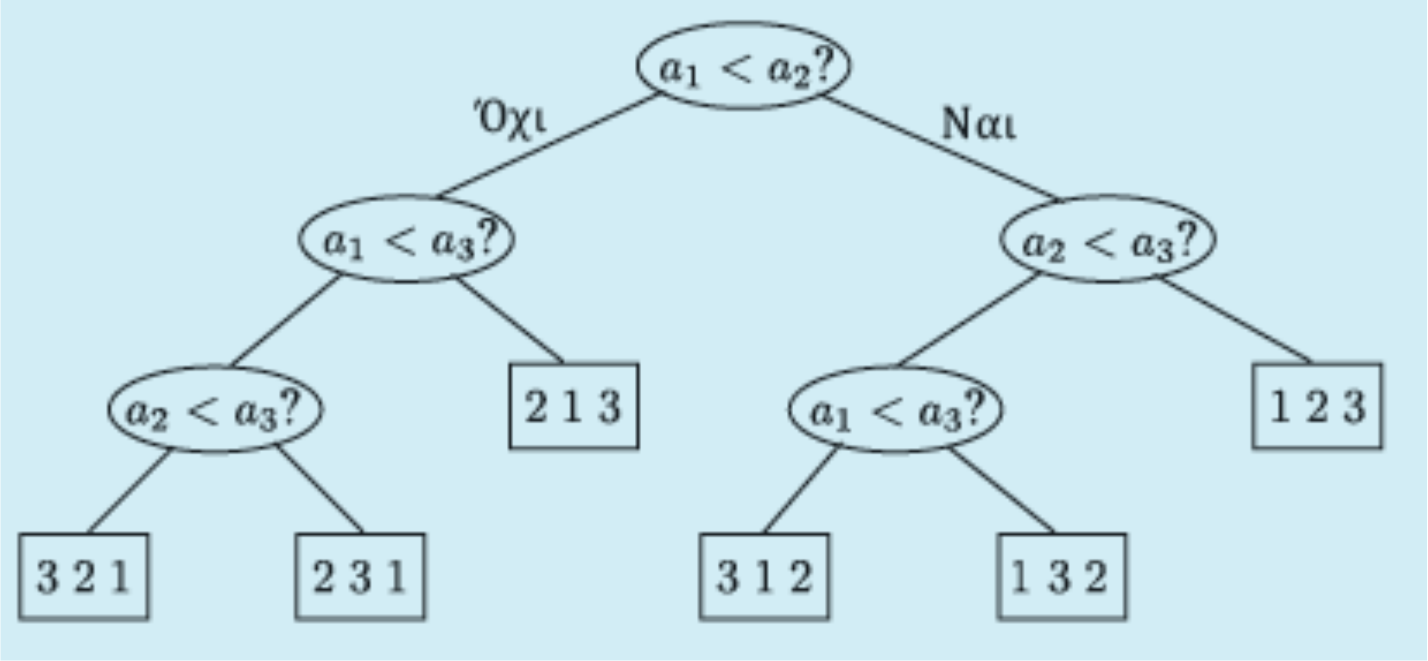
$$= 2\sum_{i=1}^{n} \frac{i-1}{i(i+1)} \leq 2\sum_{i=1}^{n} \frac{i}{i(i+1)} = 2\sum_{i=1}^{n} \frac{1}{i+1} \leq 2\sum_{i=1}^{n} \frac{1}{i} = 2H_n$$

$$T(n) = (n+1)\alpha_n \leq (n+1) \cdot 2H_n = O(n \log n)$$

Lower bound for sorting

A lower bound applicable for all algorithms that use comparisons

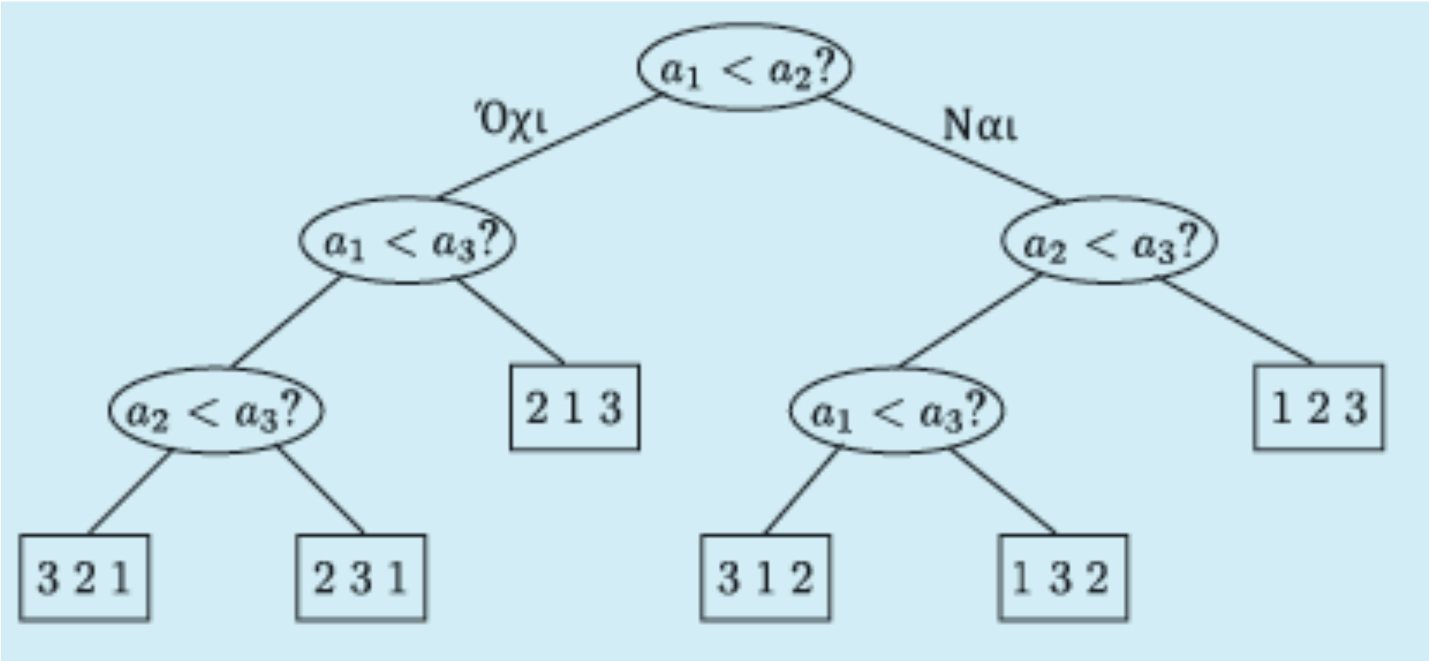
- Pairwise comparisons
- Every such sorting algorithm corresponds to a binary decision tree



Tree leaves = possible orderings (permutations)

Complexity = tree height

Lower bound for sorting



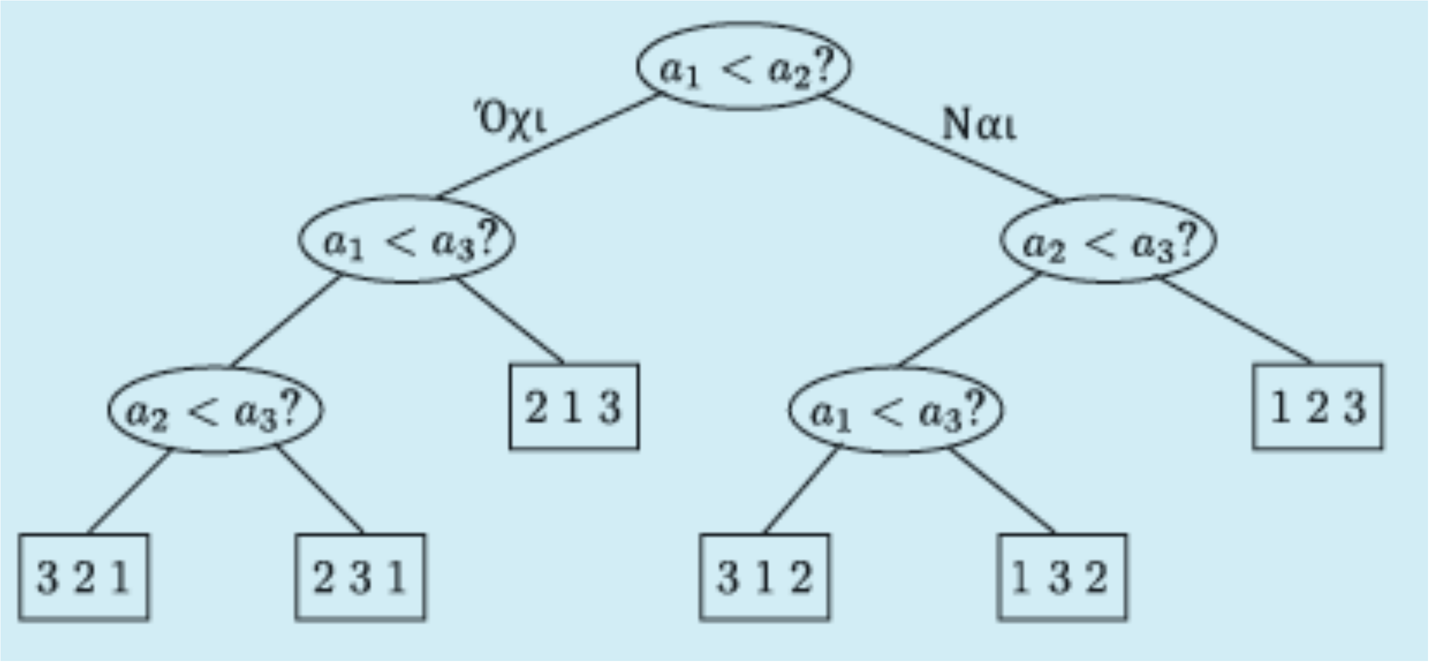
\# leaves ≥ # possible permutations = n!

No permutation can be absent

•If yes, what would the algorithm answer if the input corresponded to such a permutation?

Let d = tree height, d = Ω(?)

Lower bound for sorting



Every binary tree of height d has at most 2^d leaves

Hence:

$$n! \leq 2^d \Rightarrow d \geq \log(n!)$$

Lower bound for sorting

$$d \geq \log(n!) \quad = \quad \log\left(1 \cdot 2 \cdot \ldots \cdot \frac{n}{2} \cdot \left(\frac{n}{2}+1\right) \cdot \left(\frac{n}{2}+2\right) \cdot \ldots \cdot n\right) \geq \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right)$$

$$= \frac{n}{2}\log\left(\frac{n}{2}\right) = \frac{n}{2}(\log n - \log 2) = \frac{n}{2}(\log n - 1) = \Omega(n \log n)$$

OR:

$$d \geq \log(n!) \quad \overset{\textbf{Stirling}}{\geq} \quad \log\left(\frac{n}{e}\right)^{n} = n\log\left(\frac{n}{e}\right) = n(\log n - \log e)$$

$$= n\log n - n\log e = \Omega(n \log n)$$

Thus, any algorithm based on comparisons must have complexity at least **Ω(nlogn)**

Median and Selection

SELECTION

I: n distinct numbers, a parameter k, $1 \leq k \leq n$

Q: the k-th largest element

k = n: find minimum,   k = 1: find maximum

$k = \lfloor (n+1)/2 \rfloor$ → MEDIAN (half the elements smaller, the other half bigger)

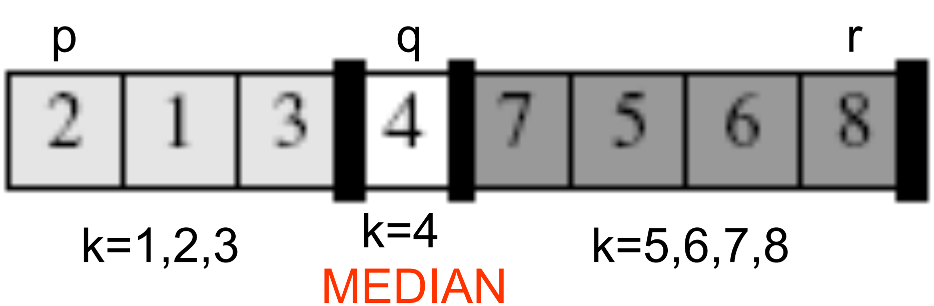
k odd: x x x **M** x x x   (n=7, k= 4)

k even: x x x **M** **x** x x x (n=8, k= 4 - lower median)

Obvious algorithm: O(n log n) – why?

Selection – Divide and Conquer

```
Select (A, p, r, k)
if p = r: return A[p]
select pivot x;
q = Partition (A,p,r)
//split A into A[p,q-1],A[q+1,r];
// A[i] ≤ x, p ≤ i ≤ q-1
// x ≤ A[i], q+1 ≤ i ≤ r
// q is the final position of x
m=q-p+1
if k=m: then return A[q]
 else: if k < m Select(A,p,q-1,k)
              else: Select (A,q+1,r, k-m)
```



Selection – Divide and Conquer

Selection vs. Quicksort

- Quicksort: divide and examine recursively both segments of the array
- Selection: divide and examine recursively only one segment

If we always end up at the largest segment:

$$\text{Complexity}: \quad \mathrm{T(n)} \leq T(\max\{q-1, n-q\}) + (n-1)$$

Best case: T(n) = T(n/2) + O(n) $\Rightarrow$ O(n)

Worst case: T(n) = T(n-1) + O(n) $\Rightarrow$ O(n^2)

Average case: ?

Selection - D&C Average Case

Assumptions:

- All permutations of the n numbers are equiprobable
- All numbers of A[1..n] are distinct

Then, the pivot can end up in any position equiprobably

- **q:** final position of the pivot after running Partition
- Pr[Partition(A, p, r) = q] = 1/n for every q

$$\mathrm{T(n)} \le T(\max\{q-1, n-q\}) + (n-1)$$

- **Expected complexity:**

$$T(n) \quad \le \quad \sum_{q=1}^{n} \frac{1}{\mathrm{n}} \cdot [T(\max\{q-1, n-q\}) + (n-1)]$$

- **T(n) = O(n)** (similar analysis with Quicksort)

AVERAGE CASE ANALYSIS

	WORST	AVERAGE
Finding the max (# of asignments)	$O(n)$	$O(\log n)$
Increment a binary counter	$O(n)$	$O(1)$
Insertionsort	$O(n^2)$	$O(n^2)$
Quicksort	$O(n^2)$	$O(n\log n)$
Selection	$O(n^2)$	$O(n)$

ΕΚΤΟΣ ΥΛΗΣ

Average case analysis for Binary Search Trees and Hashing

DICTIONARY ADT

A data structure for maintaining a dynamic set S

- A data set that keeps changing (items being inserted or deleted over time)
- Each item comes with a key

Supports the following operations

- **`SEARCH (S,k)`** `//`search according to a key k
- **`INSERT (S,x)`** `//`insert an element x
- **`DELETE (S,k)`** `//`delete an element with key k

<u>NAIVE IMPLEMENTATIONS:</u>

- Arrays or lists: O(n) both for average and worst case

DICTIONARY ADT

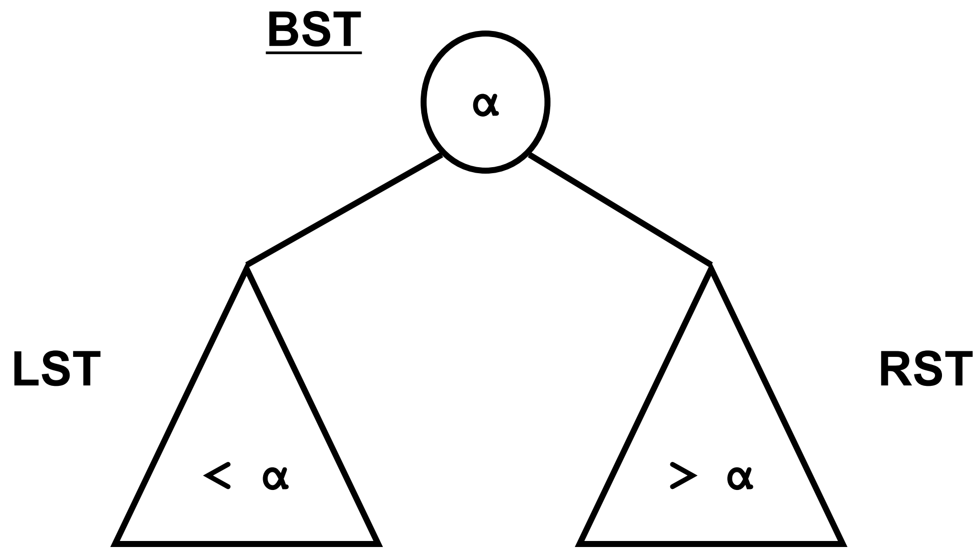
- **`SEARCH (S,k)`** //search according to a key k
- **`INSERT (S,x)`** //insert an element x
- **`DELETE (S,k)`** //delete an element with key k

<u>BETTER IMPLEMENTATIONS:</u>

- Binary Search Trees (BSTs):
  - O(n) worst case
  - O(logn) average case
- Balanced BSTs (AVL, Red-Black, 2-3-4 trees):
  - O(logn) worst case
- Splay trees:
  - O(logn) amortized
- Hash Tables
  - O(n) worst case
  - O(1) average case (under reasonable assumptions)

BINARY SEARCH TREES (BSTs)

An implementation of Dictionary



BSTs - Complexity of operations

The complexity of any operation is $O(p_k)$ where
p_k = depth of operation= <u>path length</u> from root to a node k

$$\max_{k \in S}(p_k) = h = \text{height of S}$$

<u>Best case</u>: O(log n) ***balanced tree***

<u>Worst case</u>: O(n) ***chain***

<u>Average case</u>: ?

BSTs - Average case

- Suppose a BST is built by inserting consecutively n distinct elements (assume integer keys)
- Assume all n! permutations of the keys equiprobable
- Assume we have a successful search operation (and equiprobable to search for any of the keys)
- Unsuccessful search costs just 1 more

`P(i)` = average path length in a BST of `i` nodes (average # of nodes on a path from the root to any node – not only to the leaves)

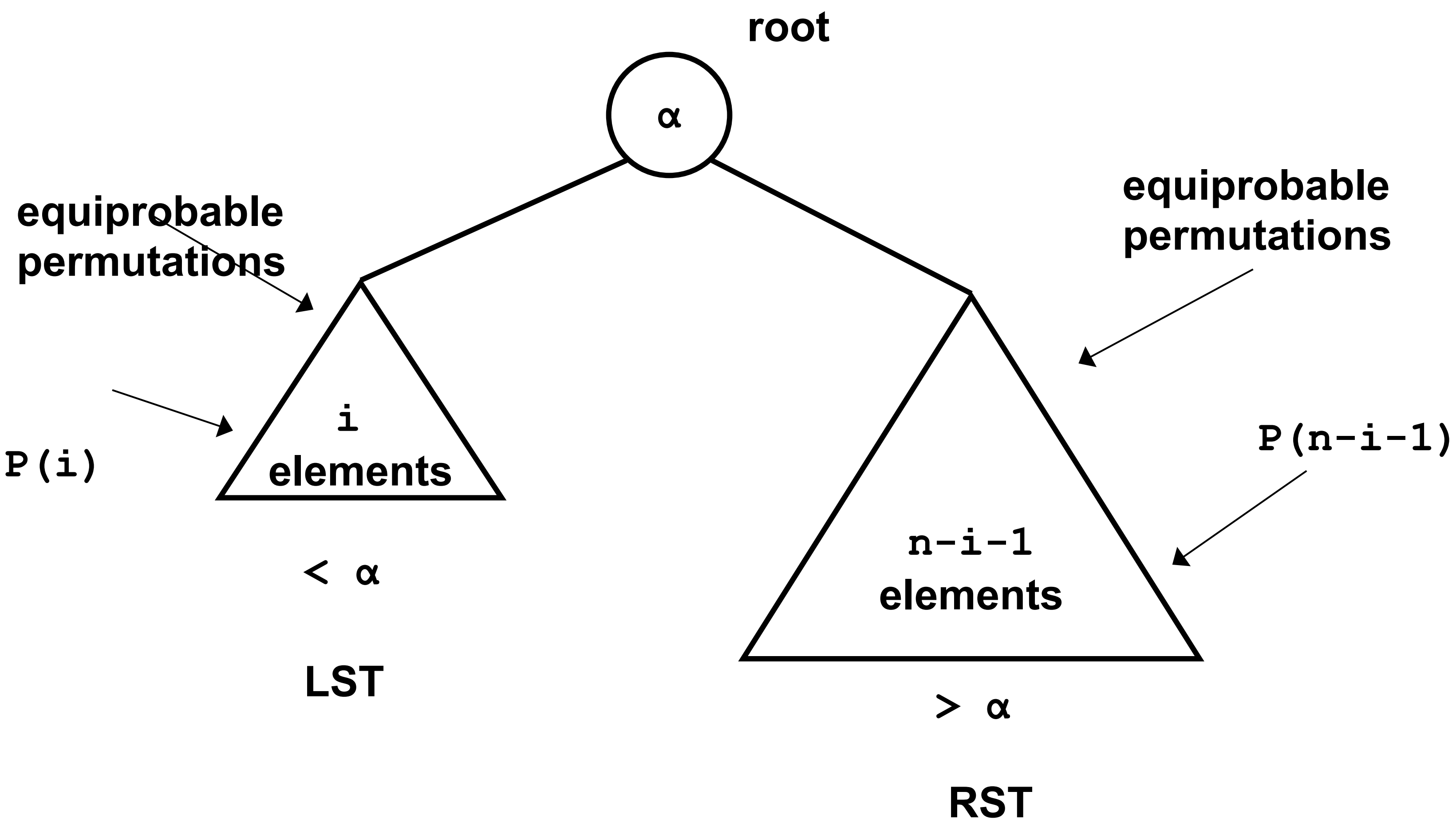
`P(0) = 0`

`P(1) = 1`

We want to estimate `P(n)`

α: the first element inserted = the root of the BST
equiprobable to be the 1^{st}, 2^{nd}, ..., i^{th}, ..., n^{th} in the sorted order of the `n` elements

BSTs - Average case



BSTs - Average case

For a given i:

P(n,i) = Average path length when we search key x, if the LST has i nodes:

- $x = a : P(n,i) = 1$
- $x \in LST : P(n,i) = 1 + P(i)$
- $x \in RST : P(n,i) = 1 + P(n-i-1)$

$$\boxed{\text{Pr[searching any of the n elements]} = \frac{1}{n} \ \text{(equiprobable)}}$$

$$P(n,i) = \frac{1}{n} \cdot 1 + \frac{i}{n}[1 + P(i)] + \frac{(n-i-1)}{n}\ [1 + P(n-i-1)]$$

$$= \frac{1 + i + (n-i-1)}{n} + \frac{i}{n}P(i) + \frac{n-i-1}{n}P(n-i-1)$$

$$= 1 + \frac{i}{n}P(i) + \frac{n-i-1}{n}P(n-i-1)$$

BSTs - Average case

Recall: we care for P(n)

$$P(n) = \sum_{i=0}^{n-1} \Pr[LST \text{ has i nodes}] \cdot P(n,i)$$

$$\Pr[LST \text{ has i nodes}] = \Pr\left[\begin{array}{l}\alpha \text{ is the } (\mathrm{i}+1)^{\text{th}} \text{ element in the} \\ \text{sorted order of the n elements}\end{array}\right] = \frac{1}{n}$$

Hence: $$\boxed{P(n) = \frac{1}{n}\sum_{i=0}^{n-1} P(n,i)}$$

BSTs - Average case

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1} P(n,i)$$

$$= \frac{1}{n}\left\{\sum_{i=0}^{n-1}\left[1 + \frac{i}{n}P(i) + \frac{n-i-1}{n}P(n-i-1)\right]\right\}$$

$$= 1 + \frac{1}{n^2}\sum_{i=0}^{n-1}\left[iP(i) + (n-i-1)\cdot P(n-i-1)\right]$$

$$\boxed{P(n) = 1 + \frac{2}{n^2}\sum_{i=0}^{n-1} iP(i)}$$

We shall show that $P(n) \le 1 + 4\log n$ **(by induction on n)**

BSTs - Average case

$$P(n) \le 1 + 4\log n$$

Induction basis

$$n = 1:\ P(1) = 1,\ 1 + 4\log 1 = 1$$

Induction hypothesis

$$P(i) \le 1 + 4\log i,\ \forall i < n$$

BSTs - Average case

<u>Inductive step</u>

$$P(n) = 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} iP(i)$$

$$\leq 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} i(1 + 4\log i)$$

$$\leq 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} 4i\log i + \frac{2}{n^2}\boxed{\sum_{i=1}^{n-1} i} \quad \frac{n(n-1)}{2} \leq \frac{n^2}{2}$$

$$\leq 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} 4i\log i + \frac{2}{n^2}\frac{n^2}{2} \Rightarrow$$

$$P(n) \leq 2 + \frac{8}{n^2}\sum_{i=1}^{n-1} i\log i$$

BSTs - Average case

$$\sum_{i=1}^{n-1} i\log i = \sum_{i=1}^{\left\lceil \frac{n}{2} \right\rceil - 1} i\log i + \sum_{i=\left\lceil \frac{n}{2} \right\rceil}^{n-1} i\log i$$

$$\leq \sum_{i=1}^{\left\lceil \frac{n}{2} \right\rceil - 1} i\log \frac{n}{2} + \sum_{i=\left\lceil \frac{n}{2} \right\rceil}^{n-1} i\log n$$

$$\leq \frac{n^2}{8}\log\frac{n}{2} + \frac{3n^2}{8}\log n$$

$$= \frac{n^2}{8}(\log n - 1) + \frac{3n^2}{8}\log n$$

$$= \frac{n^2}{2}\log n - \frac{n^2}{8}$$

BSTs - Average case

Thus,

$$P(n) \le 2 + \frac{8}{n^2}\sum_{i=1}^{n-1} i \log i$$

$$\boxed{\sum_{i=1}^{n-1} i \log i \le \frac{n^2}{2}\log n - \frac{n^2}{8}}$$

$$\le 2 + \frac{8}{n^2}\left(\frac{n^2}{2}\log n - \frac{n^2}{8}\right)$$

$$= 2 + 4\log n - 1$$

$$= 1 + 4\log n \Rightarrow$$

$$P(n) = O(\log n)$$

HASH TABLES

[CLRS 11.1, 11.2, 11.4]

An alterative implementation of DICTIONARY ADT

Recall we care to implement the operations

- **`SEARCH (S,k)`** //search according to a key k
- **`INSERT (S,x)`** //insert an element x
- **`DELETE (S,k)`** //delete an element with key k

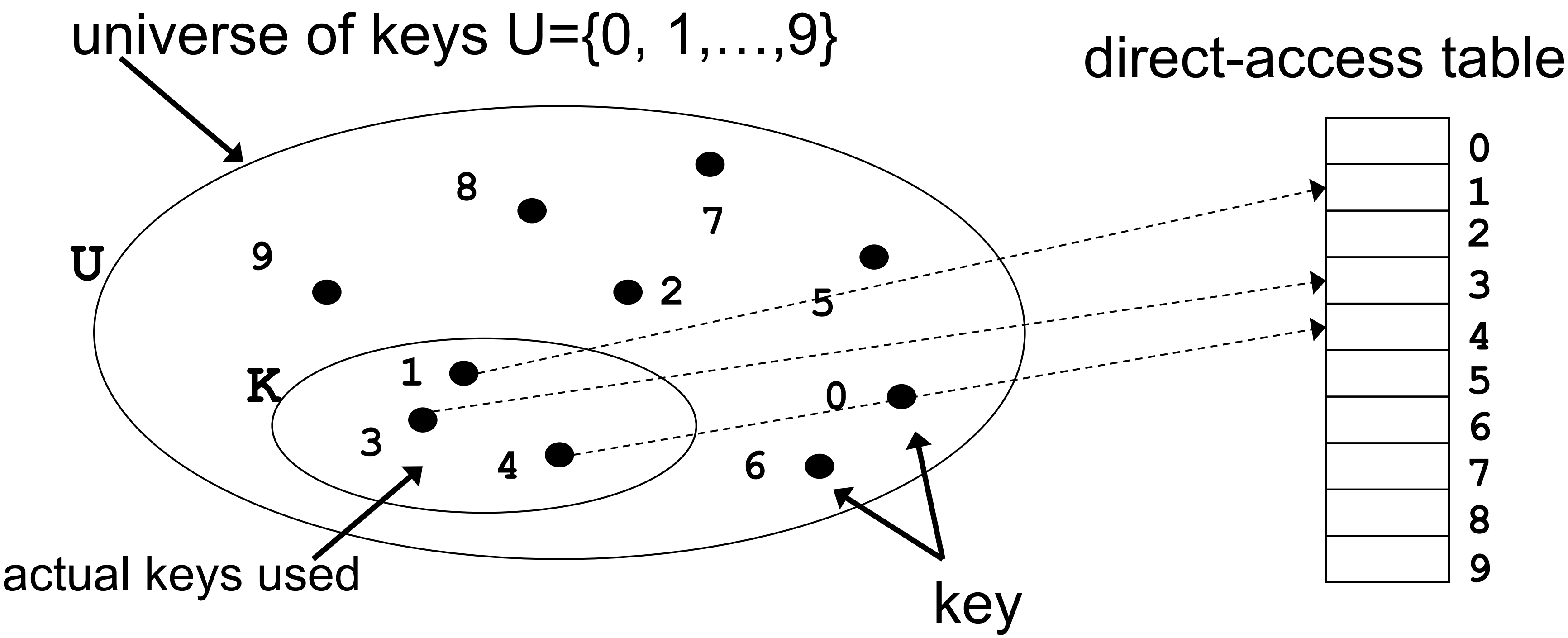
2 main approaches used in hashing:

1. Chaining
2. Open addressing

Direct Addressing

- We want to store objects that have a key field
- Let U = {0,1,2,3,…} the set of all possible key numbers – assume integer keys
- Allocate an array that has a position for each key `T[0..|U|-1]`
- `T[k]` corresponds to (the element of) key `k`
- Operations:
  - `search(k)`: return `T[k]`
  - `insert(x)`: `T[x.key]=x`
  - `delete(k)`: `T[k]=null`
- Complexity: O(1) in worst case for all operations

Direct Addressing



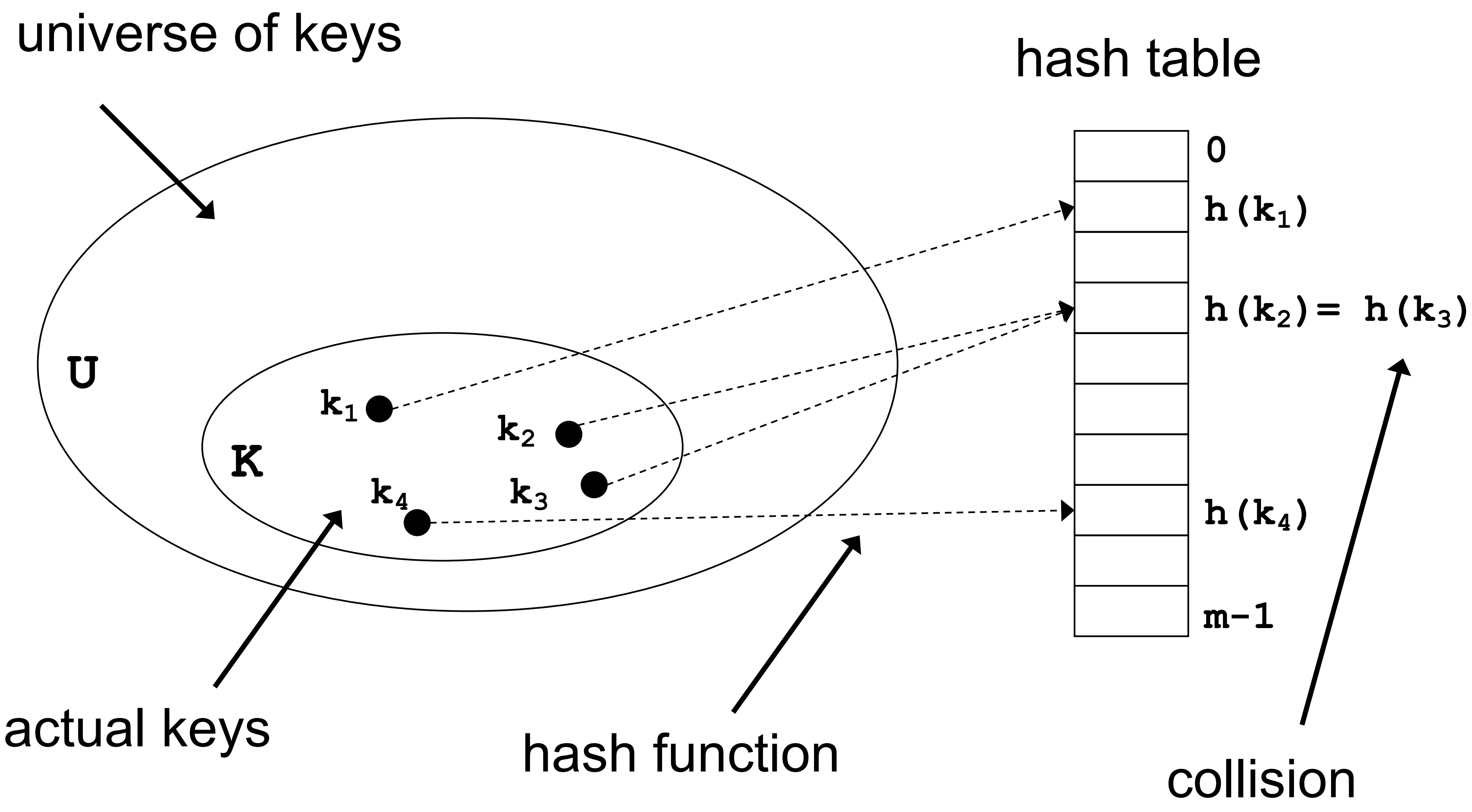
Problems:

- We may have objects with the same key
- Not all possible keys are used, we waste too much memory if U is huge
- actually stored keys $|K| << |U|$

Hashing

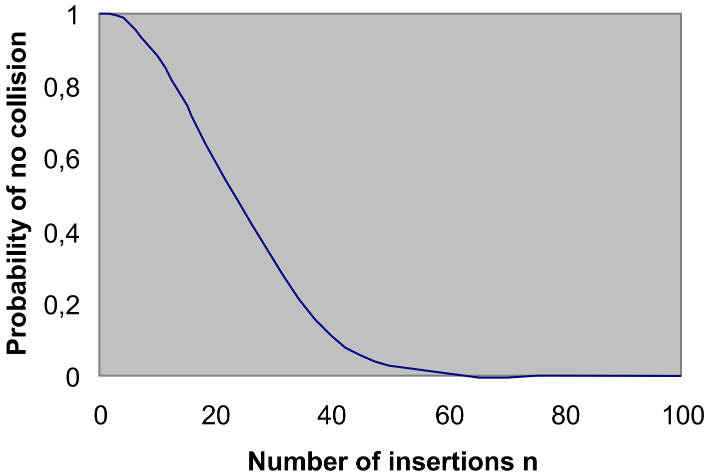
- Map the universe U of keys onto a small range of integers
- Hash function $h\colon U \rightarrow \{0,1,...,m-1\}$, for some integer m
- Use an array of size m: $T[0...m-1]$ ($m << |U|$)
- Hash collision: when $h(k)=h(k')$ for $k \neq k'$
- Goal: Obtain a hash function that is
  - cheap to evaluate (e.g., h(k) = ak mod m)
  - assumption: $h(k)$ is computed in $\Theta(1)$
  - minimizes collisions
- <u>n = # of stored elements</u>

Hashing


universe of keys
hash table
0
h(k1)
h(k2)= h(k3)
h(k4)
m-1
U
K
k1
k2
k3
k4
actual keys
hash function
collision

Collisions

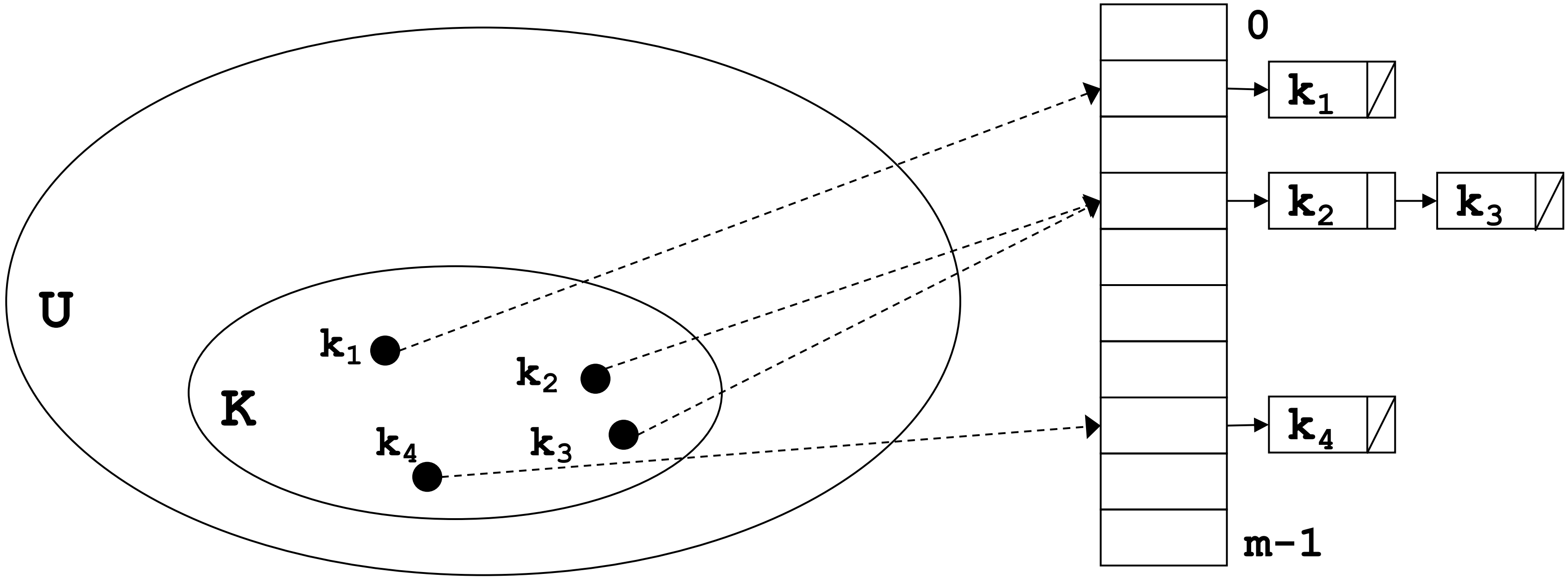
- No matter how good the hash function is, the probability of no collision is very low even for small n (birthday paradox)
- For `m=365` and n $\geq$`50` this probability goes to `0`



- How to treat hash collisions when they occur?

Chaining

Put all keys that hash to the same integer in a linked list



Use array of m lists: `T[0], T[1], T[2],...,T[m-1]`

Chaining – worst case

- DICTIONARY implementation:
  - `search(k)`: search for an element with key `k` in the list `T[h(k)]`
  - `insert(x)`: put element x **at the front** of list `T[h(x.key)]` (we do not keep the lists sorted)
  - `delete(k)`: delete element with key `k` from list `T[h(k)]`

- Complexity
  - `search(k)`: $\Theta(|$`T([h(k)]`$|)$
  - `insert(x)`: $\Theta(1)$ (no check if element x is already present)
  - `delete(k)`: $\Theta(|$`T([h(k)]`$|)$

- Worst case: all keys are hashed onto the same slot
  - `search(k)`: $\Theta(n)$
  - `insert(x)`: $\Theta(1)$ (no check if element x is already present)
  - `delete(k)`: $\Theta(n)$

Chaining - Average case

- Assumption: uniform hashing
  - each key is equally likely hashed into any of the m slots, independently of where any other element has hashed to
- Filling degree of hash table `T`: $\alpha(n,m) = n/m$
  - the average length of list `T[j]` is α
- Expected number of elements examined in `T[h(k)]` to search key `k`?

  Distinguish between
  - unsuccessful search
  - successful search

Chaining - Average case

Unsuccessful search

- Expected time to search for key `k`
  = expected time to search till the end of list `T[h(k)]`

- `T[h(k)]` has expected length α

- The computation of `h(k)` takes Θ(1) time

  that is a total of **$\Theta(1+\alpha)$**

Chaining - Average case

Successful search

- Suppose keys were inserted in the order $k_1, k_2, \ldots, k_n$
- k_i: the i^{th} inserted key
- $A(k_i)$: the expected time to search k_i

$A(k_i) = 1 +$ average # of keys inserted in $T[h(k_i)]$ after k_i was inserted

- Due to uniform hashing: $A(k_i) = 1 + \boxed{\sum_{j=i+1}^{n} \frac{1}{m}} = 1 + \frac{n-i}{m}$

# of keys inserted in T[h(k_i)] after k_i

- average over all n inserted keys $E[A] = \frac{1}{n} \sum_{i=1}^{n} A(k_i)$

Chaining - Average case

Successful search

$$E(A) = \frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{n-i}{m}\right) = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) = 1+\frac{1}{nm}\left[n^2-\sum_{i=1}^{n} i\right]$$

$$= 1+\frac{1}{nm}\left[n^2-\frac{n(n+1)}{2}\right] = 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{\alpha}{2n},$$

- Better than in the unsuccessful case
- But overall $\Theta(1+\alpha)$

Chaining - Average case

- Assume that n is O(m) (e.g., think of n = 5m or cm for a small constant c)

- Then, $\alpha = \dfrac{n}{m} = \dfrac{O(m)}{m} = O(1)$

- Hence: **all** dictionary operations take `O(1)` time on average

Open addressing

- ALL elements are stored in the array T itself
- Each entry of T contains either an element or null
- $n \leq m$, $\alpha \leq 1$

- Insertion of a key `k`:
  - Probe the entries of the hash table until an empty slot is found

- Sequence of slots probed depends on key `k` to be inserted

- The hash function depends on the key `k` and the probe #, i

$$h : U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots m-1\}$$

- The probe sequence generated for a key k

  h(k,0), h(k,1), h(k,2), …, h(k,m-1)

  should be a permutation of 0, 1, 2, 3,…, m-1

  (this guarantees that all slots are eventually considered)

Open addressing – Insert

```
Insert (T, k);
// i = probe #
i=0;
repeat
    j=h(k,i); // compute (i+1)th probe
    if T[j]=null then T[j]=k;  return j;
                 else i=i+1;
until (i=T.length);
return full
```

Open addressing – Search

```
Search (T, k);
// i = probe #
i=0;
repeat
    j=h(k,i);
    if T[j]=k then return j
              else i=i+1;
  until (i=T.length or T[j]==null);
return null
```

probes the same slots as insertion (with no deletions)

Open addressing – Delete

- Just setting `T[i]=null` for deletion is inappropriate!
- If at insertion of `k`, a visited slot `i` was occupied, and then the element there is deleted there is no way to retrieve `k` anymore !
- **Solution**: `T[i]=DELETED`(a special value)
- `Insert` needs to be adapted to treat such slots as empty
- `Search` remains unchanged as `DELETED` slots are ignored
- Search times now no longer depend on filling degree α only

If keys are to be often deleted, chaining is more commonly used than open addressing

Open addressing – Hash functions

- Requirement: for a given key `k`, generate a probing sequence

  h(k,0), h(k,1), h(k,2),… , h(k,m-1)

  which is a permutation of 0, 1, 2, 3,…, m-1 (in worst case all elements of the array need to be examined at insertion)

- Several policies/functions
  - **Linear probing:** h(k, i) = (h'(k) + i) mod m, for some appropriate single-parameter hash function (what se saw in Data Structures)
  - **Quadratic probing:** h(k, i) = (h'(k) + ci + ci^2) mod m
  - **Double hashing:** use a 2nd hash function for the probe

- Quality is judged by the number of different probe sequences each policy can generate

Open addressing – Hash function

- Assumption for our analysis: Uniform hashing
  - For each key considered, each of the m! permutations is equally likely as a probing sequence
  - too expensive or even unrealistic to implement in practice
  - But useful for analysis

- In practice: double hashing achieves a good approximation to uniform hashing

Open addressing – average case

Unsuccessful search

X= # of probes in unsuccessful search

A_i : the event {the i^{th} probe is to an occupied slot}

$$\Pr\{X \geq i\} =$$

$$= \Pr\{A_1 \cap A_2 \cap \ldots \cap A_{i-1}\}$$

$$= \Pr\{A_1\} \cdot \Pr\{A_2 | A_1\} \cdot \Pr\{A_3 | A_1 \cap A_2\} \cdot \ldots \cdot \Pr\{A_{i-1} | A_1 \cap \ldots \cap A_{i-2}\}$$

$$= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \ldots \cdot \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

(recall that n<m)

Open addressing – average case

Unsuccessful search

$$E[X]=\sum_{i=0}^{\infty} i\,\Pr\{X=i\}=\sum_{i=0}^{\infty} i\ [\ \Pr\{X\geq i\}-\Pr\{X\geq i+1\}\ ]$$

$$=\sum_{i=1}^{\infty}\Pr\{X\geq i\}\leq\sum_{i=1}^{\infty}\alpha^{i-1}=\sum_{i=0}^{\infty}\alpha^{i}$$

$$=1+a+a^{2}+a^{3}+a^{3}+\ldots=\frac{1}{1-a}\qquad (a\leq 1)$$

Intuition:

•1 probe is always made

•With probability α, the 1st probe finds an occupied slot and a 2nd probe is made

•With probability ≈ α^2, the 1st and the 2nd probe find occupied slots and a 3rd probe is made

•and so on...

Open addressing – average case

Successful search

- Follows the same probe sequence as insert
- Insert = unsuccessful search + placement → $1/(1-\alpha)$
- X_{i+1} = average # of probes for the $(i+1)^{th}$ inserted key
  = $1/(1-i/m)$
- X= # of probes in unsuccessful search over all n keys

$$E[X] = \frac{1}{n} \cdot \sum_{i=0}^{n-1} X_{i+1} = \frac{1}{n} \cdot \sum_{i=0}^{n-1} \frac{1}{1-i/m} = \frac{1}{\alpha} \cdot \sum_{i=0}^{n-1} \frac{1}{m-i}$$

Open addressing – average case

Successful search

$$E[X] = \frac{1}{a} \cdot \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{\alpha} \cdot \left(\sum_{k=m-n+1}^{m} \frac{1}{k} \right)$$

$$\leq \frac{1}{\alpha} \cdot \int_{m-n}^{m} \frac{1}{x} dx = \frac{1}{\alpha} [\ln m - \ln(m-n)]$$

$$= \frac{1}{\alpha} \cdot \ln\left(\frac{\mathrm{m}}{\mathrm{m - n}} \right) = \frac{1}{\alpha} \cdot \ln\left(\frac{1}{1 - \alpha} \right)$$

Efficiency of open addressing

Summary: Under the assumption of uniform hashing:

- An unsuccessful search takes $O\left(\frac{1}{1-\alpha}\right)$ time on average
  - If the hash table is half full, 2 probes are necessary on average
  - If the hash table is 90% full, 10 probes are necessary on average
- A successful search takes $O\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha}\right)$ time on average
  - If the hash table is half full, 1.39 probes are necessary on average
  - If the hash table is 90% full, 2.56 probes are necessary on average
- Recall that for chaining this was $\Theta(1+\alpha)$ for both cases
- Hence: as long as a = O(1), we have O(1) complexity on average for all the desired operations!