Imperial College London

CO202 – Software Engineering – Algorithms

Randomised Algorithms

Ben Glocker
Huxley Building, Room 377
b.glocker@imperial.ac.uk

The material is partly based on previous lectures by Prof Alex Wolf

Randomised Algorithms

- Surprisingly: Some of the fastest and most clever algorithms rely on **chance**
- Randomised algorithms are often simple and elegant, and their output is **correct with high probability**
- Randomness is one important strategy to provide a **performance guarantee**

Coin Flipper on RANDOM.ORG

Example: The Hiring Problem

Suppose we need to hire a new office assistant

- Candidates are sent by an employment agency (in random order)
- Interviewing a candidate is cheap, hiring a candidate is costly
- We are committed to having, at all times, the best possible person for the job. If we interview someone who is better than the current assistant, we fire and hire

```
HIRE-ASSISTANT(n)

best = 0
for i = 1 to n
    interview candidate i
    if candidate i is better than candidate best
        best = i
        hire candidate i
```

How often do we hire?

Example: The Hiring Problem

Assuming that all permutations are **equally likely**

- Using probability theory (see Section 5.2 in [Cormen]) we can show that we will hire $\ln n$ candidates (e.g. 3 for n=20)

Problem: We don't know whether the agency is sending candidates in random order, or not.

Worst-case: Candidates are sent in strictly increasing order of quality. We are going to hire n times.

Example: The Hiring Problem

Solution: We ask the agency to send a list of candidates. We choose randomly which candidate to interview.

```
RANDOMIZED-HIRE-ASSISTANT(n)
  randomly permute the list of candidates
  best = 0
  for i = 1 to n
      interview candidate i
      if candidate i is better than candidate best
          best = i
          hire candidate i
```

don't assume
but enforce
random order

Example: Quicksort

- Quicksort was invented 1960 by
  Sir Charles Anthony Richard Hoare
  (Winner of the ACM Turing Award; but not for Quicksort)

- Quicksort is a divide & conquer algorithm
- Widespread use, e.g. default library sort function in UNIX

Complexity

average case: $O(n \log n)$

worst case: $O(n^2)$

Reminder: Quicksort

Divide & Conquer Approach

1. **Divide:** Partition (rearrange) the array `A[p..r]` into two subarrays `A[p..q-1]` and `A[q+1..r]` such that each element in the first are less than or equal to `A[q]`, and elements in the second are greater or equal to `A[q]`. Index `q` is computed as part of this procedure.

2. **Conquer:** Sort the two subarrays recursively using quicksort.

3. **Combine:** Trivial (there is nothing to do).

Reminder: Quicksort

```
QUICKSORT(A,p,r)
  if p < r
        q = PARTITION(A,p,r)
        QUICKSORT(A,p,q-1)
        QUICKSORT(A,q+1,r)
```

The key in Quicksort

To sort an entire array `A`, the initial call is `QUICKSORT(A,1,A.length)`

Divide: Partition

```
PARTITION(A,p,r)
 1: x = A[r]
 2: i = p-1
 3: for j = p to r-1
 4:       if A[j] ≤ x
 5:             i = i+1
 6:             SWAP(A[i],A[j])
 7: SWAP(A[i+1],A[r])
 8: return i+1
```

proposed by N. Lomuto

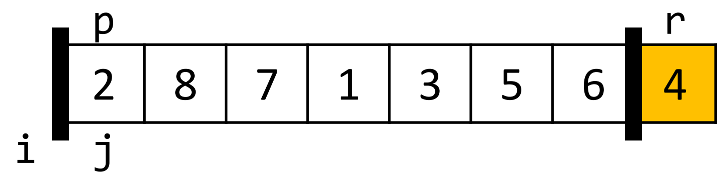
Divide: Partition

```
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])
return i+1
```

p							r
2	8	7	1	3	5	6	4

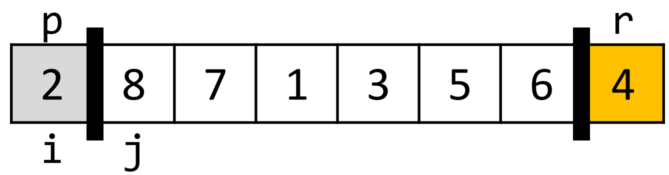
Divide: Partition

```
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])
return i+1
```



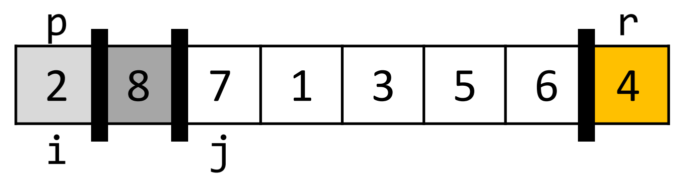
Divide: Partition

```
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])
  return i+1
```



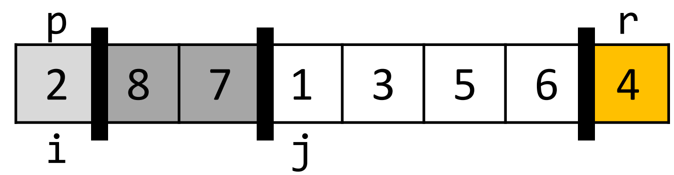
Divide: Partition

```
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])
return i+1
```



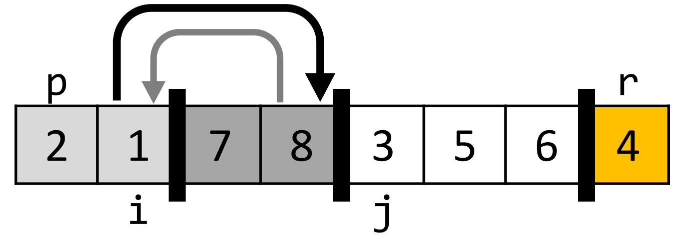
Divide: Partition

```
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])
return i+1
```



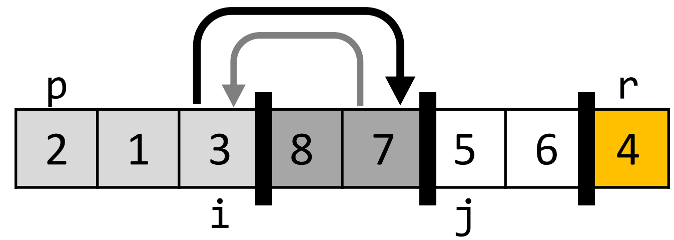
Divide: Partition

```
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])
 return i+1
```



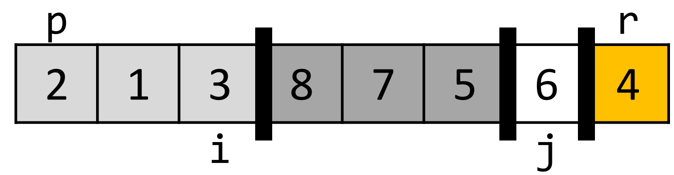
Divide: Partition

```
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])
return i+1
```



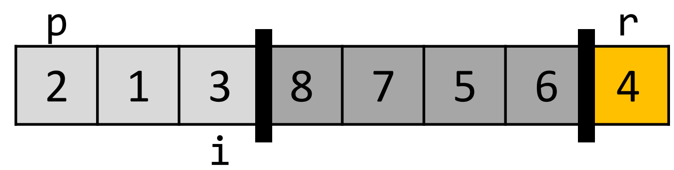
Divide: Partition

```
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])
 return i+1
```



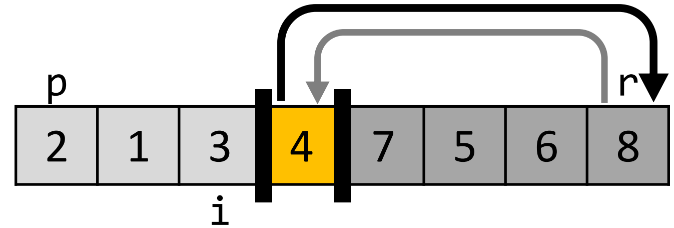
Divide: Partition

```
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])
return i+1
```



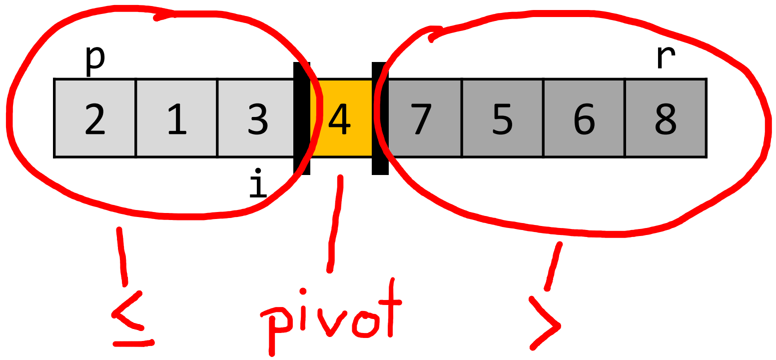
Divide: Partition

```
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])
return i+1
```



Divide: Partition

```
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])
return i+1
```



Exercise 1: Illustrate the Operations of Partition

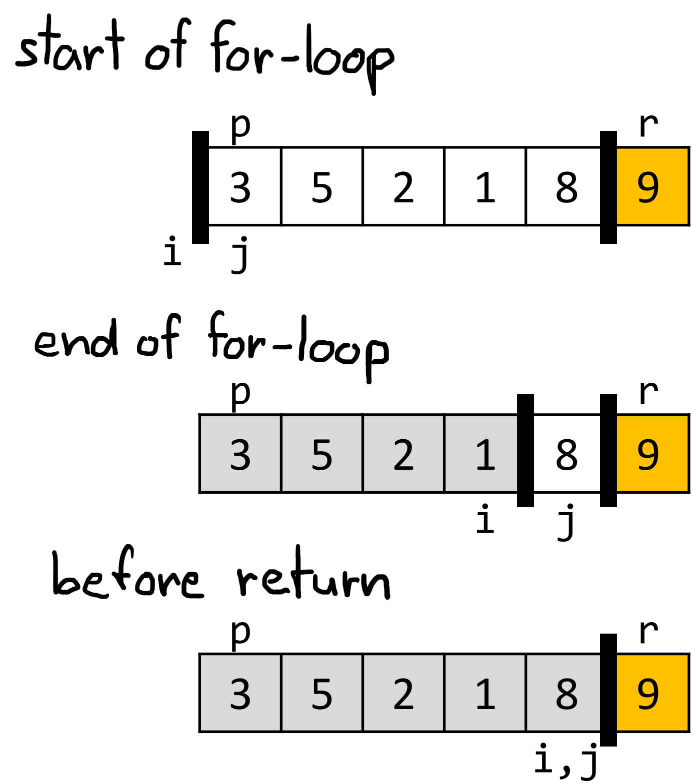
$$A = \langle 3, 5, 2, 1, 8, 9 \rangle$$

```
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])
return i+1
```

Exercise 1: Illustrate the Operations of Partition

```
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])
    return i+1
```

$$A = \langle 3, 5, 2, 1, 8, 9 \rangle$$



Performance of Quicksort

- Running time of `QUICKSORT` depends on whether the partitioning is **balanced** or **unbalanced**
- It depends on which elements are used for partitioning

Balanced

`QUICKSORT` runs asymptotically as fast as `MERGE-SORT`

Unbalanced

`QUICKSORT` can run asymptotically as fast as `INSERTION-SORT`

Performance of Quicksort

Worst-case partitioning

Partitioning produces one subproblem with $n-1$ elements and one with 0 elements

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

$$T(n) = \Theta(n^2)$$

Best-case partitioning

For most even splits, partitioning produces two subproblems each of size no more than n/2

$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(n) = \Theta(n \lg n)$$

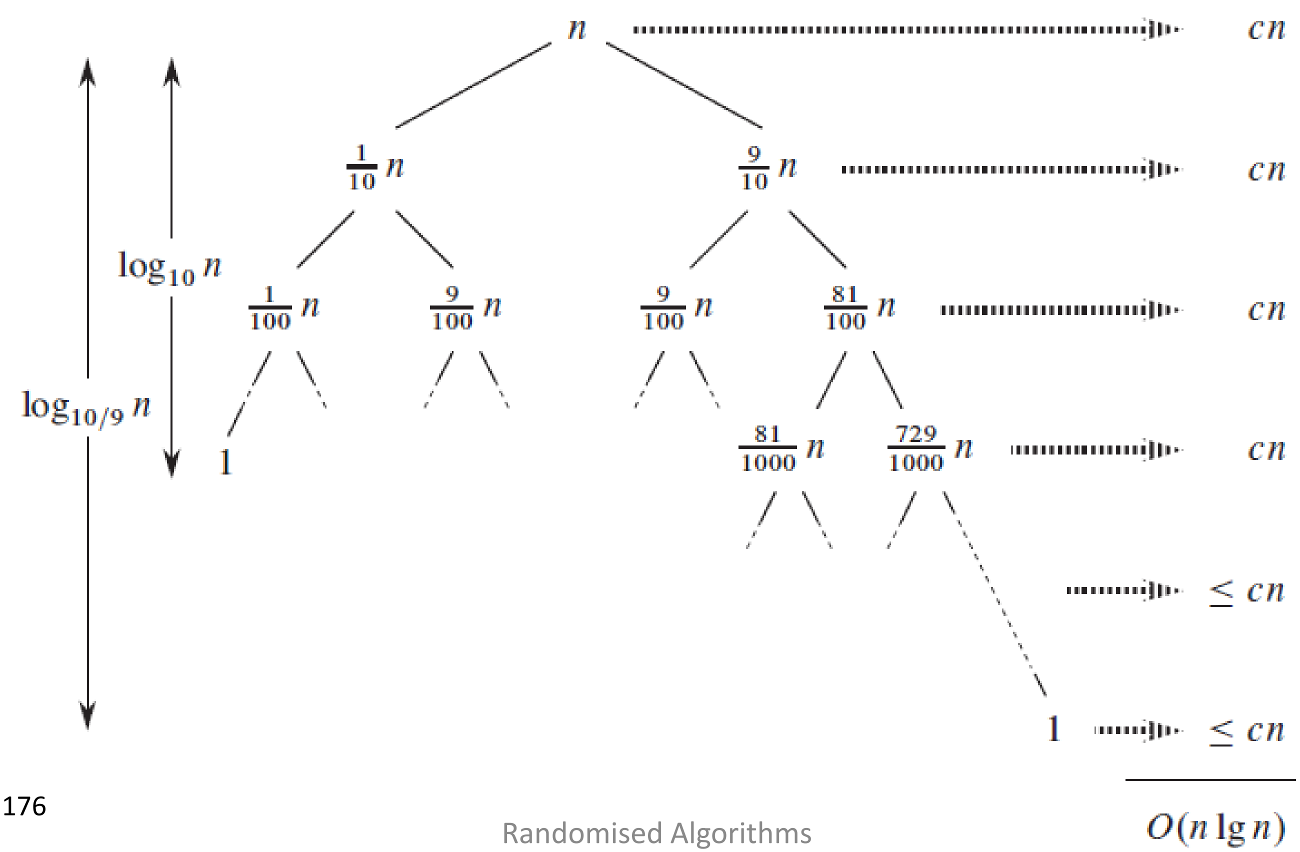
Performance of Quicksort

Average-case partitioning

Are we closer to the best-case or the worst-case?

Suppose the partitioning algorithm always produces a 9-to-1 proportional split

$$T(n) = T(9n/10) + T(n/10) + cn$$



[Cormen] p.176

Performance of Quicksort

Average-case partitioning

On average, partitioning produces a mix of “good” and “bad” splits. Assuming those alternate on the recursion tree levels, the running time is still $O(n \lg n)$ but with slightly larger constants.

This assumes that **all permutations** of the input numbers are **equally likely**. That is, unfortunately, not always the case.

How about **random shuffling** the input array? Could work, but let’s try something else…

Randomised Version of Quicksort

Pivot selection using **random sampling**

```
RANDOMISED-PARTITION(A,p,r)
 i = RANDOM(p,r)
 SWAP(A[r],A[i])
 return PARTITION(A,p,r)

RANDOMISED-QUICKSORT(A,p,r)
 if p < r
     q = RANDOMISED-PARTITION(A,p,r)
     RANDOMISED-QUICKSORT(A,p,q-1)
     RANDOMISED-QUICKSORT(A,q+1,r)
```

Improved Randomised Version of Quicksort

Median-of-3 partition

Choose the pivot as the median (middle element) of a set of 3 elements randomly selected from the subarray.

$$A = \langle 3, 5, 2, 1, 8, 9 \rangle$$

1 5 8

median

Exercise 2: The Original Partition Algorithm

$$A = \langle 3, 5, 2, 1, 8, 9 \rangle$$

```
HOARE-PARTITION(A,p,r)
x = A[p]
i = p
j = r+1
while TRUE
    repeat
        j = j-1
    until A[j] ≤ x or j == p
    repeat
        i = i+1
    until A[i] ≥ x or i == r
    if i < j
        SWAP(A[i],A[j])
    else
        SWAP(A[p],A[j])
        return j
```

Exercise 2: The Original Partition Algorithm

```
HOARE-PARTITION(A,p,r)
x = A[p]
i = p
j = r+1
while TRUE
    repeat
        j = j-1
    until A[j] ≤ x or j == p
    repeat
        i = i+1
    until A[i] ≥ x or i == r
    if i < j
        SWAP(A[i],A[j])
    else
        SWAP(A[p],A[j])
        return j
```

p					r	
3	5	2	1	8	9	
i						j

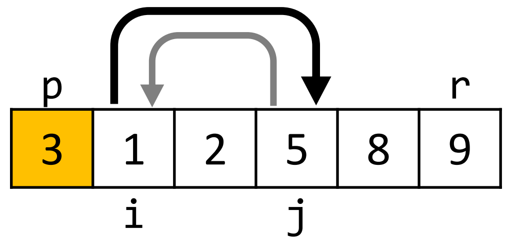
Exercise 2: The Original Partition Algorithm

```
HOARE-PARTITION(A,p,r)
 x = A[p]
 i = p
 j = r+1
 while TRUE
     repeat
         j = j-1
     until A[j] ≤ x or j == p
     repeat
         i = i+1
     until A[i] ≥ x or i == r
     if i < j
         SWAP(A[i],A[j])
     else
         SWAP(A[p],A[j])
         return j
```

p					r
3	5	2	1	8	9
	i		j		

Exercise 2: The Original Partition Algorithm

```
HOARE-PARTITION(A,p,r)
 x = A[p]
 i = p
 j = r+1
 while TRUE
     repeat
         j = j-1
     until A[j] ≤ x or j == p
     repeat
         i = i+1
     until A[i] ≥ x or i == r
     if i < j
         SWAP(A[i],A[j])
     else
         SWAP(A[p],A[j])
         return j
```



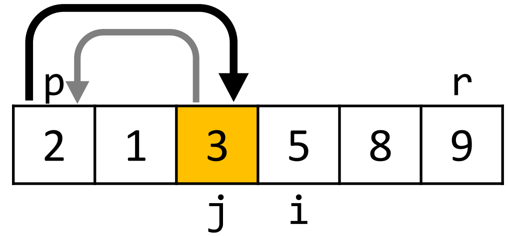
Exercise 2: The Original Partition Algorithm

```
HOARE-PARTITION(A,p,r)
x = A[p]
i = p
j = r+1
while TRUE
    repeat
        j = j-1
    until A[j] ≤ x or j == p
    repeat
        i = i+1
    until A[i] ≥ x or i == r
    if i < j
        SWAP(A[i],A[j])
    else
        SWAP(A[p],A[j])
        return j
```

p					r
3	1	2	5	8	9
		j	i		

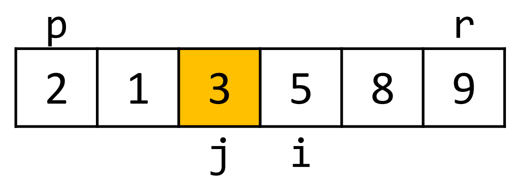
Exercise 2: The Original Partition Algorithm

```
HOARE-PARTITION(A,p,r)
x = A[p]
i = p
j = r+1
while TRUE
    repeat
        j = j-1
    until A[j] ≤ x or j == p
    repeat
        i = i+1
    until A[i] ≥ x or i == r
    if i < j
        SWAP(A[i],A[j])
    else
        SWAP(A[p],A[j])
        return j
```



Exercise 2: The Original Partition Algorithm

```
HOARE-PARTITION(A,p,r)
x = A[p]
i = p
j = r+1
while TRUE
    repeat
        j = j-1
    until A[j] ≤ x or j == p
    repeat
        i = i+1
    until A[i] ≥ x or i == r
    if i < j
        SWAP(A[i],A[j])
    else
        SWAP(A[p],A[j])
        return j
```

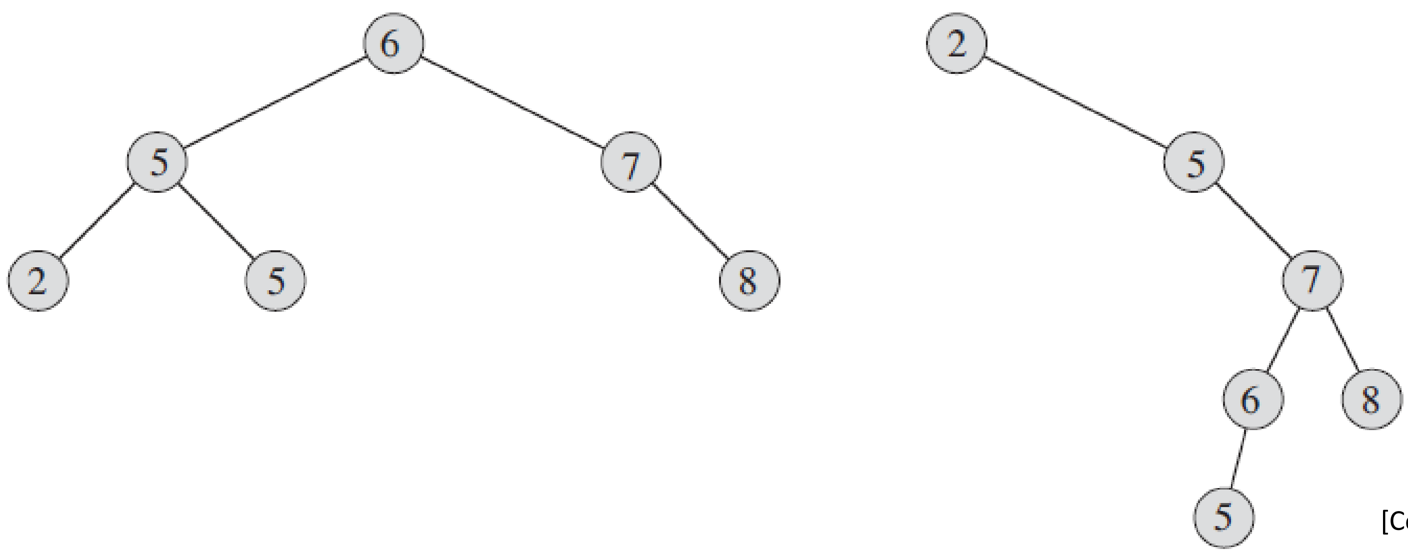


Example: Binary Search Trees

Binary search tree property

Let x be a node in a BST. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

- BSTs are used to implement look-up tables and dynamic sets.
- Operations **insert** and **search** (and others) have complexity $O(h)$ where h is the height of the tree.



[Cormen] p.287

Example: Binary Search Trees

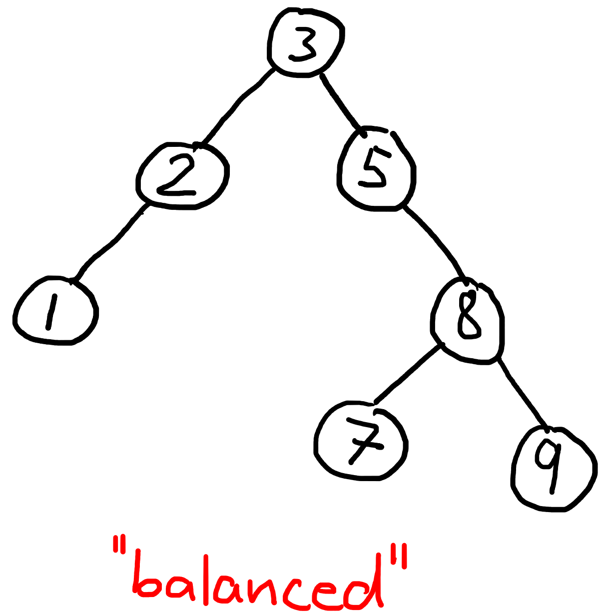
Tree nodes

- key: a value
- parent: pointer to the parent node, `NIL` if root node
- left: pointer to the left child node, `NIL` if no left child
- right: pointer to the right child node, `NIL` if no right child

BST Insert

```
TREE-INSERT(t,z)
y = NIL
x = t
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
if y == NIL
    t = z
else if z.key < y.key
    y.left = z
else
    y.right = z
return t
```

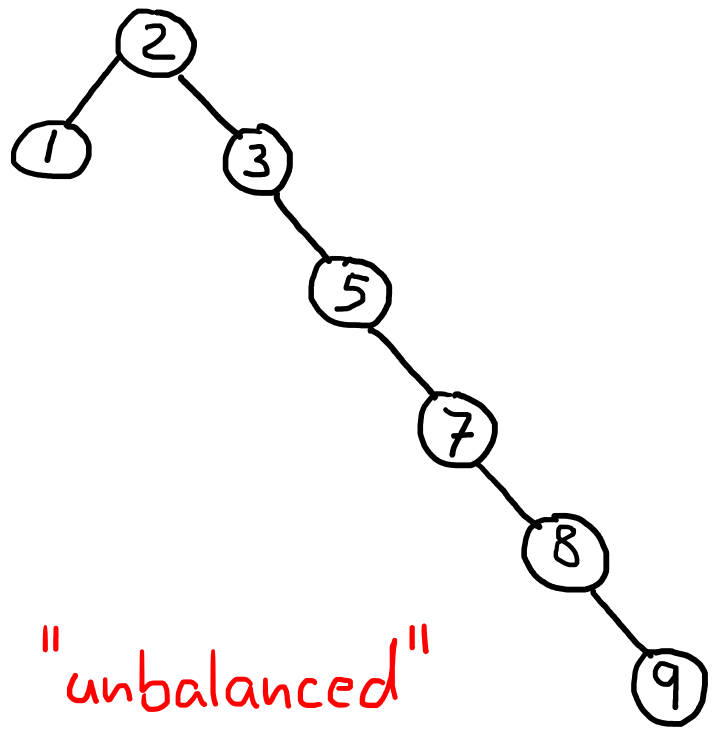
$$\langle 3, 5, 2, 1, 8, 9, 7 \rangle$$



BST Insert

```
TREE-INSERT(t,z)
y = NIL
x = t
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.parent = y
if y == NIL
    t = z
else if z.key < y.key
    y.left = z
else
    y.right = z
return t
```

$$\langle 2, 3, 1, 5, 7, 8, 9 \rangle$$



Improving BST Insert

The **expected height** of a **randomly built** binary search tree on n distinct keys is $O(\lg n)$. (see Section 12.4 in [Cormen])

Idea 1: Randomly **permute** input sequence

Problem: The sequence must be known in advance

Idea 2: Randomly **alternate** between two insert algorithms

1. tail insert: using the standard `TREE-INSERT`
   a new node is inserted as leaf in the tree
2. head insert: using `ROOT-INSERT`
   a new node is inserted as the root of the tree

Randomised BST Insert (version 1)

```
INSERT-RAND(t,z)
r = RANDOM(1,t.size+1)
if r == 1
    return ROOT-INSERT(t,z)
else
    return TREE-INSERT(t,z)
```

Does this simulate a random permutation of the input sequence?

- Are all permutations **equally likely**?

  No! Counterexample: Last element can only become the root or a leaf.

- How about applying this approach **recursively**?

Randomised BST Insert (version 2)

```
INSERT-RAND(t,z)
if t == NIL
    return z
r = RANDOM(1,t.size+1)
if r == 1
    return ROOT-INSERT(t,z)
if z.key < t.key
    t.left = INSERT-RAND(t.left,z)
else
    t.right = INSERT-RAND(t.right,z)
t.size = t.size + 1
return t
```

How do we implement `ROOT-INSERT`?

BST Root Insert

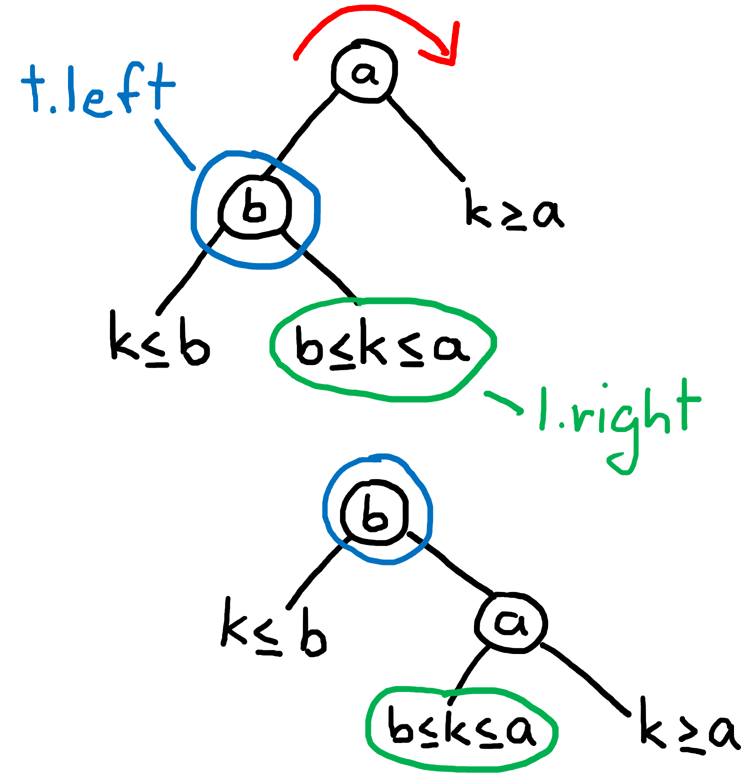
```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```

```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

BST Root Insert

```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```




```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

BST Root Insert

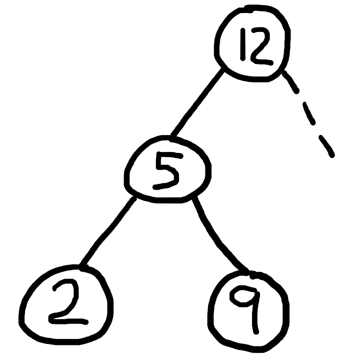
```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```

```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

BST Root Insert

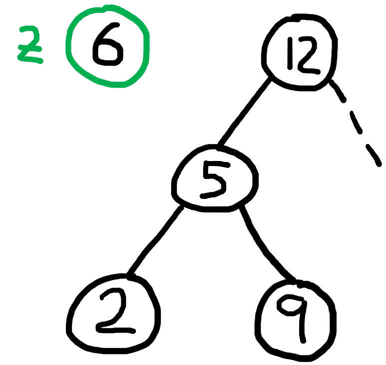
```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```




```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

BST Root Insert



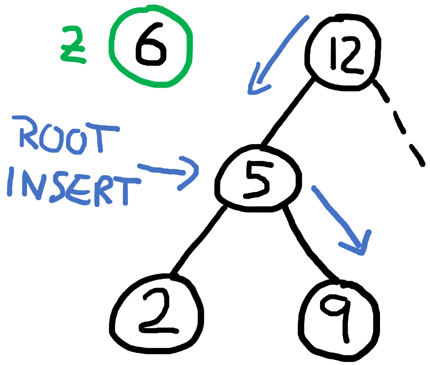

```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```

```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

BST Root Insert

```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```

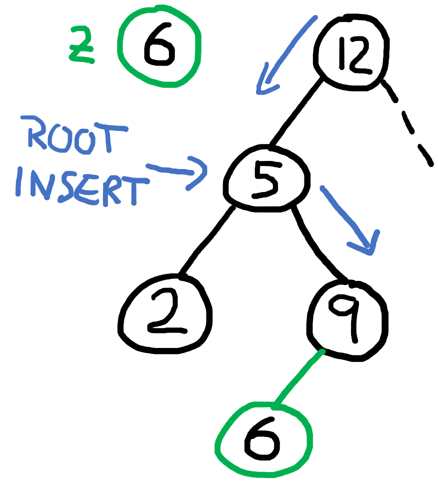



```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

BST Root Insert

```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```

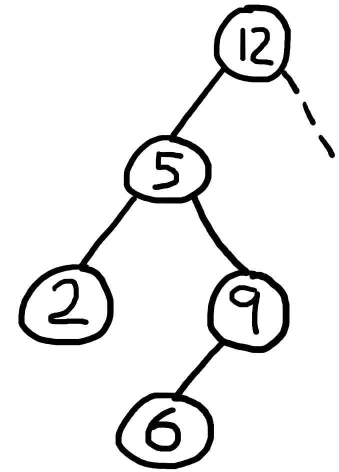



```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

BST Root Insert

```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```

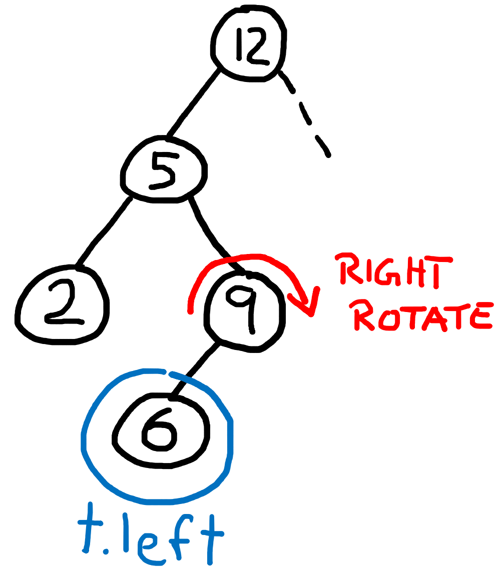



```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

BST Root Insert

```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```

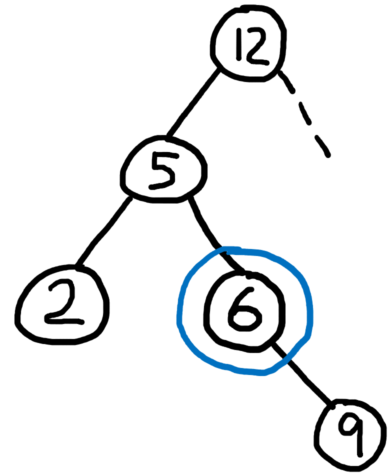



```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

BST Root Insert

```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```

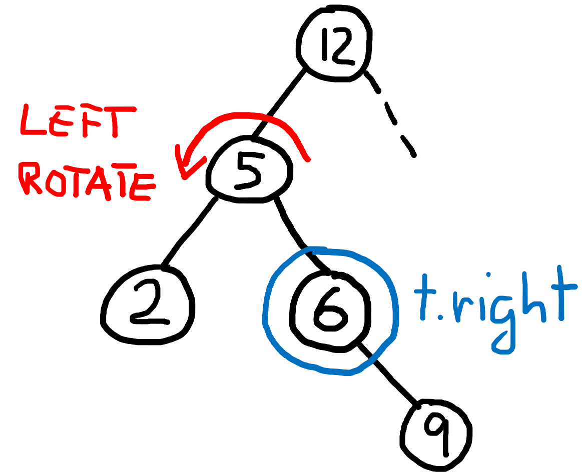



```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

BST Root Insert

```
ROOT-INSERT(t,z)
    if t == NIL
        return z
    if z.key < t.key
        t.left = ROOT-INSERT(t.left,z)
        t.size = t.size + 1
        return RIGHT-ROTATE(t)
    else
        t.right = ROOT-INSERT(t.right,z)
        t.size = t.size + 1
        return LEFT-ROTATE(t)
```

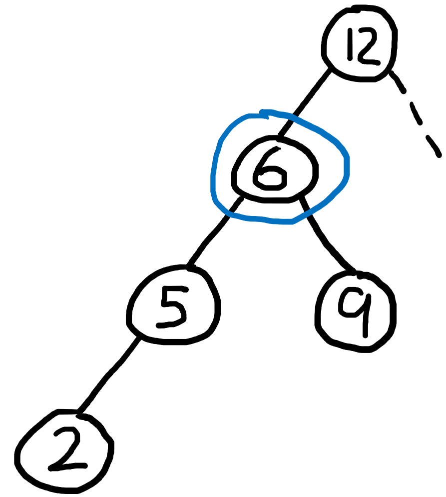



```
LEFT-ROTATE(t)
    r = t.right
    t.right = r.left
    r.left = t
    r.size = t.size
    t.size -= r.right.size + 1
    return r
```

```
RIGHT-ROTATE(t)
    l = t.left
    t.left = l.right
    l.right = t
    l.size = t.size
    t.size -= l.left.size + 1
    return l
```

BST Root Insert

```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```




```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

Exercise 3: Randomised BST Insert

$$\langle 2, 3, 1, 5, 7, 8, 9 \rangle \quad \langle 0, 1, 0, 1, 1, 0, 0 \rangle$$

```
INSERT-RAND(t,z)
 if t == NIL
     return z
 r = RANDOM(1,t.size+1)
 if r == 1
     return ROOT-INSERT(t,z)
 if z.key < t.key
     t.left = INSERT-RAND(t.left,z)
 else
     t.right = INSERT-RAND(t.right,z)
 t.size = t.size + 1
 return t
```

```
ROOT-INSERT(t,z)
 if t == NIL
     return z
 if z.key < t.key
     t.left = ROOT-INSERT(t.left,z)
     t.size = t.size + 1
     return RIGHT-ROTATE(t)
 else
     t.right = ROOT-INSERT(t.right,z)
     t.size = t.size + 1
     return LEFT-ROTATE(t)
```

```
LEFT-ROTATE(t)
 r = t.right
 t.right = r.left
 r.left = t
 r.size = t.size
 t.size -= r.right.size + 1
 return r
```

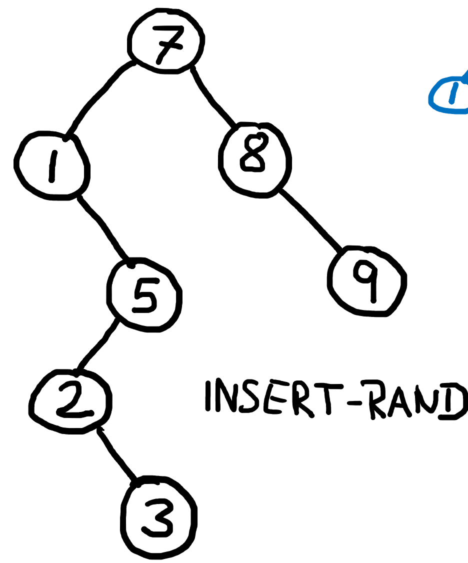
```
RIGHT-ROTATE(t)
 l = t.left
 t.left = l.right
 l.right = t
 l.size = t.size
 t.size -= l.left.size + 1
 return l
```

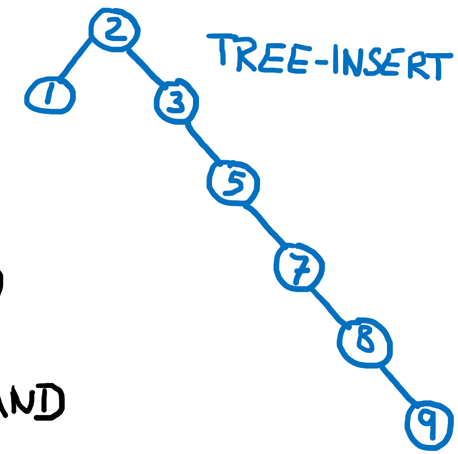
Exercise 3: Randomised BST Insert

```
INSERT-RAND(t,z)
if t == NIL
    return z
r = RANDOM(1,t.size+1)
if r == 1
    return ROOT-INSERT(t,z)
if z.key < t.key
    t.left = INSERT-RAND(t.left,z)
else
    t.right = INSERT-RAND(t.right,z)
t.size = t.size + 1
return t
```

```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```

$\langle 2,3,1,5,7,8,9\rangle \quad \langle 0,1,0,1,1,0,0\rangle$



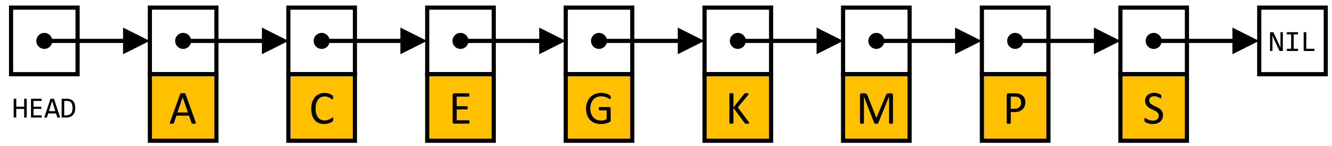



```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

Example: Skip Lists

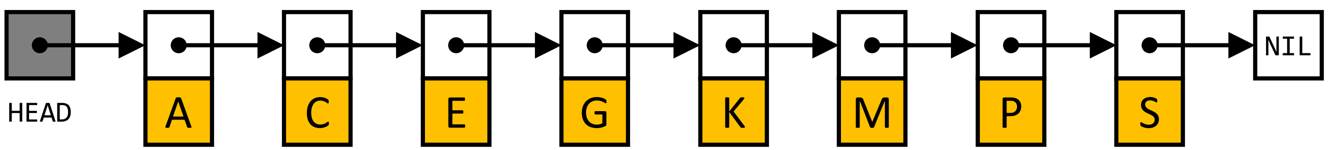
A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.



Linked list: search key P

Example: Skip Lists

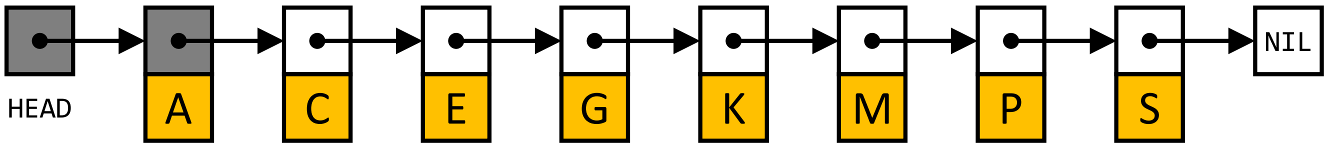
A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.



Linked list: search key P

Example: Skip Lists

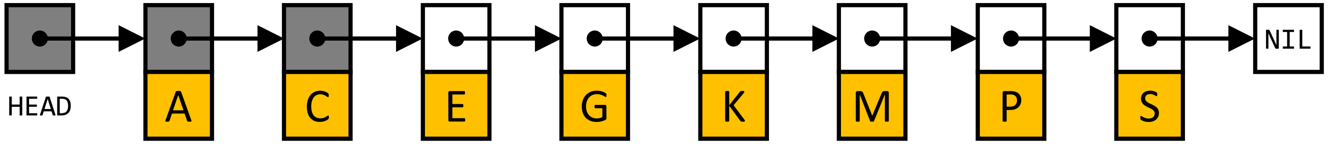
A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.



Linked list: search key P

Example: Skip Lists

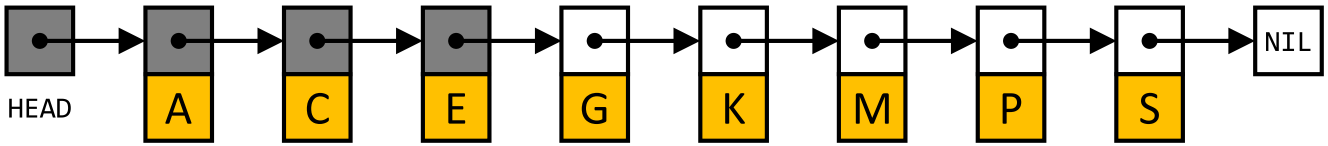
A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.



Linked list: search key P

Example: Skip Lists

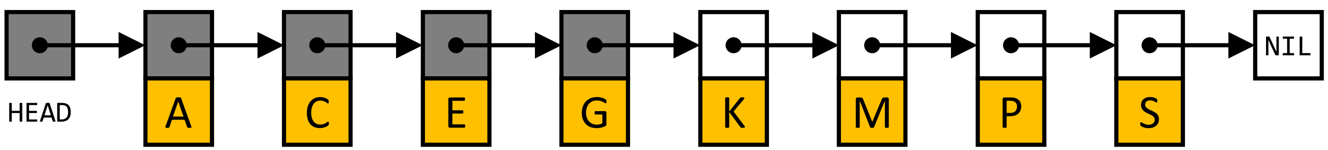
A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.



Linked list: search key P

Example: Skip Lists

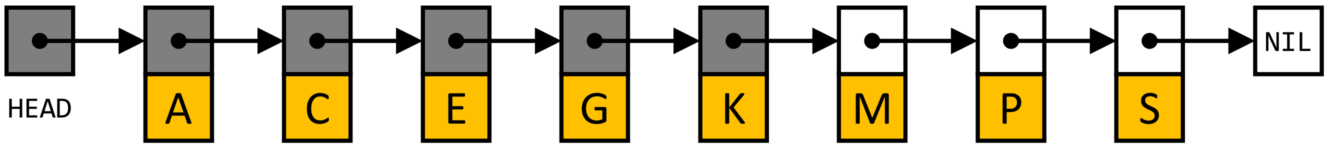
A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.



Linked list: search key P

Example: Skip Lists

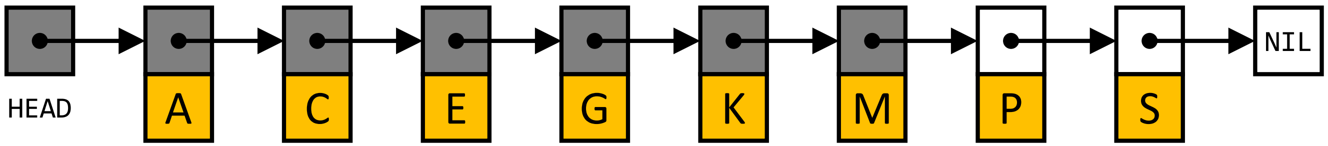
A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.



Linked list: search key P

Example: Skip Lists

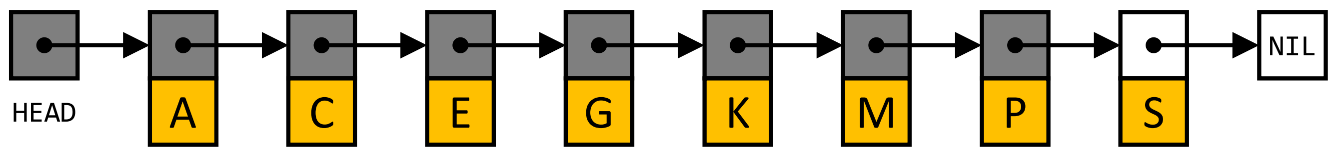
A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.



Linked list: search key P

Example: Skip Lists

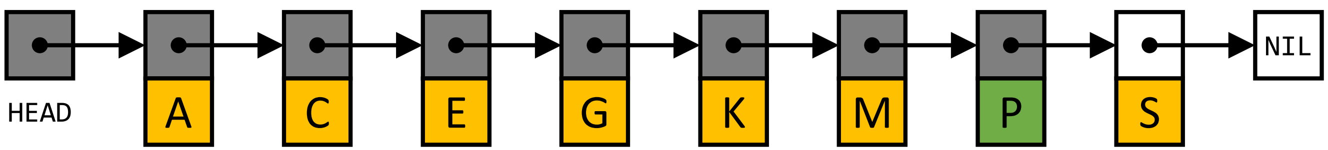
A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.



Linked list: search key P

Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

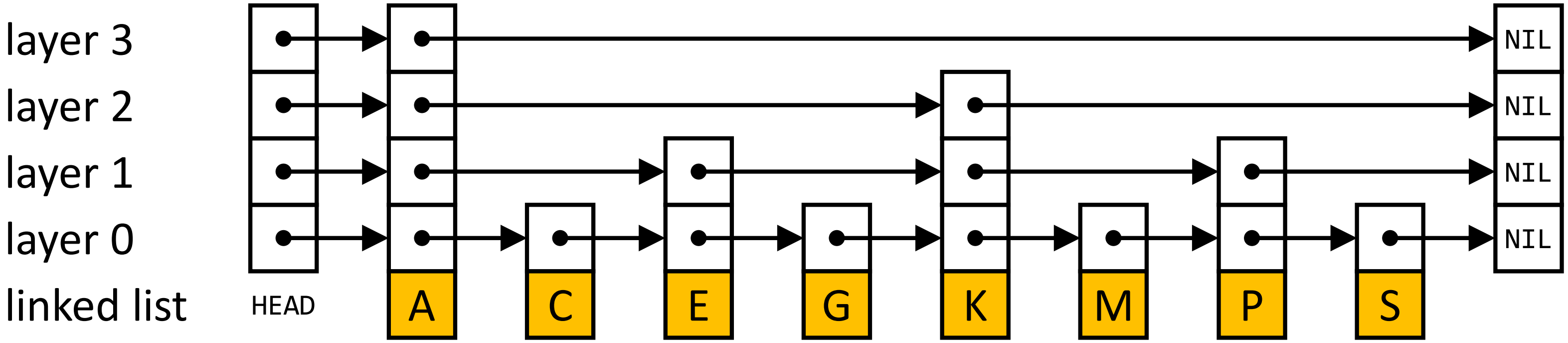


Linked list: search key P

Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

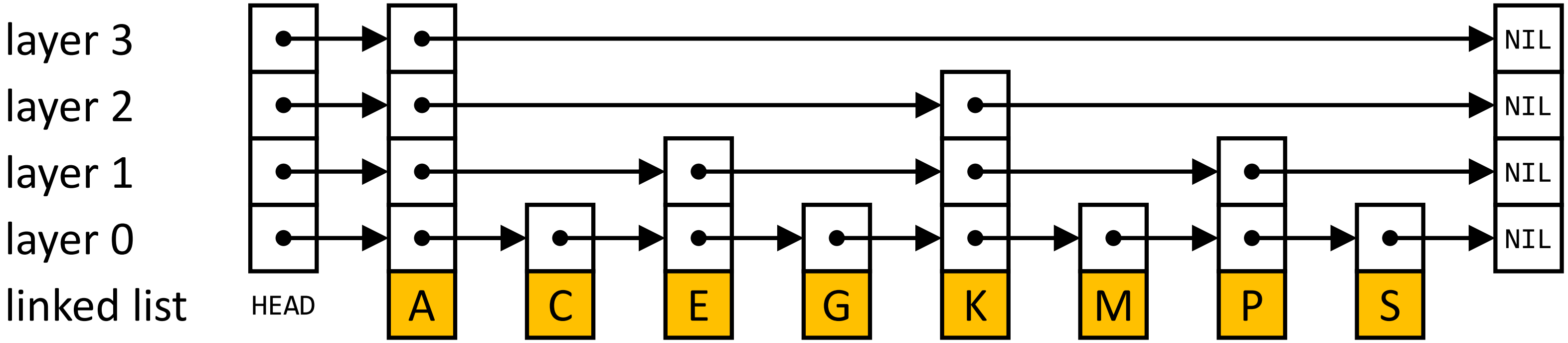
Idea: Maintain a **hierarchy of linked sublists**



Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

Idea: Maintain a **hierarchy of linked sublists**

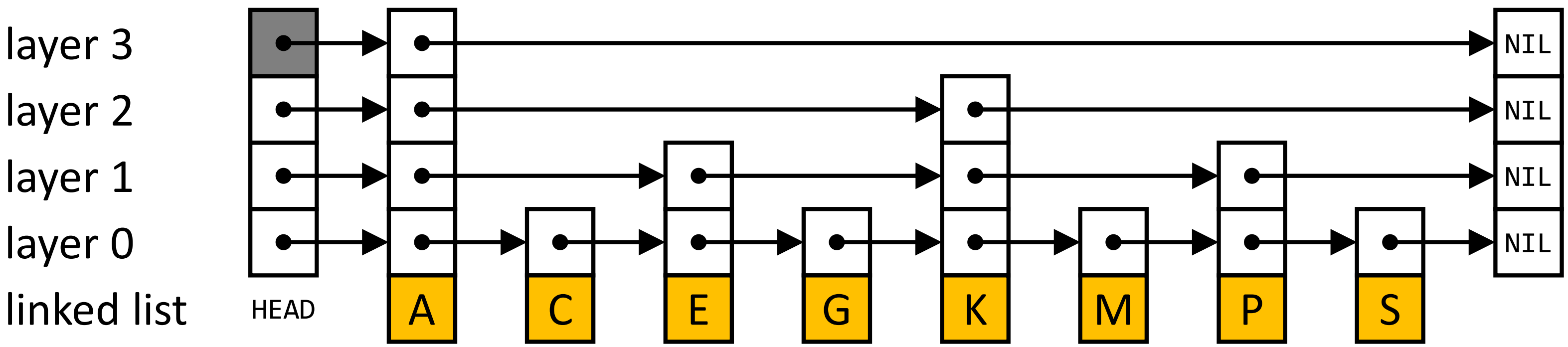


Skip list: search key P

Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

Idea: Maintain a **hierarchy of linked sublists**

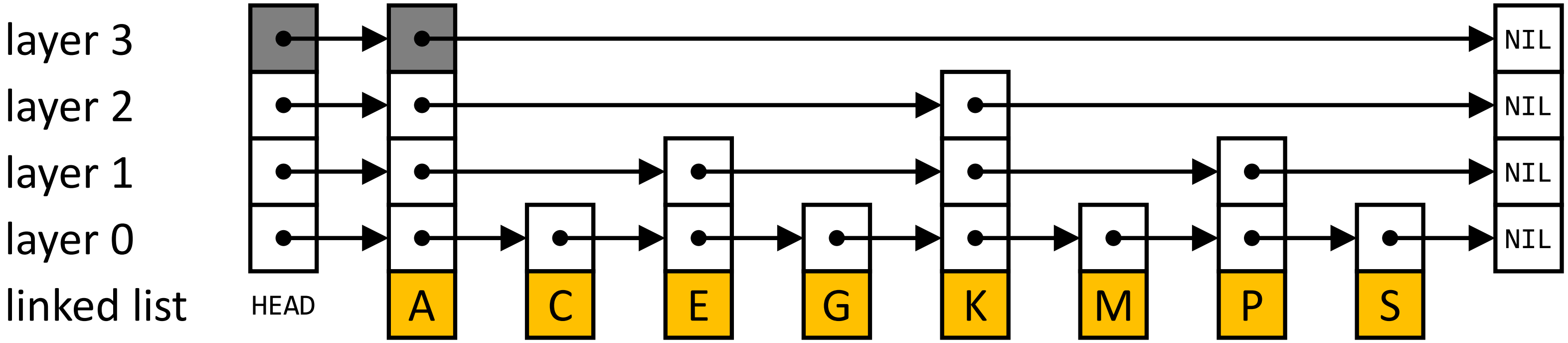


Skip list: search key P

Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

Idea: Maintain a **hierarchy of linked sublists**

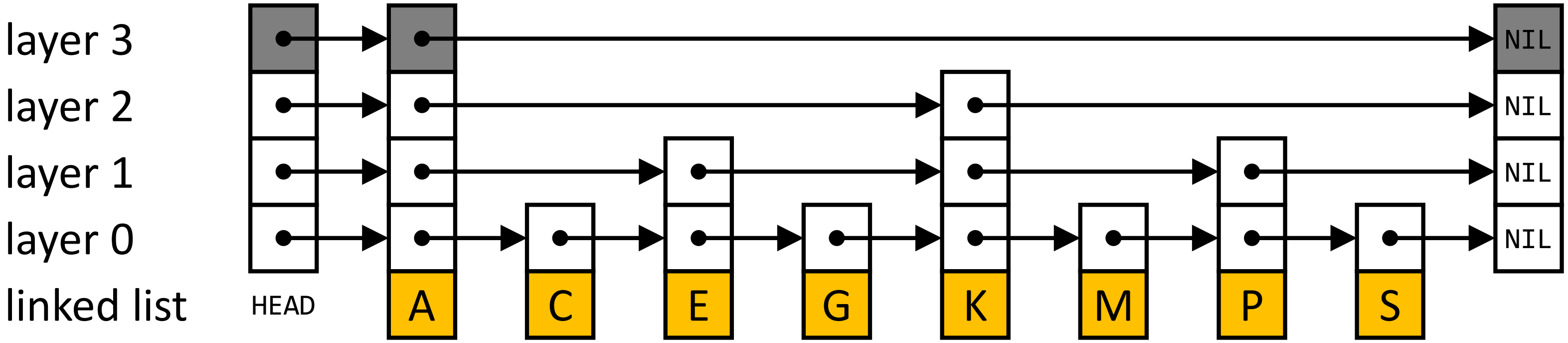


Skip list: search key P

Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

Idea: Maintain a **hierarchy of linked sublists**

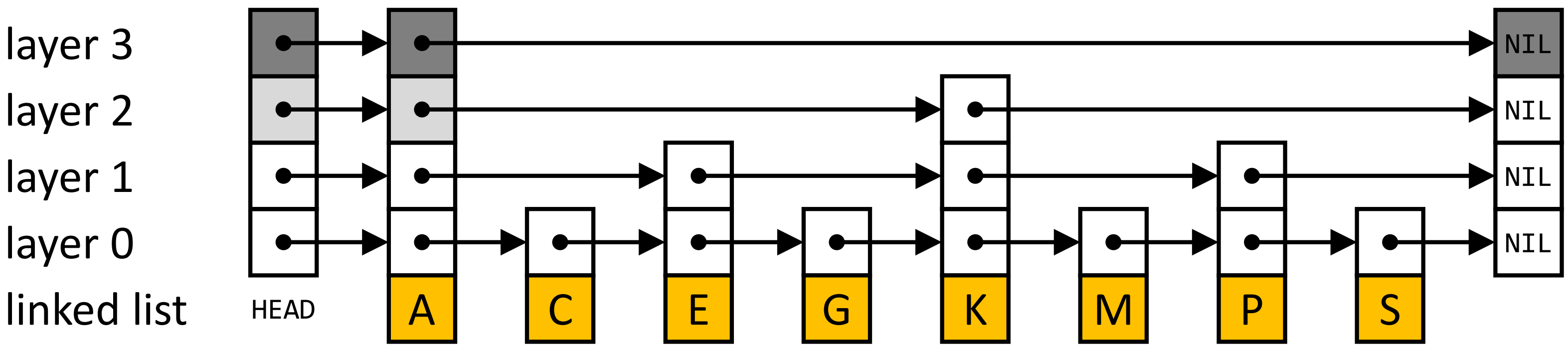


Skip list: search key P

Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

Idea: Maintain a **hierarchy of linked sublists**

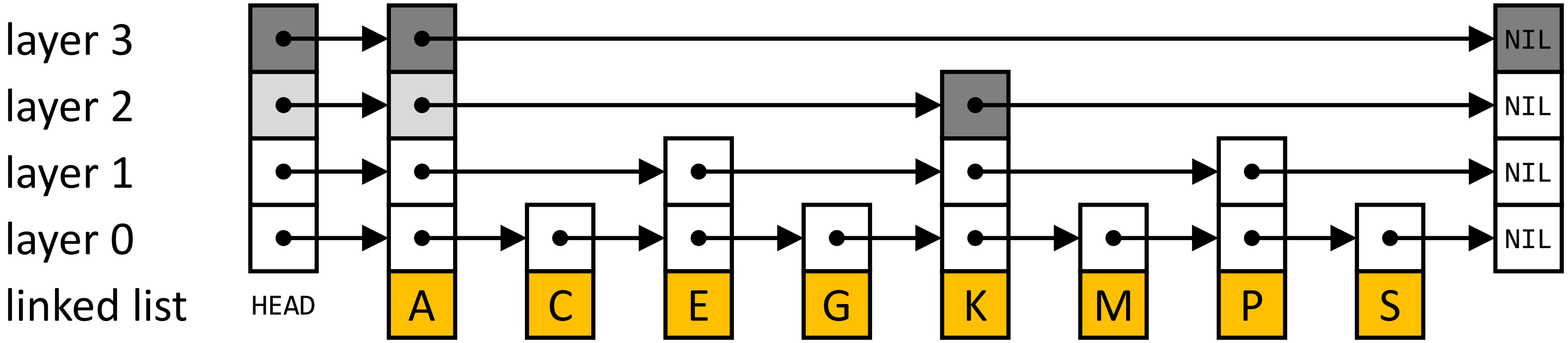


Skip list: search key P

Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

Idea: Maintain a **hierarchy of linked sublists**

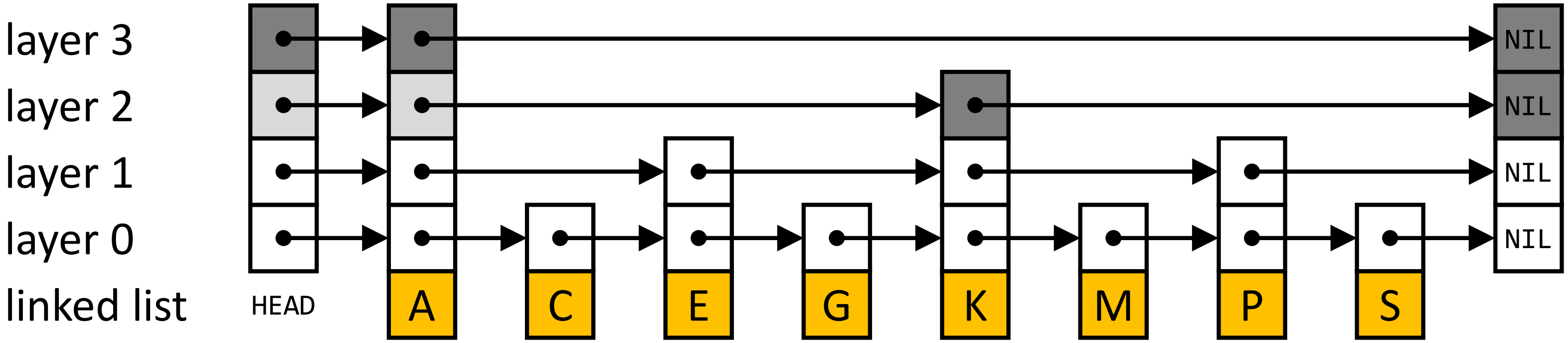


Skip list: search key P

Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

Idea: Maintain a **hierarchy of linked sublists**

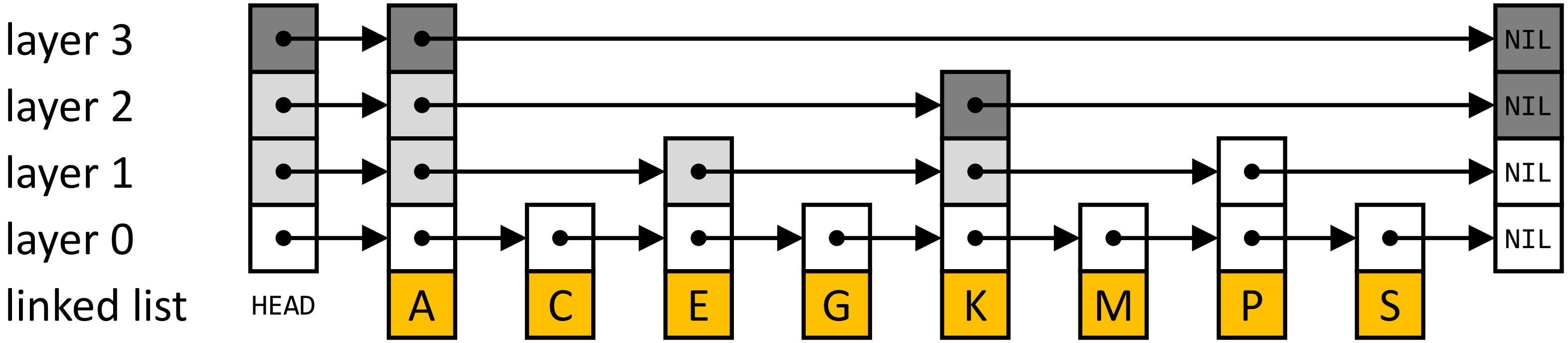


Skip list: search key P

Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

Idea: Maintain a **hierarchy of linked sublists**

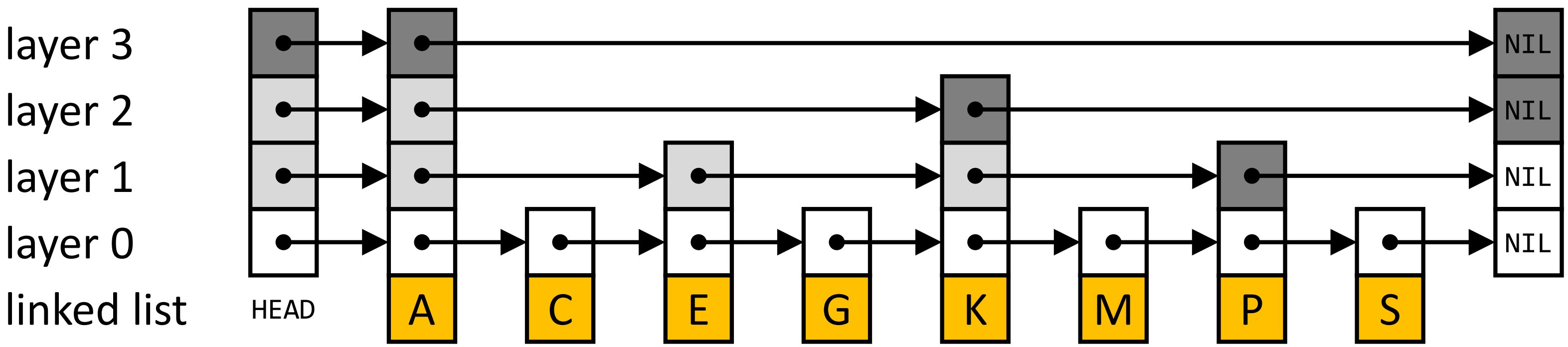


Skip list: search key P

Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

Idea: Maintain a **hierarchy of linked sublists**

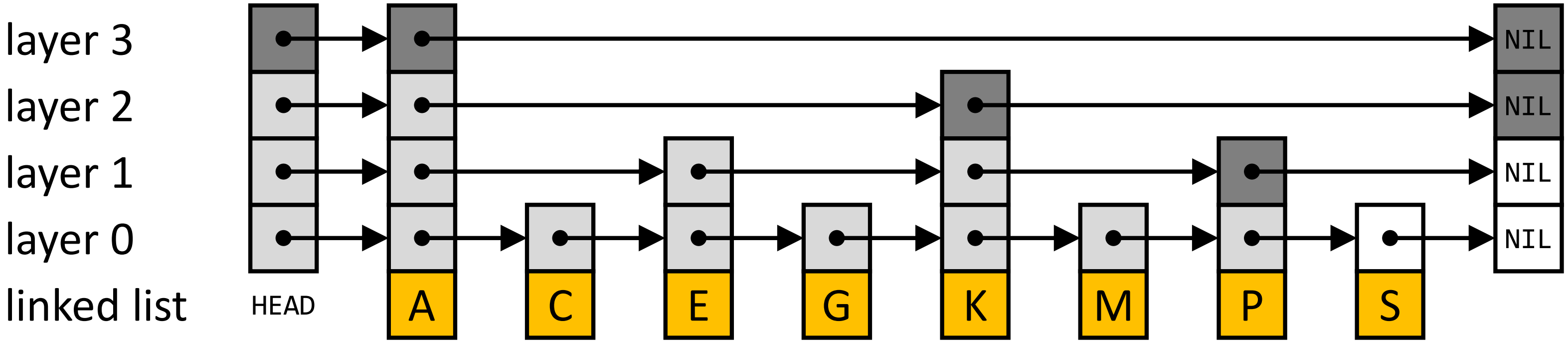


Skip list: search key P

Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

Idea: Maintain a **hierarchy of linked sublists**

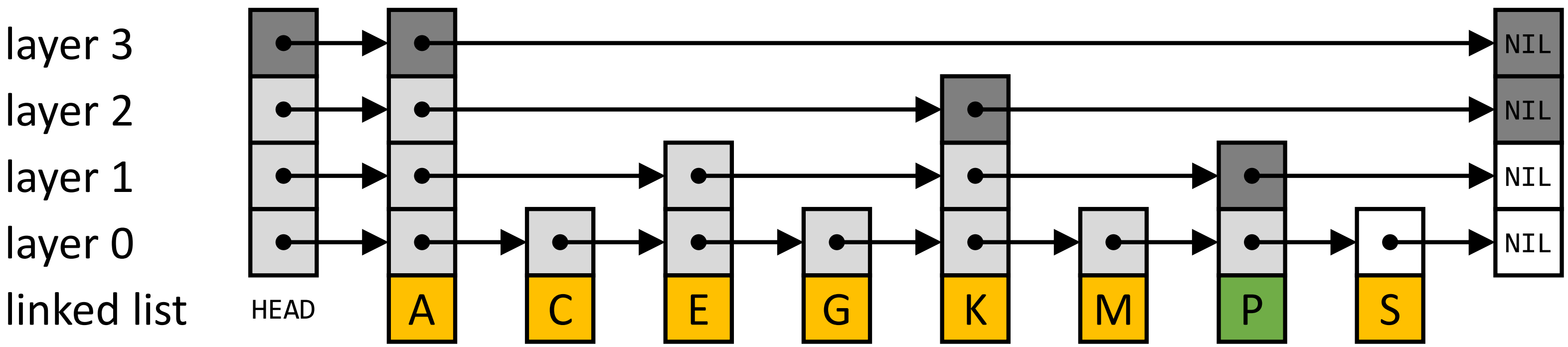


Skip list: search key P

Example: Skip Lists

A skip list is a **data structure** that supports **fast search** for an ordered sequence of elements.

Idea: Maintain a **hierarchy of linked sublists**

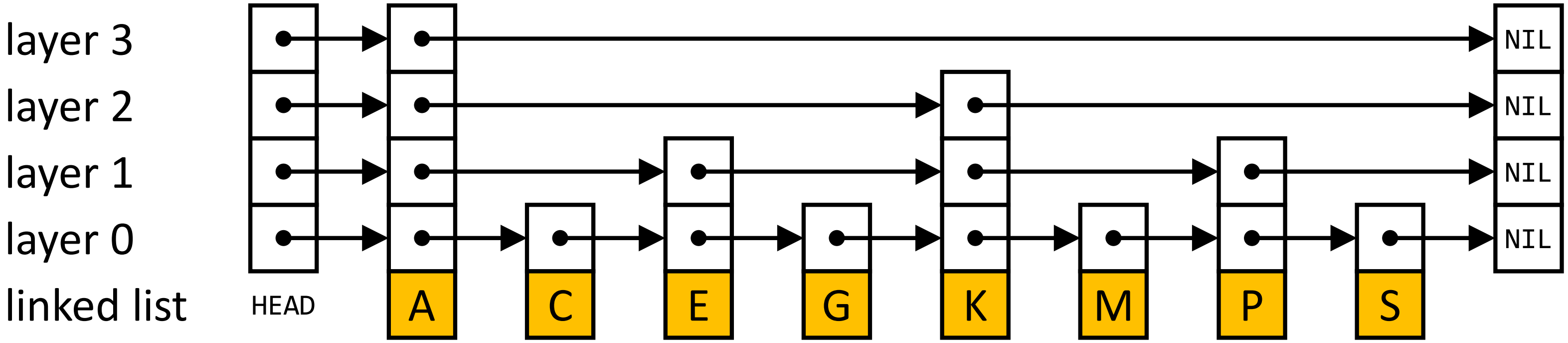


Skip list: search key P

Running time of binary search $O(\lg n)$

Example: Skip Lists

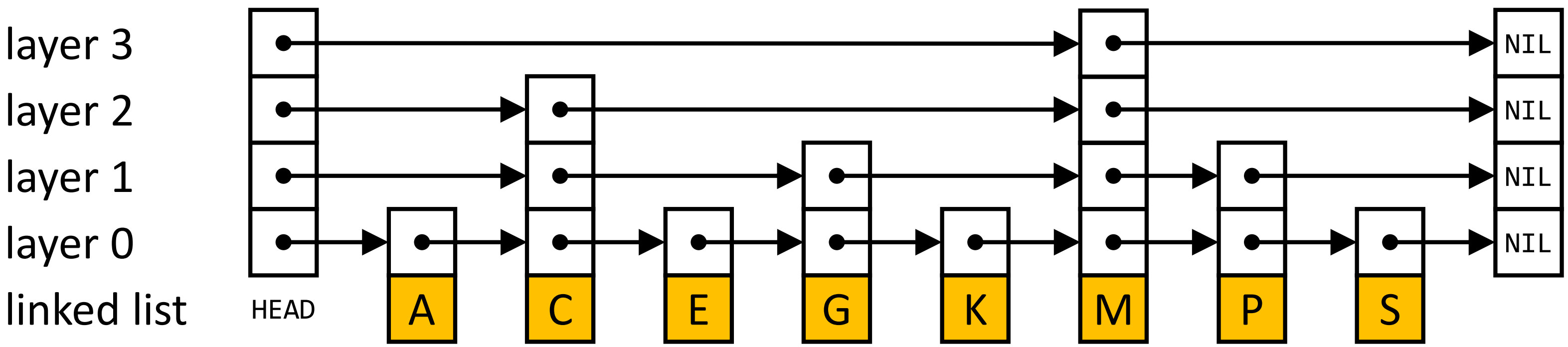
The **below skip list** is “ideal”. However, in practice this is **not feasible** to maintain as we would need to reorganise after each insert.



Randomised Skip Lists

Idea: Approximate the ideal using **randomisation**!

An element in layer i appears with some probability in layer $i + 1$

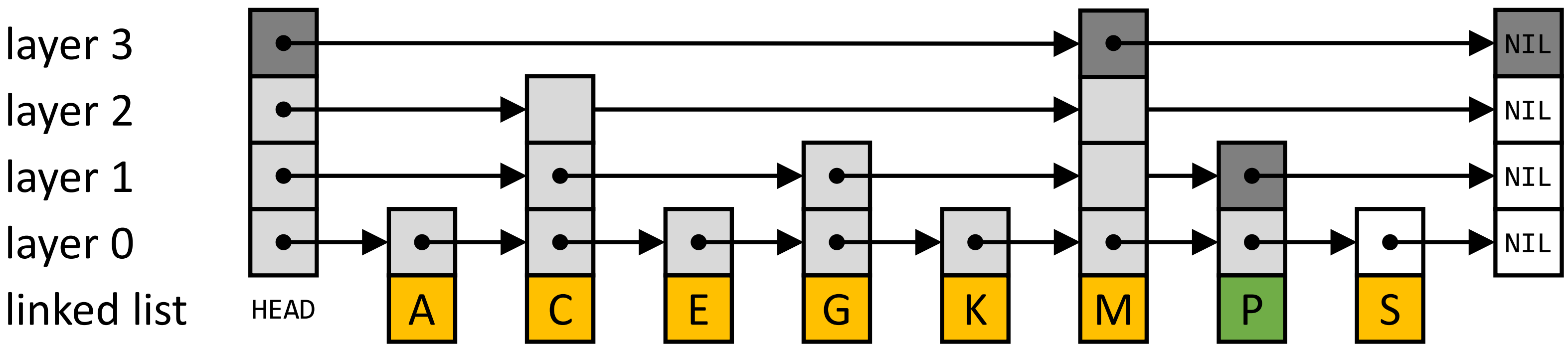


Skip list: search key P

Randomised Skip Lists

Idea: Approximate the ideal using **randomisation**!

An element in layer i appears with some probability in layer $i + 1$

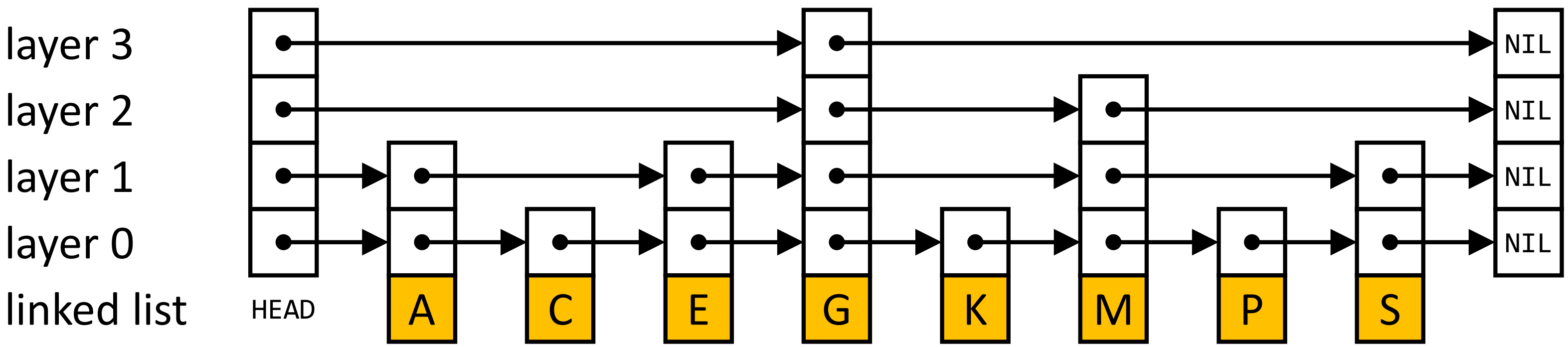


Skip list: search key P

Randomised Skip Lists

Idea: Approximate the ideal using **randomisation**!

An element in layer i appears with some probability in layer $i + 1$

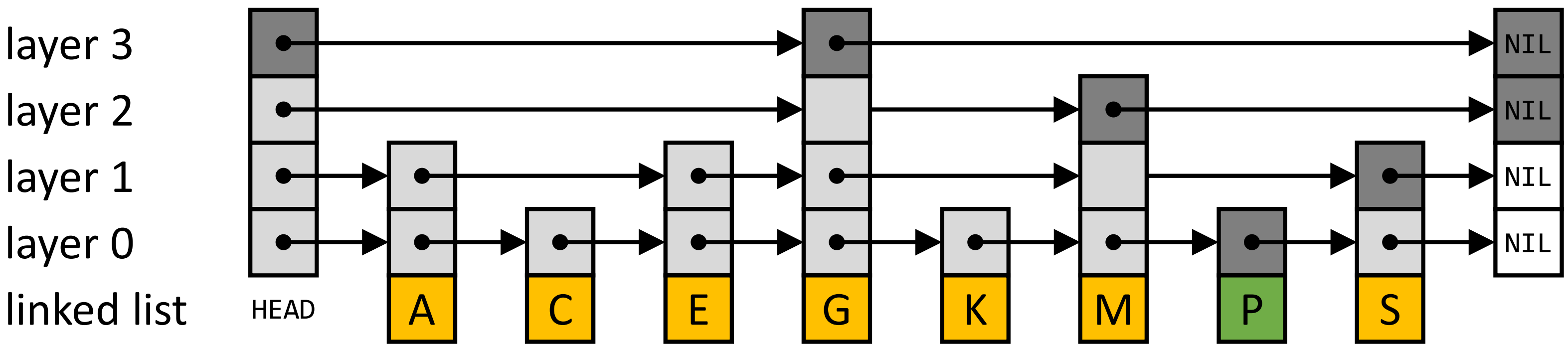


Skip list: search key P

Randomised Skip Lists

Idea: Approximate the ideal using **randomisation**!

An element in layer i appears with some probability in layer $i + 1$

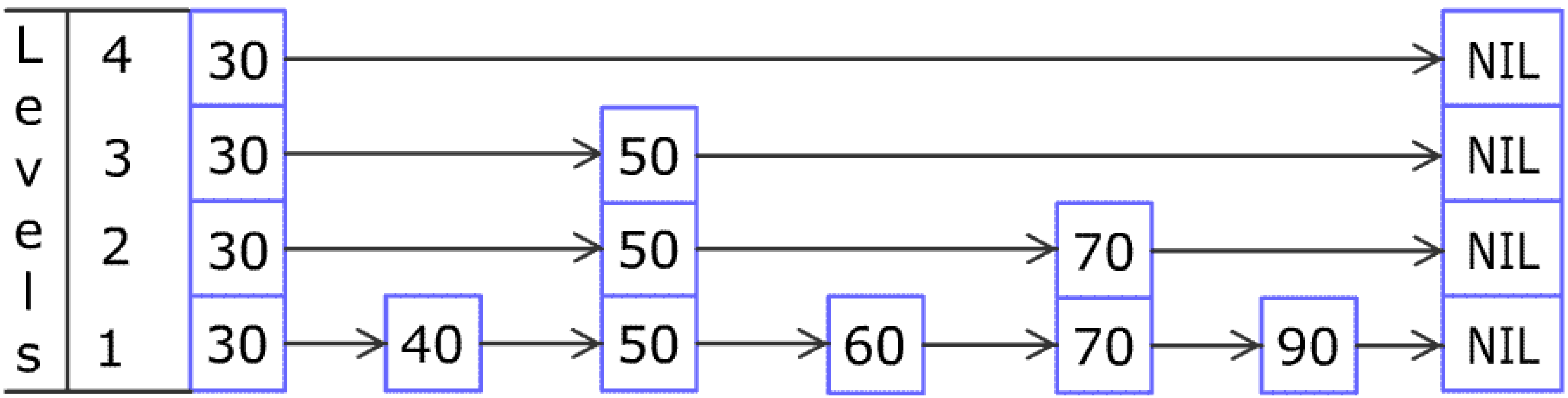


Skip list: search key P

Expected running time $O(\lg n)$

Randomised Skip Lists

Inserting new elements



By Artyom Kalinin - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=30222103

Example: Find a Zero Bit Problem

- **Input**: Sequence of n bits with approximately equal number of 1s and 0s, $\langle b_1, b_2, \dots, b_n \rangle$
- **Output:** An index i such that $b_i = 0$, or 0 if none exists

```
FIND-A-ZERO(B)
 1: for i = 1 to B.length
 2:     if B[i] == 0
 3:         return i
 4: return 0
```

Vulnerable to pathologic inefficiencies

What if the sequence is sorted so all 1-bits come before 0-bits?

Example: Find a Zero Bit Problem

It is not always desirable to randomize the input, so we use randomness directly

```
RANDOMISED-FIND-A-ZERO-V1(B)
repeat
    i = RANDOM(1,B.length)
until B[i] == 0
return i
```

Las Vegas Algorithm
- always correct
- unbounded resources

Generate & Test

```
RANDOMISED-FIND-A-ZERO-V2(B,k)
for j = 1 to k
    i = RANDOM(1,B.length)
    if B[i] == 0
        return i
return 0
```

Monte Carlo Algorithm
- not always correct
- bounded resources

Conclusions

Introducing **randomness** can

- help to **avoid pathologic inputs**
- yield good **expected running time**
- allow to deal with **large input domains**

Randomisation strategies

- randomise the **input**
  (e.g. random permutations, hiring problem)
- randomise the **computation**
  (e.g. random choices, pivot selection, BST insert)

Randomisation itself is based on algorithms

References

Books

- [Cormen] **Introduction to Algorithms**
  T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. MIT Press. 2009 (3rd Edition)
- [Sedgewick] **Algorithms**
  R. Sedgewick, K. Wayne. Addison-Wesley. 2011 (4th Edition)
- [Dasgupta] **Algorithms**
  S. Dasgupta, C. Papadimitriou, U. Vazirani. McGraw-Hill Higher Education. 2006

Online

- http://algs4.cs.princeton.edu/lectures/
- https://www.coursera.org/courses?query=algorithms

CO202 – Software Engineering – Algorithms

Randomised Algorithms - Exercises

Exercise 1: Illustrate the Operations of Partition

$$A = \langle 3, 5, 2, 1, 8, 9 \rangle$$

```
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])
return i+1
```

Exercise 2: The Original Partition Algorithm

$$A = \langle 3, 5, 2, 1, 8, 9 \rangle$$

```
HOARE-PARTITION(A,p,r)
x = A[p]
i = p
j = r+1
while TRUE
    repeat
        j = j-1
    until A[j] ≤ x or j == p
    repeat
        i = i+1
    until A[i] ≥ x or i == r
    if i < j
        SWAP(A[i],A[j])
    else
        SWAP(A[p],A[j])
        return j
```

Exercise 3: Randomised BST Insert

$$\langle 2, 3, 1, 5, 7, 8, 9 \rangle \quad \langle 0, 1, 0, 1, 1, 0, 0 \rangle$$

```
INSERT-RAND(t,z)
if t == NIL
    return z
r = RANDOM(1,t.size+1)
if r == 1
    return ROOT-INSERT(t,z)
if z.key < t.key
    t.left = INSERT-RAND(t.left,z)
else
    t.right = INSERT-RAND(t.right,z)
t.size = t.size + 1
return t
```

```
ROOT-INSERT(t,z)
if t == NIL
    return z
if z.key < t.key
    t.left = ROOT-INSERT(t.left,z)
    t.size = t.size + 1
    return RIGHT-ROTATE(t)
else
    t.right = ROOT-INSERT(t.right,z)
    t.size = t.size + 1
    return LEFT-ROTATE(t)
```

```
LEFT-ROTATE(t)
r = t.right
t.right = r.left
r.left = t
r.size = t.size
t.size -= r.right.size + 1
return r
```

```
RIGHT-ROTATE(t)
l = t.left
t.left = l.right
l.right = t
l.size = t.size
t.size -= l.left.size + 1
return l
```

Exercise 4: Approximating Pi

How can we approximate pi using random numbers?

Exercise 5: Finding the k-th Smallest Element

Given set $A = \{a_1, \ldots, a_n\}$, find the k-th smallest Element