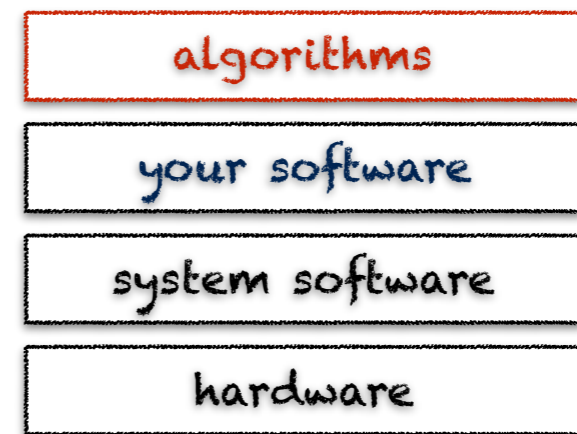




searching
algorithms

learning objectives



- ♦ learn what the searching problem is about
- ♦ learn two algorithms for solving this problem
- ♦ learn the importance of data structures

the problem

the searching problems comes in two variants:

- ◆ does a collection contain a given element?

$$\exists e \in C \subseteq \mathbb{N} \mid e = 10$$

- ◆ what is the value associated with some key in a given associative array ?

$$\text{let } v \in (k, v) \mid (k, v) \in \mathbb{N} \times \mathbb{R} \wedge k = 7$$

sequential search

the **simplest** searching algorithm
based on **brute-force** approach

also called **linear** search

```
SEQUENTIAL SEARCH of  $key$  in  $A[1..n]$   
for  $i \leftarrow 1$  to  $n$   
    if  $A[i] = key$   
        return true  
return false
```

worst case: $O(n)$

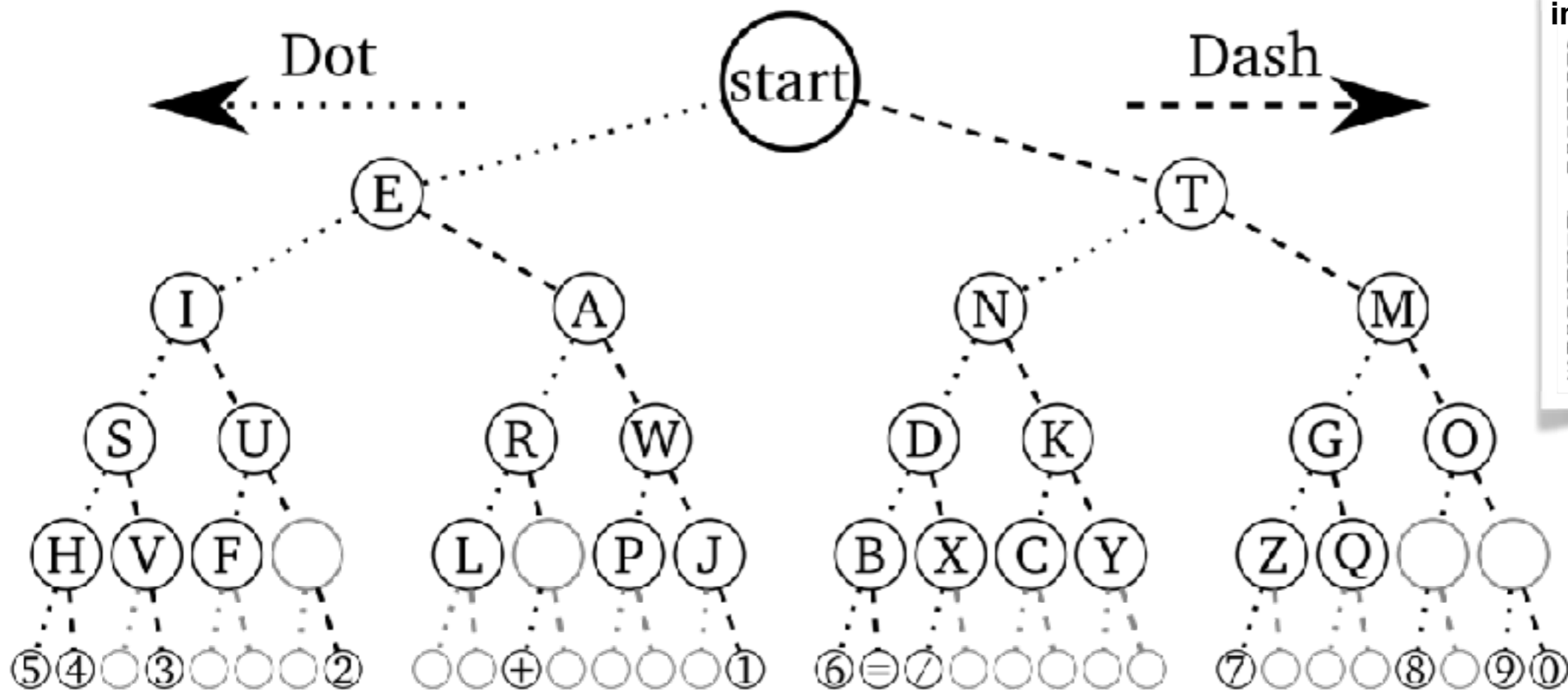
best case: $O(1)$

```
def sequentialSearch(theKey : Int, theArray: Array[Int]) : Boolean = {  
    for (key <- theArray)  
        if (key == theKey)  
            return true  
    return false  
}
```



dichotomic search

a **dichotomic search** consists in selecting between **two mutually exclusive alternatives (dichotomies)** at each step of the algorithm



international morse code

A	· — · — ·	U	· — · — ·
B	· — — · —	V	· — · — —
C	— · — — ·	W	· — — · —
D	— · — · —	X	— · — · —
E	· — — —	Y	— · — — ·
F	· — · — ·	Z	— — · — ·
G	— · — — —		
H	· — · — ·	1	· — — — —
I	· — — —	2	· — — — —
J	— · — — —	3	· — — — —
K	— · — — —	4	· — — — —
L	· — — — —	5	· — — — —
M	— — — —	6	· — — — —
N	— · — — —	7	· — — — —
O	— — — —	8	· — — — —
P	— · — — —	9	· — — — —
Q	— — — —	0	· — — — —
R	· — — — —		
S	· — — — —		
T	— — — —		

binary search

this algorithm requires a sorted collection

also called half-interval search or logarithmic search

BINARY SEARCH of key in $A[1..n]$

$low = 1$

$high = n$

while $low \leq high$ do

$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$

if $A[mid] > key$ then

$high = mid - 1$

else if $A[mid] < key$ then

$low = mid + 1$

else return *true*

return *false*

at each step, it reduces the search space by half by excluding the half that cannot contain the searched key

worst case: $O(\log n)$

best case: $O(1)$

```
def binarySearch(theKey : Int, theArray: Array[Int]) : Boolean = {
  var low = 0
  var high = theArray.size - 1

  while (low <= high) {
    val mid = (low + high)/2;
    if (theKey < theArray(mid)) {
      high = mid - 1
    } else if (theKey > theArray(mid))
      low = mid + 1
    else
      return true
  }
  return false;
}
```

binary search



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
3	8	13	17	25	34	41	46	55	63	74	78	81	82	93	99

↑
mid
74 > 46

3	8	13	17	25	34	41	46	55	63	74	78	81	82	93	99
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

↑
mid
74 < 78

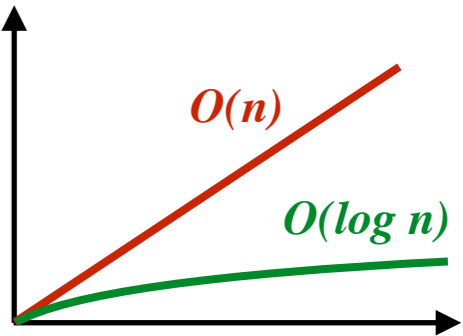
3	8	13	17	25	34	41	46	55	63	74	78	81	82	93	99
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

↑
mid
74 > 63

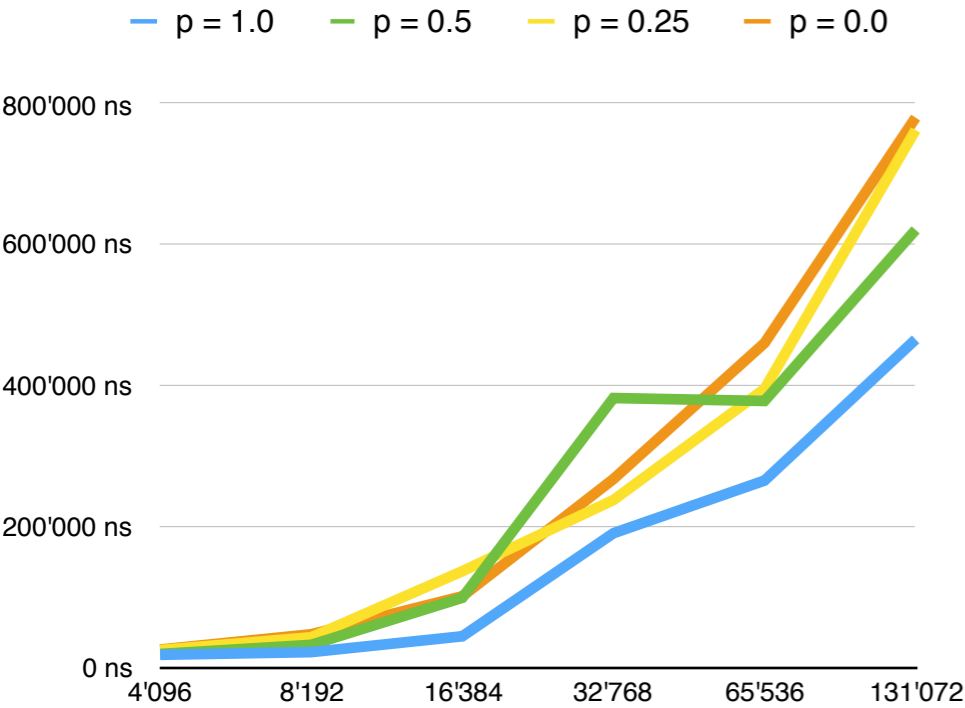
3	8	13	17	25	34	41	46	55	63	74	78	81	82	93	99
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

↑
mid
74 = 74

search performance



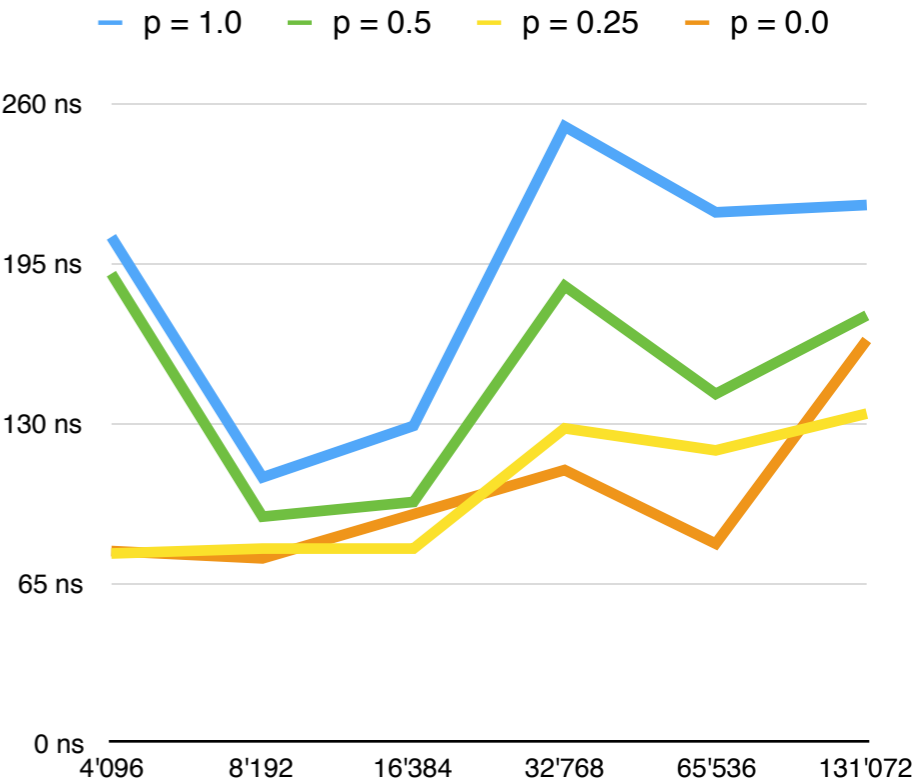
sequential search



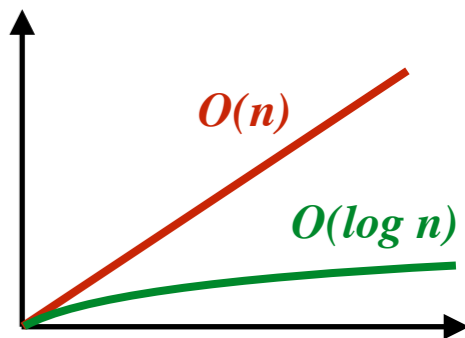
n	p = 1.0	p = 0.5	p = 0.25	p = 0.0
4'096	17'130 ns	18'306 ns	23'912 ns	24'645 ns
8'192	20'962 ns	31'311 ns	42'125 ns	46'303 ns
16'384	43'458 ns	98'216 ns	135'982 ns	100'311 ns
32'768	189'951 ns	382'093 ns	237'106 ns	266'952 ns
65'536	264'791 ns	377'919 ns	395'050 ns	460'229 ns
131'072	465'458 ns	621'136 ns	763'112 ns	780'680 ns

binary search

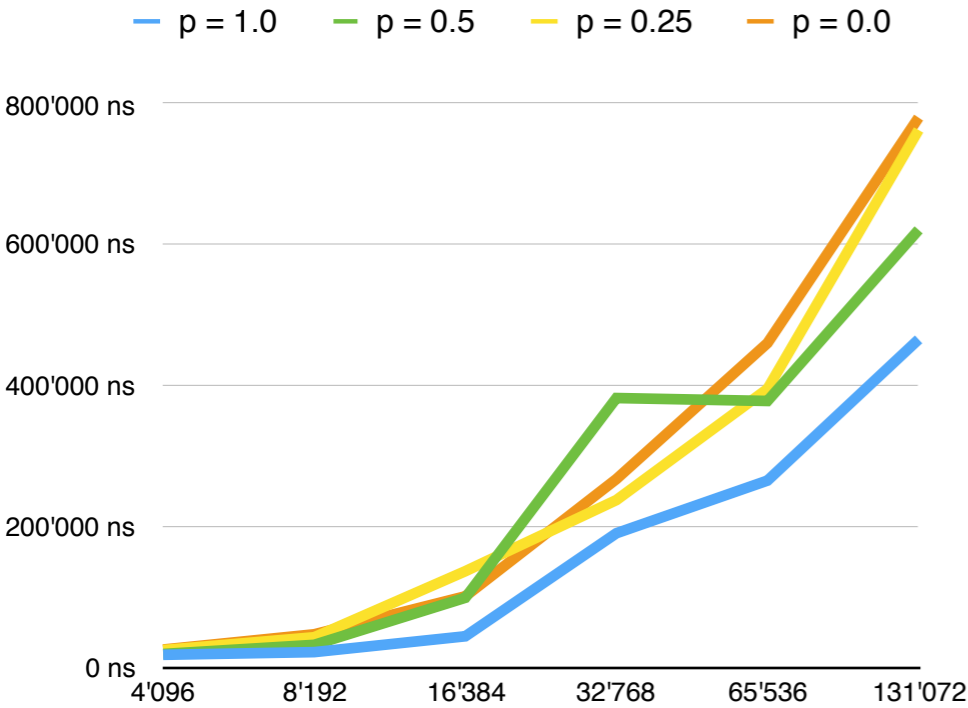
n	p = 1.0	p = 0.5	p = 0.25	p = 0.0
4'096	206 ns	191 ns	77 ns	78 ns
8'192	108 ns	92 ns	79 ns	75 ns
16'384	129 ns	98 ns	79 ns	93 ns
32'768	251 ns	186 ns	128 ns	111 ns
65'536	216 ns	142 ns	119 ns	81 ns
131'072	219 ns	174 ns	134 ns	164 ns



search performance



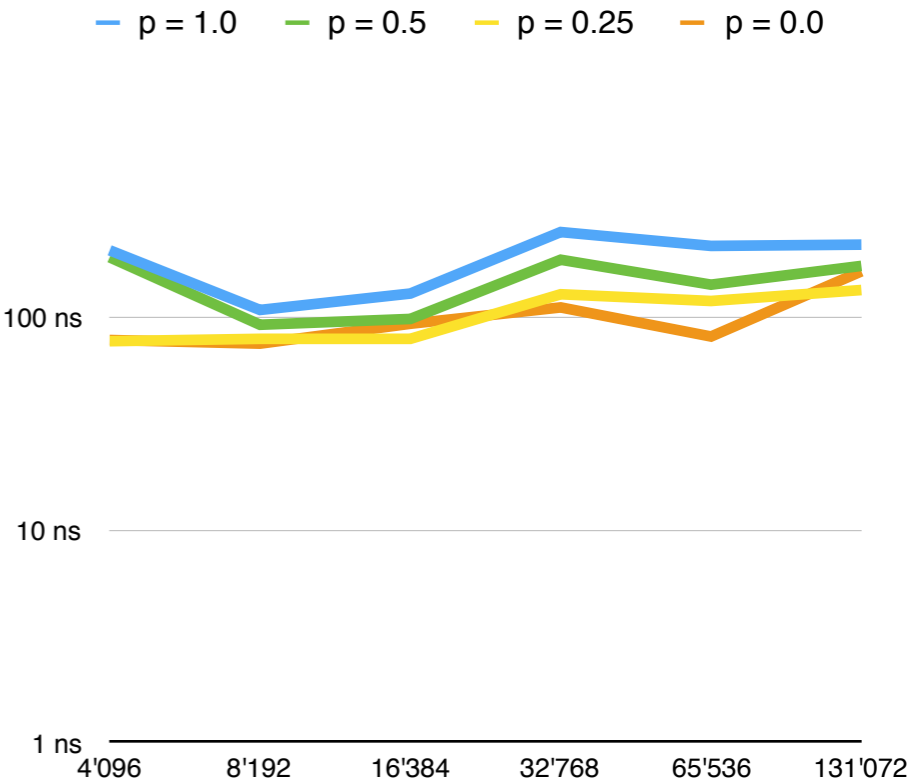
sequential search



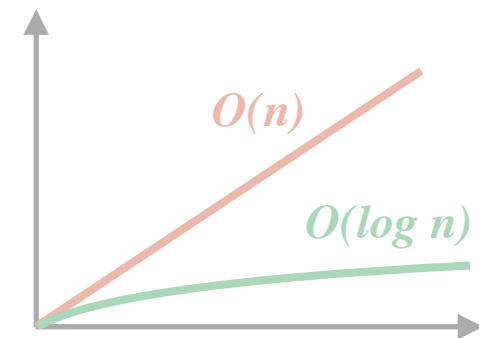
n	p = 1.0	p = 0.5	p = 0.25	p = 0.0
4'096	17'130 ns	18'306 ns	23'912 ns	24'645 ns
8'192	20'962 ns	31'311 ns	42'125 ns	46'303 ns
16'384	43'458 ns	98'216 ns	135'982 ns	100'311 ns
32'768	189'951 ns	382'093 ns	237'106 ns	266'952 ns
65'536	264'791 ns	377'919 ns	395'050 ns	460'229 ns
131'072	465'458 ns	621'136 ns	763'112 ns	780'680 ns

binary search

n	p = 1.0	p = 0.5	p = 0.25	p = 0.0
4'096	206 ns	191 ns	77 ns	78 ns
8'192	108 ns	92 ns	79 ns	75 ns
16'384	129 ns	98 ns	79 ns	93 ns
32'768	251 ns	186 ns	128 ns	111 ns
65'536	216 ns	142 ns	119 ns	81 ns
131'072	219 ns	174 ns	134 ns	164 ns



search performance



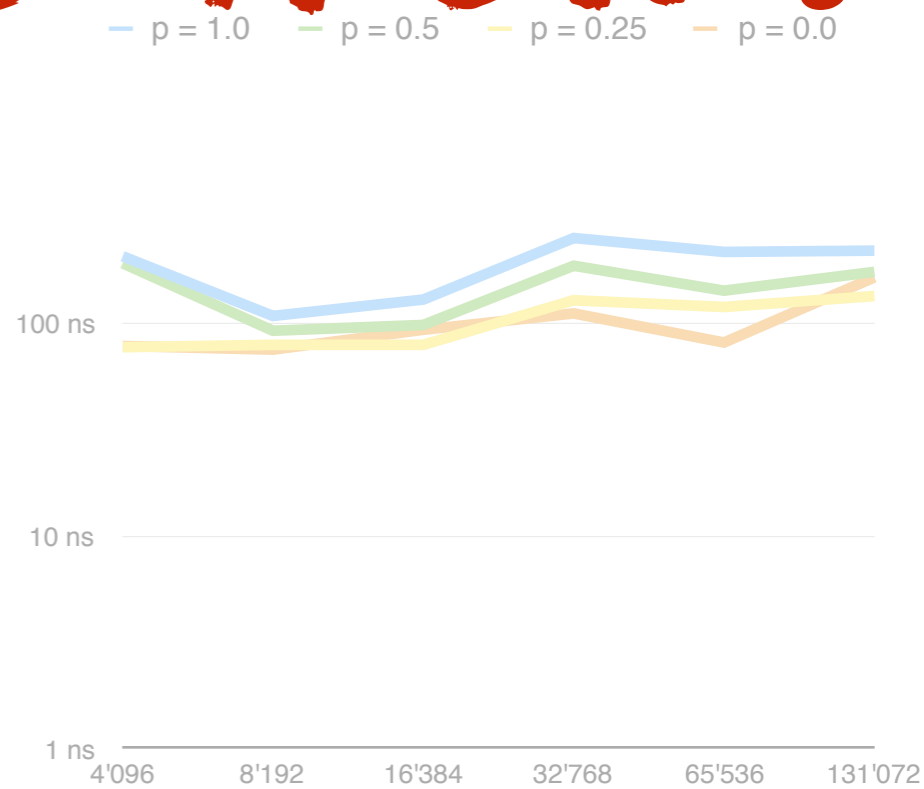
sequential search

n	p = 1.0	p = 0.5	p = 0.25	p = 0.0
4'096	17'130 ns	18'306 ns	23'912 ns	24'645 ns
8'192	20'962 ns	31'311 ns	42'125 ns	46'303 ns
16'384	42'458 ns	98'216 ns	135'982 ns	100'311 ns
32'768	18'455 ns	31'093 ns	27'112 ns	266'952 ns
65'536	264'791 ns	377'919 ns	395'050 ns	460'229 ns
131'072	465'458 ns	621'136 ns	763'112 ns	780'680 ns



binary search

n	p = 1.0	p = 0.5	p = 0.25	p = 0.0
4'096	206 ns	191 ns	77 ns	78 ns
8'192	108 ns	92 ns	79 ns	75 ns
16'384	129 ns	98 ns	79 ns	93 ns
32'768	251 ns	186 ns	128 ns	111 ns
65'536	216 ns	142 ns	119 ns	81 ns
131'072	219 ns	174 ns	134 ns	164 ns



where does the "magic" come from?

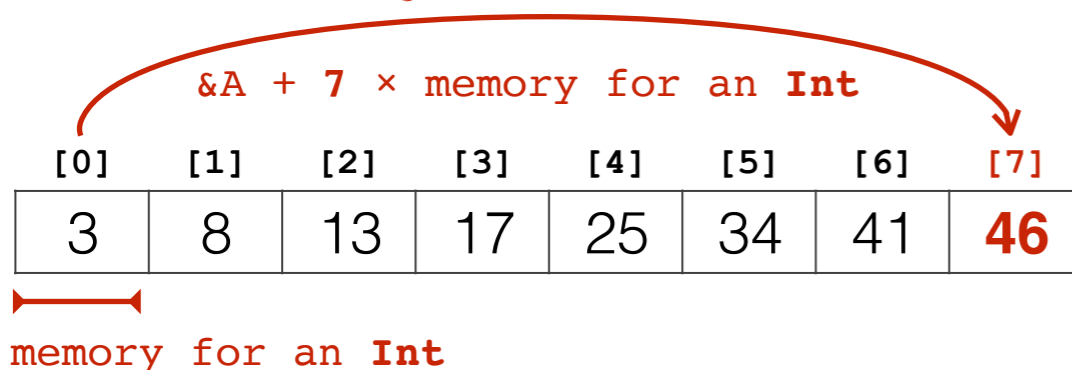
data structures

the **performance** of an algorithm often also **depends on the data structure**

the **binary search** requires a **sorted collection**, so part of the cost is goes into **sorting the collection**

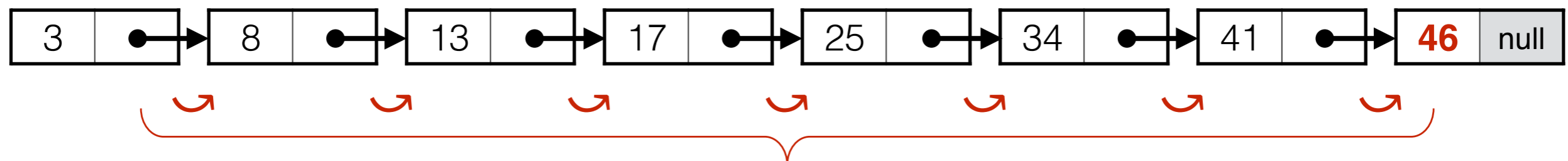
array

single operation



accessing a particular element in a collection, say $A[7]$

linked list



several complex operations to follow links until the searched element

data structures

sorted array

remove element

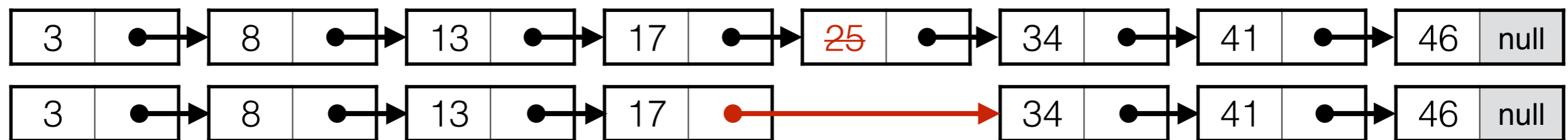
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
3	8	13	17	25	34	41	46
3	8	13	17	34		41	46
3	8	13	17	34	41		46
3	8	13	17	34	41	46	

add element

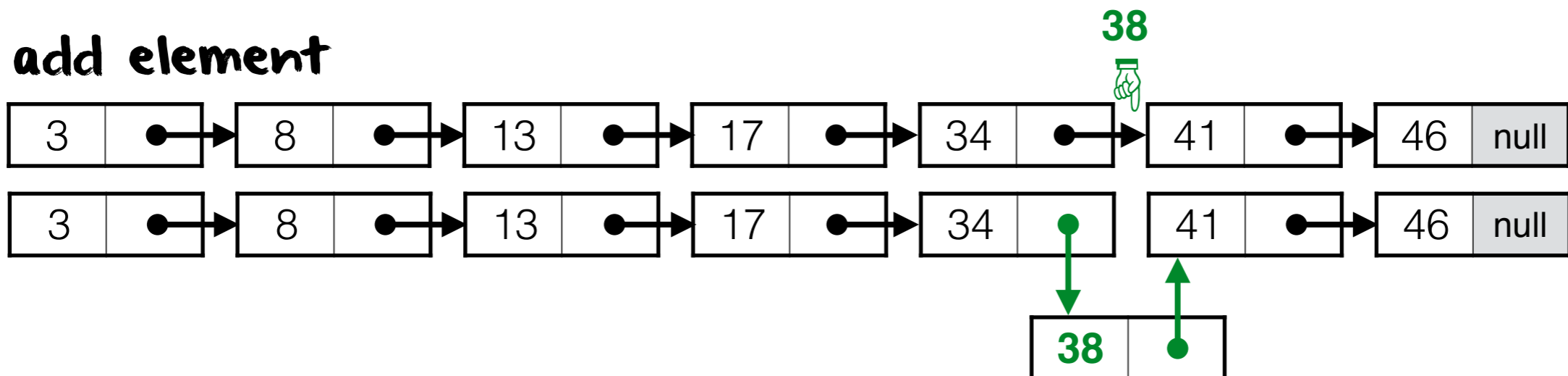
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
3	8	13	17	34	41	46	
3	8	13	17	34	41		46
3	8	13	17	34		41	46
3	8	13	17	34	38	41	46

sorted linked list

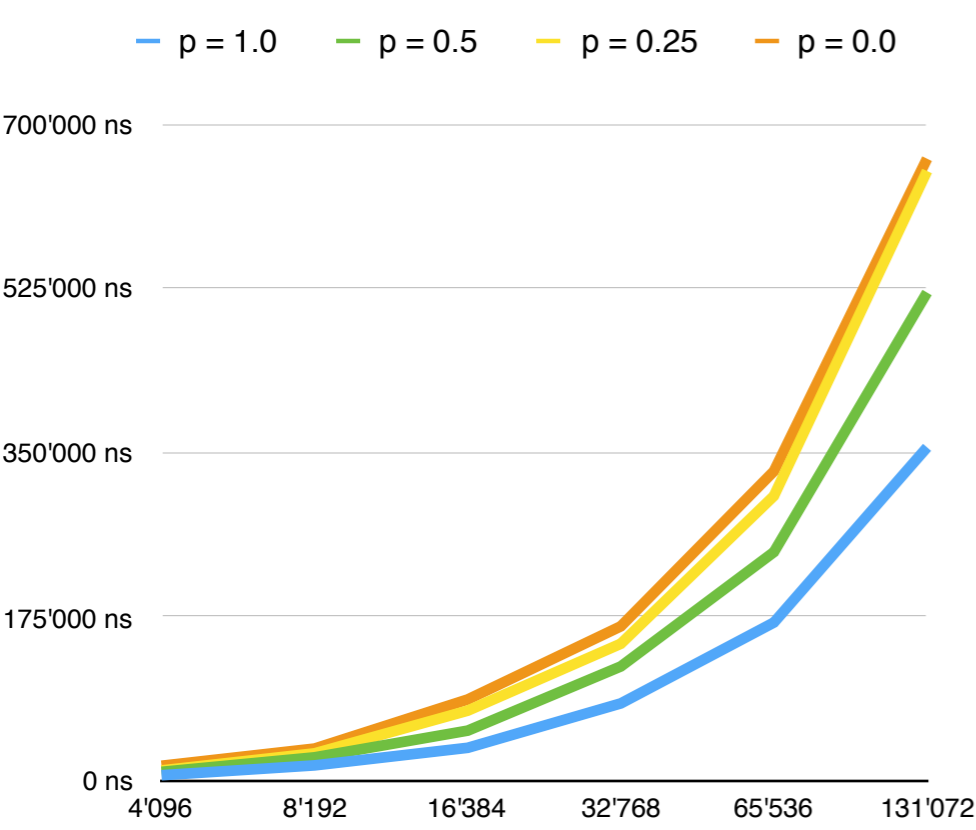
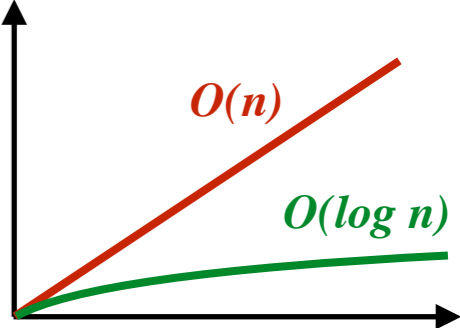
remove element



add element



search performance

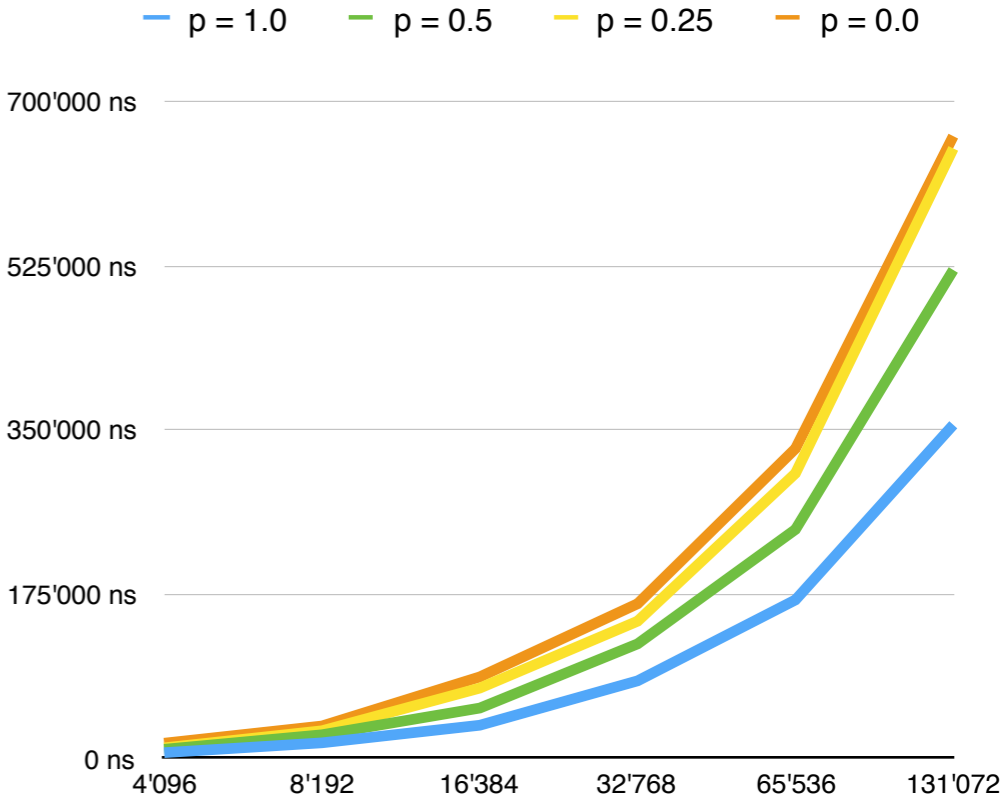


sequential search in a linked list

n	p = 1.0	p = 0.5	p = 0.25	p = 0.0
4'096	6'376 ns	9'951 ns	11'516 ns	16'447 ns
8'192	16'865 ns	25'425 ns	29'715 ns	34'780 ns
16'384	35'585 ns	53'881 ns	75'148 ns	87'058 ns
32'768	82'872 ns	122'246 ns	146'411 ns	164'986 ns
65'536	169'044 ns	244'068 ns	303'836 ns	330'198 ns
131'072	355'536 ns	520'393 ns	649'836 ns	662'913 ns

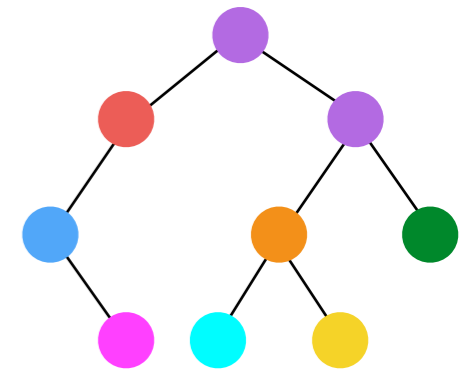
binary search in a linked list

n	p = 1.0	p = 0.5	p = 0.25	p = 0.0
4'096	89'863 ns	174'655 ns	200'748 ns	238'440 ns
8'192	193'307 ns	360'377 ns	457'862 ns	565'766 ns
16'384	482'478 ns	886'582 ns	1'038'265 ns	1'187'520 ns
32'768	1'098'888 ns	1'765'568 ns	2'146'730 ns	2'693'595 ns
65'536	2'593'438 ns	4'261'790 ns	5'019'401 ns	5'911'436 ns
131'072	5'265'582 ns	9'724'940 ns	11'840'437 ns	14'867'349 ns



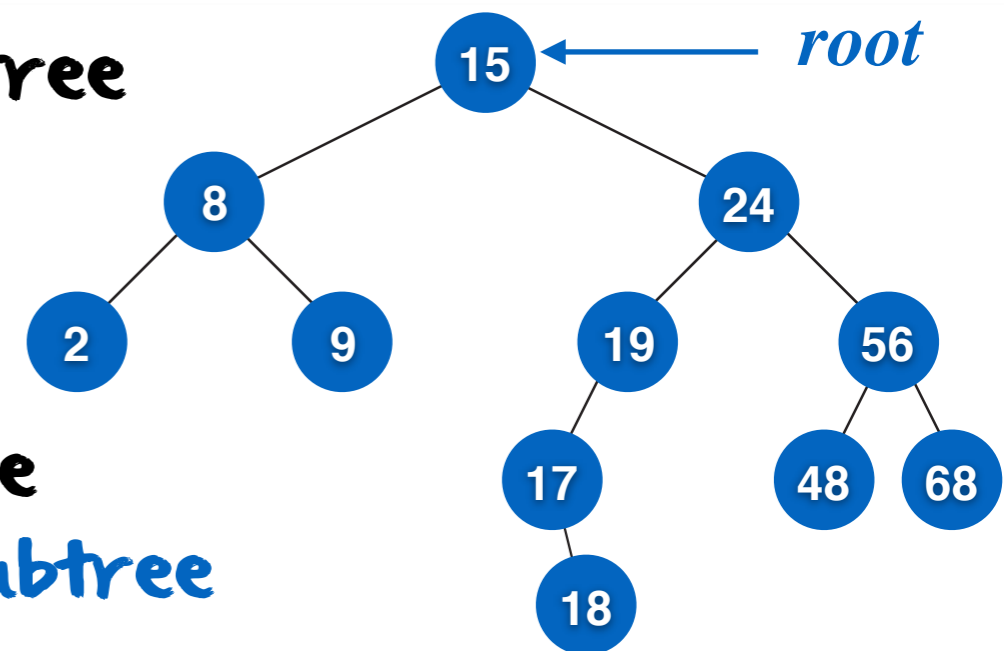
binary search trees

a **binary tree** is a tree data structure where each node has **at most two children links**, which are referred to as the **left child** and the **right child**



a **binary search tree** is a **rooted** binary tree with the following properties:

- ◆ each node has a **comparable key**
- ◆ the key of any node is **larger than** the keys of all nodes in that node's **left subtree**
- ◆ the key of any node is **smaller than** the keys of all nodes in that node's **right subtree**



a **subtree** is simply the tree that is a child of a node

binary search trees

in addition, each node might also have:

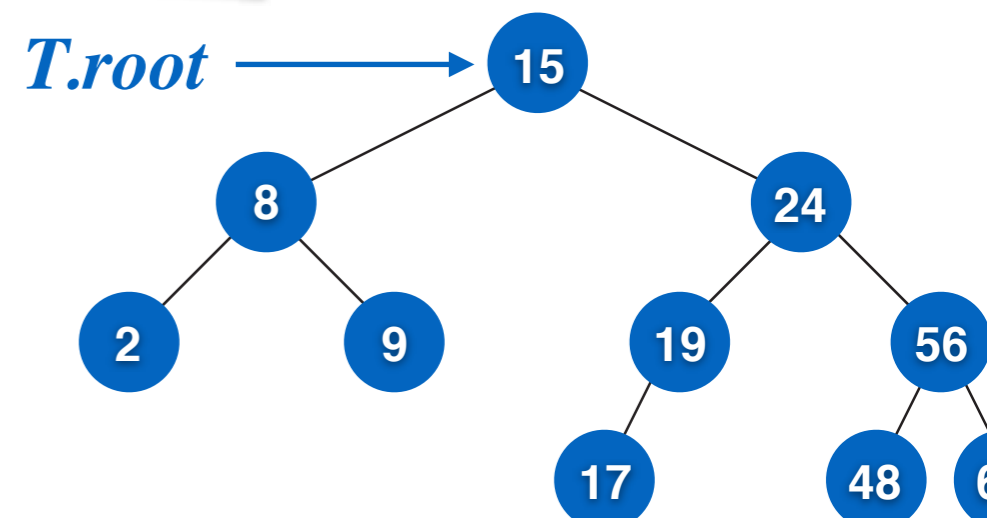
- ◆ a value (in the case of associative arrays)
- ◆ a link to its parent in the tree, often noted p

in full generality, a node of the binary search tree is thus a tuple of the form $(key, value, left, right, p)$

these tuple elements are usually designed as

$x.key$ $x.value$ $x.left$ $x.right$ $x.p$

the tree itself is usually noted T and has a root attribute, noted $T.root$ pointing to the first node of T



binary search trees

some algorithms

INORDER-TREE-WALK(x)

```
1 if  $x \neq \text{NIL}$ 
2   INORDER-TREE-WALK( $x.\text{left}$ )
3   print  $x.\text{key}$ 
4   INORDER-TREE-WALK( $x.\text{right}$ )
```

TREE-SEARCH(x, k)

```
1 if  $x == \text{NIL}$  or  $k == x.\text{key}$ 
2   return  $x$ 
3 if  $k < x.\text{key}$ 
4   return TREE-SEARCH( $x.\text{left}, k$ )
5 else return TREE-SEARCH( $x.\text{right}, k$ )
```

ITERATIVE-TREE-SEARCH(x, k)

```
1 while  $x \neq \text{NIL}$  and  $k \neq x.\text{key}$ 
2   if  $k < x.\text{key}$ 
3      $x = x.\text{left}$ 
4   else  $x = x.\text{right}$ 
5 return  $x$ 
```

TREE-MINIMUM(x)

```
1 while  $x.\text{left} \neq \text{NIL}$ 
2    $x = x.\text{left}$ 
3 return  $x$ 
```

no need to compare keys!



TREE-MAXIMUM(x)

```
1 while  $x.\text{right} \neq \text{NIL}$ 
2    $x = x.\text{right}$ 
3 return  $x$ 
```

TREE-SUCCESSOR(x)

```
1 if  $x.\text{right} \neq \text{NIL}$ 
2   return TREE-MINIMUM( $x.\text{right}$ )
3  $y = x.p$ 
4 while  $y \neq \text{NIL}$  and  $x == y.\text{right}$ 
5    $x = y$ 
6    $y = y.p$ 
7 return  $y$ 
```

TREE-INSERT(T, z)

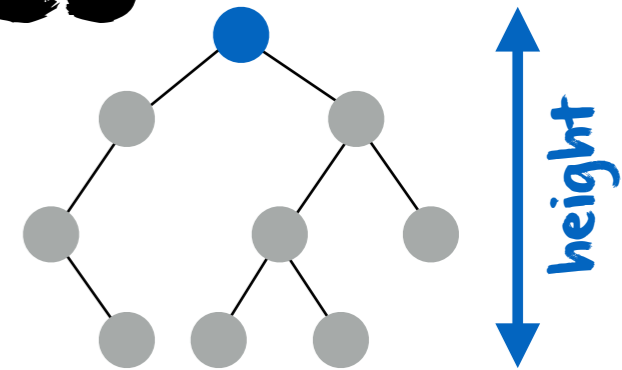
```
1  $y = \text{NIL}$ 
2  $x = T.\text{root}$ 
3 while  $x \neq \text{NIL}$ 
4    $y = x$ 
5   if  $z.\text{key} < x.\text{key}$ 
6      $x = x.\text{left}$ 
7   else  $x = x.\text{right}$ 
8  $z.p = y$ 
9 if  $y == \text{NIL}$ 
10    $T.\text{root} = z$ 
11 elseif  $z.\text{key} < y.\text{key}$ 
12    $y.\text{left} = z$ 
13 else  $y.\text{right} = z$ 
```

tree was empty



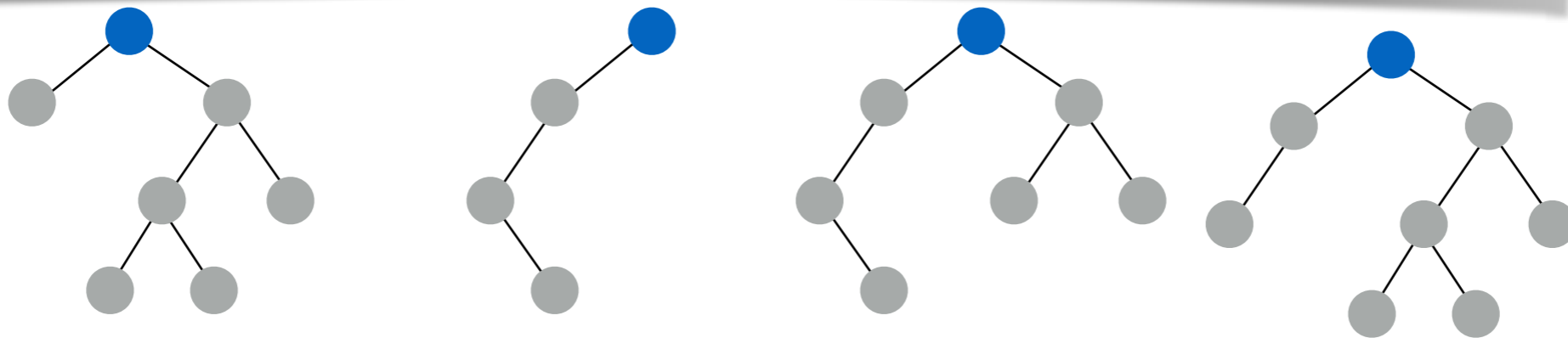
balanced trees

the height of a tree is the maximum distance of any node from the root



a height-balanced (or simply balanced) tree is a tree whose subtrees have the following properties:

- ◆ they differ in height by no more than one
- ◆ they are height-balanced as well



why is it interesting to use a balanced binary search tree?

