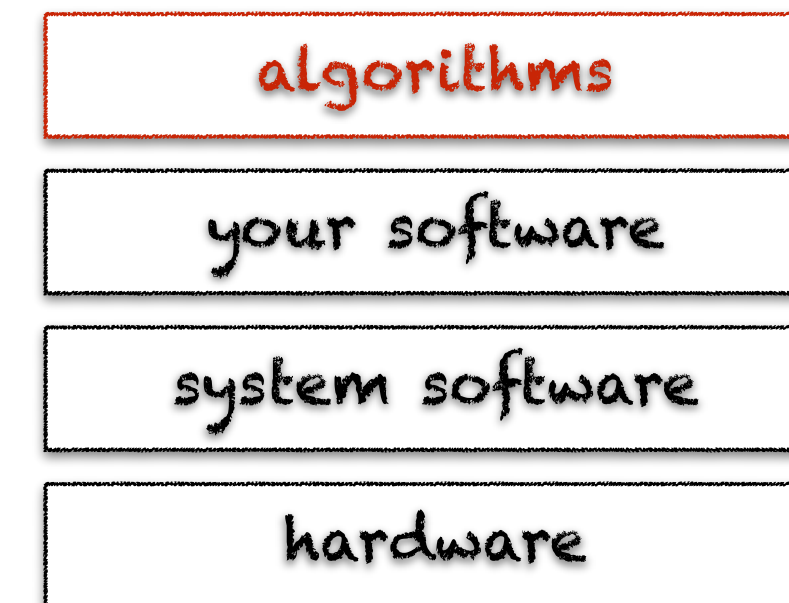




search
algorithms

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 a **brute-force** approach

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

also called **linear search**

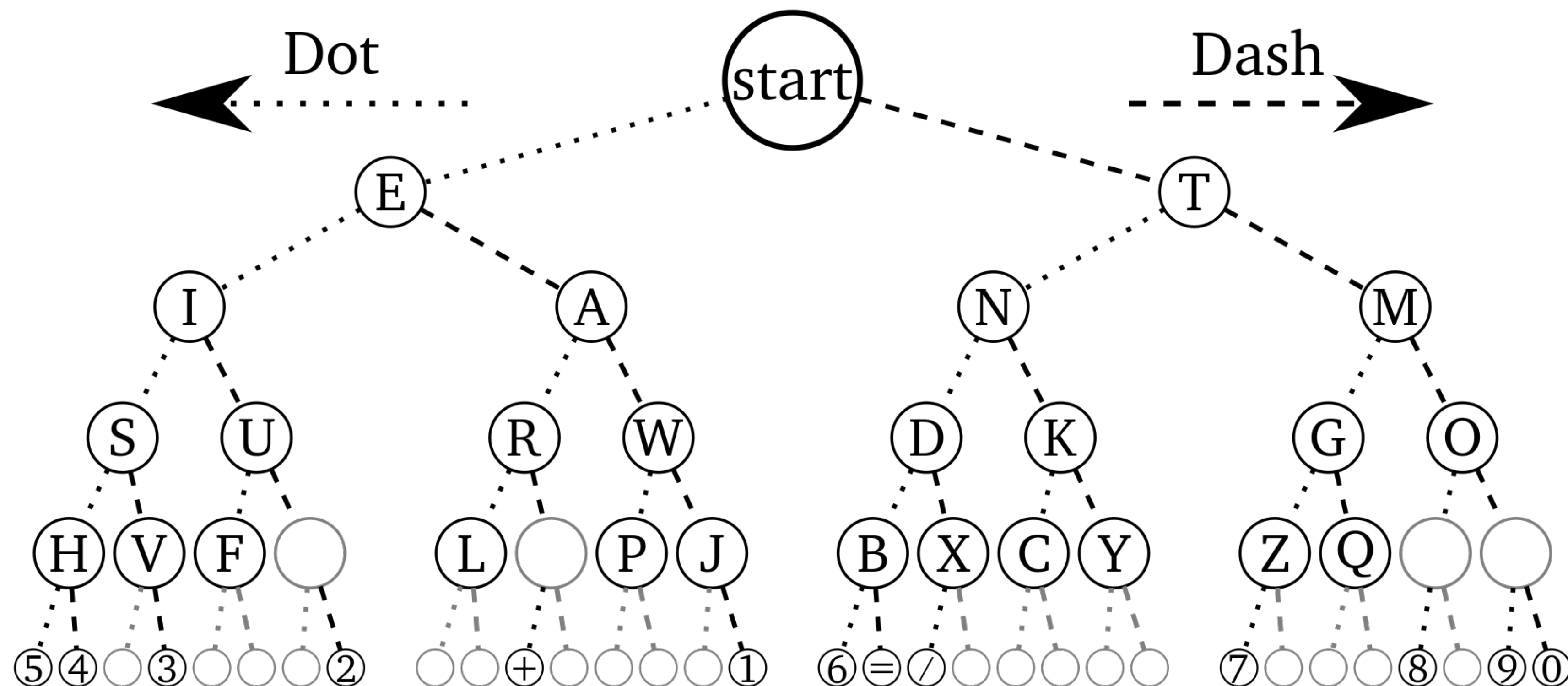
worst case: $O(n)$
best case: $O(1)$

```
int sequentialSearch(int theKey, int[] theArray) {  
    for (int i = 0; i < theArray.length; i++) {  
        if (theKey == theArray[i]) {  
            return i;  
        }  
    }  
    return -1;  
}
```



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	• • • •		
I	• •		
J	• — — —		
K	— — — •		
L	• — • •		
M	— —		
N	— •		
O	— — —		
P	• — — •		
Q	— — • —		
R	• — •		
S	• • •		
T	—		

1	• — — — —
2	• • — — —
3	• • • — —
4	• • • • —
5	• • • • •
6	— • • • •
7	— • • • •
8	— — • • •
9	— — — • •
0	— — — — •

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* ≤ *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: $O(1)$

binary search

```
int binarySearch(int theKey, int[] theArray) {  
    int low = 0;  
    int high = theArray.length - 1;  
  
    while (low <= high) {  
        int mid = (low + high) / 2;  
        if (theKey < theArray[mid]) {  
            high = mid - 1;  
        } else if (theKey > theArray[mid]) {  
            low = mid + 1;  
        } else { return mid; }  
    }  
    return -1;  
}
```



[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

binary search

```
int binarySearch(int theKey, int[] theArray) {  
    int low = 0;  
    int high = theArray.length - 1;  
  
    while (low <= high) {  
        int mid = (low + high) / 2;  
        if (theKey < theArray[mid]) {  
            high = mid - 1;  
        } else if (theKey > theArray[mid]) {  
            low = mid + 1;  
        } else { return mid; }  
    }  
    return -1;  
}
```



[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

binary search

```
int binarySearch(int theKey, int[] theArray) {  
    int low = 0;  
    int high = theArray.length - 1;  
  
    while (low <= high) {  
        int mid = (low + high) / 2;  
        if (theKey < theArray[mid]) {  
            high = mid - 1;  
        } else if (theKey > theArray[mid]) {  
            low = mid + 1;  
        } else { return mid; }  
    }  
    return -1;  
}
```



[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

```

int binarySearch(int theKey, int[] theArray) {
    int low = 0;
    int high = theArray.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (theKey < theArray[mid]) {
            high = mid - 1;
        } else if (theKey > theArray[mid]) {
            low = mid + 1;
        } else { return mid; }
    }
    return -1;
}

```



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

```

int binarySearch(int theKey, int[] theArray) {
    int low = 0;
    int high = theArray.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (theKey < theArray[mid]) {
            high = mid - 1;
        } else if (theKey > theArray[mid]) {
            low = mid + 1;
        } else { return mid; }
    }
    return -1;
}

```



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

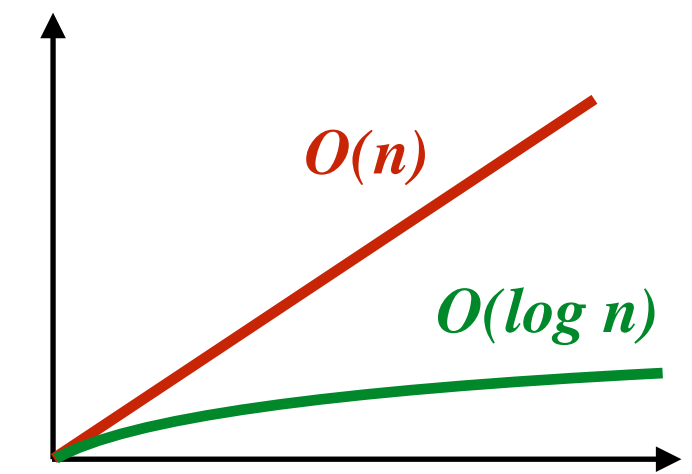
3	8	13	17	25	34	41	46	55	63	74	78	81	82	93	99
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

↑
mid
74 = 74



search performance

worse case



```
int binarySearch(int theKey, int[] theArray) {
    int low = 0;
    int high = theArray.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (theKey < theArray[mid]) {
            high = mid - 1;
        } else if (theKey > theArray[mid]) {
            low = mid + 1;
        } else { return mid; }
    }
    return -1;
}
```

```
int sequentialSearch(int theKey, int[] theArray) {
    for (int i = 0; i < theArray.length; i++) {
        if (theKey == theArray[i]) {
            return i;
        }
    }
    return -1;
}
```

Sequential	n = 10	n = 100	n = 1'000	n = 10'000	n = 100'000	n = 1'000'000
Time	812 ns	2'910 ns	24'460 ns	264'193 ns	1'629'898 ns	2'308'789 ns
Iterations	10	100	1'000	10'000	100'000	1'000'000
Iteration Time	81 ns	29 ns	24 ns	26 ns	16 ns	2 ns

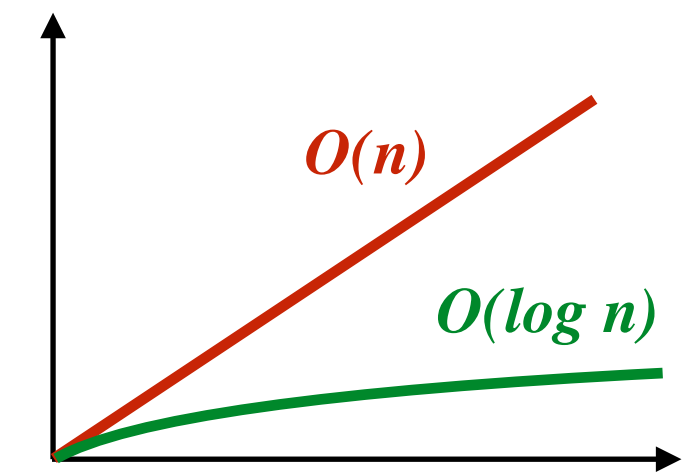
Binary	n = 10	n = 100	n = 1'000	n = 10'000	n = 100'000	n = 1'000'000
Time	603 ns	683 ns	859 ns	1'293 ns	2'353 ns	5'976 ns
Iterations	4	7	10	14	17	20
Iteration Time	151 ns	98 ns	86 ns	92 ns	138 ns	299 ns

Ratio	1.35	4.26	28.47	204.33	692.69	386.34
-------	------	------	-------	--------	--------	--------



search performance

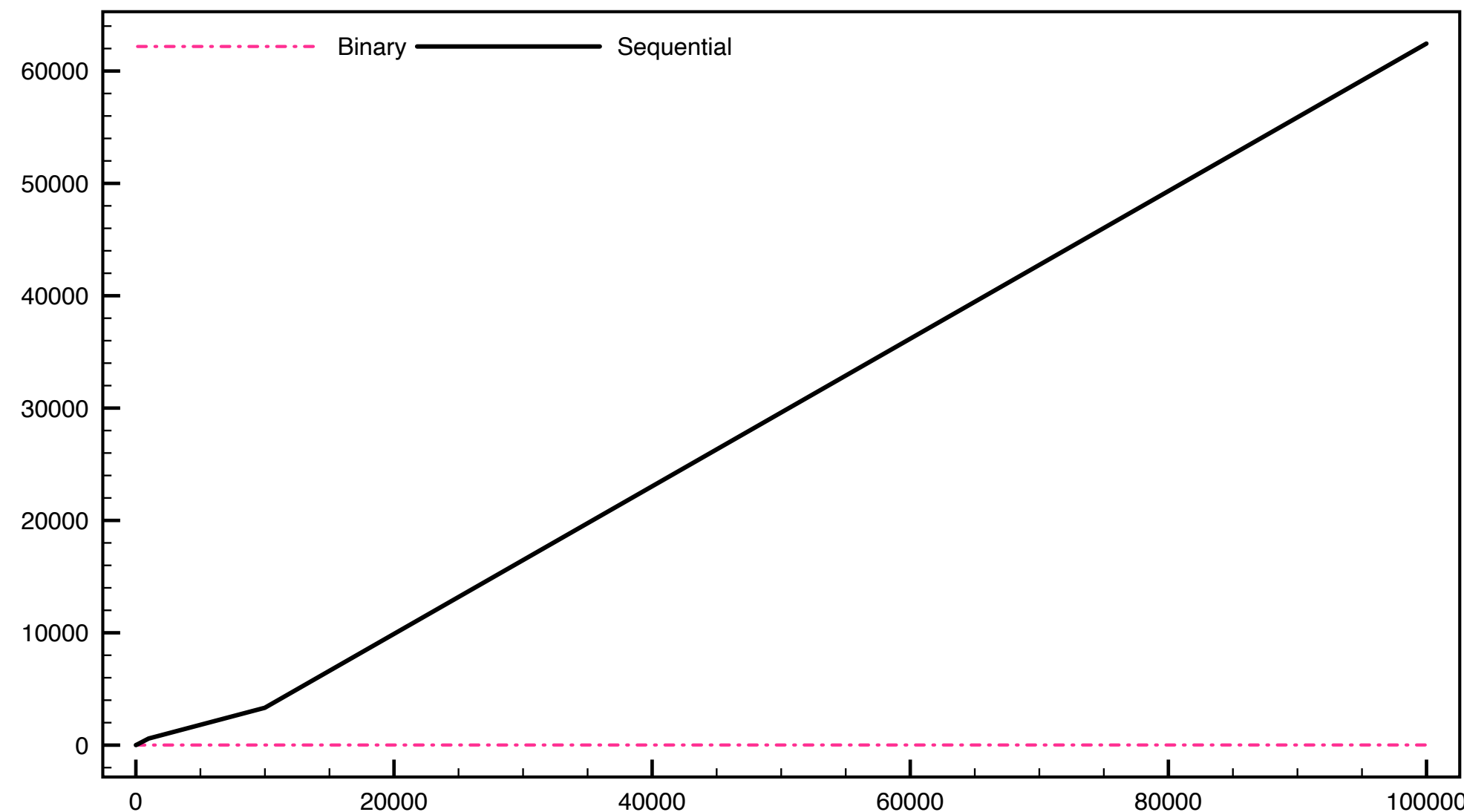
average case



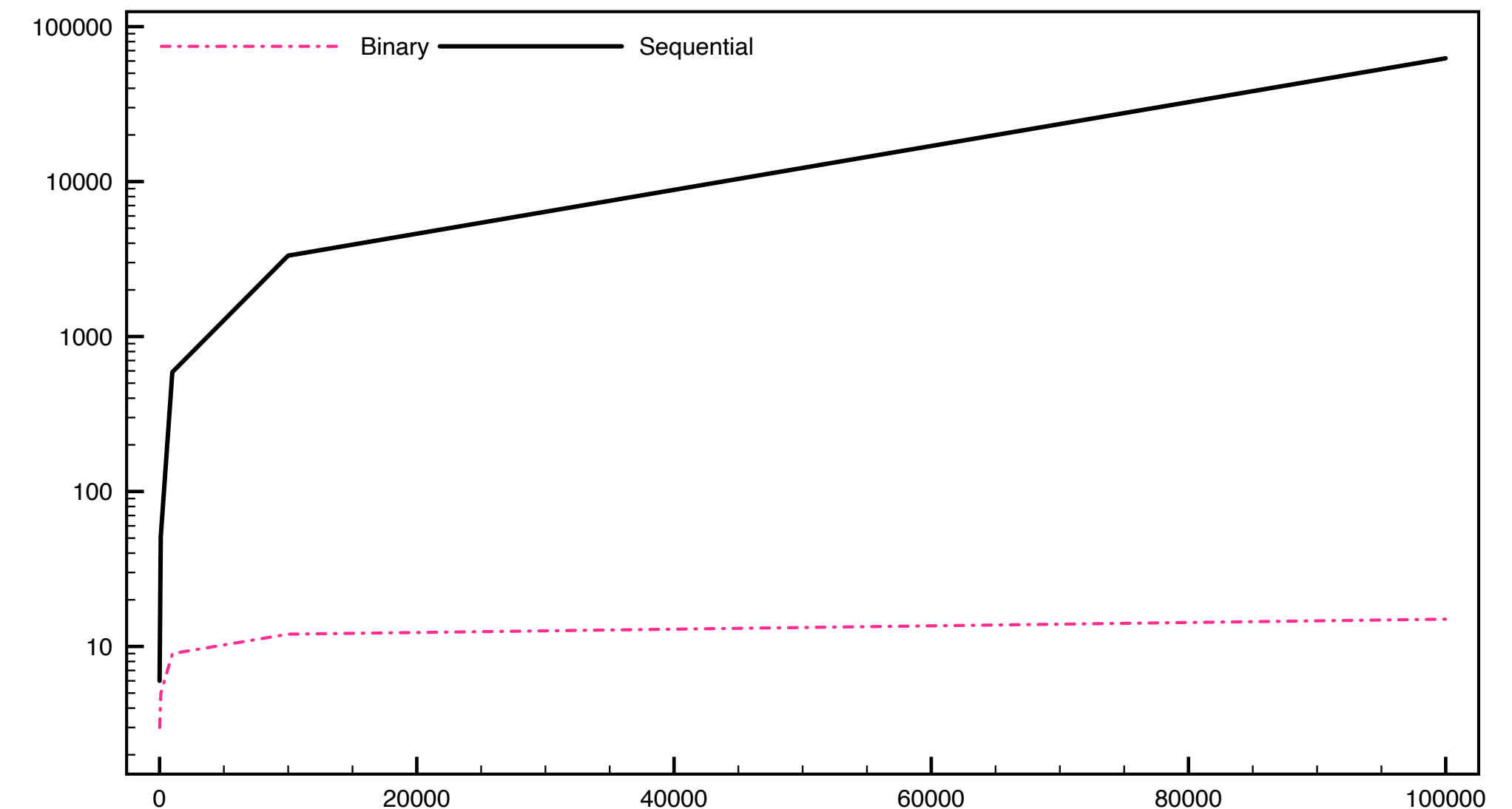
```
int binarySearch(int theKey, int[] theArray) {  
    int low = 0;  
    int high = theArray.length - 1;  
  
    while (low <= high) {  
        int mid = (low + high) / 2;  
        if (theKey < theArray[mid]) {  
            high = mid - 1;  
        } else if (theKey > theArray[mid]) {  
            low = mid + 1;  
        } else { return mid; }  
    }  
    return -1;  
}
```

```
int sequentialSearch(int theKey, int[] theArray) {  
    for (int i = 0; i < theArray.length; i++) {  
        if (theKey == theArray[i]) {  
            return i;  
        }  
    }  
    return -1;  
}
```

linear scale



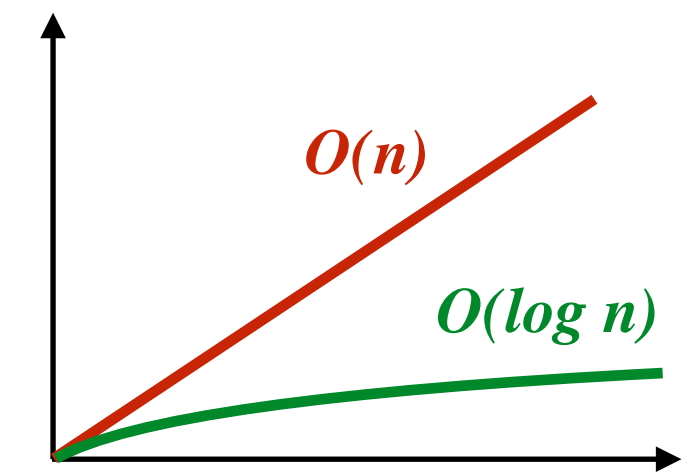
logarithmic scale





search performance

average case



```
int binarySearch(int theKey, int[] theArray) {
    int low = 0;
    int high = theArray.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (theKey < theArray[mid]) {
            high = mid - 1;
        } else if (theKey > theArray[mid]) {
            low = mid + 1;
        } else { return mid; }
    }
    return -1;
}
```

```
int sequentialSearch(int theKey, int[] theArray) {
    for (int i = 0; i < theArray.length; i++) {
        if (theKey == theArray[i]) {
            return i;
        }
    }
    return -1;
}
```

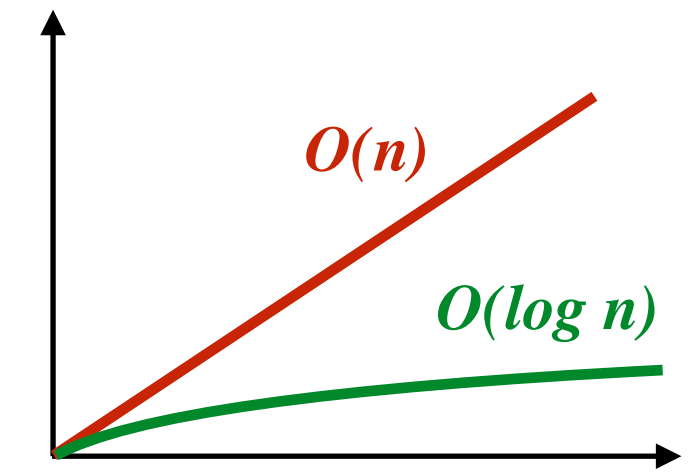
Sequential	n = 10	n = 100	n = 1'000	n = 10'000	n = 100'000	n = 1'000'000
Time	661 ns	1'834 ns	14'651 ns	85'793 ns	1'494'208 ns	2'506'030 ns
Iterations	6	51	589	3'326	62'442	429'271
Iteration Time	110 ns	36 ns	25 ns	26 ns	24 ns	6 ns

Binary	n = 10	n = 100	n = 1'000	n = 10'000	n = 100'000	n = 1'000'000
Time	566 ns	716 ns	877 ns	1'176 ns	2'781 ns	5'949 ns
Iterations	3	5	9	12	15	19
Iteration Time	189 ns	143 ns	97 ns	98 ns	185 ns	313 ns

Ratio	1.17	2.56	16.71	72.95	537.29	421.25
-------	------	------	-------	-------	--------	--------



search performance



average case
ArrayList

```
int binarySearch(int theKey, List<Integer> theList) {
    int low = 0;
    int high = theList.size() - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (theKey < theList.get(mid)) {
            high = mid - 1;
        } else if (theKey > theList.get(mid)) {
            low = mid + 1;
        } else { return mid; }
    }
    return -1;
}
```

```
int sequentialSearch(int theKey, List<Integer> theList) {
    for (int i = 0; i < theList.size(); i++) {
        if (theKey == theList.get(i)) {
            return i;
        }
    }
    return -1;
}
```

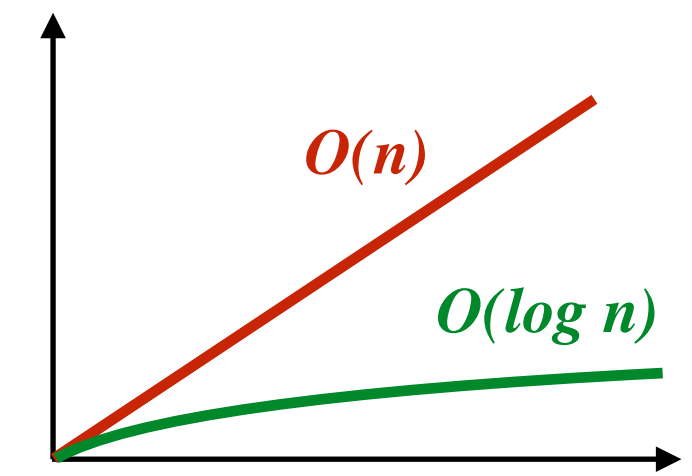
Sequential	n = 10	n = 100	n = 1'000	n = 10'000	n = 100'000	n = 1'000'000
Time	3'587 ns	28'774 ns	178'565 ns	556'390 ns	2'264'037 ns	1'944'778 ns
Iterations	6	52	672	7'126	60'140	338'636
Iteration Time	598 ns	553 ns	266 ns	78 ns	38 ns	6 ns

Binary	n = 10	n = 100	n = 1'000	n = 10'000	n = 100'000	n = 1'000'000
Time	2'119 ns	4'659 ns	2'904 ns	1'964 ns	5'270 ns	5'899 ns
Iterations	2	6	8	12	16	18
Iteration Time	1'060 ns	777 ns	363 ns	164 ns	329 ns	328 ns

Ratio	1.69	6.18	61.49	283.29	429.61	329.68
-------	------	------	-------	--------	--------	--------



search performance



average case

LinkedList

```
int binarySearch(int theKey, List<Integer> theList) {
    int low = 0;
    int high = theList.size() - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (theKey < theList.get(mid)) {
            high = mid - 1;
        } else if (theKey > theList.get(mid)) {
            low = mid + 1;
        } else { return mid; }
    }
    return -1;
}
```

```
int sequentialSearch(int theKey, List<Integer> theList) {
    for (int i = 0; i < theList.size(); i++) {
        if (theKey == theList.get(i)) {
            return i;
        }
    }
    return -1;
}
```

Sequential	n = 10	n = 100	n = 1'000	n = 10'000	n = 100'000	n = 1'000'000
Time	3'147 ns	24'782 ns	460'278 ns	21'326'699 ns	1'851'252'103 ns	199'371'399'466 ns
Iterations	3	25	547	5'102	45'376	260'550
Iteration Time	1'049 ns	991 ns	841 ns	4'180 ns	40'798 ns	765'194 ns

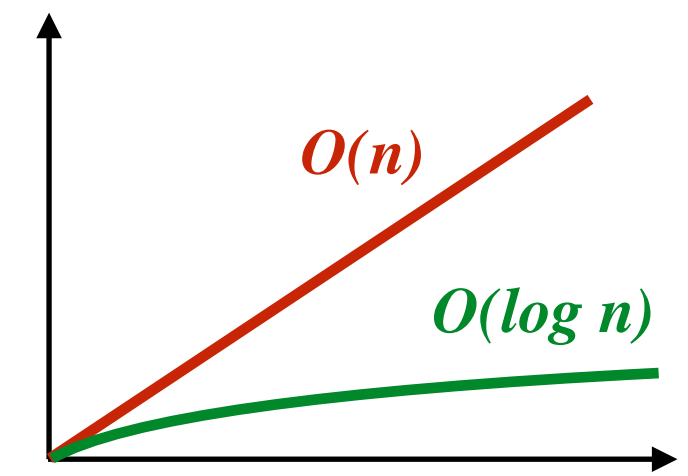
Binary	n = 10	n = 100	n = 1'000	n = 10'000	n = 100'000	n = 1'000'000
Time	2'084 ns	7'586 ns	11'975 ns	61'329 ns	1'045'852 ns	63'417'680 ns
Iterations	2	5	7	11	15	17
Iteration Time	1'042 ns	1'517 ns	1'711 ns	5'575 ns	69'723 ns	3'730'452 ns

Ratio	1.51	3.27	38.44	347.74	1'770.09	3'143.78
-------	------	------	-------	--------	----------	----------



search performance

average case



```
int binarySearch(int theKey, List<Integer> theList) {
    int low = 0;
    int high = theList.size() - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (theKey < theList.get(mid)) {
            high = mid - 1;
        } else if (theKey > theList.get(mid)) {
            low = mid + 1;
        } else { return mid; }
    }
    return -1;
}
```

```
int sequentialSearch(int theKey, List<Integer> theList) {
    for (int i = 0; i < theList.size(); i++) {
        if (theKey == theList.get(i)) {
            return i;
        }
    }
    return -1;
}
```

```
int sequentialSearch(int theKey, int[] theArray) {
    for (int i = 0; i < theArray.length; i++) {
        if (theKey == theArray[i]) {
            return i;
        }
    }
    return -1;
}
```

int[]

Sequential

	n = 10	n = 100	n = 1'000	n = 10'000	n = 100'000	n = 1'000'000
Time	661 ns	1'834 ns	14'651 ns	85'793 ns	1'494'208 ns	2'506'030 ns
Iterations	6	51	589	3'326	62'442	429'271

ArrayList

Sequential

	n = 10	n = 100	n = 1'000	n = 10'000	n = 100'000	n = 1'000'000
Time	3'587 ns	28'774 ns	178'565 ns	556'390 ns	2'264'037 ns	1'944'778 ns
Iterations	6	52	672	7'126	60'140	338'636

LinkedList

Binary

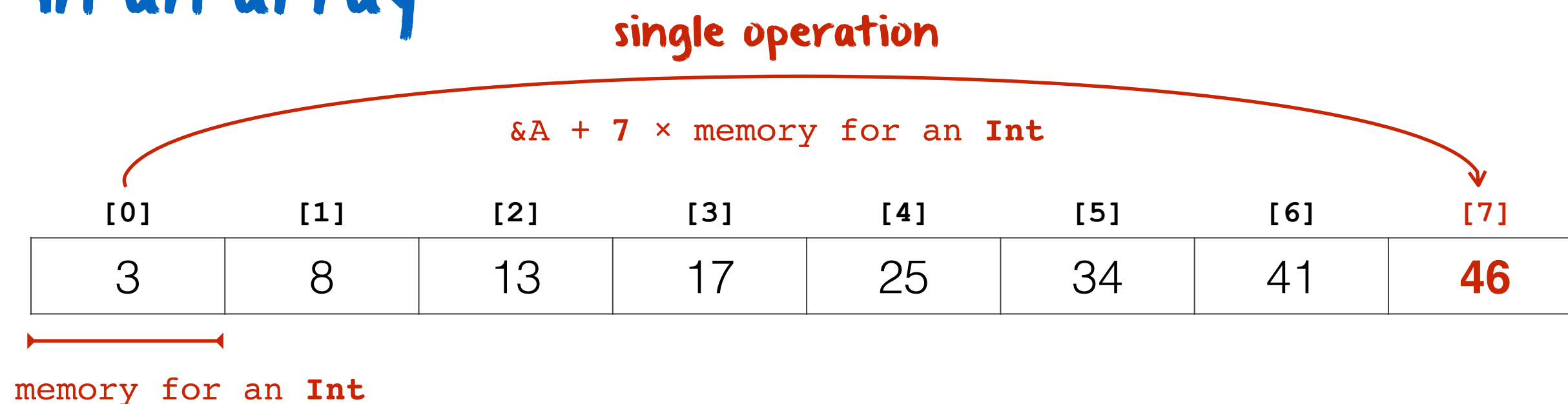
	n = 10	n = 100	n = 1'000	n = 10'000	n = 100'000	n = 1'000'000
Time	2'084 ns	7'586 ns	11'975 ns	61'329 ns	1'045'852 ns	63'417'680 ns
Iterations	2	5	7	11	15	17

data structure

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

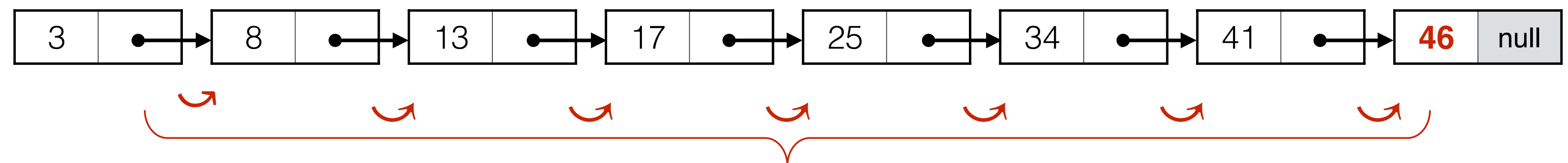
the binary search requires a sorted collection, so part of the cost goes into sorting the collection

in an array



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

in a linked list



potentially long list of operations to follow links until the searched element

data structure

sorted array

remove element

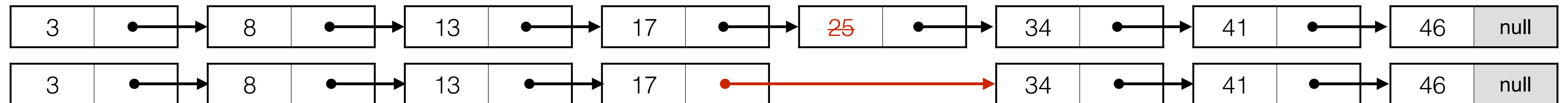
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
3	8	13	17	25	34	41	46
3	8	13	17	34		41	46
3	8	13	17	34	41		46
3	8	13	17	34	41	46	

add element

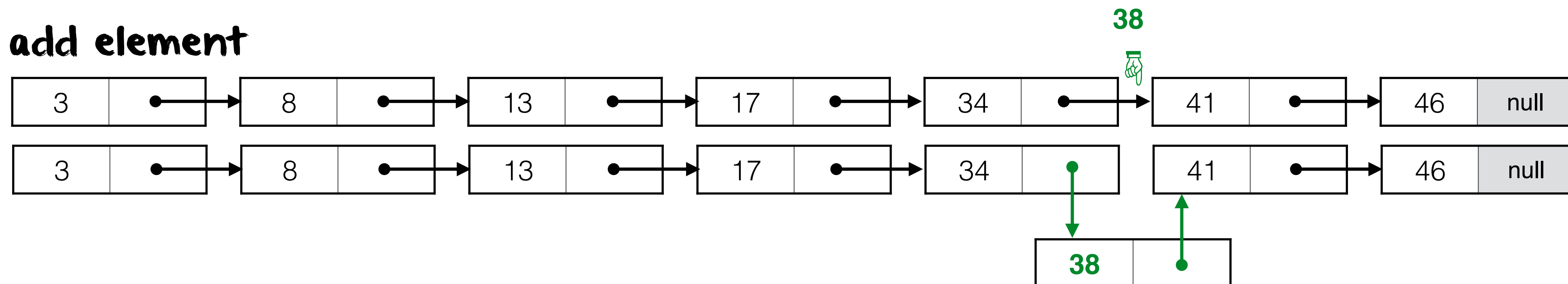
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
3	8	13	17	34		41	46
3	8	13	17	34	41	46	
3	8	13	17	34	41		46
3	8	13	17	34	38	41	46

sorted linked list


remove element




add element




data structure & search performance

 Python, sequential : 65739000 ns

 Java, sequential, int[] : 4683126 ns

 Java, sequential, ArrayList : 3793184 ns

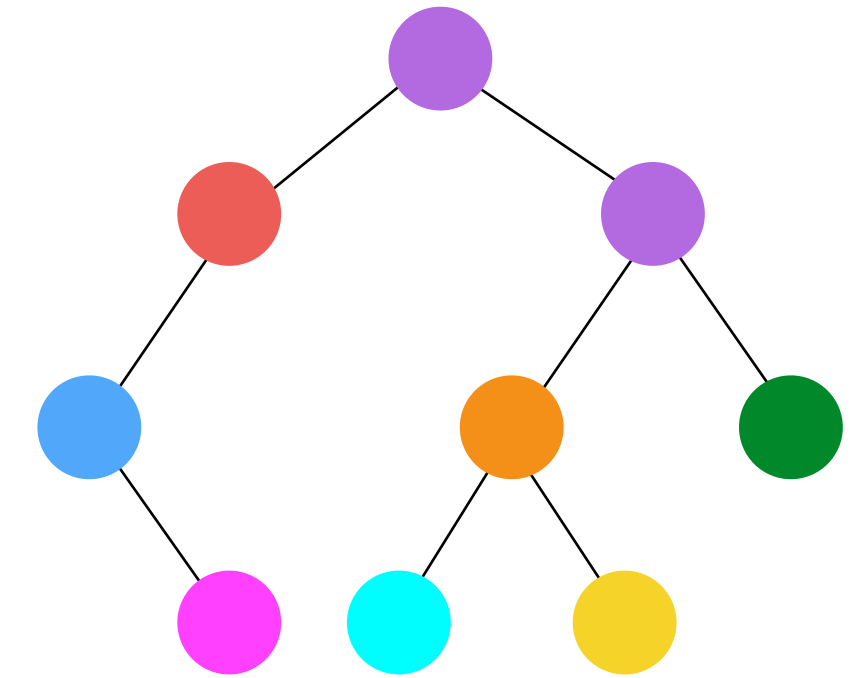
 Python, binary : 11000 ns

 Java, binary, int[] : 5304 ns

 Java, binary, ArrayList : 10069 ns

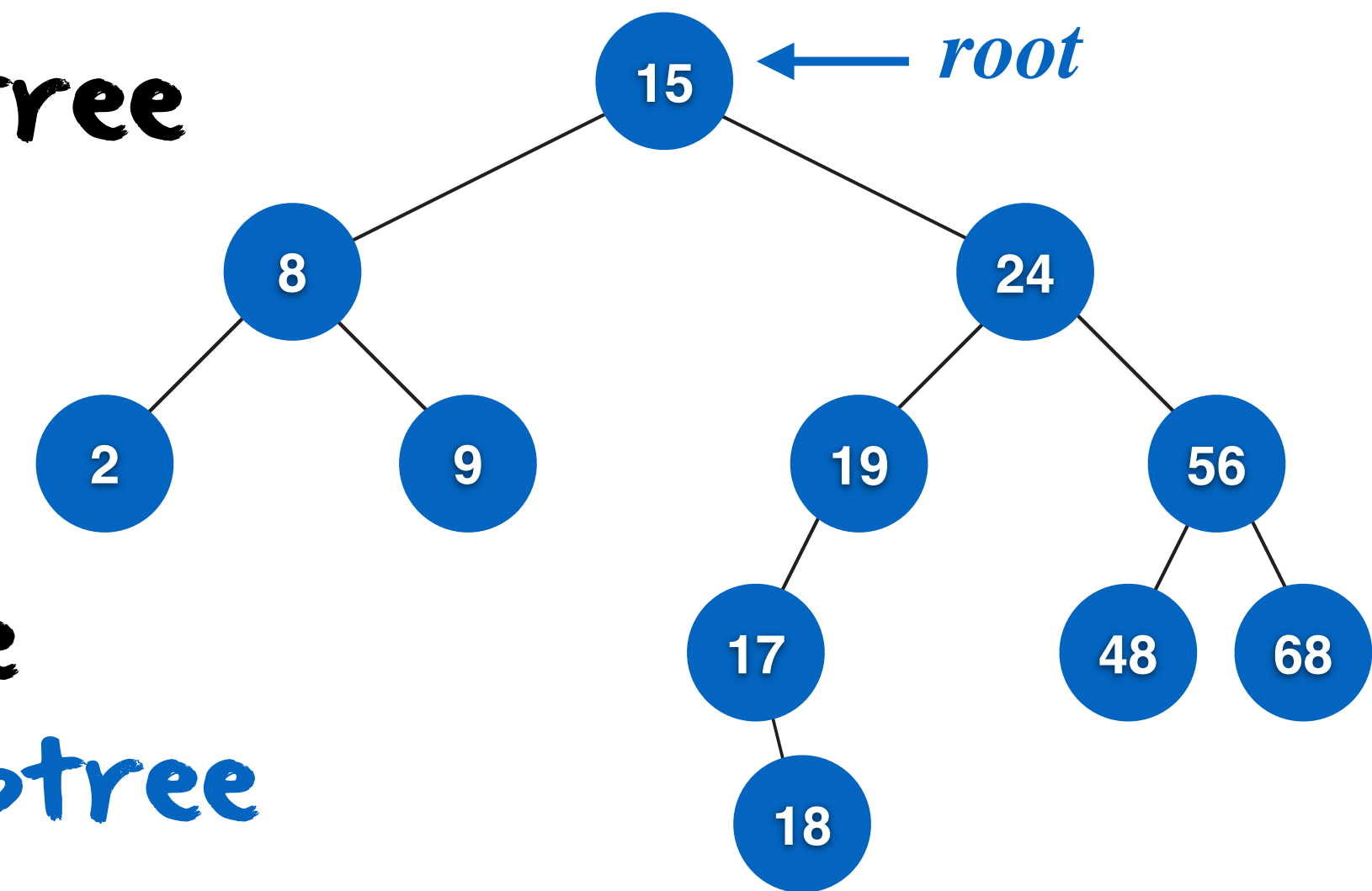
binary search trees

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

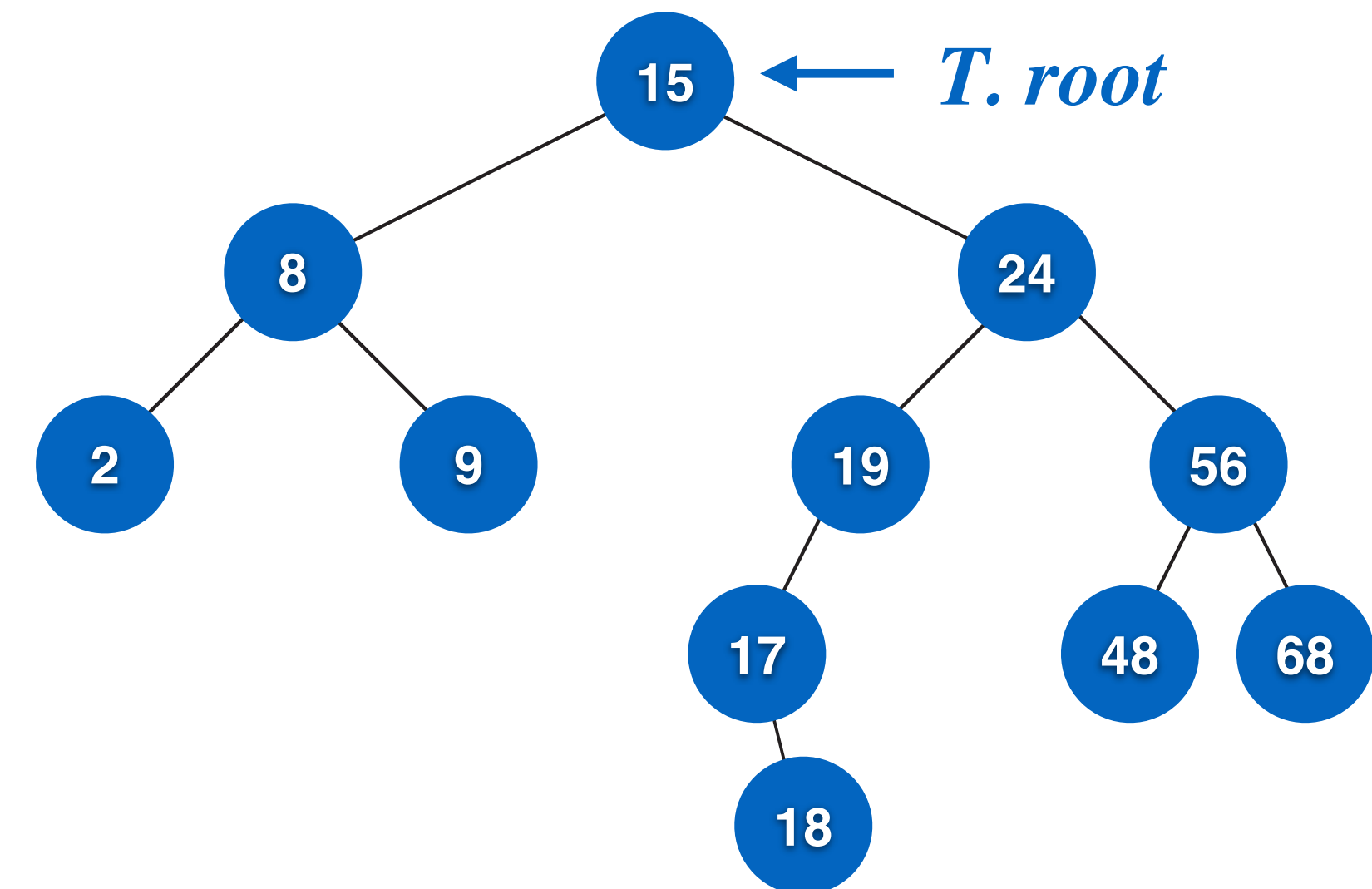
in addition, each node might also contain:

- ◆ 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 designated 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( $x.\text{right}, k$ )
```

TREE-MINIMUM(x)

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

no need to compare keys!



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-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  $y$ 
```

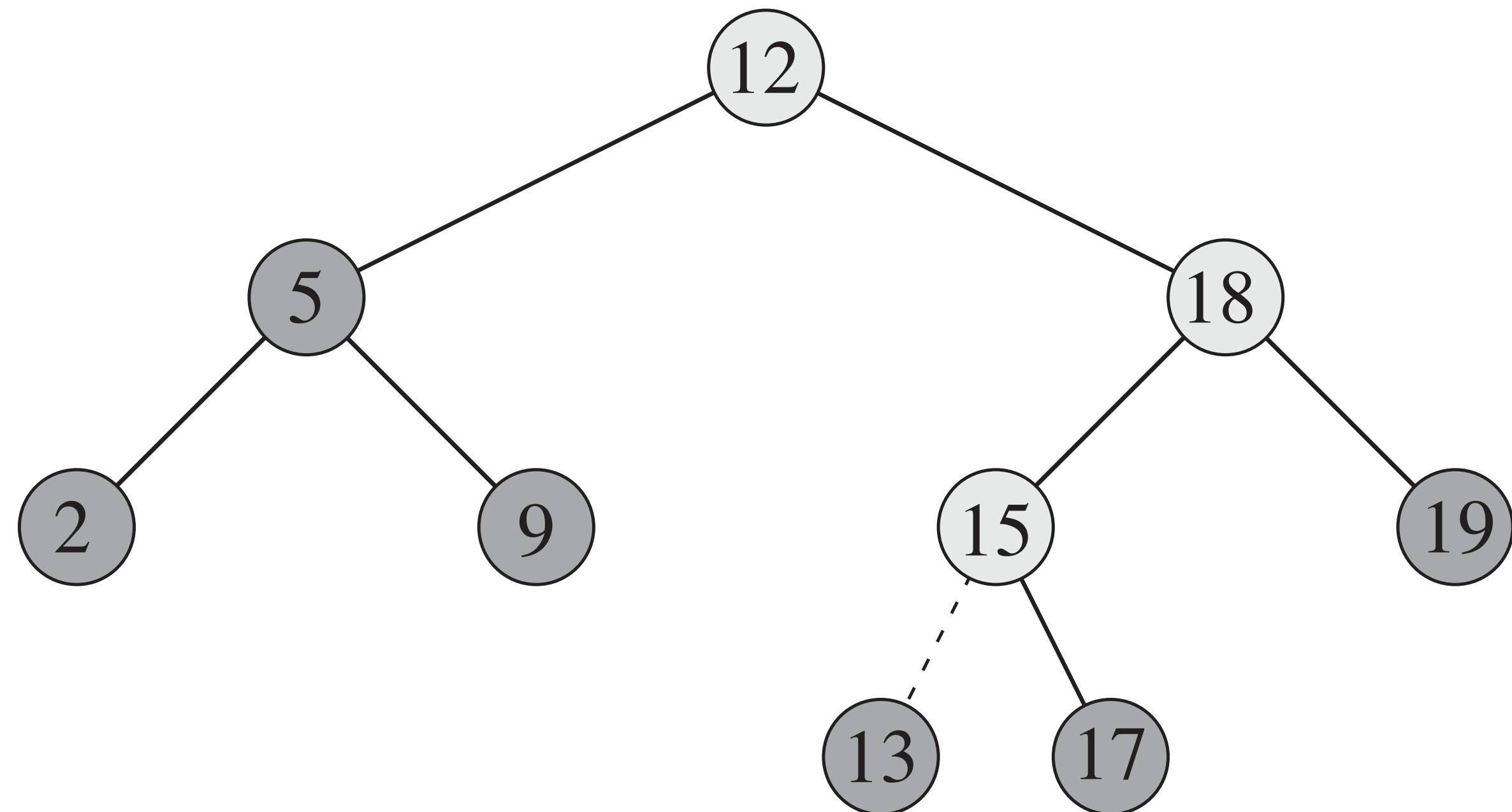
binary search trees

insertion

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



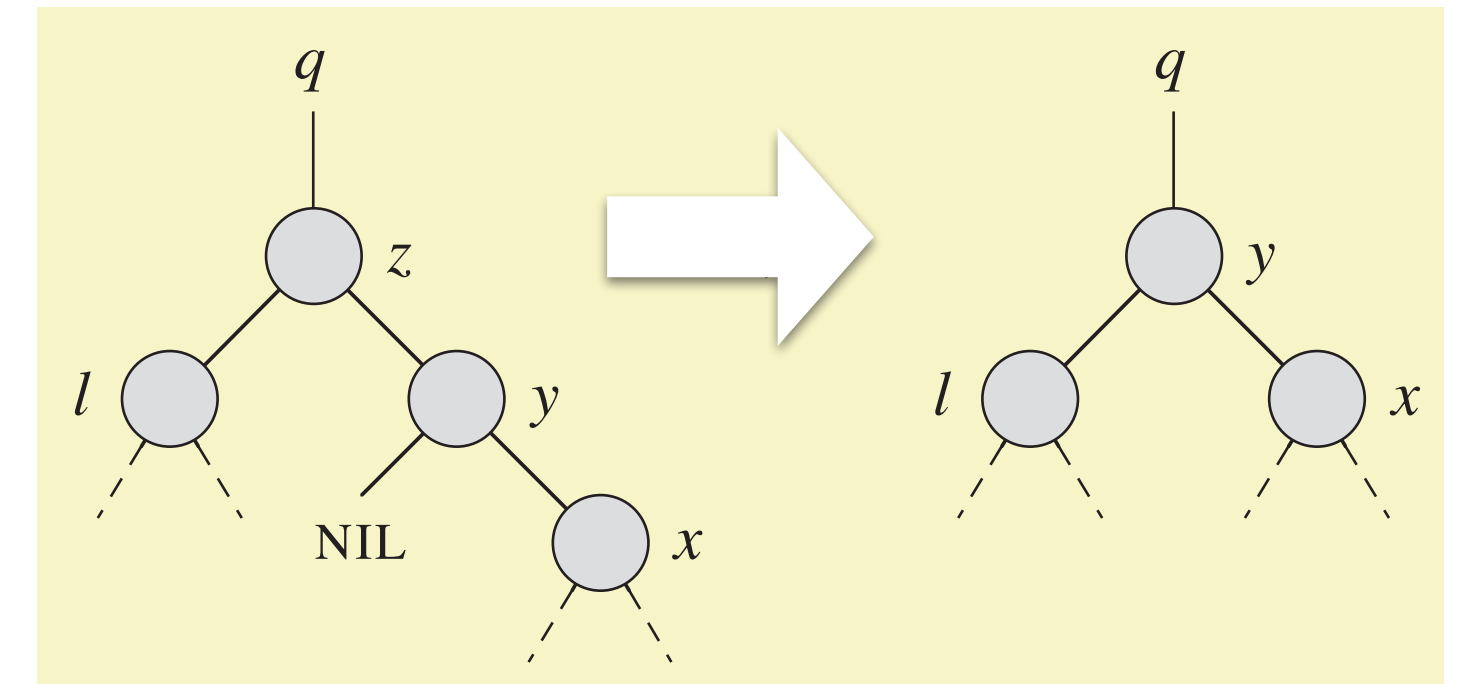
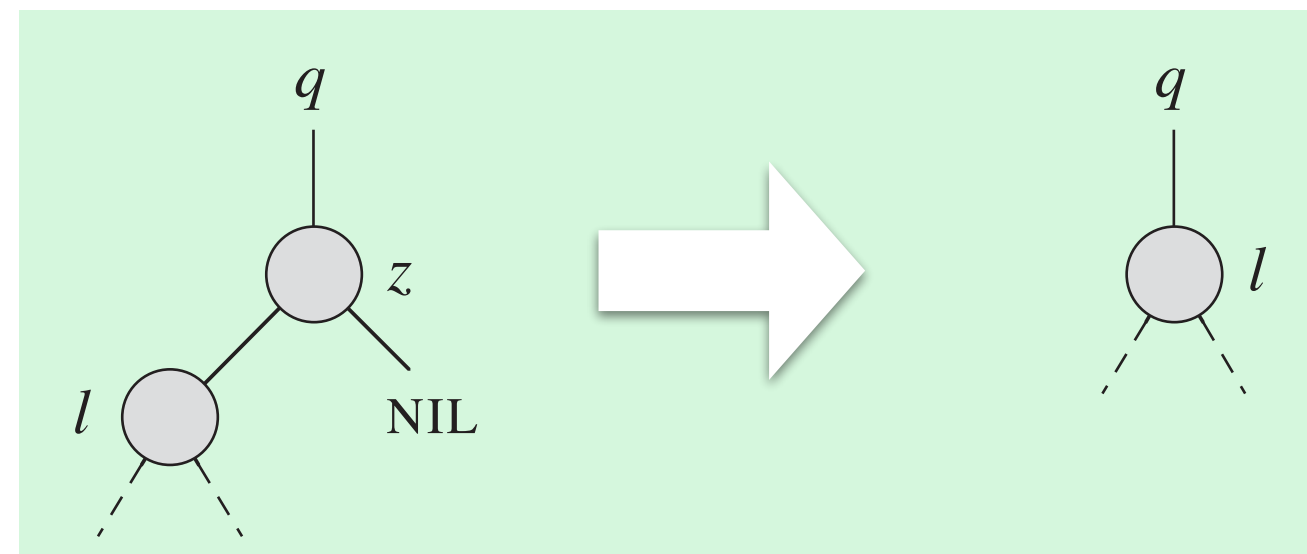
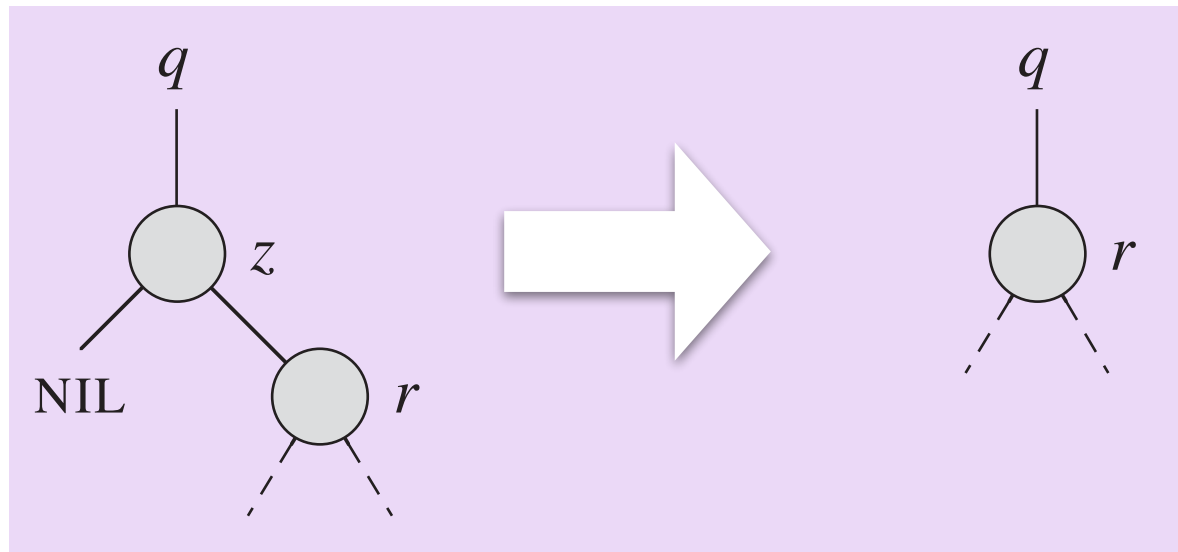
binary search trees

deletion

TREE-DELETE(T, z)

```

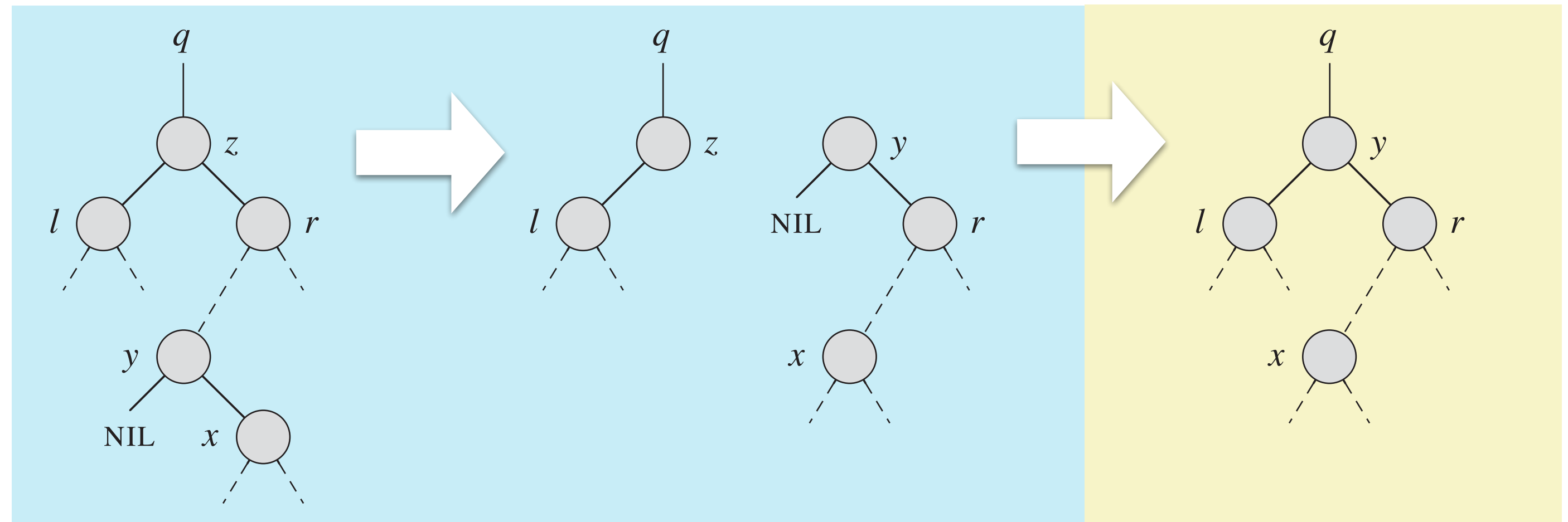
1  if  $z.left == \text{NIL}$ 
2    TRANSPLANT( $T, z, z.right$ )
3  elseif  $z.right == \text{NIL}$ 
4    TRANSPLANT( $T, z, z.left$ )
5  else  $y = \text{TREE-MINIMUM}(z.right)$ 
6    if  $y.p \neq z$ 
7      TRANSPLANT( $T, y, y.right$ )
8       $y.right = z.right$ 
9       $y.right.p = y$ 
10   TRANSPLANT( $T, z, y$ )
11    $y.left = z.left$ 
12    $y.left.p = y$ 
  
```



TRANSPLANT(T, u, v)

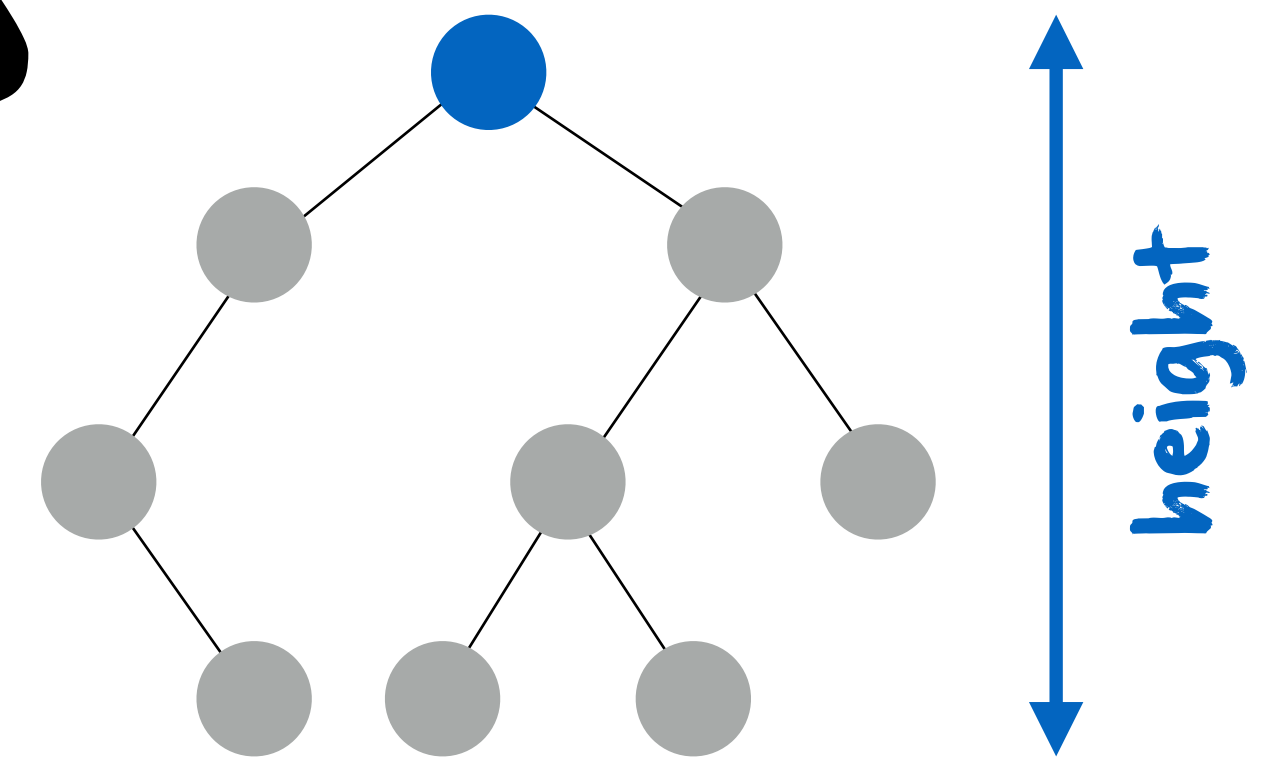
```

1  if  $u.p == \text{NIL}$ 
2     $T.root = v$ 
3  elseif  $u == u.p.left$ 
4     $u.p.left = v$ 
5  else  $u.p.right = v$ 
6  if  $v \neq \text{NIL}$ 
7     $v.p = u.p$ 
  
```



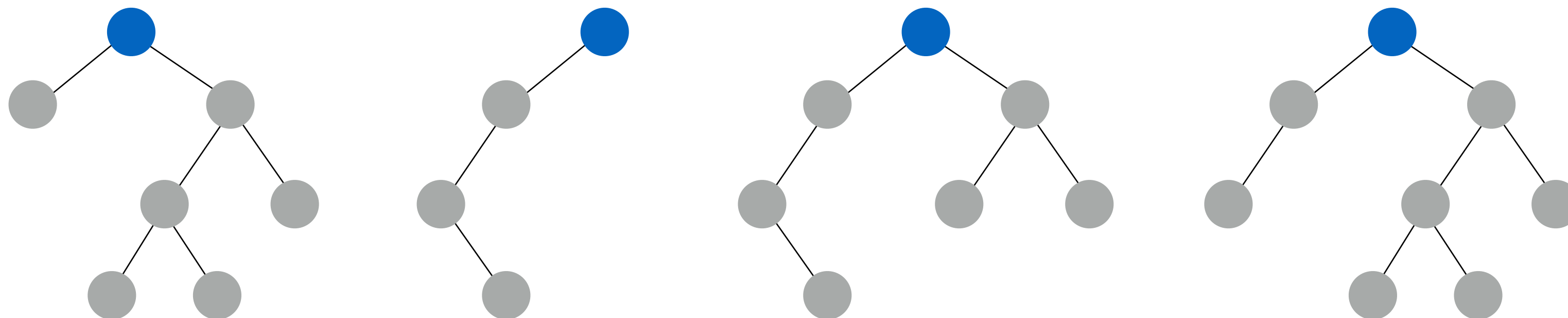
balanced trees

the **height** of a tree is the **maximum distance** of any node **from the root** in terms of **number of edges** to traverse



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?