Algorithms and Data Structures

Yahor Murashka

2017

Contents

- Complexity
- Stack, queue and linked list
- Sorting algorithms
- Hash tables
- Trees
- Selected Topics

O(n)

I. Complexity

Algorithm

Input

- sequence of computational steps

Output

Random-access machine (RAM) model

- Instructions are executed one after another.
- *Primitive* instructions takes a constant amount of time.
- There is a limit on the word size.

Running time

- It is the number of primitive operations (steps) executed:
- Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling, shift left/shift right.
- Data movement: load, store, copy.
- Control: conditional/unconditional branch, subroutine call and return.

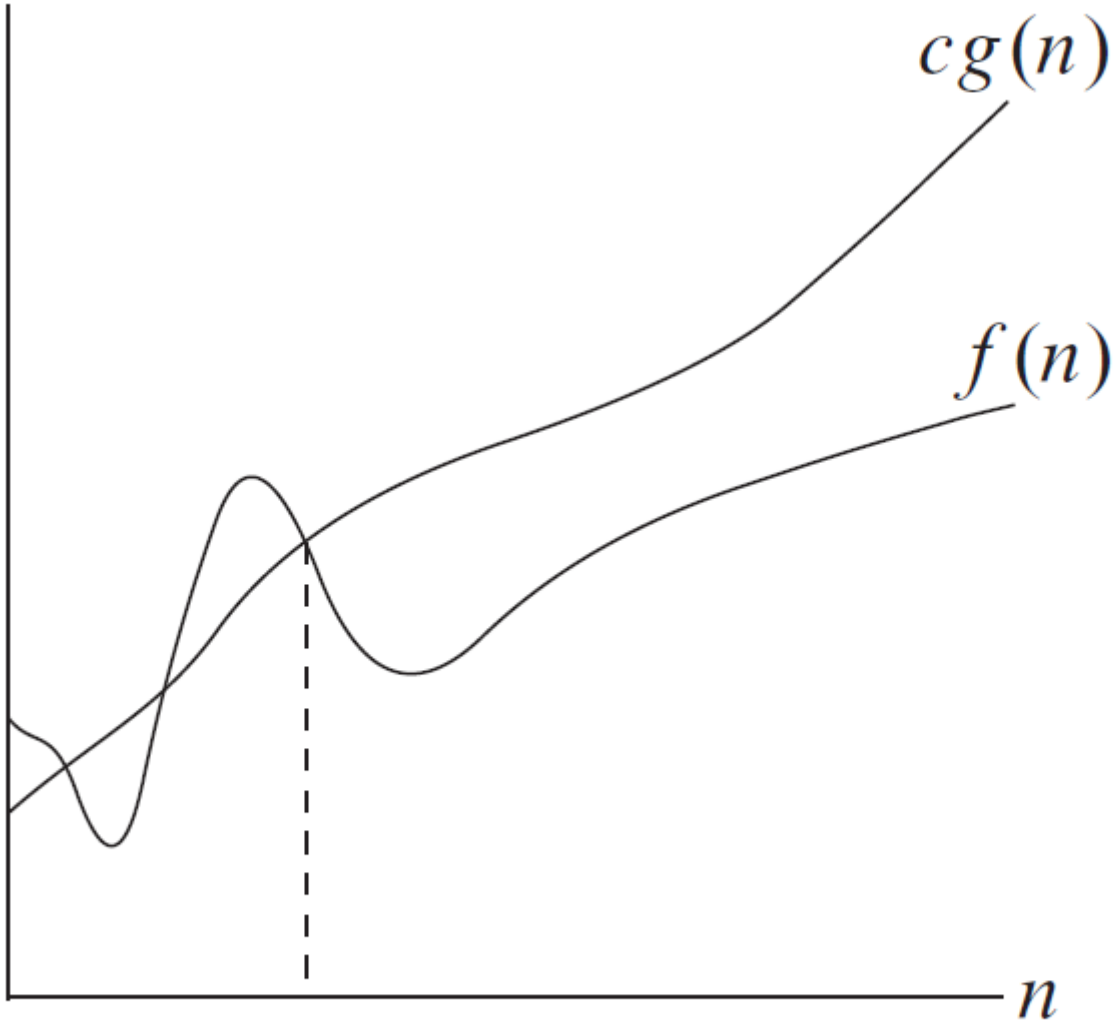
Asymptotic efficiency

That is how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases without bound.

O-notation

Asymptotic upper bound:

$O(g(n))$ = { $f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$ }



$$f(n) = O(g(n))$$

Examples:

n^2

$n^2 + n$

$n^2 + 1000n$

$1000n^2 + 1000n$

n^3

$n^{2.00001}$

$n^2 \lg\lg\lg n$

n

$n/1000$

$n^{1.99999}$

$n^2/\lg\lg\lg n$

Task

Rank the following functions by order of growth:

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

Example: *Fibonacci numbers*

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$

Code example

Fibonacci numbers

Best, worst and average case

Express what the resource usage (running time or memory) is at least, at most and on average, respectively

Example

Algorithm	Best	Worst	Average
linear search			
find minimum			
binary search			
bubble sort			
quicksort			

1. const	UI
2. $\log(\log(N))$	UI
3. $\log(N)$	UI
4. N^C, $0<C<1$	Click
5. N	Click
6. $N \log(N)$	Click
7. N^C, $C>1$	Background
8. C^N, $C>1$	x
9. $N!$	x

Questions?

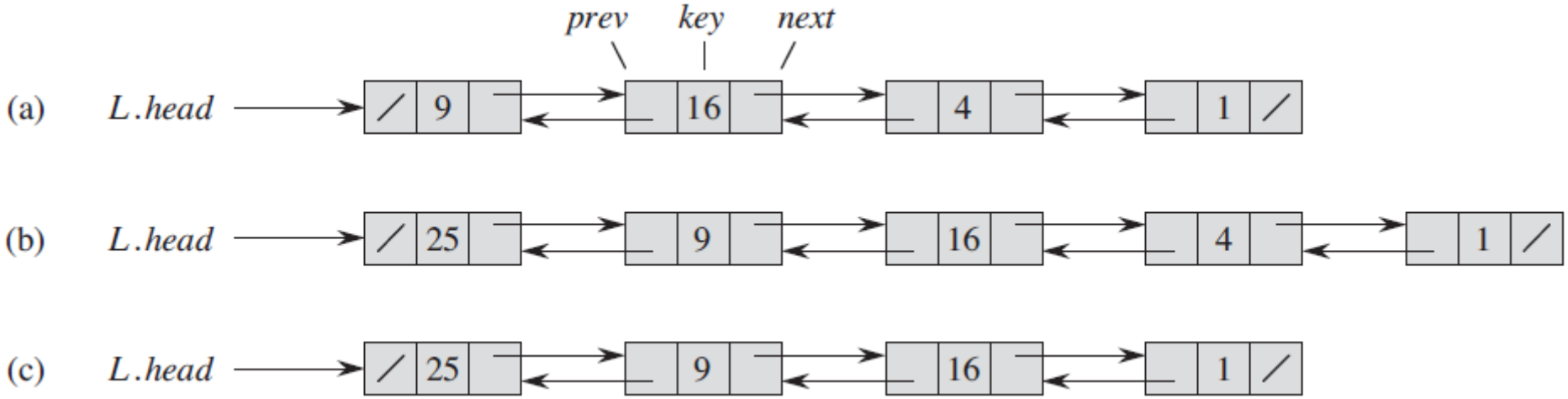
Queue<T>

II. Stack, queue and linked list

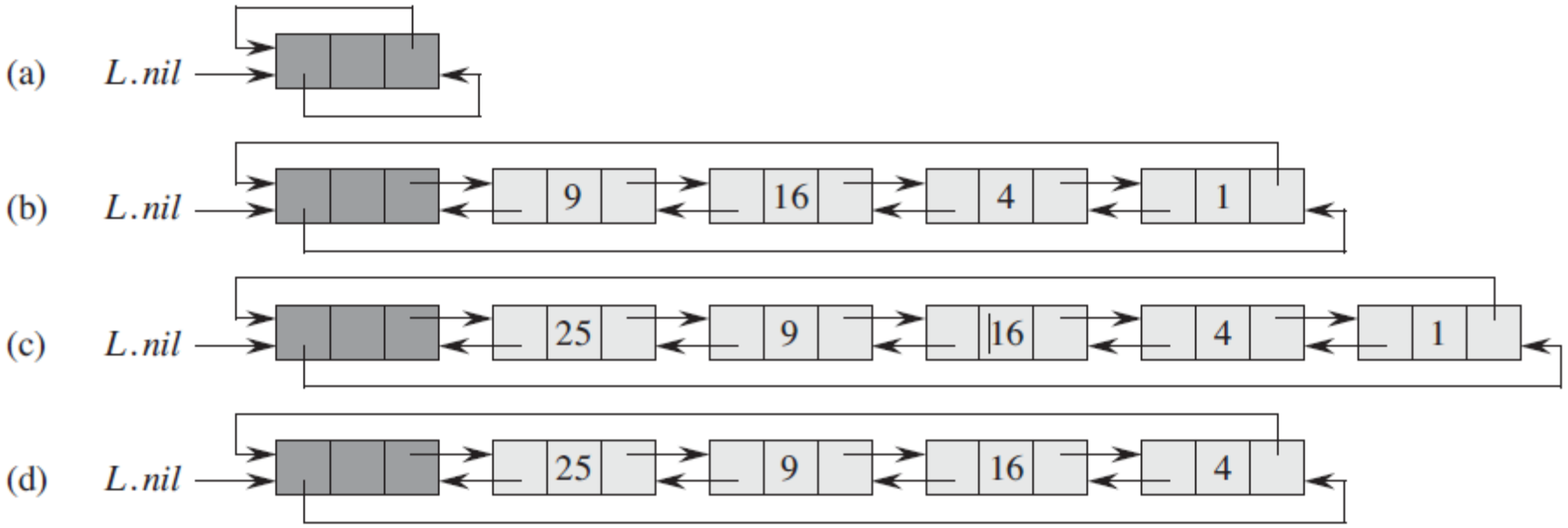
Linked lists

- Singly linked or doubly linked
- Sorted or not
- Circular or not

Doubly linked list



Circular doubly linked list with a sentinel.

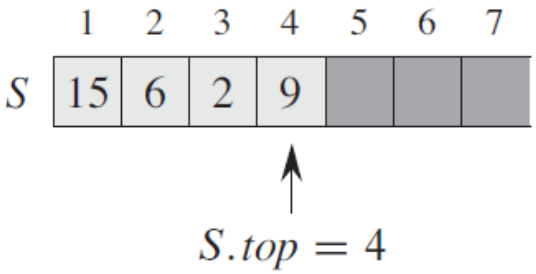


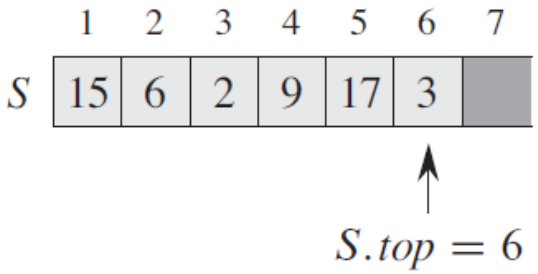
Stack

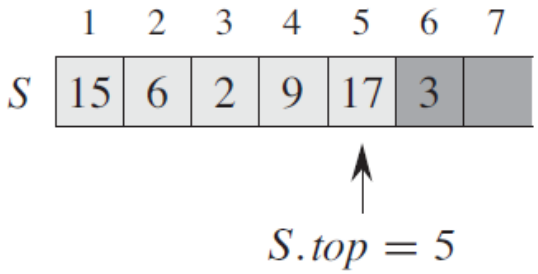
- The stack implements a ***last-in***, ***first-out***, or ***LIFO***, policy.
- PUSH
- POP
- PEEK

Stack implementation

- Array
- Linked list





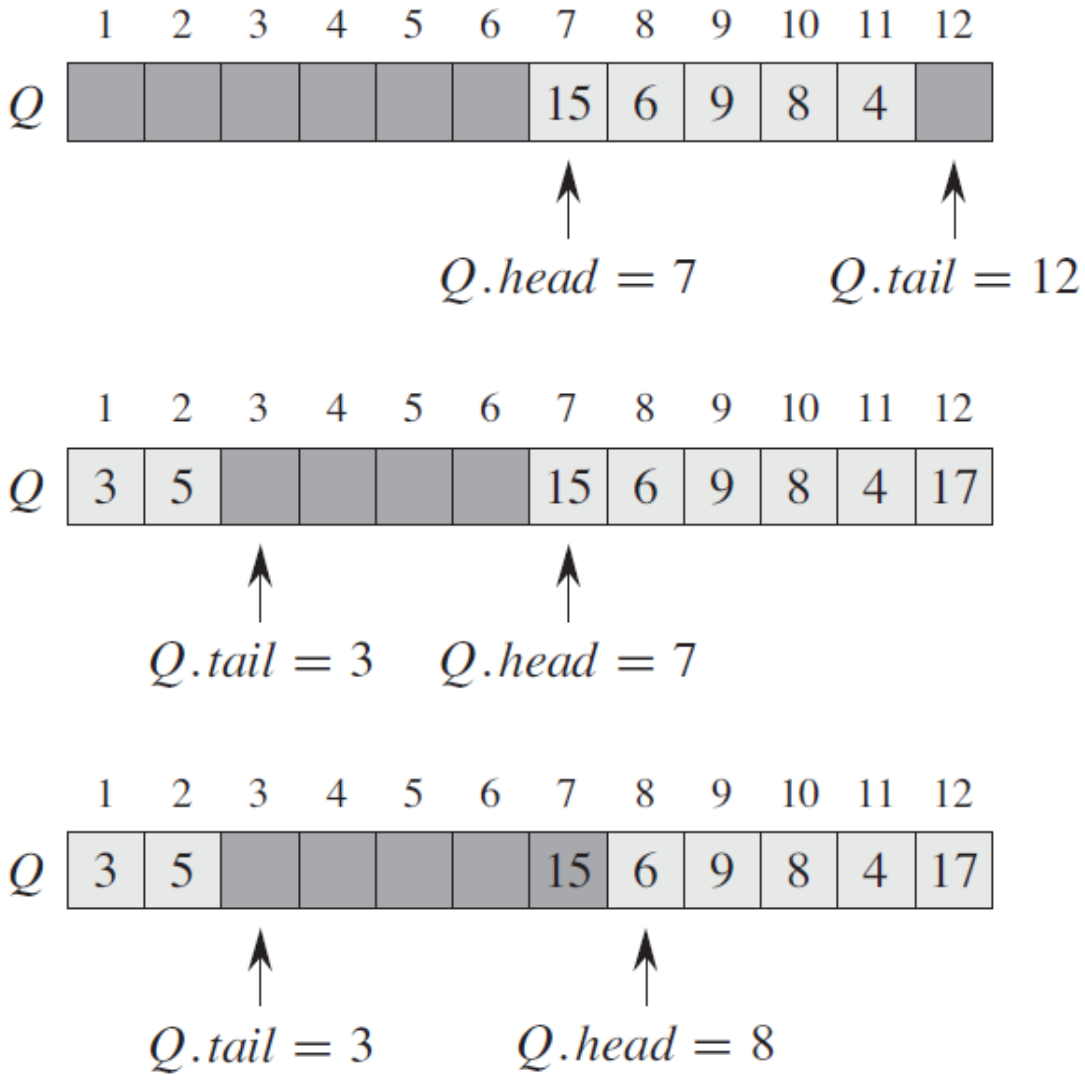


Queue

- The queue implements a ***first-in***, ***first-out***, or ***FIFO***, policy.
- Enqueue
- Dequeue
- Peek

Queue implementation

- Array
- Linked list
- Two stacks



Questions?

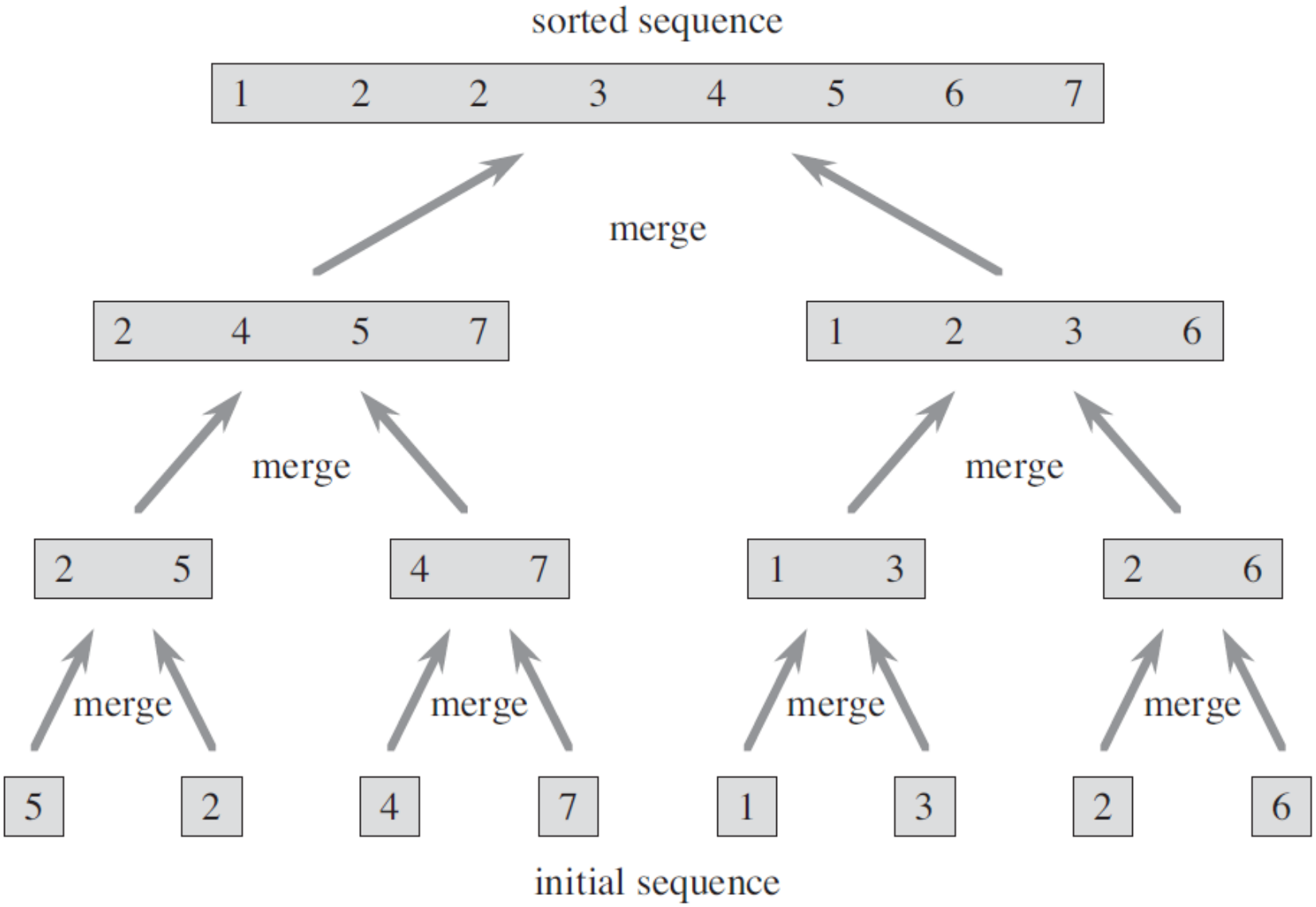
Sort<T>

III. Sorting algorithms

Sorting algorithms

- Complexity
- Memory usage
- Stability
- Recursion
- Comparison sort
- General method: insertion, exchange, selection, merging, etc

Merge sort


sorted sequence
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence

Quicksort

Applies the divide-and-conquer paradigm:

- **Divide:** Partition (rearrange) the array A into two (possibly empty) subarrays A1 and A2 such that each element of A1 is less than or equal to A[i], which is, in turn, less than or equal to each element of A2.

Quicksort

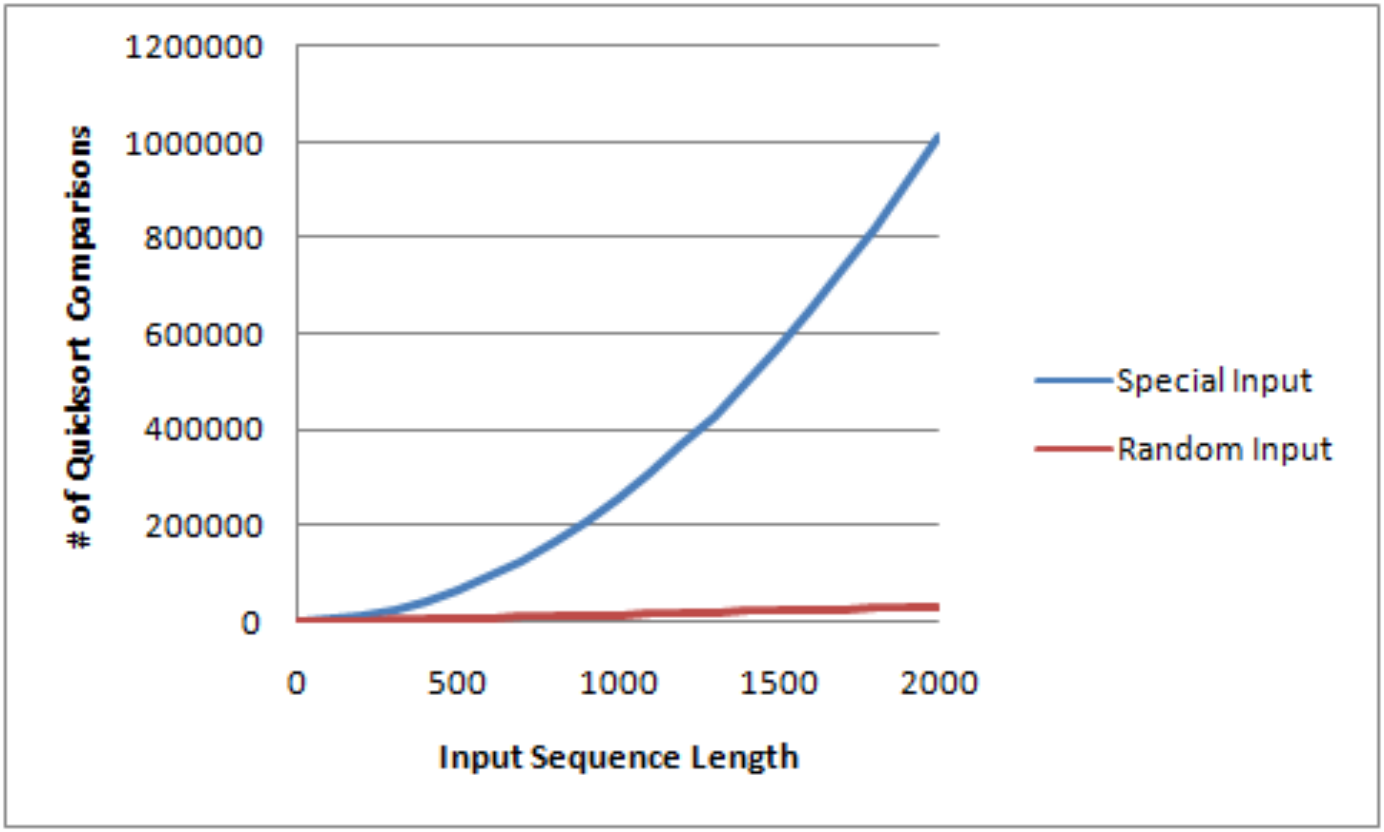
- **Conquer:** Sort the two subarrays A1 and A2 by recursive calls to quicksort.

- **Combine:** Because the subarrays are already sorted, no work is needed to combine them.

Complexity

Best	N log N
Average	N log N
Worst	N^2
Memory	log N
Stable	No

Worst case


1200000
1000000
800000
600000
400000
200000
0
# of Quicksort Comparisons
0
500
1000
1500
2000
Input Sequence Length
Special Input
Random Input

Theorem

Any **comparison sort *** algorithm requires O(n log n) comparisons in the worst case.

** the sorted order is based only on comparisons between the input elements.*

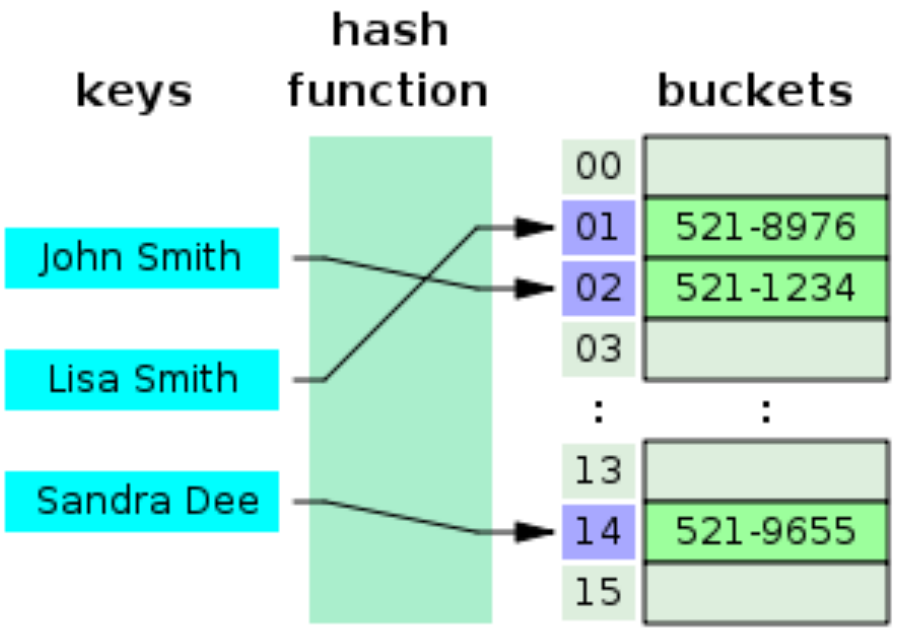
Questions?

Dictionary<Tkey, TValue>

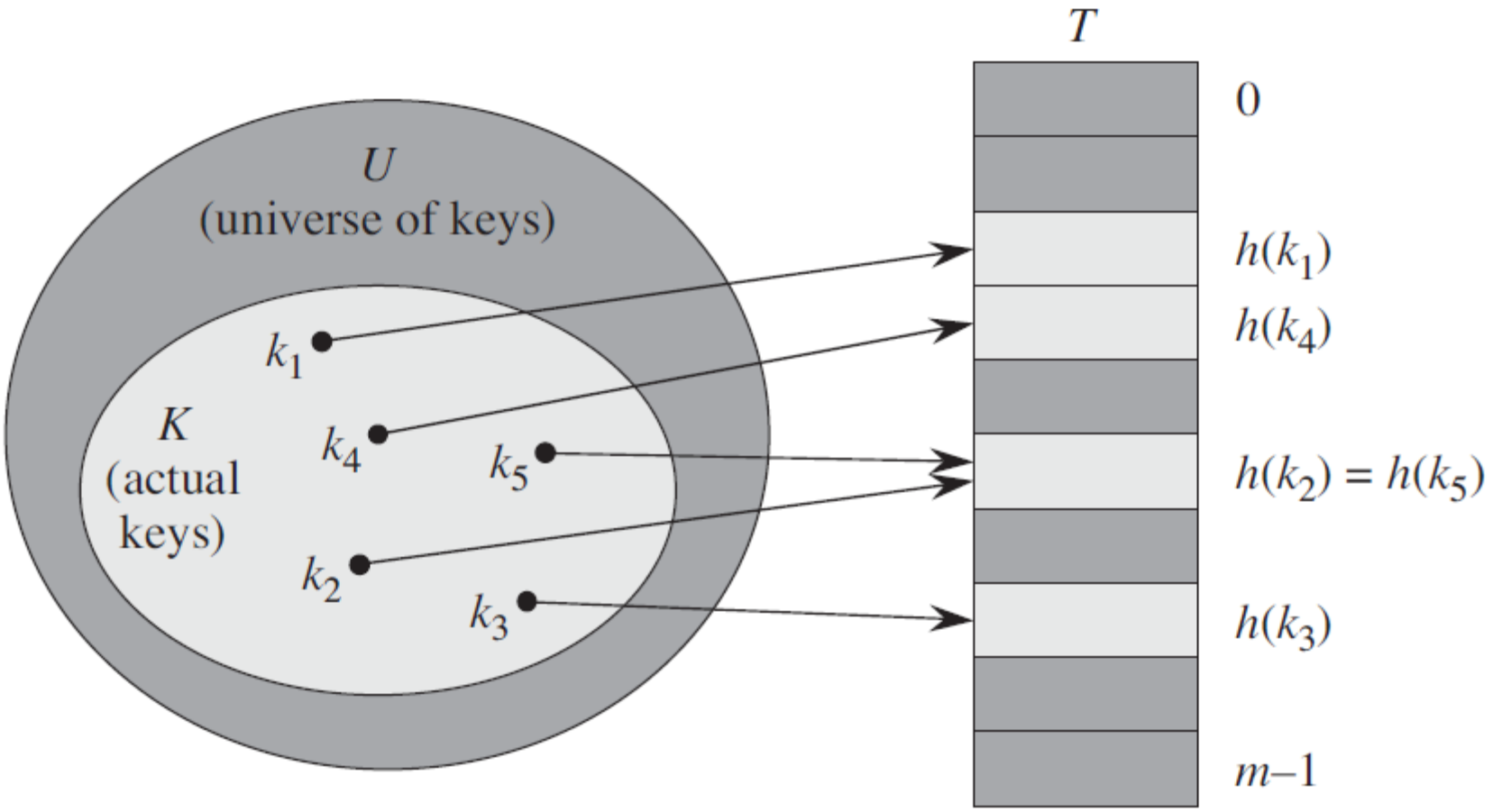
IV. Hash tables

Hash table

- Data structure that uses a hash function to map identifying values.
- The hash function is used to transform the key into the index (the hash).



Hash tables


T
U
(universe of keys)
K
(actual
keys)
k_1
k_4
k_5
k_2
k_3
0
$h(k_1)$
$h(k_4)$
$h(k_2) = h(k_5)$
$h(k_3)$
$m-1$

Collision

- There is one hitch: two keys may hash to the same slot. We call this situation a ***collision***.

Collision resolution

- Choose a suitable hash function h.
- The idea is to make h appear to be “random”.

What makes a good hash function?

A good hash function satisfies the assumption of **simple uniform hashing**:

each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to.

Interpreting keys as natural numbers

if the keys are not natural numbers,

find a way to interpret them as natural numbers

1. The division method

- $h(k) = k \bmod m$
- $m \neq 2^p$, because h(k) is just the p lowest-order bits of k
- m is prime not too close to an exact power of 2

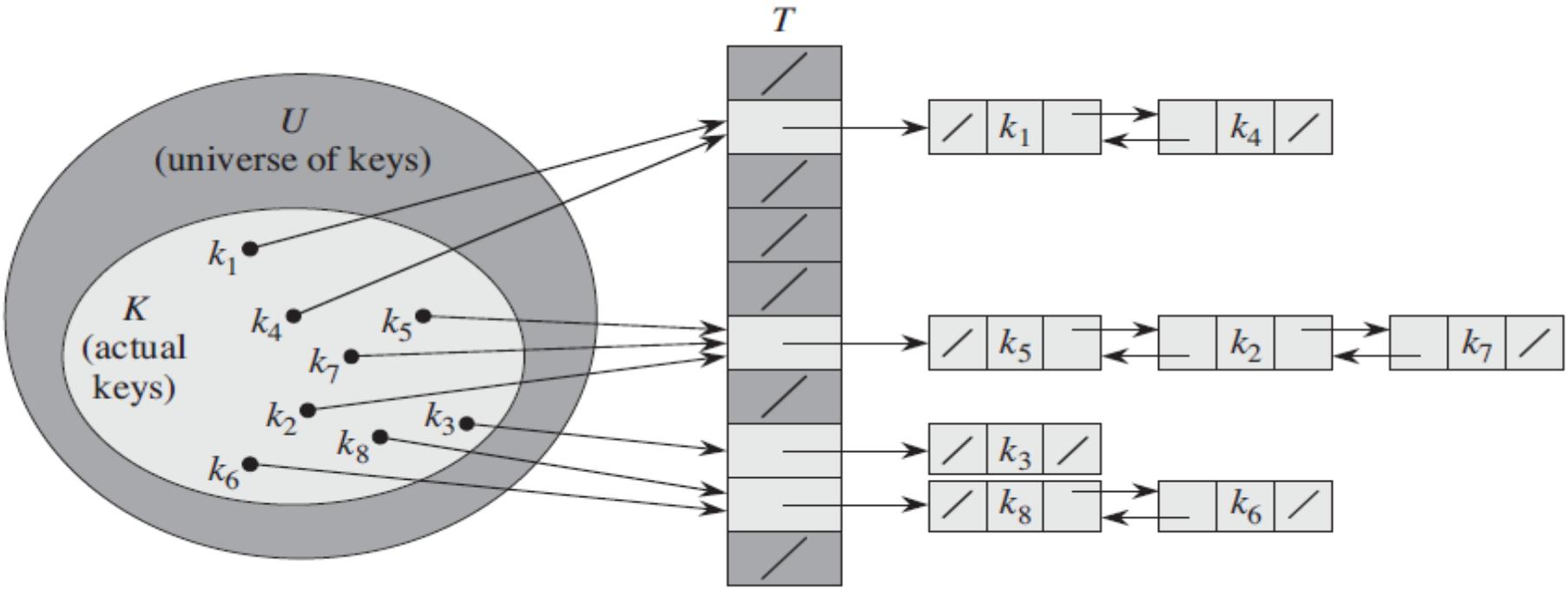
2. The multiplication method

- $h(k) = [m \{k A\}]$, where $0<A<1$
- $m = 2^p$ — can be easily implemented
- $A = (\sqrt{5}-1)/2 \approx 0.6180339887$ (*D. Knuth*)

3. Universal hashing

- Let H be a finite collection of hash functions h: keys -> {0, 1, … , m-1}
- H is **universal** if for each pair of distinct keys k and l, the number of hash functions h for which h(k)=h(l) is at most |H|/m.
- $h_{ab}(k)=((a\ k + b) \bmod p) \bmod m$

I. Collision resolution by chaining



Running time

Insertion — O(1), if we assume that the element is not already present in the table (**!**); else the same as searching.

Deletion — O(1), if the lists are doubly linked; else the same as searching.

Running time

Hash table T with m slots that stores n elements.

Define the ***load factor a*** for T as n/m, that is, the average number of elements stored in a chain.

Theorem

In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $O(1+\alpha)$, under the assumption of simple uniform hashing.

II. Open addressing

- All elements occupy the hash table itself
- Each table entry contains an element or *null*
- ***Compute*** the sequence of slots to be examined

Hash function

- The hash function is extended to include the probe number (starting from 0) as a second input
- h: U x {0, 1, …, m-1} -> {0, 1, …, m-1}
- The ***probe sequence*** must be a permutation of <0, 1, …, m-1>

Probing methods

- Linear probing

  $h(k,i) = (h_1(k) + i) \bmod m$

- Quadratic probing

  $h(k,i) = (h_1(k) + c_1 i + c_2 i^2) \bmod m$

- Double hashing

  $h(k,i) = (h_1(k) + h_2(k)\, i) \bmod m$

Associative array

- Associative array or map or dictionary

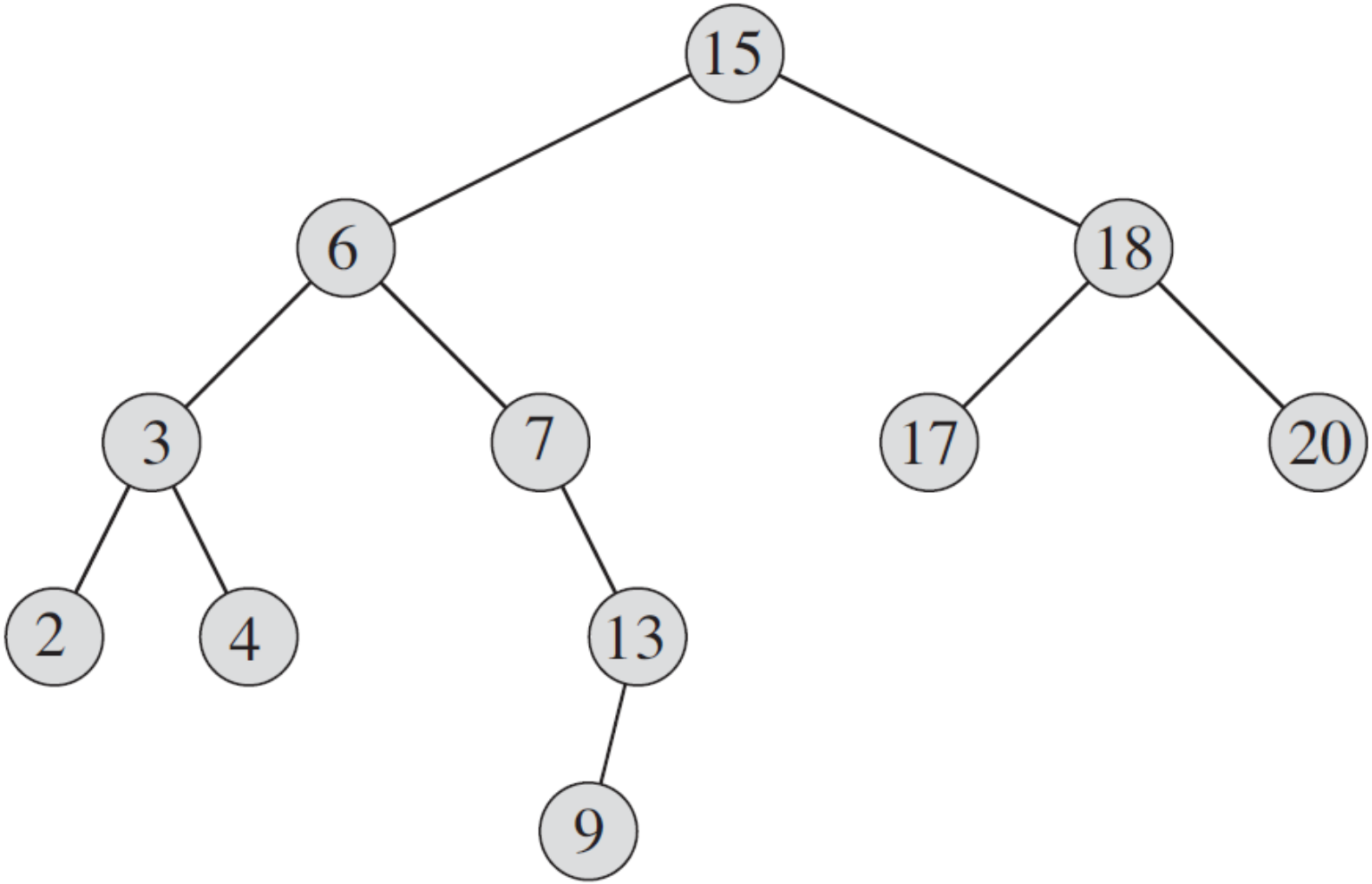
- INSERT(key, value)
- FIND(key)
- REMOVE(key)

Questions?

std::map

V. Trees

Binary search tree



Binary-search-tree property

Let x be a node in a binary search tree.

If y is a node in the left subtree of x, then y.*key* ≤ x.*key*.

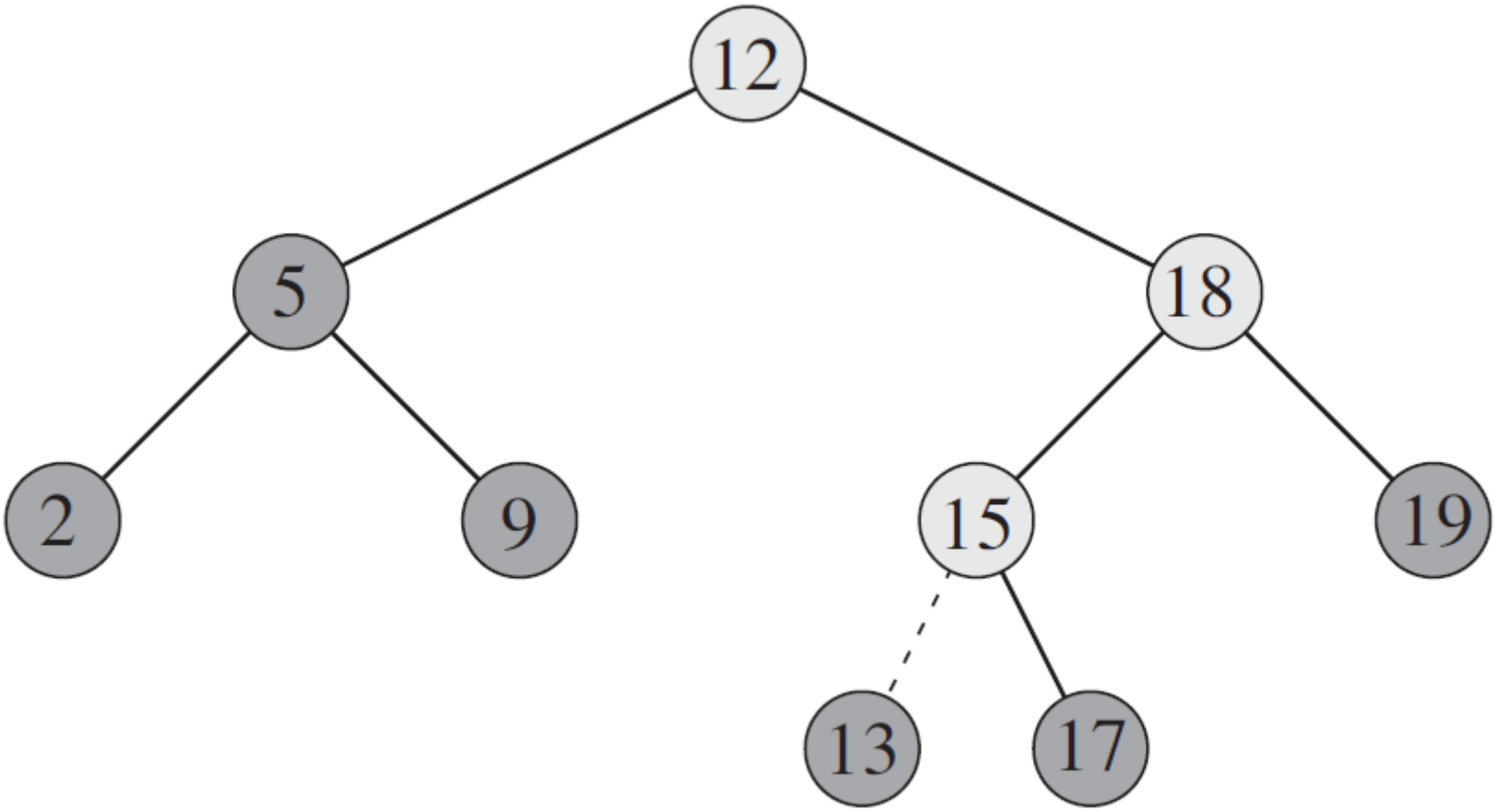
If y is a node in the right subtree of x, then y.*key* ≥ x.*key*.

Querying a binary search tree

- Searching
- Minimum and maximum
- Insertion
- Deletion

Complexity — O(h)

Insertion

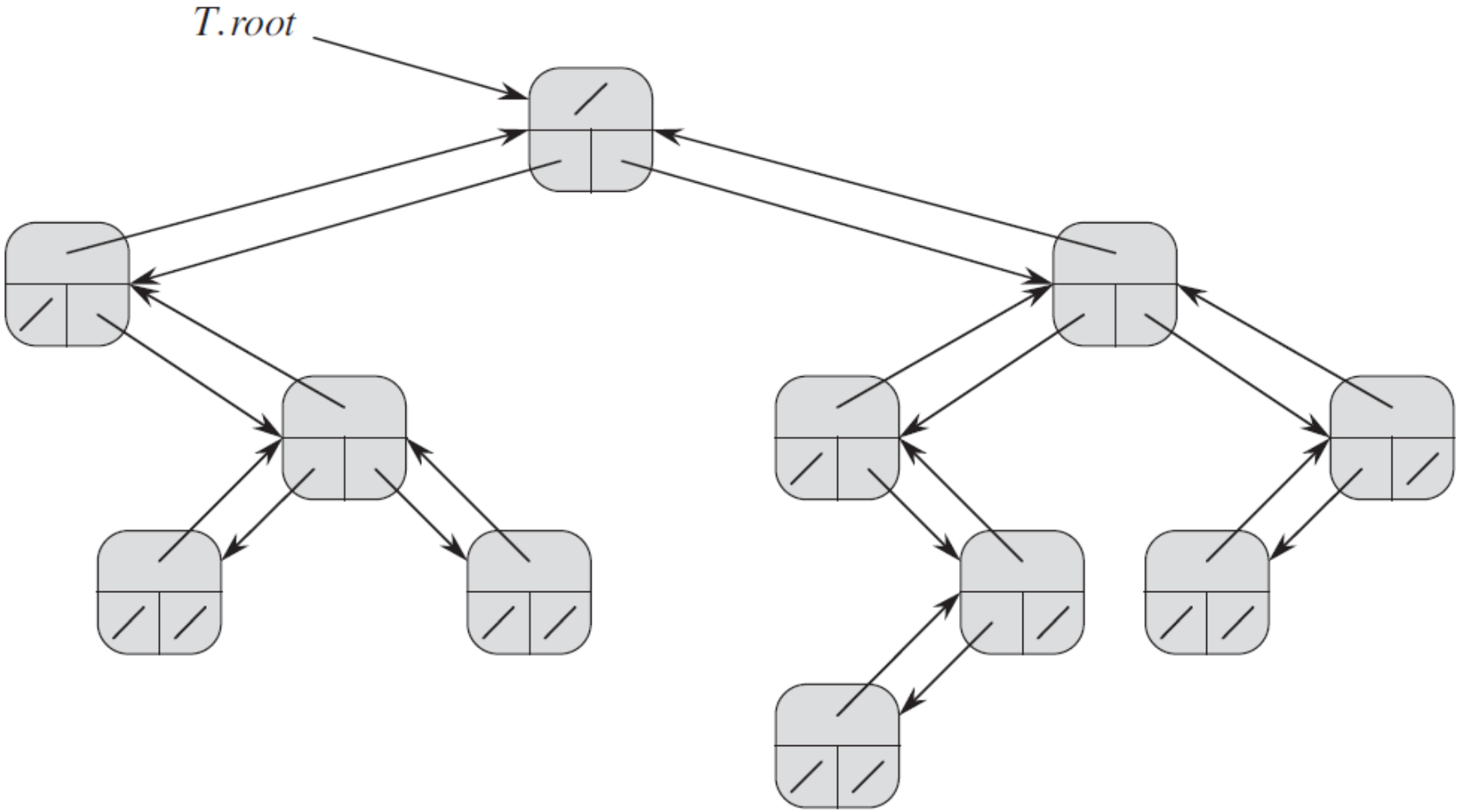


Running time

	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Representing rooted trees

Each node x has the attributes x.*p*, x.*left*, and x.*right*



Tree traversal

DFS

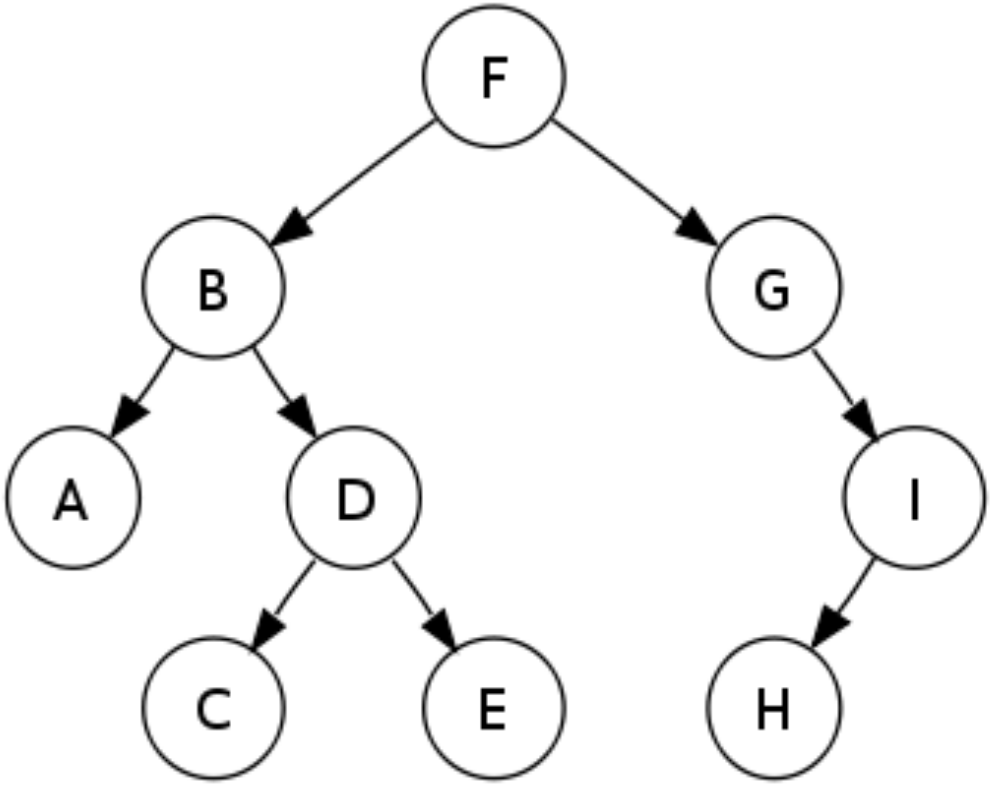
- Preorder traversal
- Inorder (symmetric) traversal
- Postorder traversal

BFS

- Level-order traversal

Preorder traversal sequence

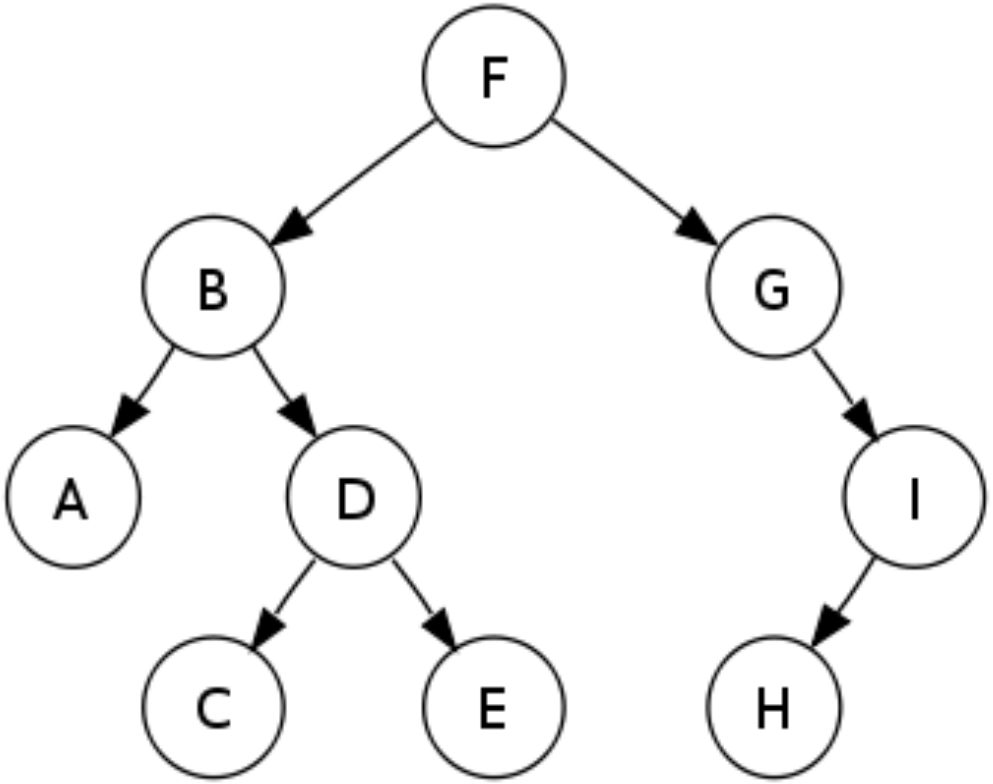
F, B, A, D, C, E, G, I, H (root, left, right)



Inorder traversal sequence

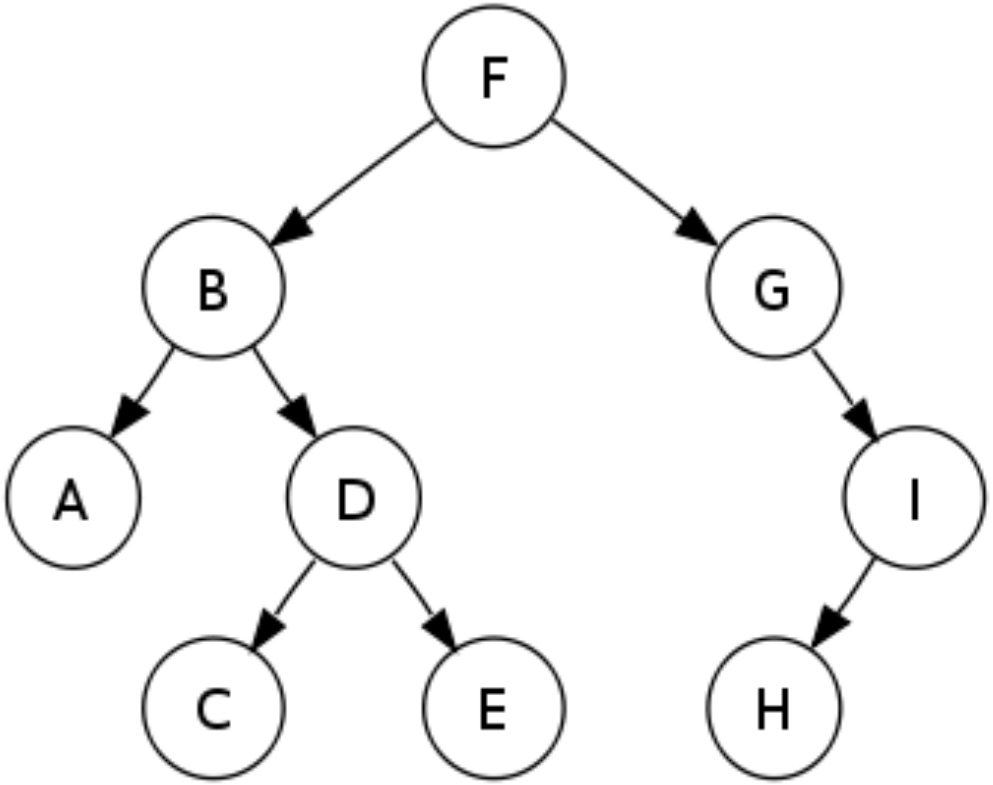
A, B, C, D, E, F, G, H, I (left, root, right)

Note: produce a sorted sequence



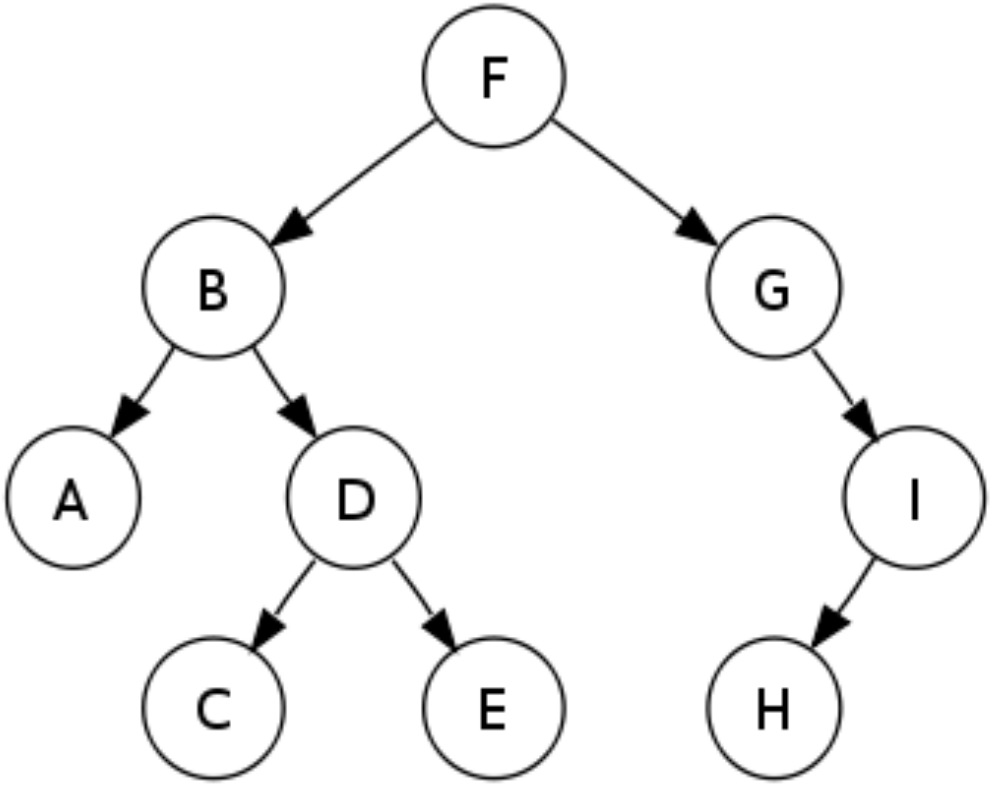
Postorder traversal sequence

A, C, E, D, B, H, I, G, F (left, right, root)



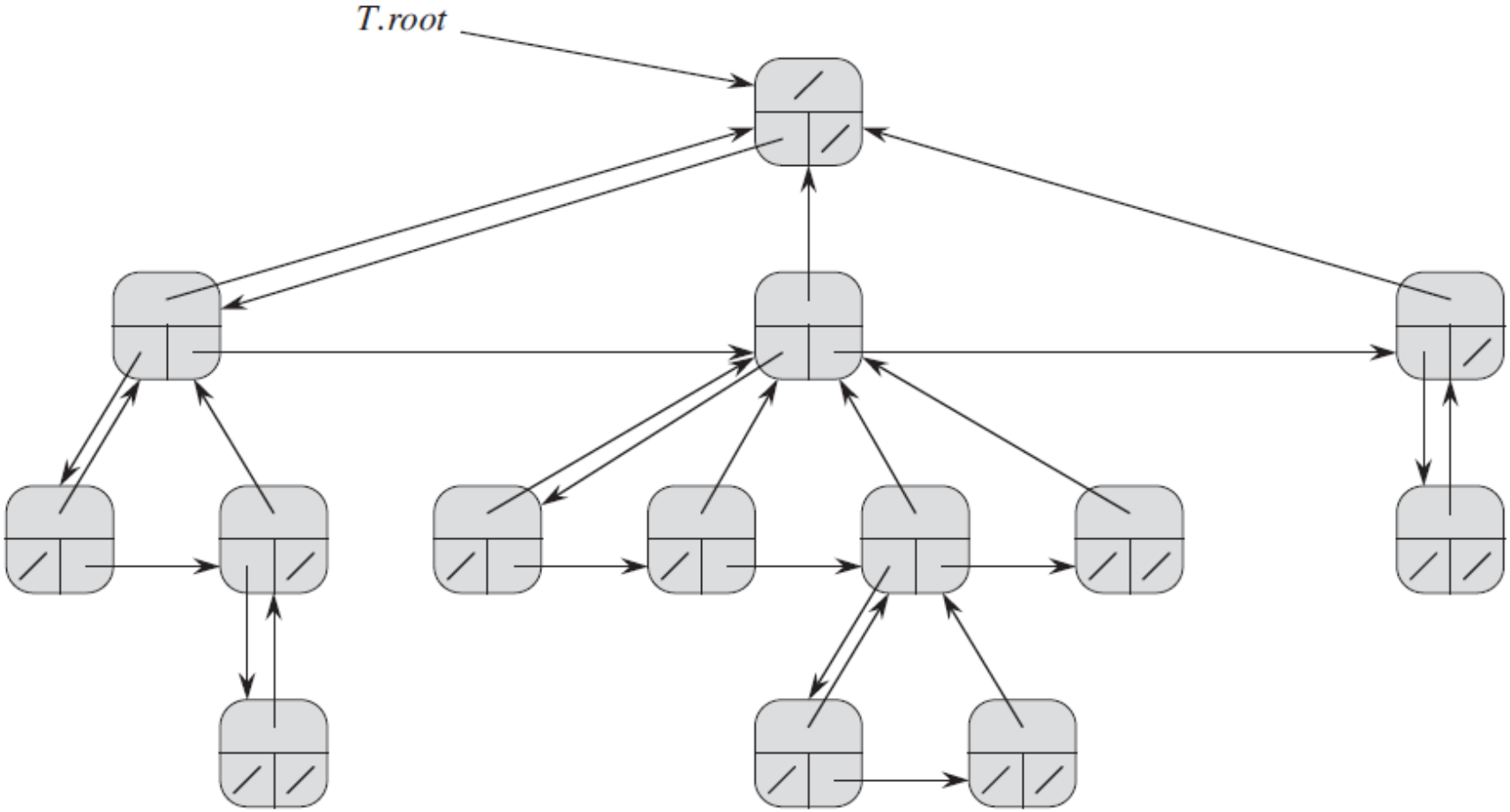
Level-order traversal sequence

F, B, G, A, D, I, C, E, H



Representing rooted trees

Each node x has attributes x.*p*, x.*left-child*, and x:*right-sibling*



Task

Implement preorder, postorder and level-order traversal which print node key. Each node is defined as:

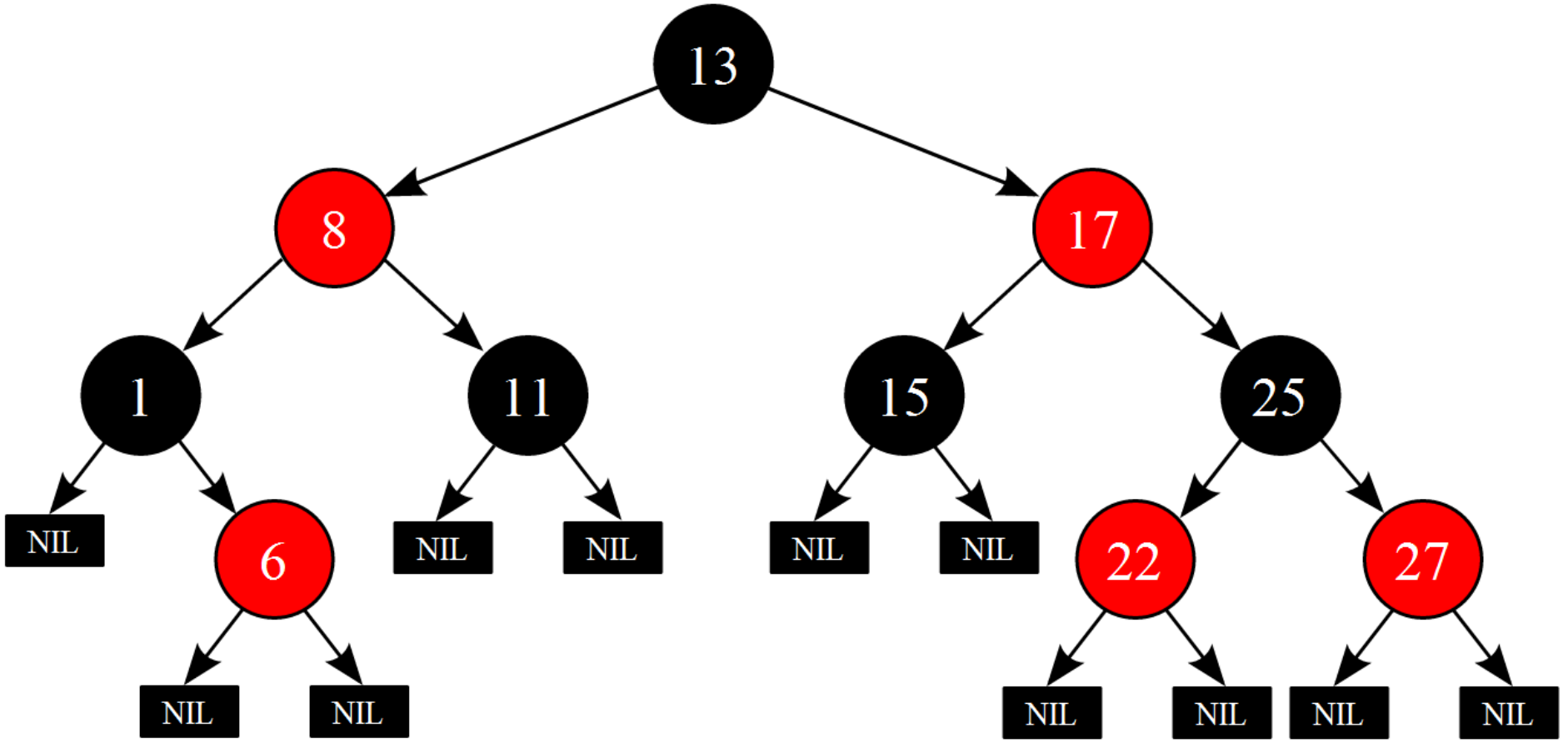
```
class Node
{
        T Key;
        Node Parent;
        Node FirstChild;
        Node NextSibling;
}
```

Red–black tree

Rules:

- A node is either red or black.
- The root is black.
- All leaves (NIL) are black.
- Both children of every red node are black.
- Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

Red-black tree



Running time

	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Hash table vs. Tree

Деревья	Хеш-таблицы
Если: важна упорядоченность элементов; важно стабильное время отклика на ЛЮБЫХ входных данных; данные хранятся на дисках (B-деревья); требуется неограниченная масштабируемость; о количестве данных ничего не известно наперед	Если: набор используемых ключей известен наперед (идеальное хеширование); ключи очень длинные и их быстрое сравнение невозможно; объем данных достаточно маленький (поместится в кэш); известно наперед примерное количество ключей, и требуется хорошее время отклика в среднем

Questions?

VI. Selected Topics

Задачи

- Обратить порядок расположения элементов в односвязном списке за O(n) времени и O(1) памяти.
- Определить за O(1) является ли целое число степенью 2.
- Не используя дополнительных переменных обменять два целых числа за O(1).
- Удалить узел из односвязного списка по указателю на удаляемый элемент за O(1).

Задачи

- Проверить, является ли односвязный список кольцевым за O(n) времени и O(1) памяти.
- Дано целое число. Вставить в него другое данное целое число с позиции i до позиции j в двоичном представлении за O(1).

References

- Thomas H. Cormen [et al.] (2009). Introduction to Algorithms (3rd ed.)

Questions?

skype: egormurashko
e_murashko@wargaming.net