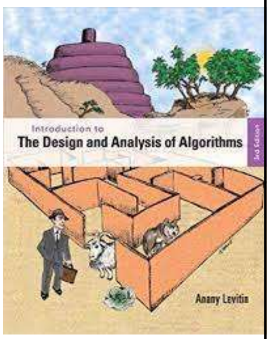
4-Decrease and Conquer



1

Decrease and Conquer

1. Reduce problem instance to smaller instance of the same problem
2. Solve smaller instance
3. Extend solution of smaller instance to obtain solution to original instance

- Can be implemented either top-down or bottom-up
- Also referred to as *inductive* or *incremental* approach

2.11.2021 BLM19307E Algorithm Analysis and Design 2

2

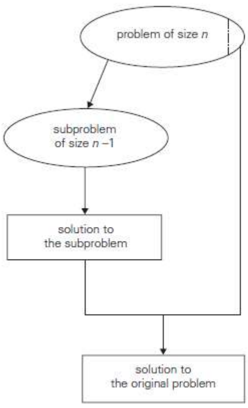
Decrease and Conquer

Three Types

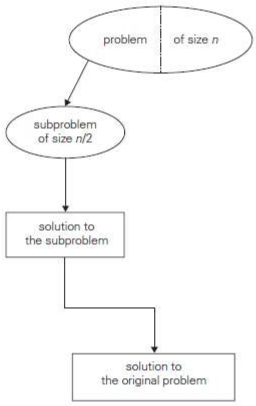
- *Decrease by a constant* (usually by 1): $n \rightarrow (n-1)$
  1. insertion sort
  2. topological sorting
  3. algorithms for generating permutations, subsets

$n \rightarrow n/2$

- *Decrease by a constant factor* (usually by half)
  - binary search and bisection method
  - exponentiation by squaring
  - multiplication à la russe
- *Variable-size decrease*
  - Euclid's algorithm
  - selection by partition
  - Nim-like games



Decrease-(by one)-and-conquer technique.



Decrease-(by half)-and-conquer technique.

What's the difference?

Consider the problem of exponentiation: Compute a^n

- Brute Force: $\underbrace{a * a * \ldots * a}_{n \text{ times}}$
- Divide and conquer: $a^{\lfloor n/2 \rfloor} * a^{\lceil n/2 \rceil}$
- Decrease by one: $a^{n-1} * a$
- Decrease by constant factor: similar to divide-and-conquer,
  $\begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even} \\ (a^{\lfloor n/2 \rfloor})^2 * a & n \text{ is odd} \\ 1 & n = 0 \end{cases}$

5

Insertion Sort

To sort array A[0..n-1], sort A[0..n-2] recursively and then insert A[n-1] in its proper place among the sorted A[0..n-2]

Usually implemented bottom up (nonrecursively)

6

Insertion Sort

(keep the left portion of the array sorted)

Example: Sort 6, 4, 1, 8, 5

sorted

6 | 4 1 8 5
4 6 | 1 8 5
1 4 6 | 8 5
1 4 6 8 | 5
$\boxed{1\ 4\ 5\ 6\ 8}$

Pseudocode of Insertion Sort

ALGORITHM *InsertionSort*($A[0..n-1]$)

//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order

for $i \leftarrow 1$ **to** $n-1$ **do**
  $v \leftarrow A[i]$
  $j \leftarrow i-1$
  while $j \geq 0$ **and** $A[j] > v$ **do**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j-1$
  $A[j+1] \leftarrow v$

basic operation

Best case: (already sorted)

1 2 5 9 10

$C_b(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$

at each iteration do 1 comparison

Worst order (reserved order)

10 9 5 2 1

sorted

Analysis of Insertion Sort

$C_w(n) = \sum_{i=1}^{n-1}\sum_{j=0}^{i-1} 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$

- Time efficiency

$$C_{worst}(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$
$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$
$$C_{best}(n) = n - 1 \in \Theta(n)$$ (also fast on almost sorted arrays)

- Space efficiency: in-place
- Stability: yes

1 3* 5 3**
1 3* 3** 5

- Best elementary sorting algorithm overall

10 8 5 2 1

like binary search

- Binary insertion sort

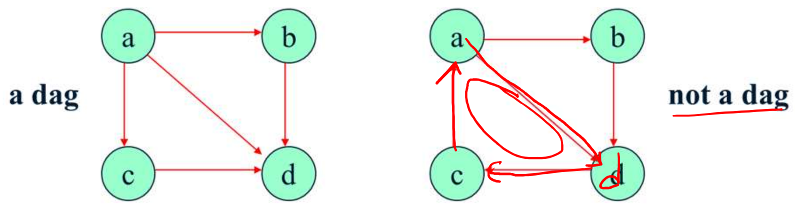
20 | 18 15
18 | 20 15 new element

2.11.2021 BLM19307E Algorithm Analysis and Design 9

9

Dags and Topological Sorting

A *dag*: a directed acyclic graph, i.e. a directed graph with no (directed) cycles



Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control)

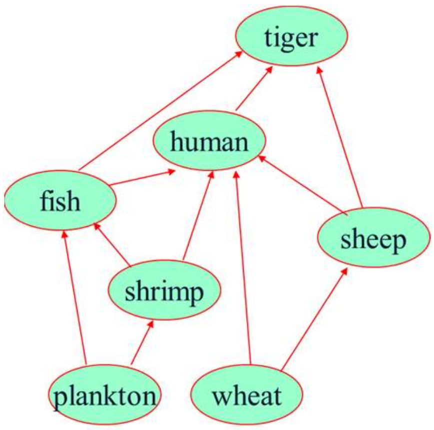
Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex (*topological sorting*). Being a dag is also a necessary condition for topological sorting be possible.

2.11.2021 BLM19307E Algorithm Analysis and Design 10

10

Topological Sorting Example

Order the following items in a food chain



plankton
shrimp
fish
wheat
sheep
human
tiger

wheat
sheep
plankton
shrimp
fish
human
tiger

11

DFS-based Algorithm

DFS-based algorithm for topological sorting

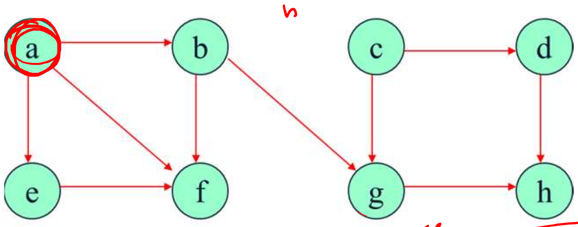
- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Back edges encountered?→ NOT a dag!

12

DFS-based Algorithm

Example

$f_{3,1}$ $h_{5,2}$ $g_{4,3}$ $b_{2,4}$ $e_{6,5}$ $d_{8,7}$ $a_{1,6}$ $c_{7,8}$



pop order f h g b e a d c $\xRightarrow{\text{reverse}}$ $\boxed{\text{c d a e b g h f}}$

topological sort.

Efficiency: same as the DFS

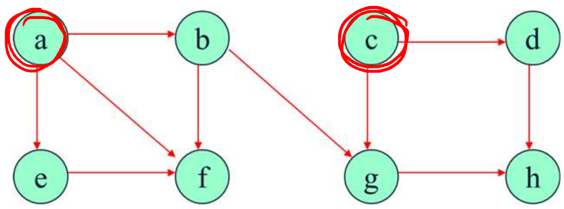
13

Source Removal Algorithm

<u>Source removal algorithm</u>

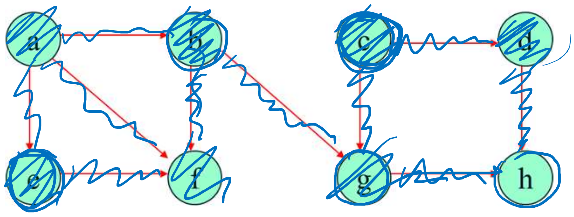
Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left (problem is solved) or there is no source among remaining vertices (not a dag)

Example:



Efficiency: same as efficiency of the DFS-based algorithm

14



a b c d e f g h

1) Source nodes a&c remove a

2) Source nodes b&c&e , remove b

e → f

3) SN: e&c , remove c

4) SN : e , g,d , remove d e → f g → h

5) e&g , remove e

6) f, g , remove f

7) g , remove g

8) h , remove h

Generating Permutations

Minimal-change decrease-by-one algorithm

If n = 1 return 1; otherwise, generate recursively the list of all permutations of 12…n-1 and then insert n into each of those permutations by starting with inserting n into 12...n-1 by moving right to left and then switching direction for each new permutation

Generating Permutations

1 2 3

Example: n=3

start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 12 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

12 21

123 132 312
213 231 321

n=4

1 2 3 4	1 3 2 4	3 1 2 4
1 2 4 3	1 3 4 2	3 1 4 2
1 4 2 3	1 4 3 2	3 4 1 2
4 1 2 3	4 1 3 2	4 3 1 2
2 1 3 4	2 3 1 4	3 2 1 4
2 1 4 3	2 3 4 1	3 2 4 1
2 4 1 3	2 4 3 1	3 4 2 1
4 2 1 3	4 2 3 1	4 3 2 1

n=4

Other permutation generating algorithms

- Johnson-Trotter (p. 145)
- Lexicographic-order algorithm (p. 146)
- Heap's algorithm (Problem 4 in Exercises 4.3)

Generating Subsets

Binary reflected Gray code: minimal-change algorithm for generating 2^n bit strings corresponding to all the subsets of an n-element set where $n > 0$

subset

$\{a,b,c\} \rightarrow$ 000 $\rightarrow \emptyset$

100 $\{a\}$

110 $\{a,b\}$

111 $\{a,b,c\}$

If n=1 make list L of two bit strings 0 and 1
else
  generate recursively list $L1$ of bit strings of length n-1
  copy list $L1$ in reverse order to get list $L2$
  add 0 in front of each bit string in list $L1$
  add 1 in front of each bit string in list $L2$
  append $L2$ to $L1$ to get L
return L

(1)
000
001 add c
011 add b } minimal change (2) 001 100 110 } not
010 remove c

(n=3) {a,b,c}

L1 00 01 } 00 01 11 10 → L
L2 11 10

000 001 011 010 } 000 001 011 010 110 111 101 100
110 111 101 100

⇓

$\phi \to \{c\} \to \{b,c\} \to \{b\} \to \{a,b\} \to \{a,b,c\} \to \{a,c\} \to \{a\}$

21

Decrease-by-Constant-Factor Algorithms

In this variation of decrease-and-conquer, instance size is reduced by the same factor (typically, 2)

Examples:
- binary search and the method of bisection
- exponentiation by squaring
- multiplication à la russe (Russian peasant method)
- fake-coin puzzle
- Josephus problem

22

Binary Search

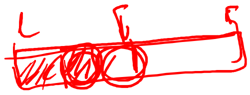
Very efficient algorithm for searching in sorted array:

K

vs

A[0] . . . A[m] . . . A[n-1]

If K = A[m], stop (successful search); otherwise, continue searching by the same method in A[0..m-1] if K < A[m] and in A[m+1..n-1] if K > A[m]



```
l ← 0;   r ← n-1
while l ≤ r do
    m ← ⌊(l+r)/2⌋
    if  K = A[m]  return m
    else if K < A[m]  r ← m-1
    else l ← m+1
return -1
```


Analysis of Binary Search

- Time efficiency
- worst-case recurrence: $C_w(n) = 1 + C_w(\lfloor \frac{n}{2} \rfloor)$, $C_w(1) = 1$
  solution: $C_w(n) = \lceil \log_2(n+1) \rceil$

  This is VERY fast: e.g., $C_w(10^6) = 20$
- Optimal for searching a sorted array
- Limitations: must be a sorted array (not linked list)
- Bad (degenerate) example of divide-and-conquer
- Has a continuous counterpart called bisection method for solving equations in one unknown $f(x) = 0$

$C_w(n) = \underbrace{C_w(\lfloor n/2 \rfloor)}_{} + \underbrace{1}_{}$ ⇒ # of comparisons.

I compare with the middle and then I continue with the half of the array.

$\underline{C_w(1) = 1}$ ⇒ do 1 comparison if I have just a single element.

$n = 2^k$ $\quad C_w(2^k) = 1 + C_w(2^{k-1})$

$= 1 + 1 + C_w(2^{k-2})$

$= 1 + 1 + 1 + C_w(2^{k-3})$

$\vdots$

$= k + \underbrace{C_w(2^{k-k})}_{C_w(2^0) = 1} = k+1 = \log n + 1 \in \Theta(\log n)$

best case.

$\underline{C_b(n) = 1}$

Exponentiation by Squaring

$T(n) = aT(n/b) + f(n)$ $\quad f(n)$ → d degree.

The problem: compute a^n where n is nonnegative integer

The problem can be solved by applying recursively the formulas:

For even values of n

$$a^n = \left(a^{\frac{n}{2}}\right)^2 \quad if\ n > 0\ and\ a^0 = 1$$

For odd values of n

$$a^n = \left(a^{\frac{n-1}{2}}\right)^2 a$$

Basic operation; multiplication.

Recurrence: $M(n) = M\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + f(n)$ $\;where\;\; f(n) = 1\ or\ 2,$ → 2 multiplication.

↳ do 1 multiplication

$M(0) = 0$

Master theorem: $M(n) \in \Theta(logn) = \Theta(b)\ \ where\ \ b = \lceil \log_2(n+1) \rceil$

Russian Peasant Multiplication

The problem: Compute the product of two positive integers

Can be solved by a decrease-by-half algorithm based on the following formulas.

For even values of n

$$n * m = \frac{n}{2} * 2m$$

For odd values of n

$$n * m = \frac{n-1}{2} * 2m + m \quad if\ n > 1\ and\ \boxed{m\ if\ n = 1}$$

Example of Russian Peasant Multiplication

even $n * m$ $\quad \frac{n}{2} * 2m$

odd $n * m$ $\quad \frac{n-1}{2} * 2m + m$

Compute 20 * 26

n	m	
20	26	
10	52	
5	104	104
2	208	+
1	416	416
		520

/2 ×2

12	52
6	104
3	208
1	416

208
416
624

Efficiency: O(log n)
the smallest number

Note: Method reduces to adding m's values corresponding to odd n's.

Fake-Coin Puzzle (simpler version)

There are *n* identically looking coins one of which is fake. There is a balance scale but there are no weights; the scale can tell whether two sets of coins weigh the same and, if not, which of the two sets is heavier (but not by how much). Design an efficient algorithm for detecting the fake coin. Assume that the fake coin is known to be lighter than the genuine ones.

$n/2$ $n/2$

Decrease by factor 2 algorithm

$W(n) = W(\lfloor n/2 \rfloor) + 1$ $W(1) = 0 \Rightarrow \Theta(\log_2 n)$

rest

$n/3$ $n/3$ $n-2\lfloor n/3 \rfloor$

Decrease by factor 3 algorithm $n/3$

$\Theta(\log_3 n)$

Variable-Size-Decrease Algorithms

In the variable-size-decrease variation of decrease-and-conquer, instance size reduction varies from one iteration to another

Examples:

- Euclid's algorithm for greatest common divisor
- partition-based algorithm for selection problem
- interpolation search
- some algorithms on binary search trees
- Nim and Nim-like games

Euclid's Algorithm

Euclid's algorithm is based on repeated application of equality

gcd(*m, n*) = gcd(*n, m* mod *n*)

Ex.:

gcd(80,44) = gcd(44,36) = gcd(36, 12) = gcd(12,0) = 12

One can prove that the size, measured by the second number, decreases at least by half after two consecutive iterations.

Hence, T(*n*) $\in$ O(log *n*)

Selection Problem

Find the *k*-th smallest element in a list of *n* numbers

- *k* = 1 or *k* = *n*

  min max.

- *median*: *k* = ⌈n/2⌉

  9

  Example: 4, 1, 10, 9, 7, 12, 8, 2, 15 median = ?

  1 2 4 7 8 9 10 12 15

The median is used in statistics as a measure of an average value of a sample. In fact, it is a better (more robust) indicator than the mean, which is used for the same purpose.

1 1.2 1.1 1.2 1.3 1.1 1.2 1.25 1 10000 outlier

mean ≈ 1000

median = 1.2

Algorithms for the Selection Problem

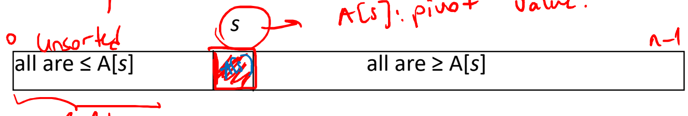
The sorting-based algorithm: Sort and return the *k*-th element
Efficiency (if sorted by mergesort): $\Theta(n \log n)$

Algorithms for the Selection Problem

left part contains all the elements smaller than A[s]

A faster algorithm is based on the array *partitioning*:

A[s]: pivot value.



Assuming that the array is indexed from 0 to n-1 and s is a split position obtained by the array partitioning:

- If $s = k$-1, the problem is solved;
- if $s > k$-1, look for the k-th smallest element in the left part;
- if $s < k$-1, look for the (k-s)-th smallest element in the right part.

Note: The algorithm can simply continue until $s = k$-1.

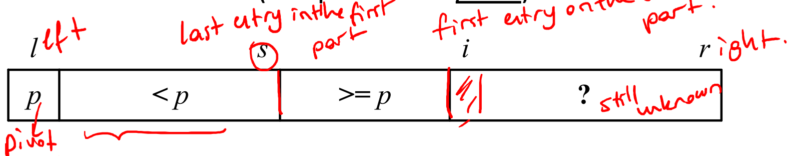
Two Partitioning Algorithms

There are two principal ways to partition an array:

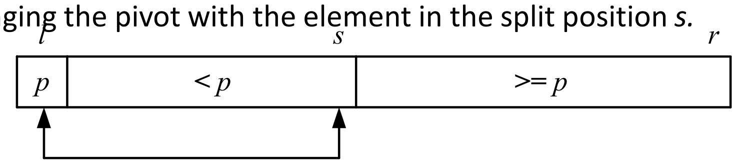
- One-directional scan (Lomuto's partitioning algorithm)
- Two-directional scan (Hoare's partitioning algorithm)

Lomuto's Partitioning Algorithm

Scans the array left to right maintaining the array's partition into three contiguous sections: $< p$, $\geq p$, and unknown, where p is the value of the first element (the partition's *pivot*).

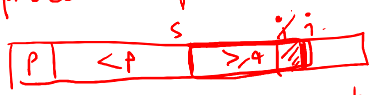


On each iteration the unknown section is decreased by one element until it's empty and a partition is achieved by exchanging the pivot with the element in the split position *s*.



At each iteration, it compares the first element in the unknown part with the pivot.

if $A[i] \geq p$, i is simply incremented to expand the segment of the elements greater than or equal to p, while shrinking the unprocessed part.

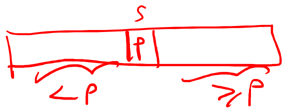


if $A[i] < p$, it is the segment of the elements smaller than p that needs to be expanded.

This is done by incrementing s, swapping $A[s]$ and $A[i]$, then incrementing i to point the new first element of shrunk unprocessed segments.

37

38



pivot

ALGORITHM *LomutoPartition*$(A[l..r])$

```
//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray A[l..r] of array A[0..n − 1], defined by its left and right
//       indices l and r (l ≤ r)
//Output: Partition of A[l..r] and the new position of the pivot
p ← A[l]
s ← l
for i ← l + 1 to r do
    if A[i] < p
        s ← s + 1;   swap(A[s], A[i])
swap(A[l], A[s])
return s
```


Tracing Lomuto's Partioning Algorithm

s	i							
4	1	10	8	7	12	9	2	15
	s	i						
4	1	10	8	7	12	9	2	15
	s						i	
4	1	10	8	7	12	9	2	15
		s						i
4	1	2	8	7	12	9	10	15
		s						
4	1	2	8	7	12	9	10	15
2	1	**4**	8	7	12	9	10	15

ALGORITHM *LomutoPartition*$(A[l..r])$

```
//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray A[l..r] of array A[0..n − 1], defined by its left and right
//       indices l and r (l ≤ r)
//Output: Partition of A[l..r] and the new position of the pivot
p ← A[l]
s ← l
for i ← l + 1 to r do
    if A[i] < p
        s ← s + 1;   swap(A[s], A[i])
swap(A[l], A[s])
return s
```

p: 4

s: ~~0~~ ~~1~~ 2

i: ~~1~~ ~~2~~ ~~7~~ ... ~~7~~ 8

Tracing Quickselect (Partition-based Algorithm)

Find the median of 4, 1, 10, 9, 7, 12, 8, 2, 15

Here: n = 9, $k = \lceil 9/2 \rceil = 5$, k -1=4

after 1st partitioning: s=2<k-1=4

pivot

after 2nd partitioning: s=4=k-1

0	1	2	3	**4**	5	6	7	8
4	1	10	8	7	12	9	2	15
2	1	**4**	8	7	12	9	10	15
			8	7	12	9	10	15
			7	**8**	12	9	10	15

The median is A[4]= 8

Best case: Partitioning an n-element array always require n-1 key comparisons.

$C_b(n) = n-1 \in \Theta(n)$ linear

ALGORITHM *Quickselect*($A[l..r]$, k)

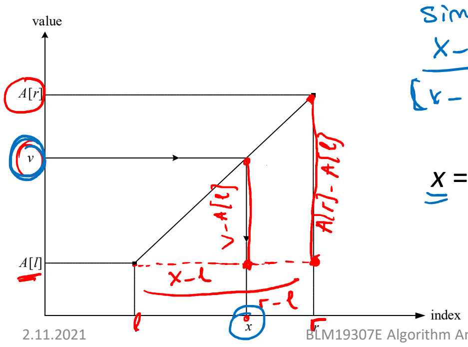
//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k ($1 \le k \le r - l + 1$)
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow$ *LomutoPartition*($A[l..r]$) //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ *Quickselect*($A[l..s-1]$, k)
else *Quickselect*($A[s+1..r]$, $k - 1 - s$)

worst case: $k=n$

$C_w(n) = (n-1) + (n-2) + \cdots + 1 \approx \Theta(n^2)$

Interpolation Search

Searches a sorted array similar to binary search but estimates location of the search key in $A[l..r]$ by using its value v. Specifically, the values of the array's elements are assumed to grow linearly from $A[l]$ to $A[r]$ and the location of v is estimated as the x-coordinate of the point on the straight line through $(l, A[l])$ and $(r, A[r])$ whose y-coordinate is v:



similarity of triangles

$$\frac{x-l}{(v-A[l])} = \frac{r-l}{(A[r]-A[l])}$$

$$x = l + \lfloor (v - A[l])(r - l)/(A[r] - A[l]) \rfloor$$

43

Analysis of Interpolation Search

- Efficiency

  average case: $C(n) < \log_2 \log_2 n + 1$

  worst case: $C(n) = n$

- Preferable to binary search only for VERY large arrays and/or expensive comparisons

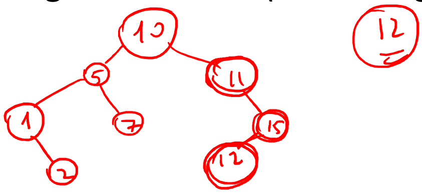
- Has a counterpart, the *method of false position* (*regula falsi*), for solving equations in one unknown

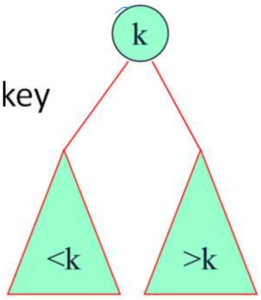
44

Binary Search Tree Algorithms

Several algorithms on BST requires recursive processing of just one of its subtrees, e.g.,

- Searching ✓
- Insertion of a new key ✓
- Finding the smallest (or the largest) key



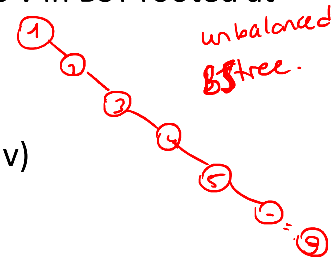


45

Searching in Binary Search Tree

already sorted
1 2 3 4 5 6 7 8 9

Algorithm BTS(x, v)
//Searches for node with key equal to v in BST rooted at node x
if x = NIL return -1
else if v = K(x) return x
else if v < K(x) return BTS(left(x), v)
else return BTS(right(x), v)



unbalanced BS tree.

Efficiency
worst case: $C(n) = n$
average case: $C(n) \approx 2\ln n \approx 1.39\log_2 n$

46