

# **Priority Queues, Heaps Trees and Heap sort**

# Priority Queues

- A priority queue is a collection of zero or more elements → each element has a **priority** or value
- Elements are deleted by increasing or decreasing order of priority rather than by the order in which they arrived in the queue
- Operations performed on priority queues
  - 1) Find an element, 2) insert a new element, 3) delete an element, etc.
- Two kinds of (Min, Max) priority queues exist
- In a **Min priority queue**, find/delete operation finds/deletes the element with minimum priority
- In a **Max priority queue**, find/delete operation finds/deletes the element with maximum priority
- Two or more elements can have the same priority

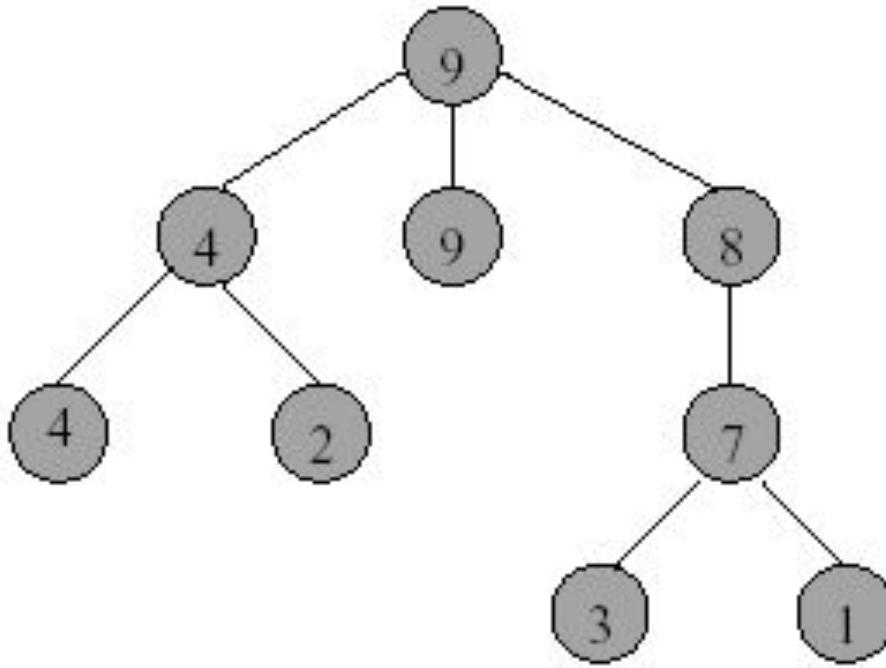
# Implementation of Priority Queues

- Implemented using **heaps**
- **Heap** is a complete binary tree that is efficiently stored using the array-based representation

# Max (Min) Tree

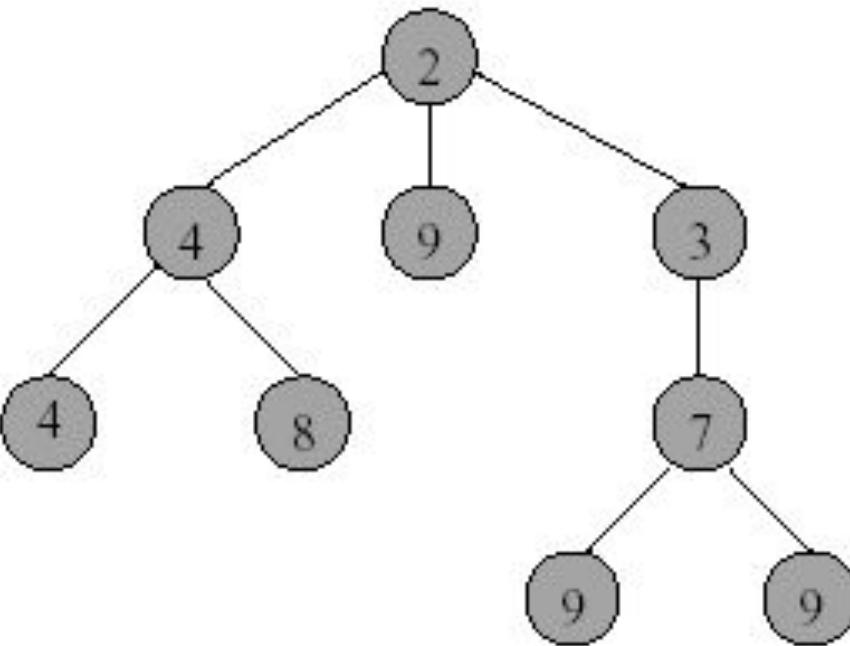
- A max tree (min tree) is a tree in which the value in each node is greater (less) than or equal to those in its children (if any)
- Nodes of a max or min tree may have more than two children (i.e., may not be binary tree)

# Max Tree Example



Root has maximum element.

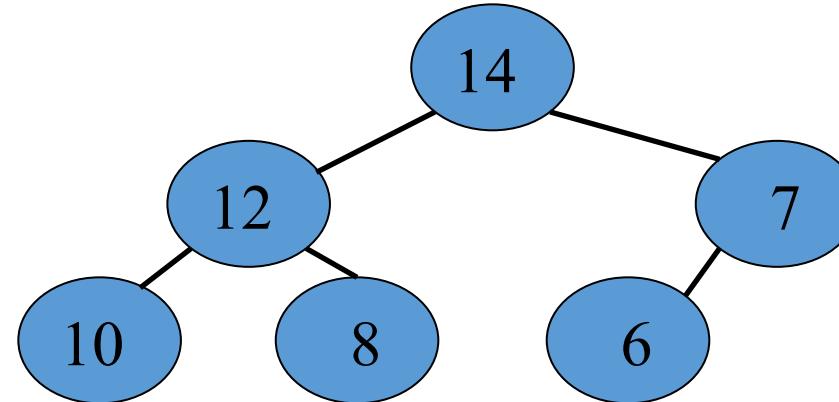
# Min Tree Example



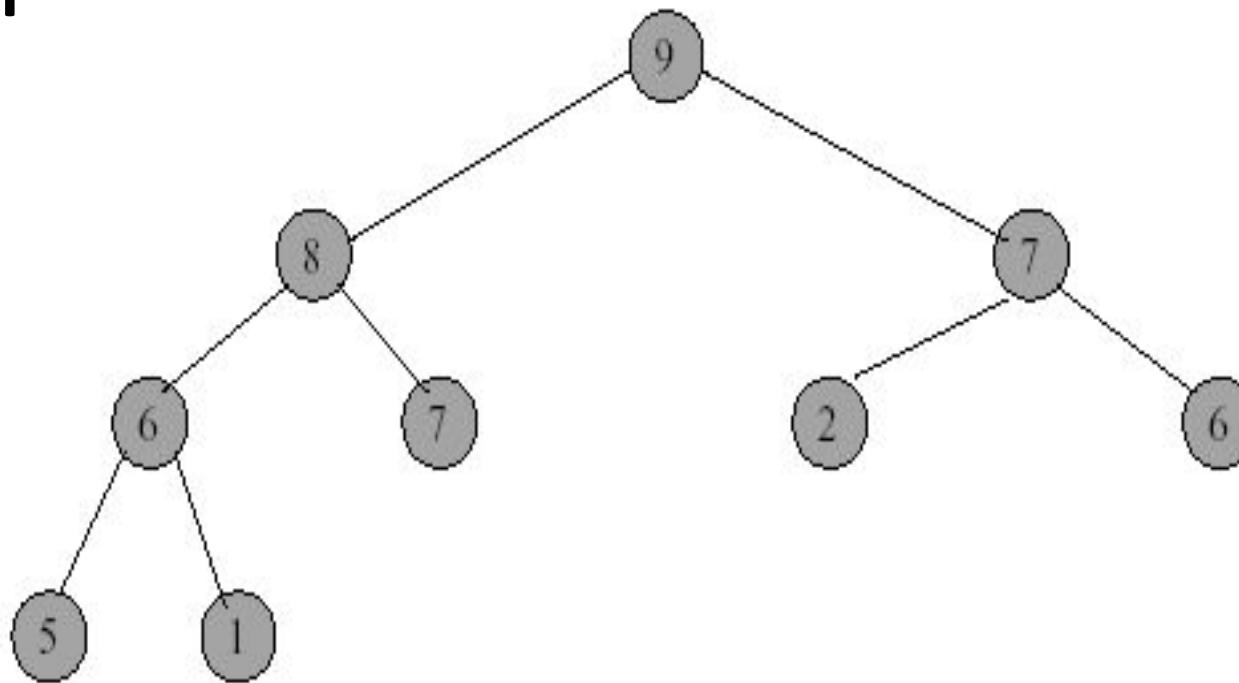
Root has minimum element.

# Heaps - Definitions

- A max heap (min heap) is a **max (min) tree** that is also a complete binary tree

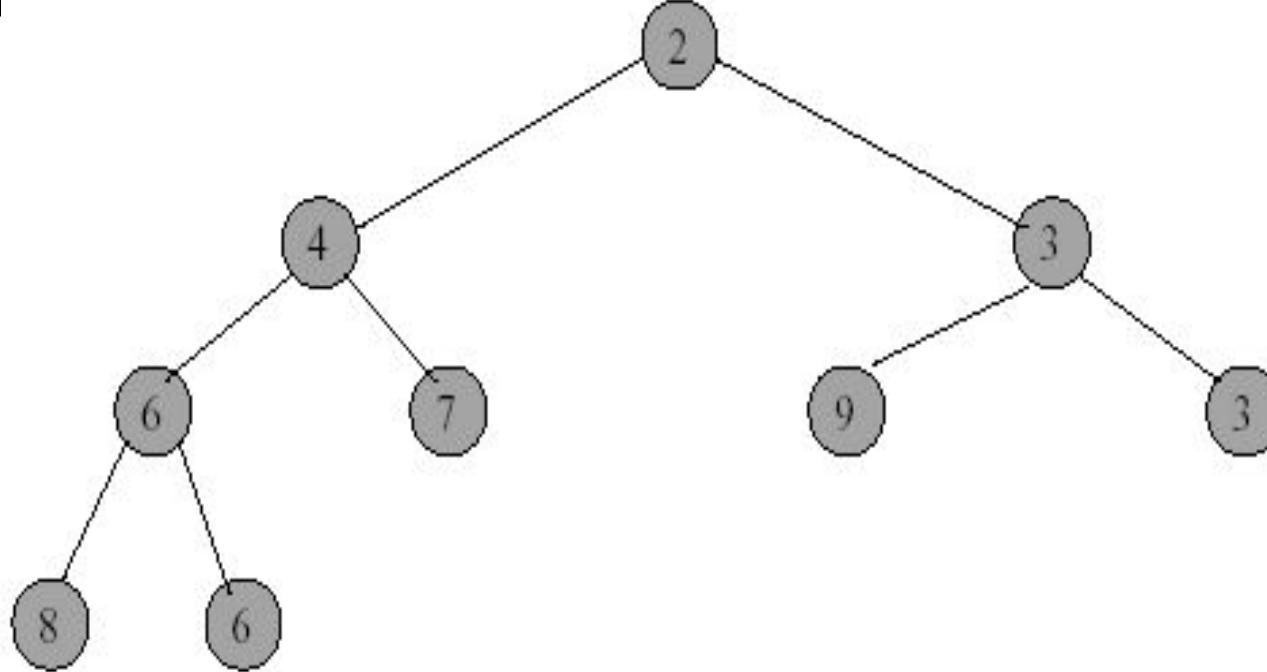


# Max Heap with 9 Nodes



Complete binary tree with 9 nodes  
that is also a max tree.

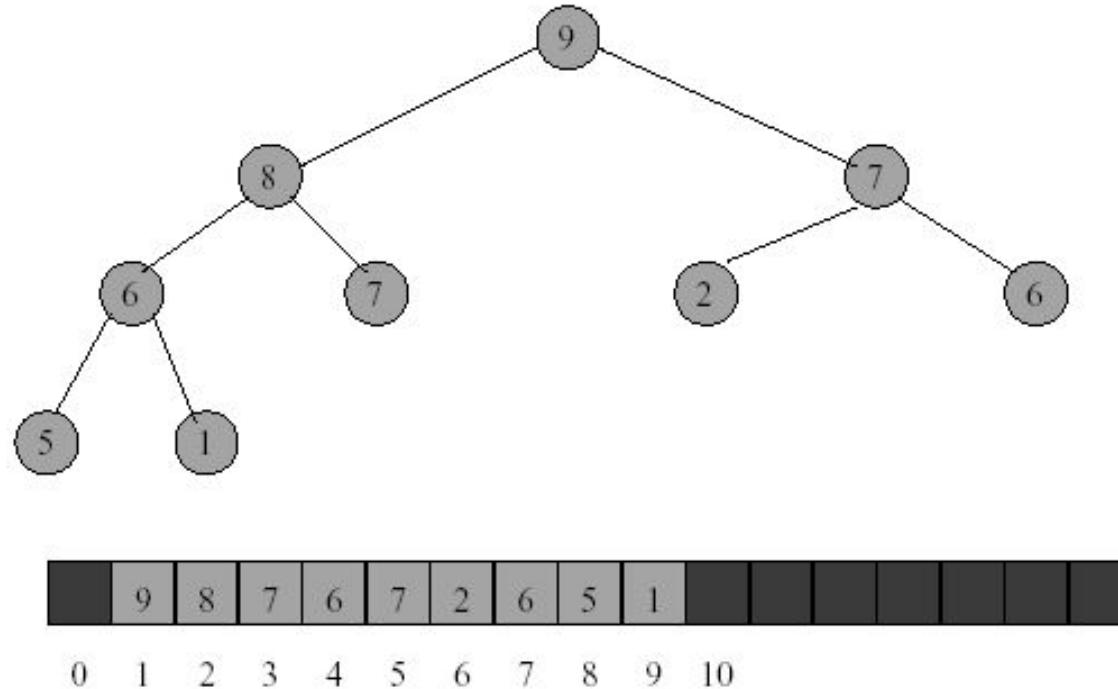
# Min Heap with 9 Nodes



Complete binary tree with 9 nodes  
that is also a min tree.

# Array Representation of Heap

- A heap is effici



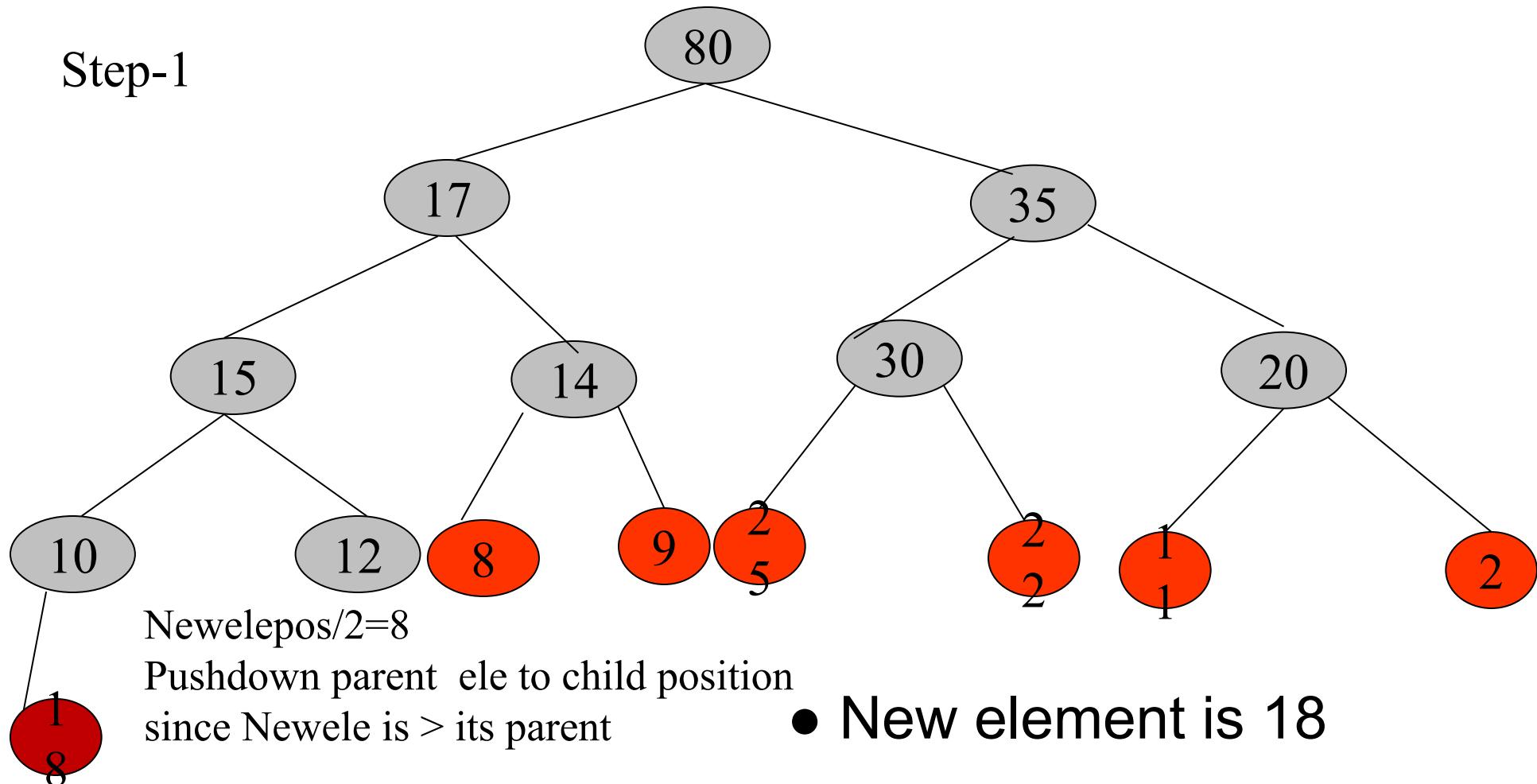
# Heap Operations

When  $n$  is the number of elements (heap size),

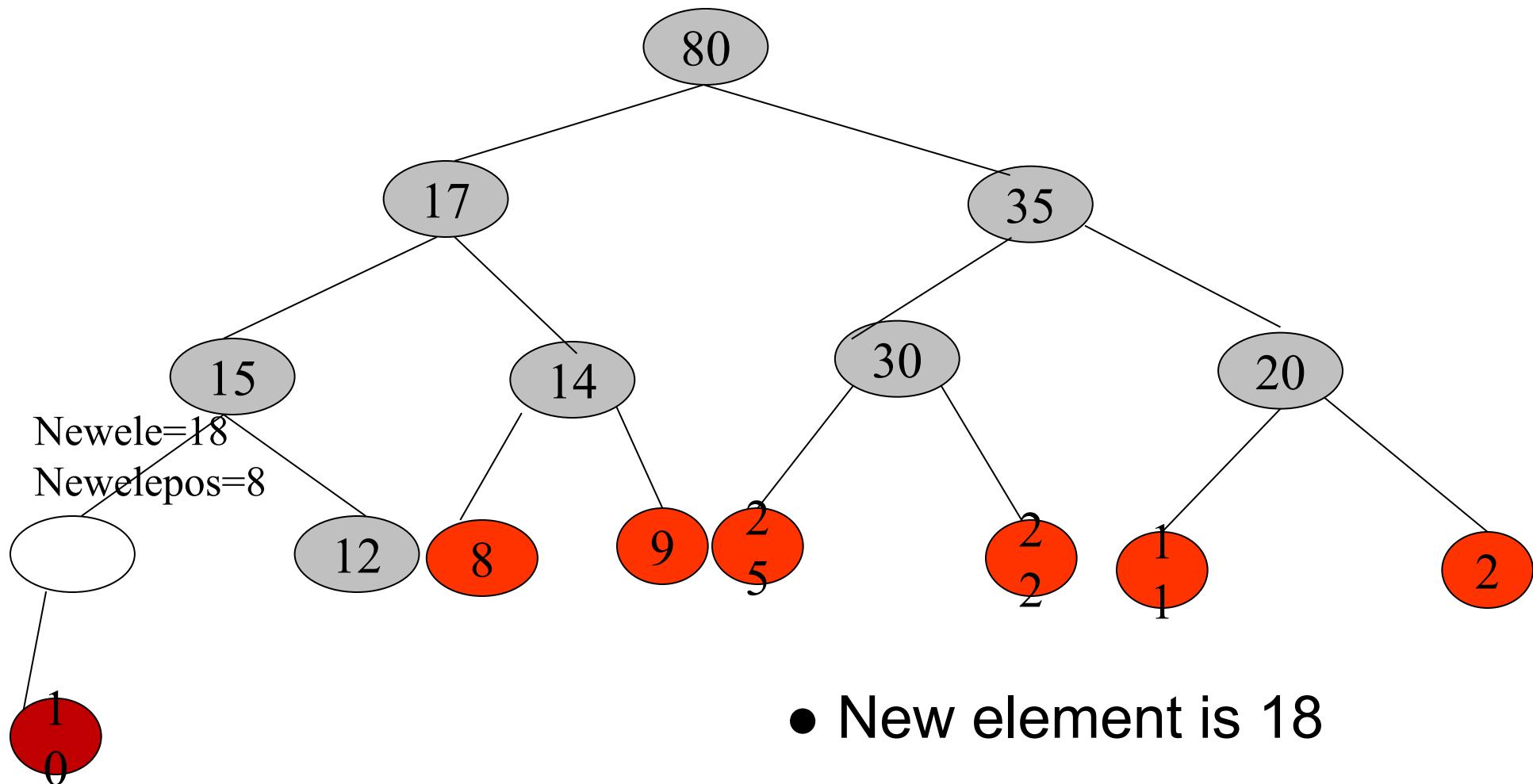
- Insertion       $\rightarrow O(\log_2 n)$
- Deletion       $\rightarrow O(\log_2 n)$
- Initialization    $\rightarrow O(n)$

# Insertion into a Max Heap

Step-1

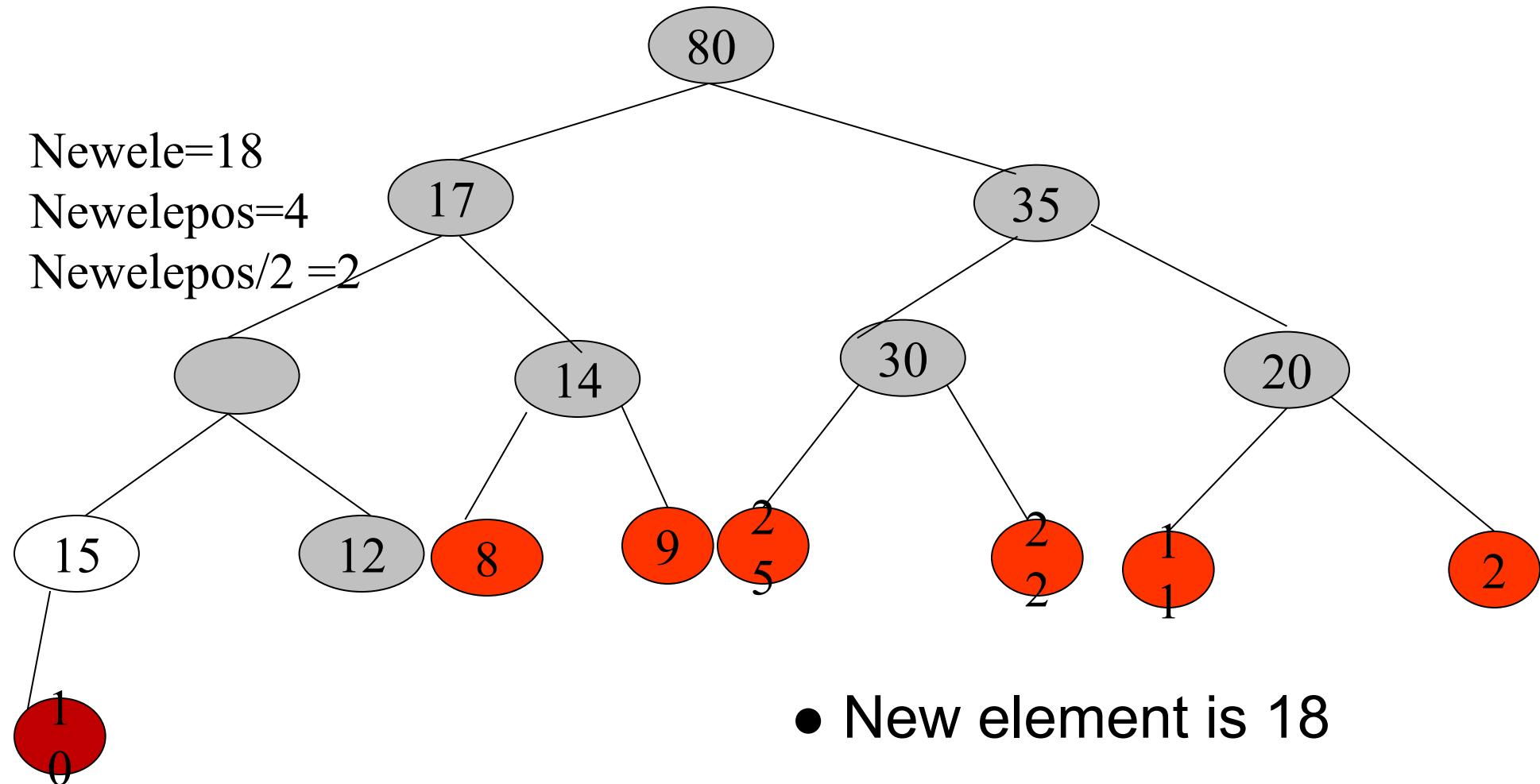


## Step-2

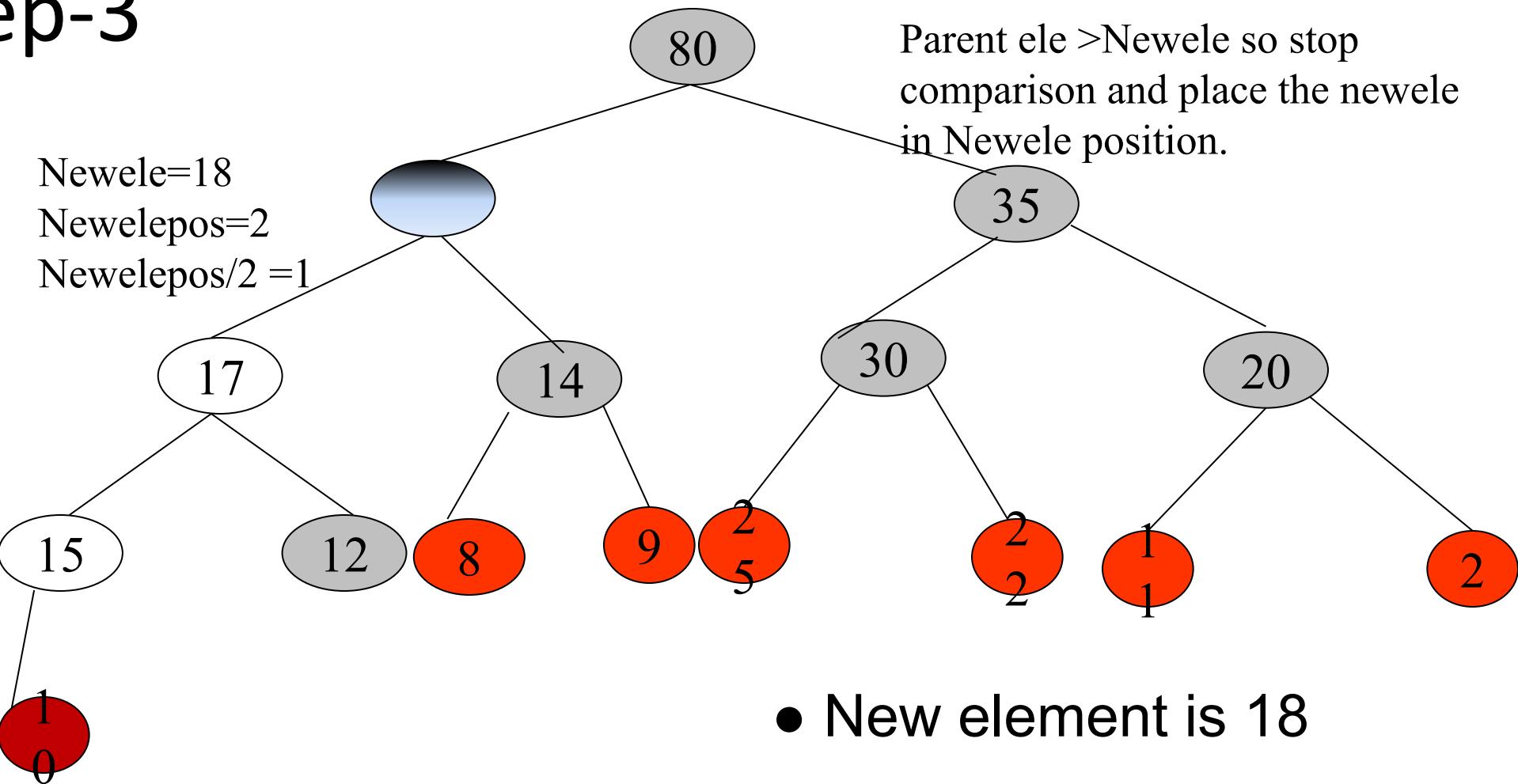


Pushdown parent ele to child position since Newele is > its parent

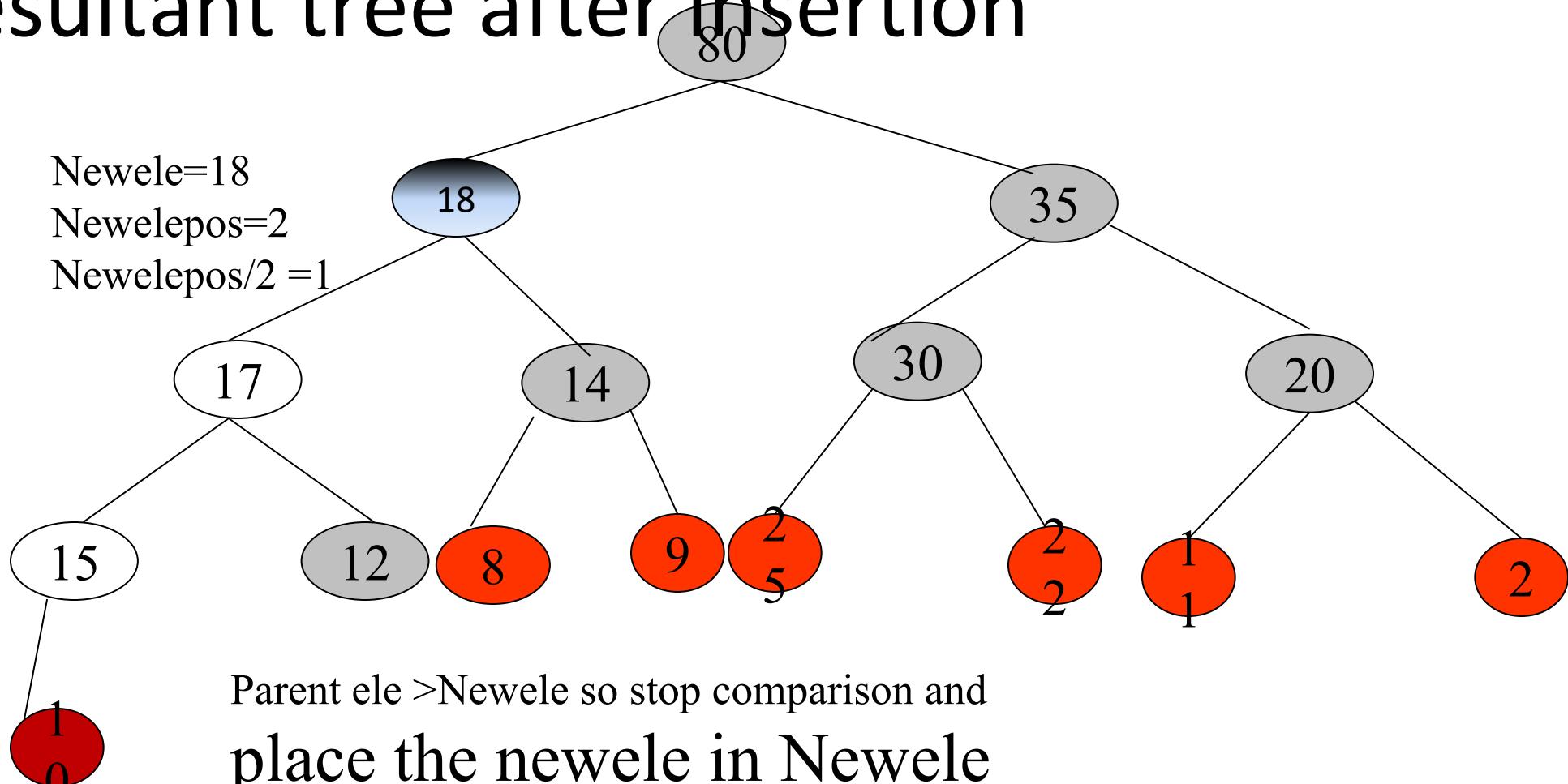
# Resultant Tree



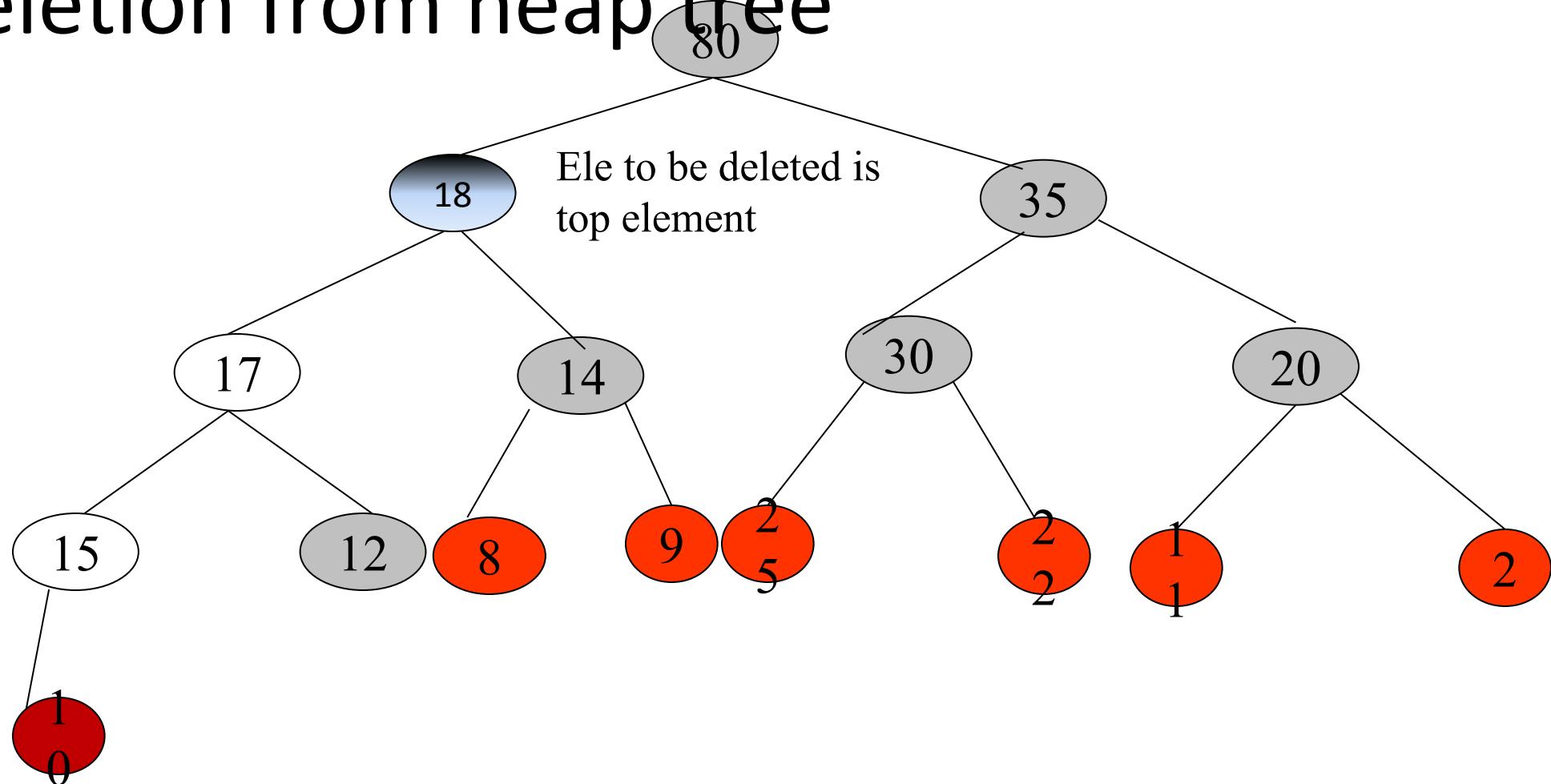
## Step-3



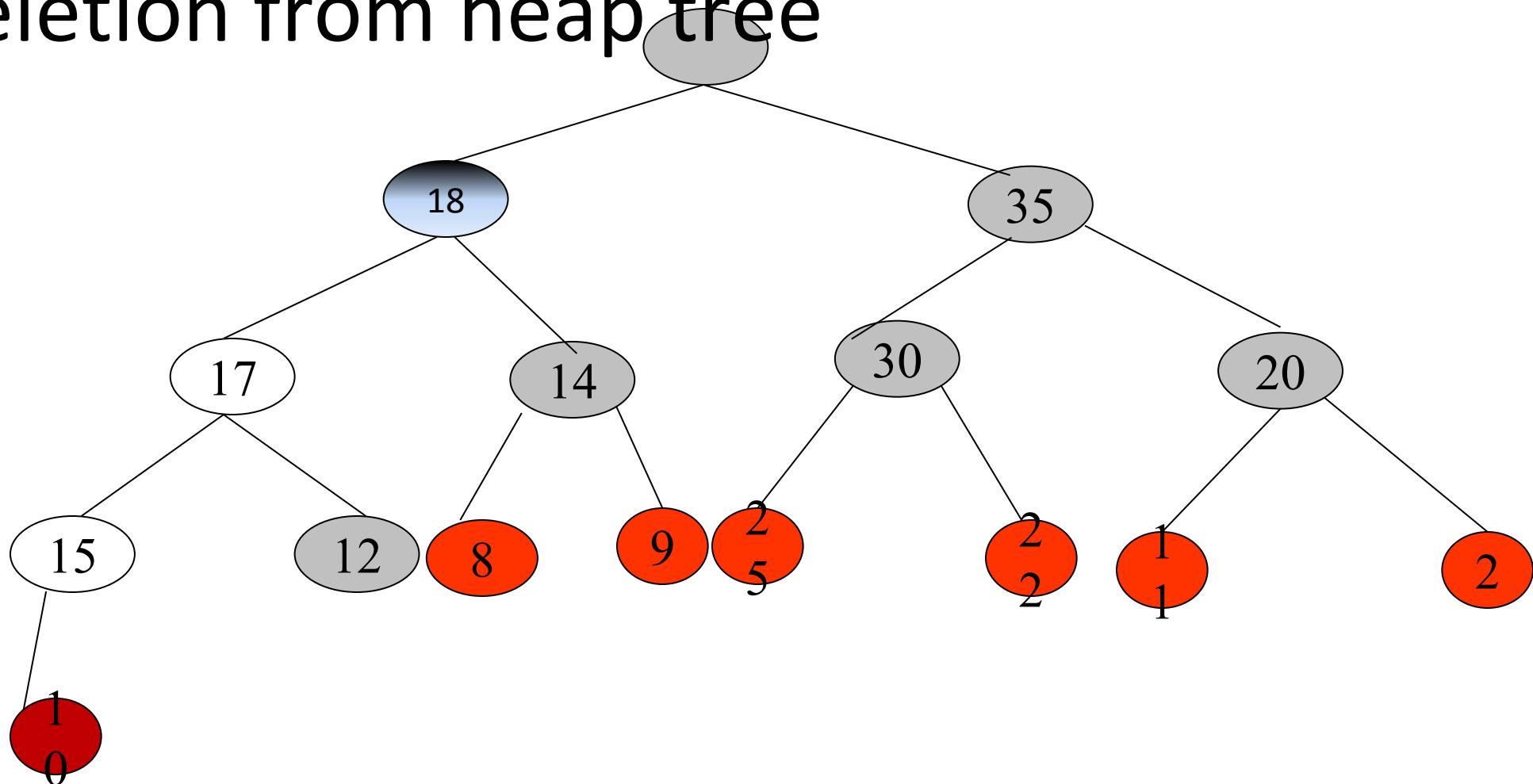
# Resultant tree after insertion



# Deletion from heap tree



# Deletion from heap tree

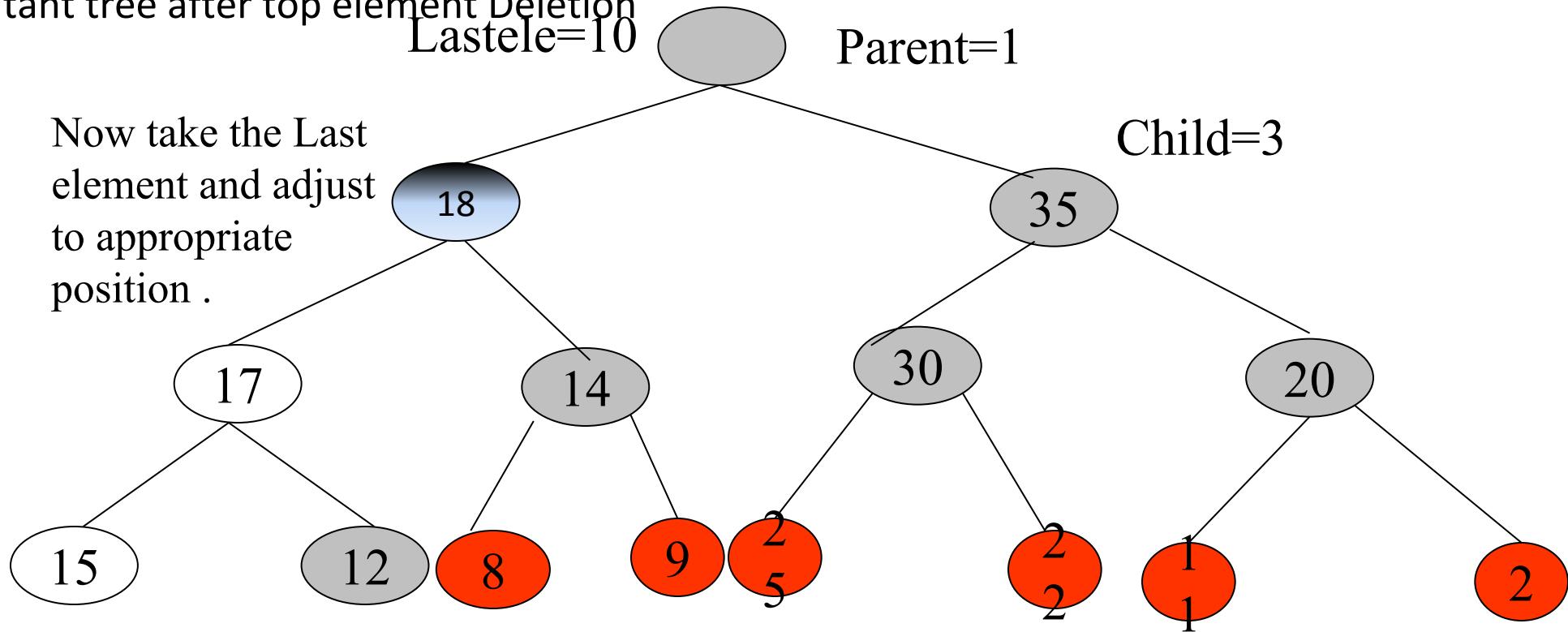


Resultant tree after top element Deletion

Lastele=10

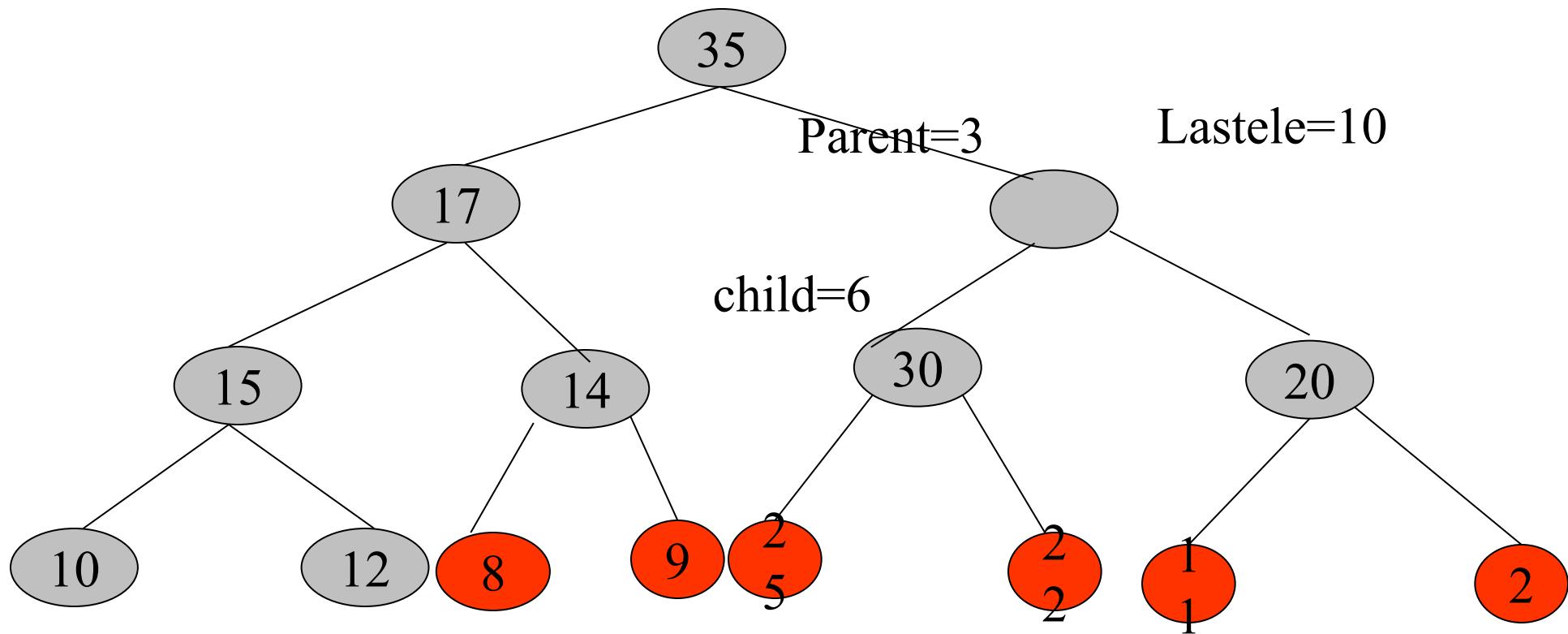
Parent=1

Now take the Last  
element and adjust  
to appropriate  
position .

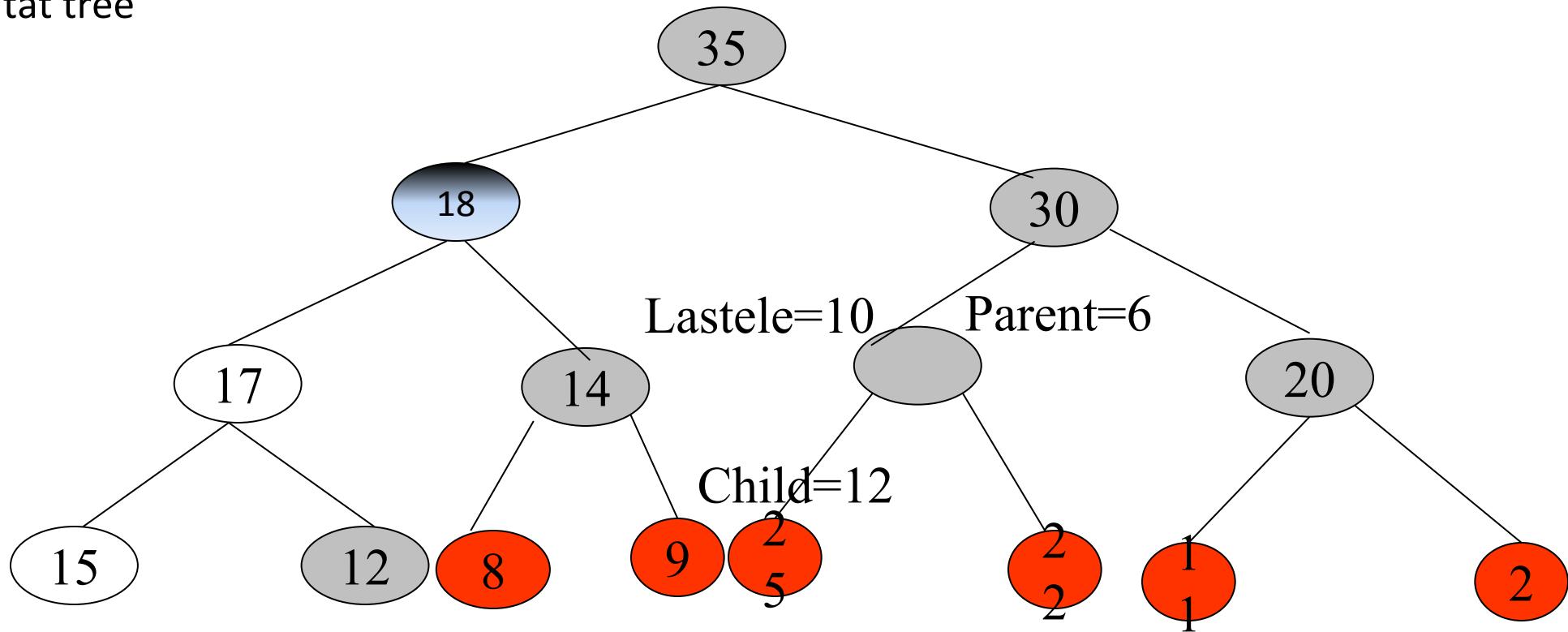


Compare Lastele with child and move up the  
child ele since child ele is greater

# Resultant Tree

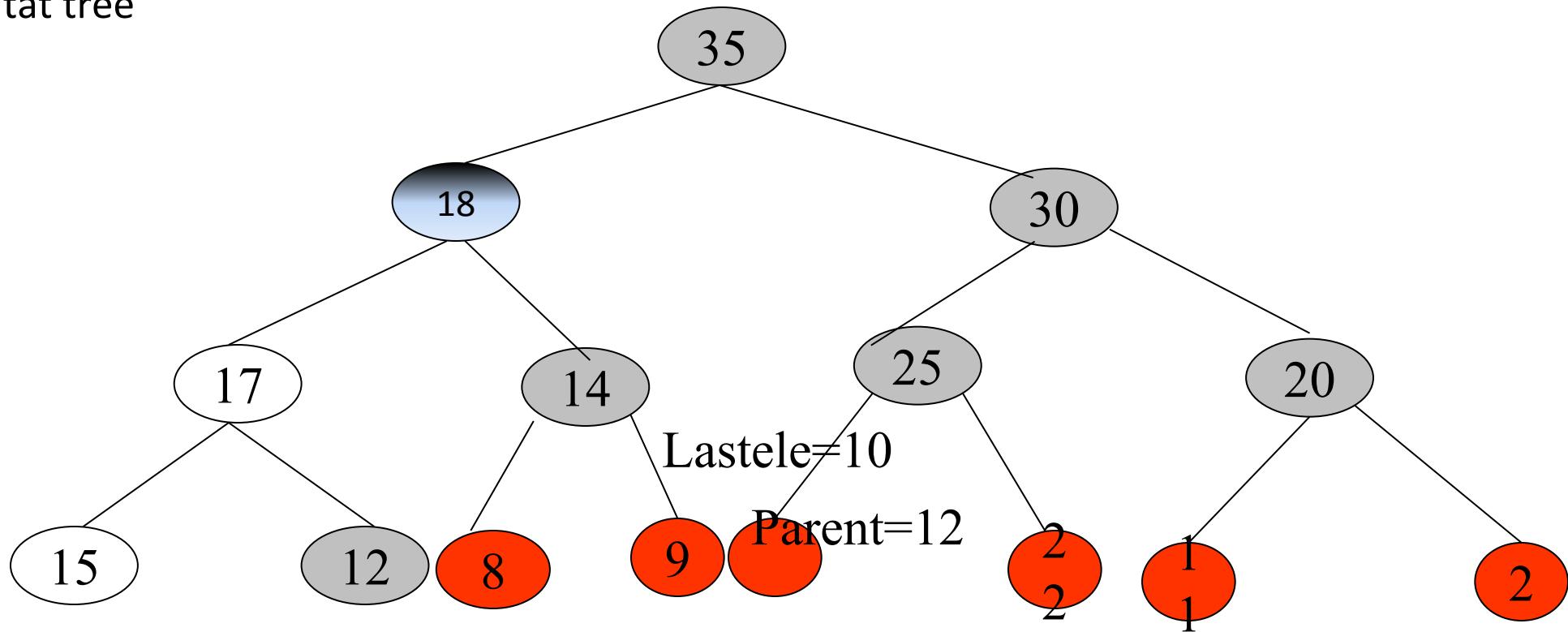


Resultat tree



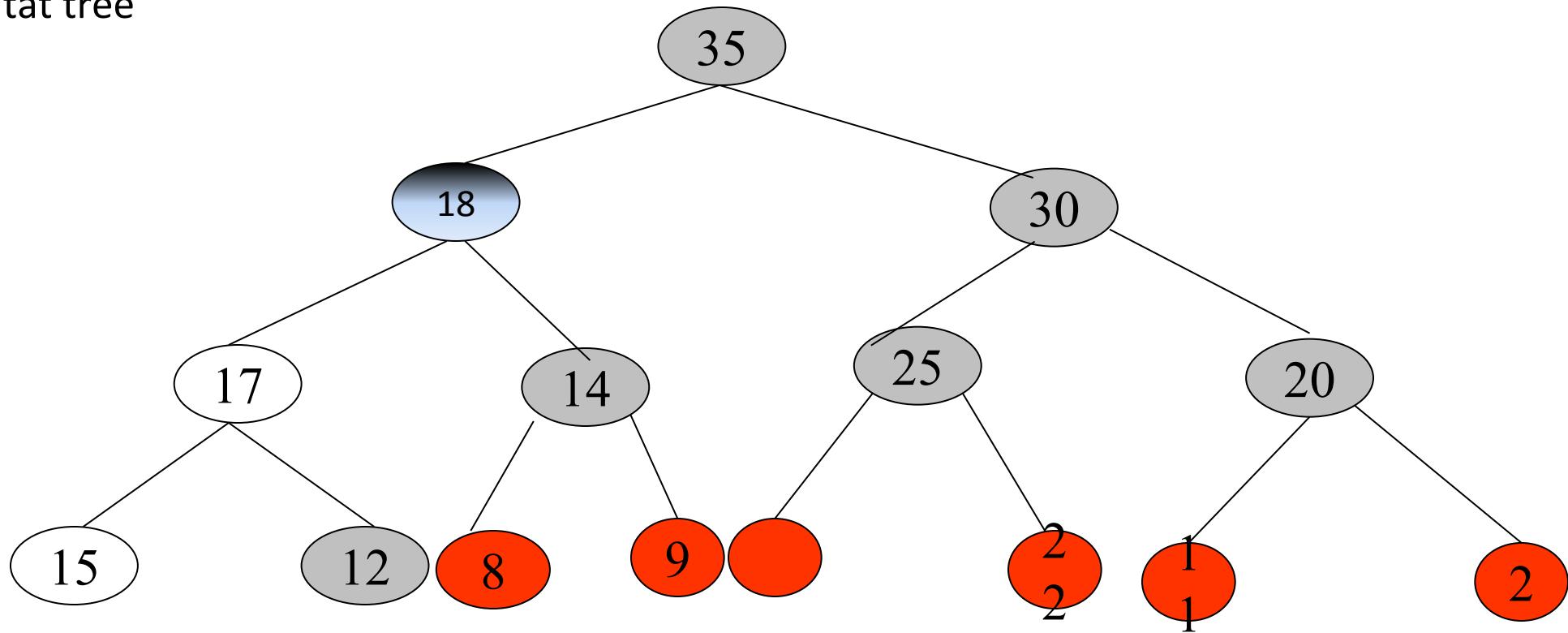
again Compare Lastele with child and move up the child ele  
since child ele is greater than Lastele

Resultat tree



again Compare Lastele with child and move up the child ele  
since child ele is greater than Lastele

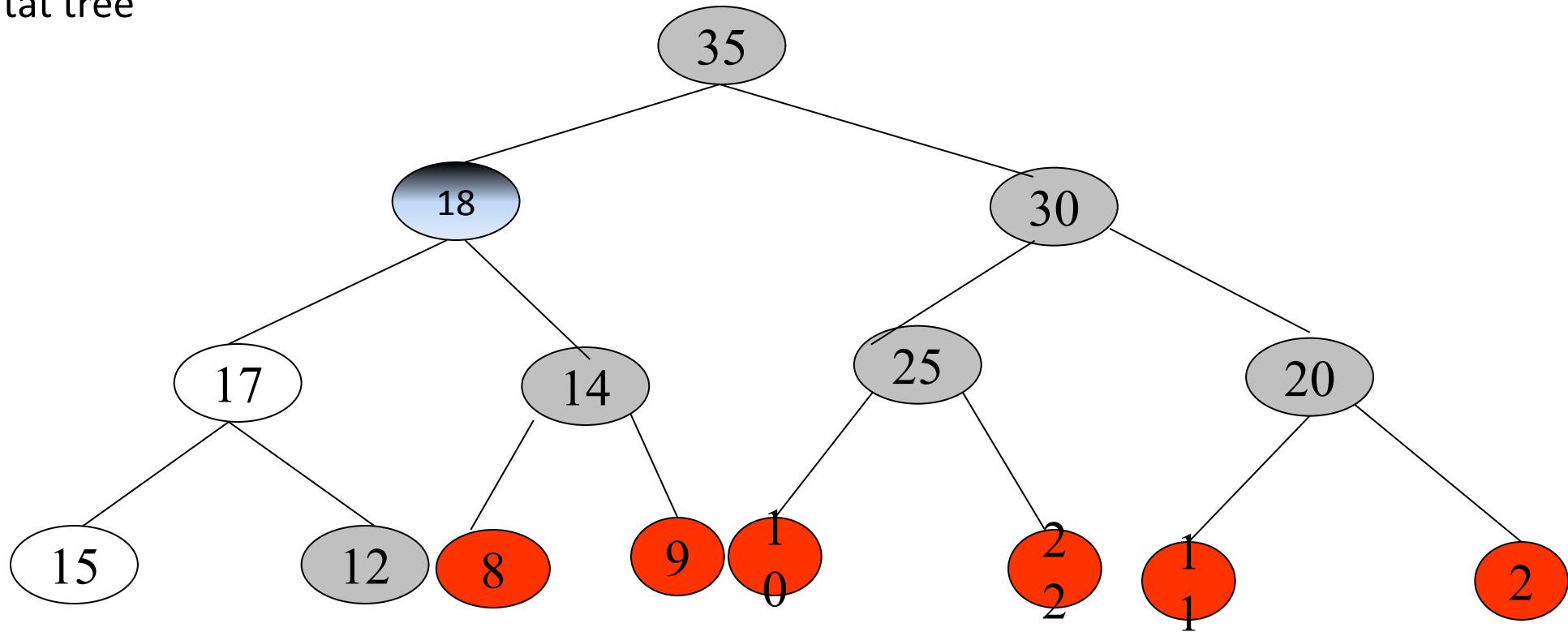
Resultat tree



Lastele=10 , parent=12

child =24 , Since 24 is greater than heapsize stop comparison and finally place the Last ele in the current Parent.

Resultat tree



# Complexity of Insertion

- Thus the time complexity is  $O(\text{height}) = O(\log_2 n)$ , where  $n$  is the heap size

# Complexity of Deletion

- The time complexity of deletion is the same as insertion
- At each level, we do  $\Theta(1)$  work
- Thus the time complexity is  $O(\text{height}) = O(\log_2 n)$ , where  $n$  is the heap size

# Sorting using max heap tree

- Elements can be sorted in ascending order using max heap tree in  $O(n \log n)$  time.
- We begin by initializing a max heap with the  $n$  elements to be sorted .
- Then we extract(i.e delete) elements from the heap one at a time.
- Initialization takes  $O(n)$  time , and each deletion takes  $O(\log n)$  time.So, the total time is  $O(n \log n)$ .

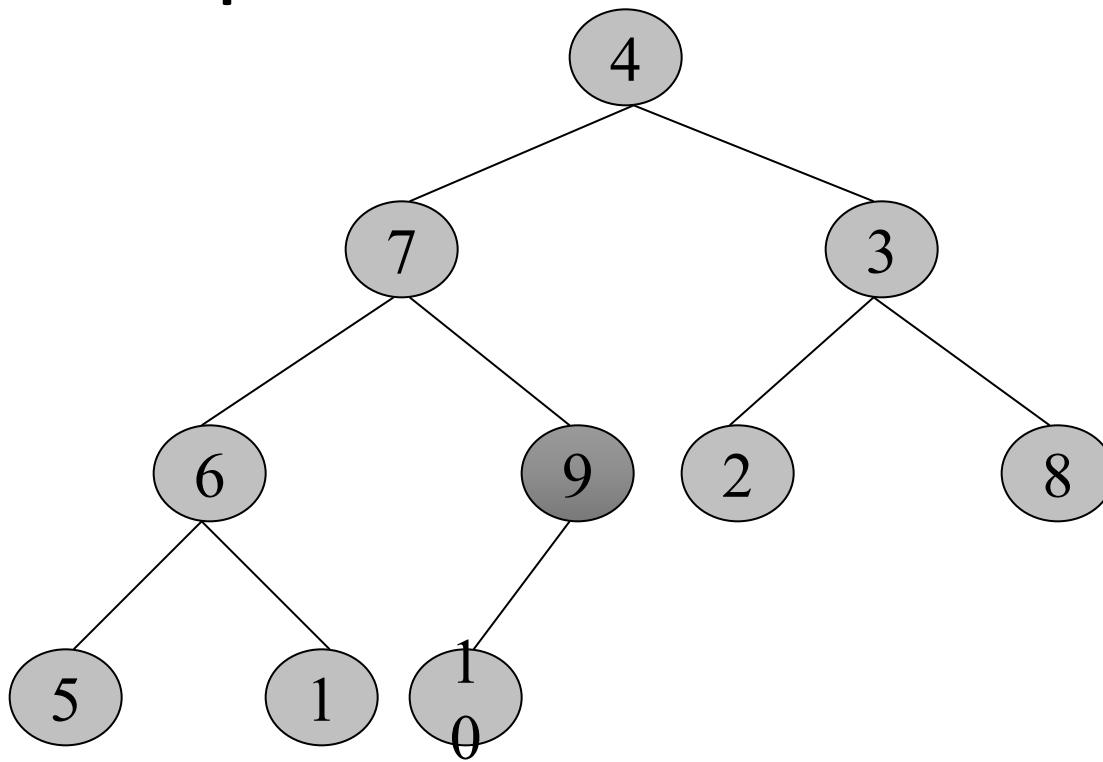
# Complexity of Initialization

- 

- Height of heap =  $h$ .
- Number of nodes at level  $j$  is  $\leq 2^{j-1}$ .
- Time for each node at level  $j$  is  $O(h-j+1)$ .
- Time for all nodes at level  $j$  is  $\leq 2^{j-1}(h-j+1) = t(j)$ .
- Total time is  $t(1) + t(2) + \dots + t(h) = O(2^h) = O(n)$ .

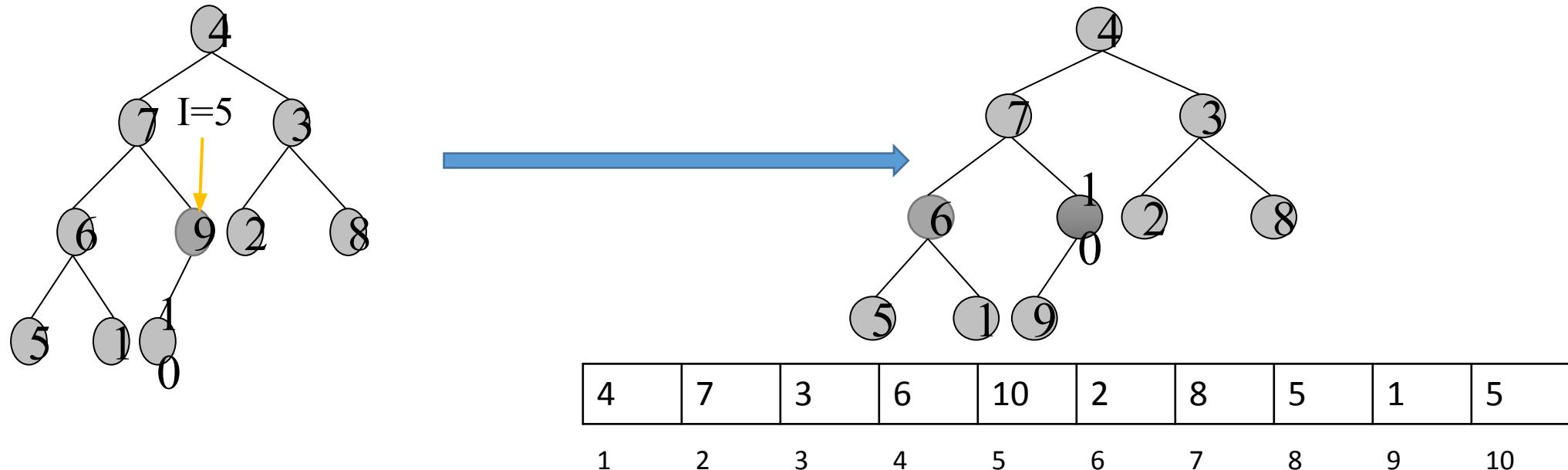
Example is illustrated in next slides

# Max heap Tree Initialization :Initial tree



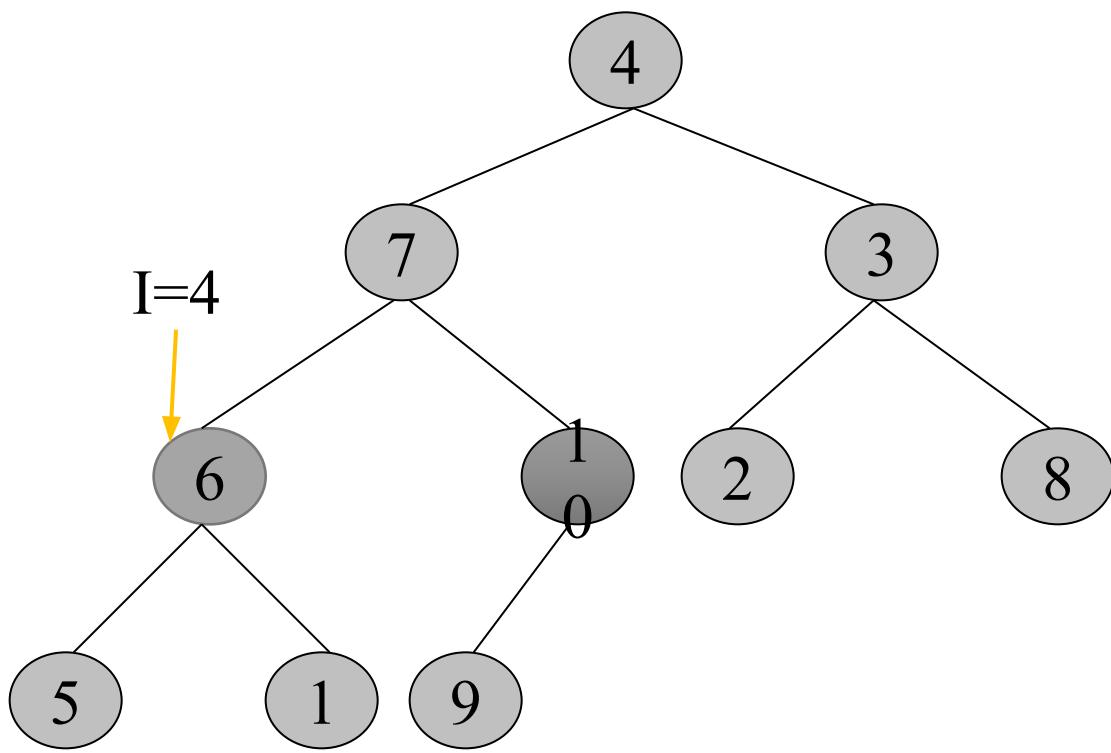
4	7	3	6	9	2	8	5	1	10
1	2	3	4	5	6	7	8	9	10

After adjusting element in the 5<sup>th</sup> position



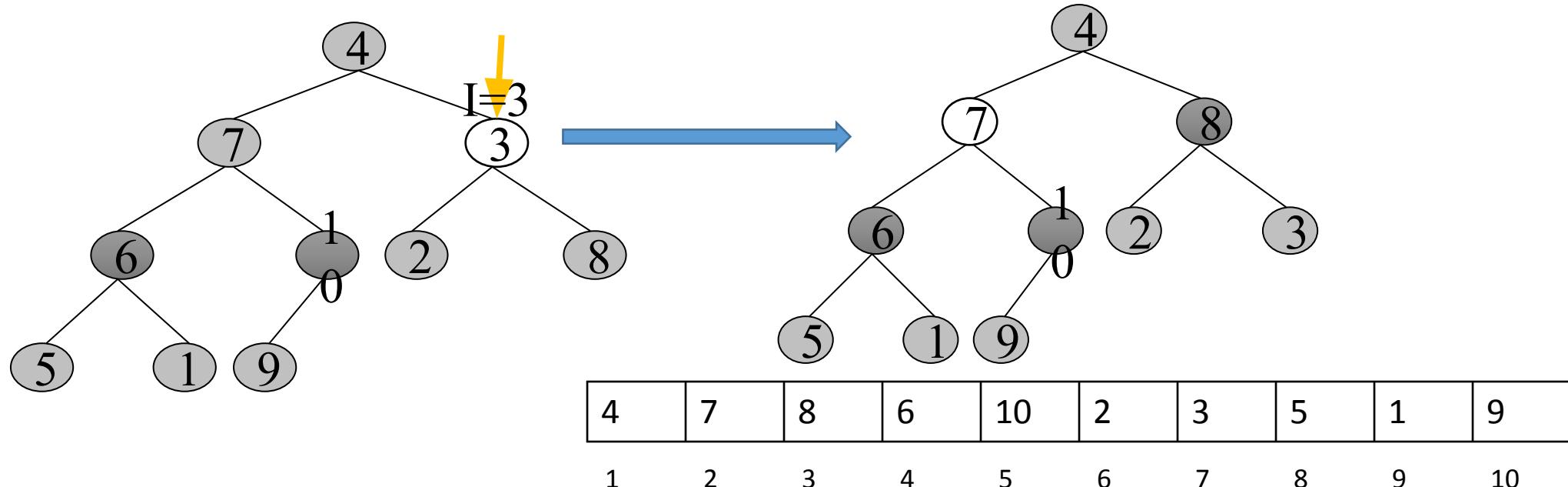
Element in the 4<sup>th</sup> position is already in the max heap order .Hence no need of adjustment

# Max heap Tree Initialization

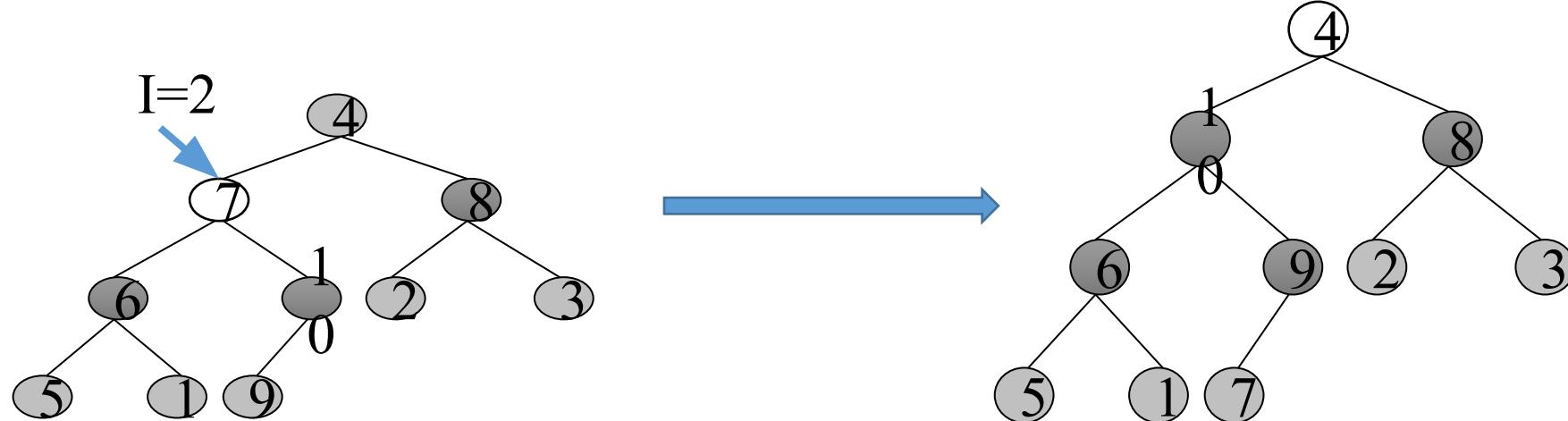


4	7	3	6	10	2	8	5	1	9
1	2	3	4	5	6	7	8	9	10

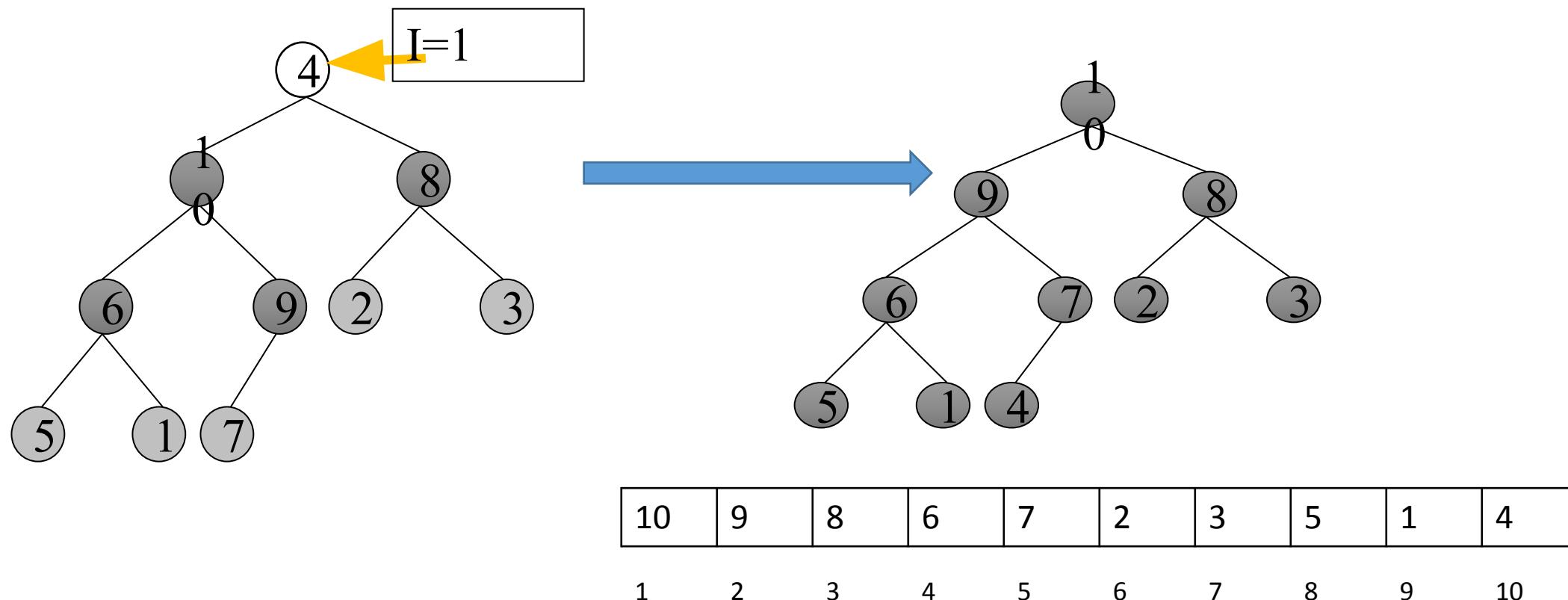
After adjusting element in the 3<sup>rd</sup> position



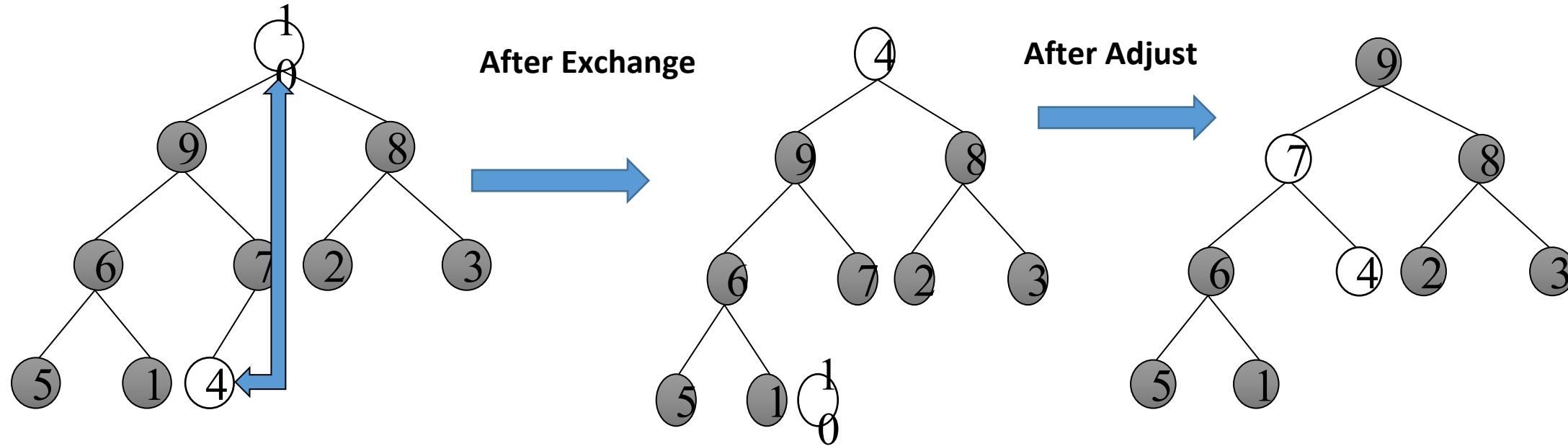
After adjusting element in the 2<sup>nd</sup> position



## After adjusting element in the 1st position

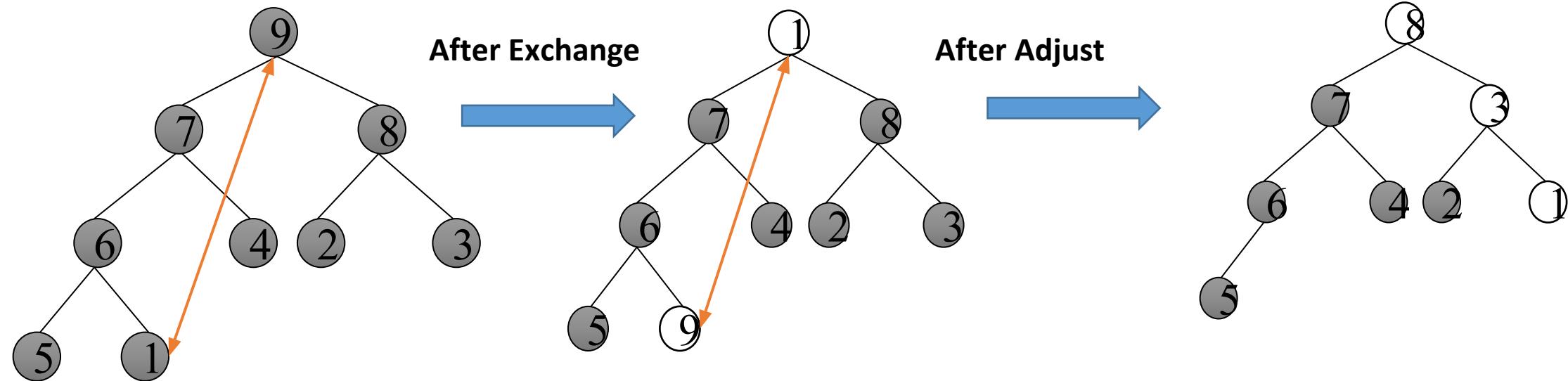


## Max heap Tree Exchange and adjust: Iteration 1



9	7	8	6	4	2	3	5	1	10
1	2	3	4	5	6	7	8	9	10

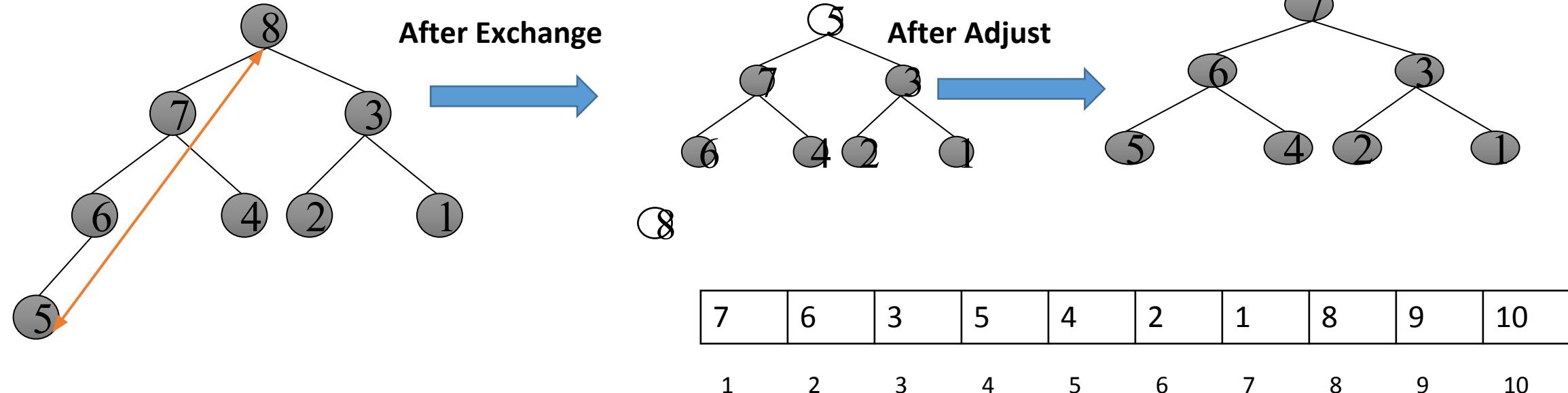
## Max heap Tree Exchange and Adjust: Iteration 2



8	7	3	6	4	2	1	5	9	10
1	2	3	4	5	6	7	8	9	10

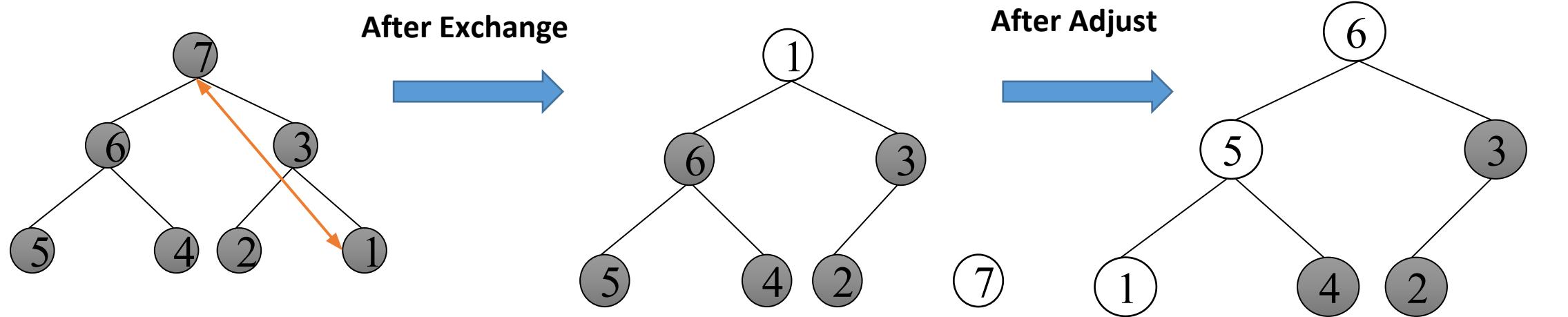
# Max heap Tree Exchange and Adjust: Iteration

3



# Max heap Tree Exchange and Adjust: Iteration

4

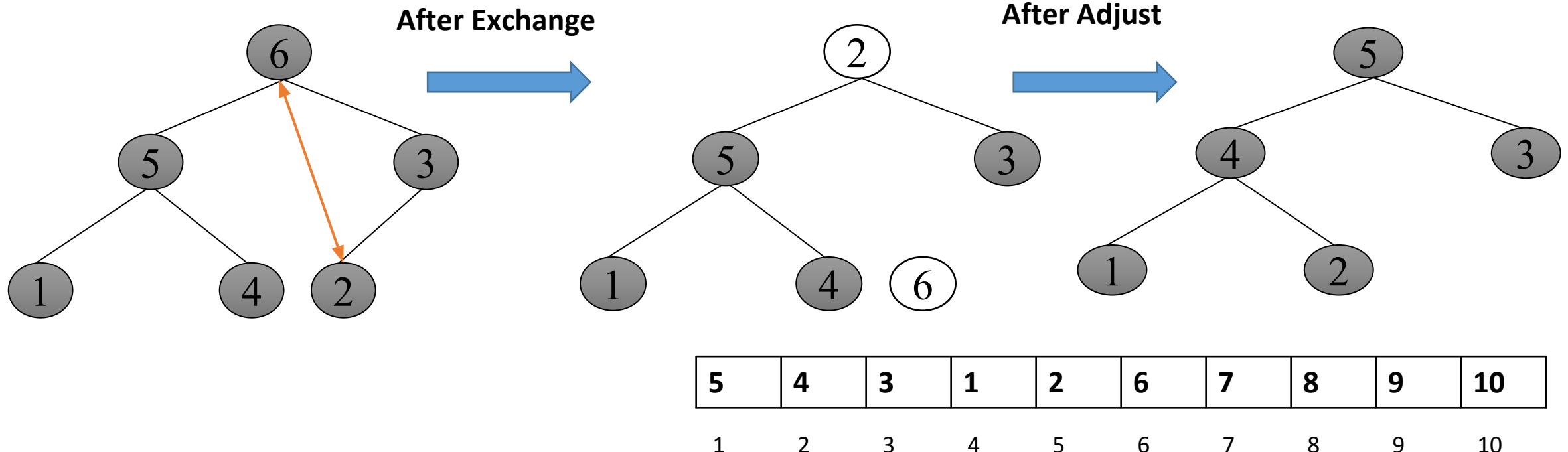


6	5	3	1	4	2	7	8	9	10
---	---	---	---	---	---	---	---	---	----

1 2 3 4 5 6 7 8 9 10

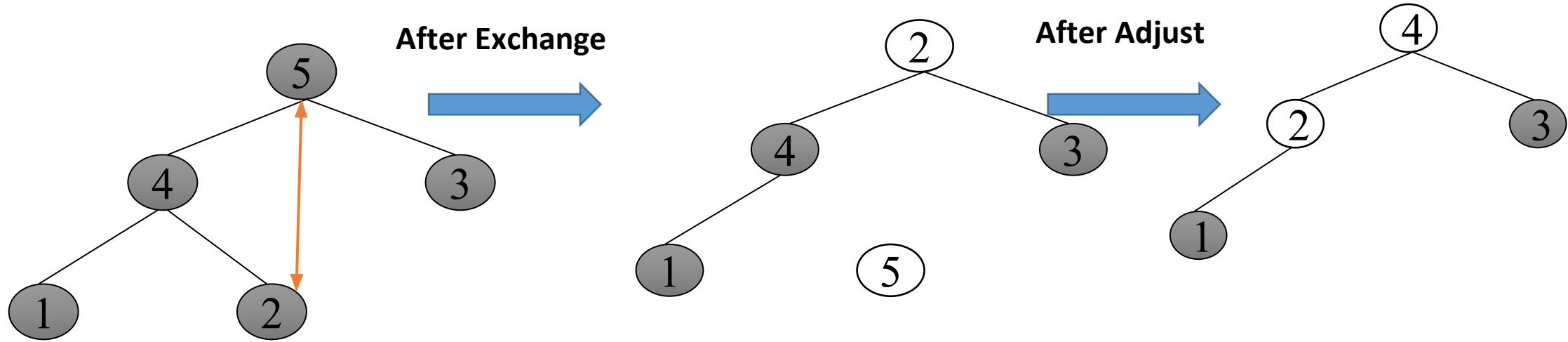
# Max heap Tree Exchange and Adjust: Iteration

5



# Max heap Tree Exchange and Adjust: Iteration

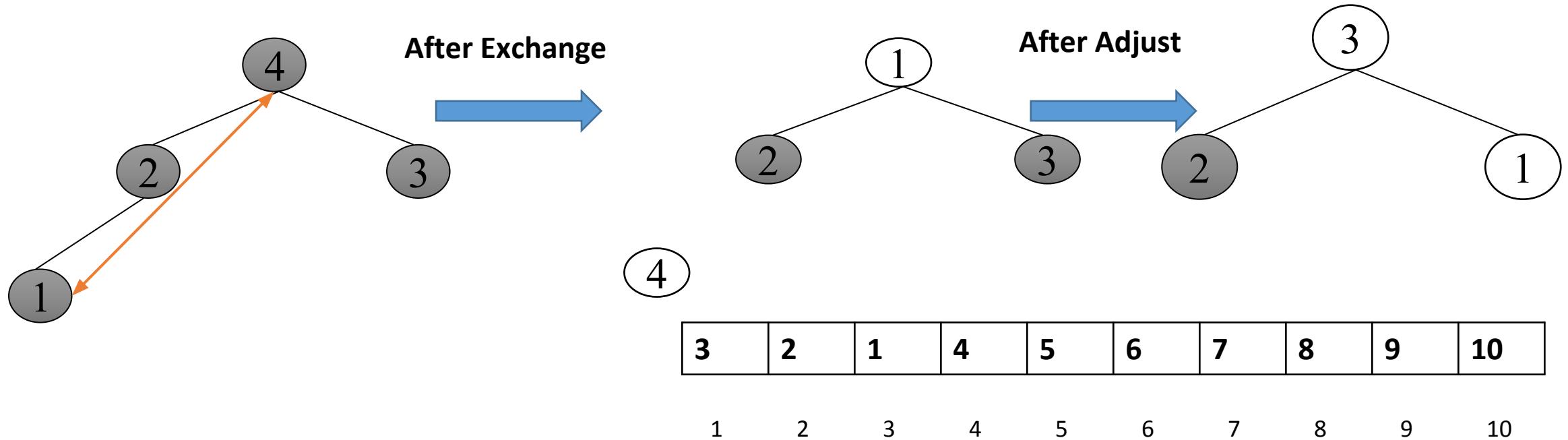
6



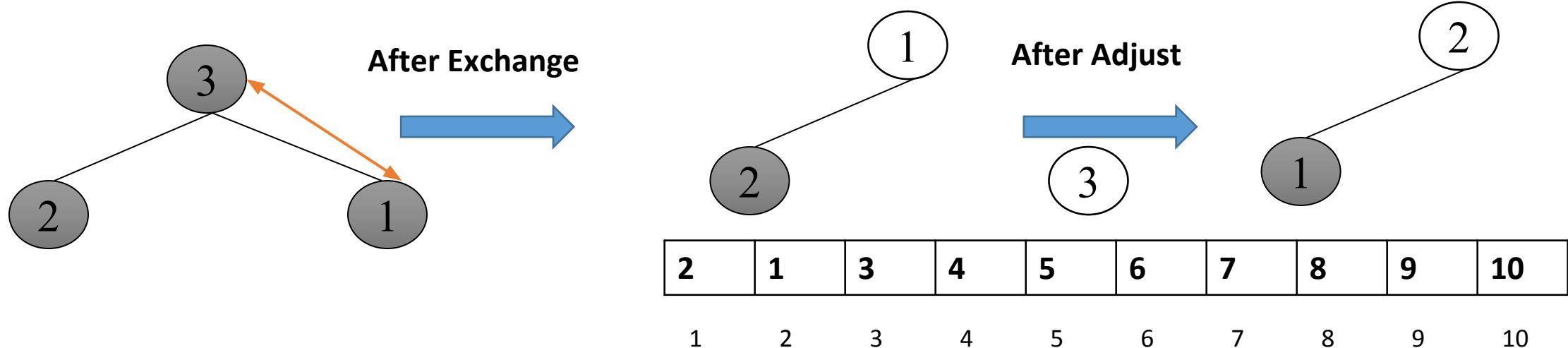
4	2	3	1	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

# Max heap Tree Exchange and Adjust:Iteration

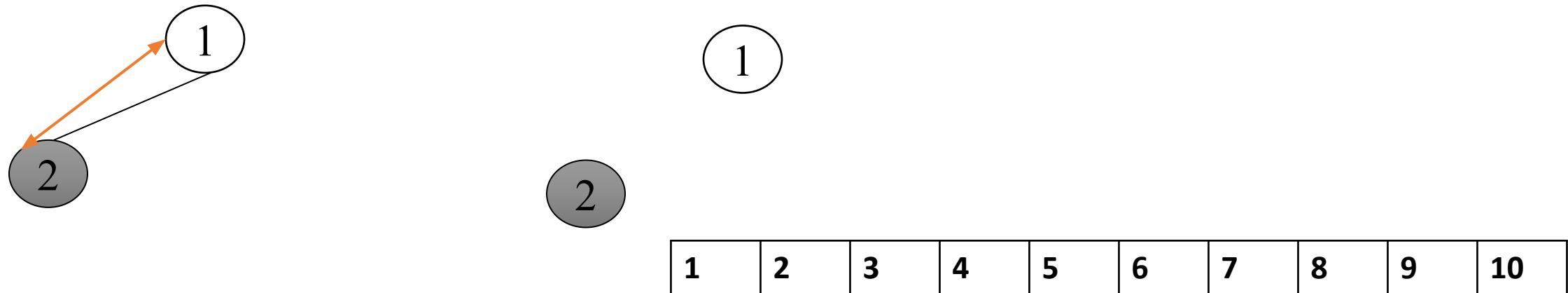
6



## Max heap Tree Exchange and Adjust : Iteration 7



## Max heap Tree Exchange and Adjust : Iteration 8



# Pattern Matching Algorithms

1. Brute Force algorithm
2. Boyer-Moore algorithm1
3. Knuth-Morris Pratt algorithm

# Notations

- T refers to the main string or Text
- P refers to pattern
- n refers to length of main string
- m refers to length of pattern

## Brute –force pattern matching

- The brute-force pattern matching algorithm compares the pattern  $P$  with the text  $T$  for each possible shift of  $P$  relative to  $T$ , until either a match is found, or all placements of the pattern have been tried
- Brute-force pattern matching runs in time  $O(nm)$
- Example of worst case:
- $T = aaa \dots ah$
- $P = aaah$

# Brute-Force Pattern Matching

Y O U M A K E Y O U R o w n D E S T I N Y

This is a watermark for trial version, register to get full one!

Y  
O  
U  
R  
Y  
O  
U  
R

Benefits for registered user:

Y 10 Can remove all trial watermark.

2. No trial watermark on the output documents.

Y  
O  
U  
R  
Y  
O  
U  
R

Remove it Now

Y  
O  
U  
R

Y  
O  
U  
R

Y  
O  
U  
R

Y  
O  
U  
R

## Brute –Force Plan for algorithm

Main String

T

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

Pattern

P

0	1	2	3
0	1	2	3

T length is n

P length is m

i/k	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5

- Diff=0
- Diff=1
- Diff=2
- Diff=3
- Diff=4
- Diff=5
- Diff=6

0	1	2	3
0	1	2	3

0	2	3
0	2	3

If (  $P[k] == T[i + k]$  )  
for all m characters of p

## Algorithm for BruteForce Pattern Matching

**Algorithm** *BruteForceMatch*( $T, P$ )

**Input** text  $T$  of size  $n$  and pattern  $P$  of size  $m$

**Output** starting index of a substring of  $T$  equal to  $P$  or  $-1$  if no such substring exists

**for**  $i \leftarrow 0$  **to**  $n - m$

$j \leftarrow 0$

**while**  $j < m \wedge T[i + j] = P[j]$

$j \leftarrow j + 1$

**if**  $j = m$

**return**  $i$  {match at  $i$ }

**else**

**break** while loop {mismatch}

**return**  $-1$  {no match anywhere}

## One possible Java code Brute Force

```
Class Brute_Force--_Pattern_Match{
    Int brute_Force(String T , String P){
        Int n=T.length();
        Int m=P.length();
        For(i=0;i<=n-m;i++){
            k=0;
            while(k<m && T.charAt(i+k) == P.chatAt(k))
                k++;
            if(k==m) return i; } //for close
        return -1; } // FUNCTION CLOSE
    public static void main(String[] args){
        Scanner s1=new Scanner(System.in);
        Scanner s2=new Scanner(System.in);
        String T,P;
        System.out.println("Enter main and substring");
        T=s1.nextLine(); P=s1.next();
        Brute_Force--_Pattern_Match B=new Brute_Force--_Pattern_Match();
        Int pos= B.brute_Force(String T , String P);
        If(pos==-1)
            System.out.println("Pattern not found");
        Else
            System.out.println("Pattern found at"+ pos +"position");}//min
    close}//class close
```

# Boyer-Moore Heuristics

- The Boyer-Moore's pattern matching algorithm is based on two heuristics

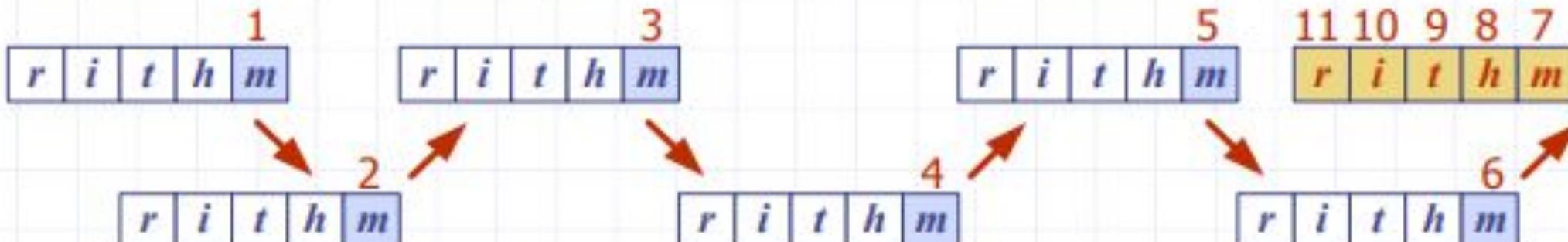
**Looking-glass heuristic:** Compare  $P$  with a subsequence of  $T$  moving backwards

**Character-jump heuristic:** When a mismatch occurs at  $T[i] = c$

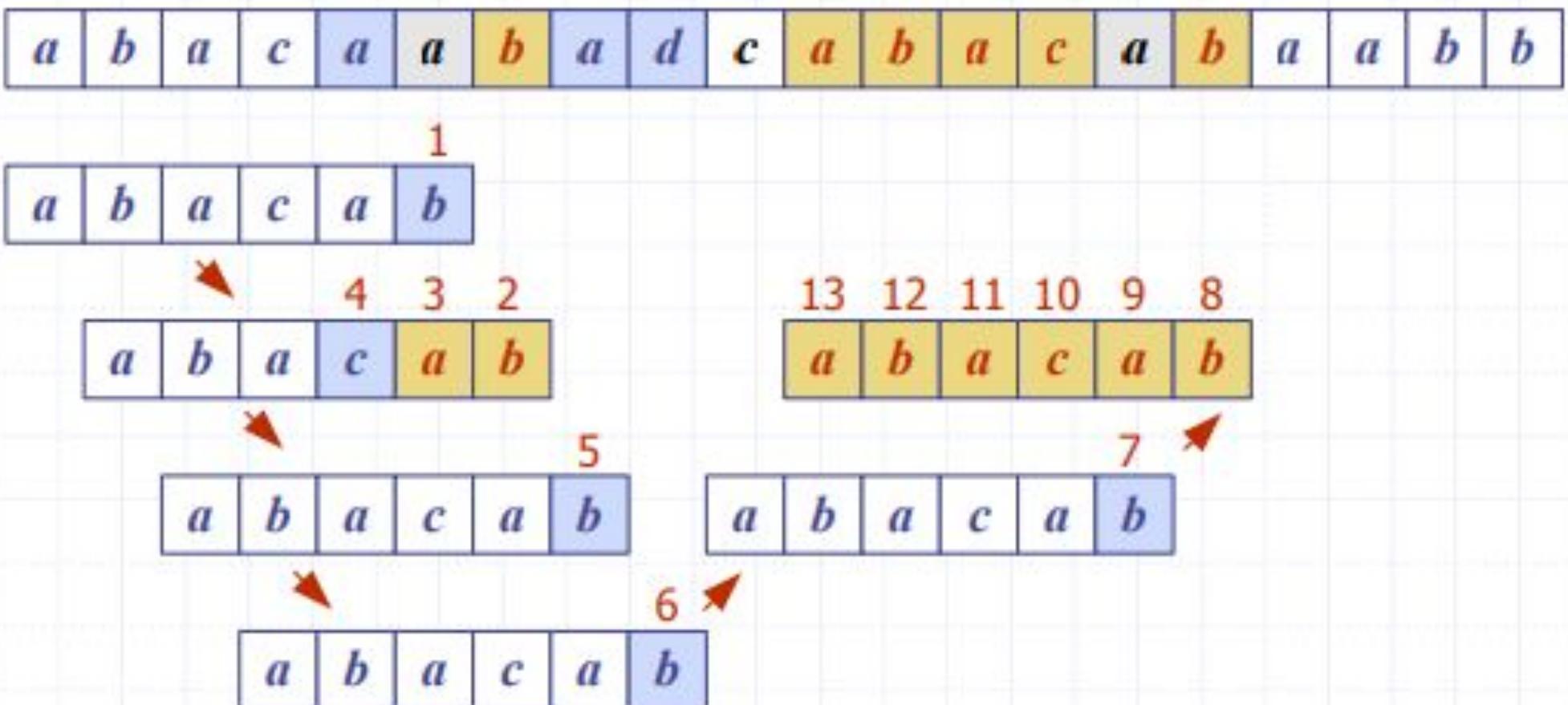
- If  $P$  contains  $c$ , shift  $P$  to align the last occurrence of  $c$  in  $P$  with  $T[i]$
- Else, shift  $P$  to align  $P[0]$  with  $T[i + 1]$

- Example

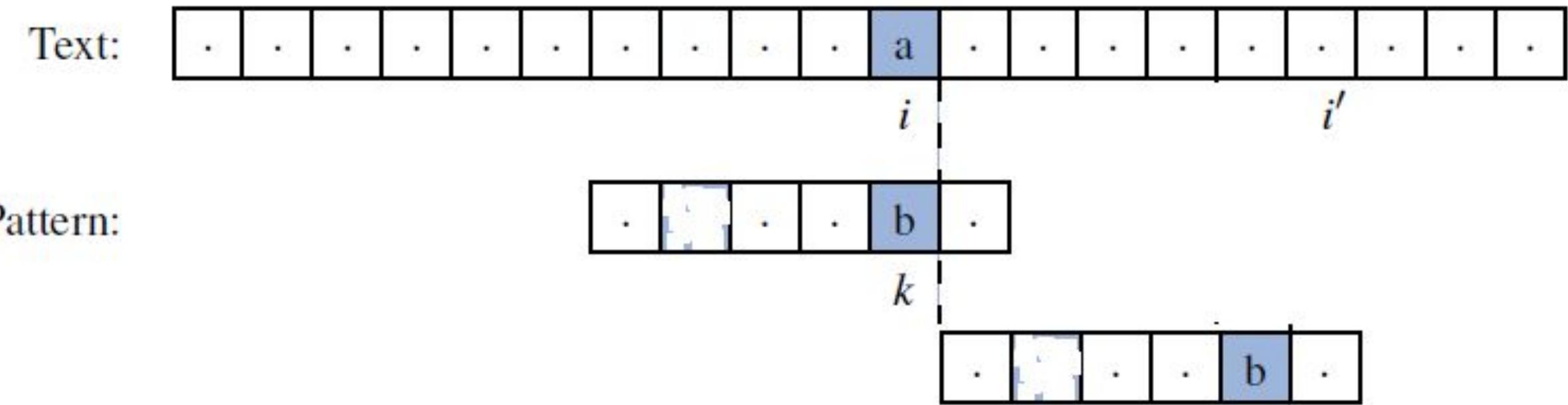
a	p	a	t	t	e	r	n	m	a	t	c	h	i	n	g	a	l	g	o	r	i	t	h	m
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



# Example

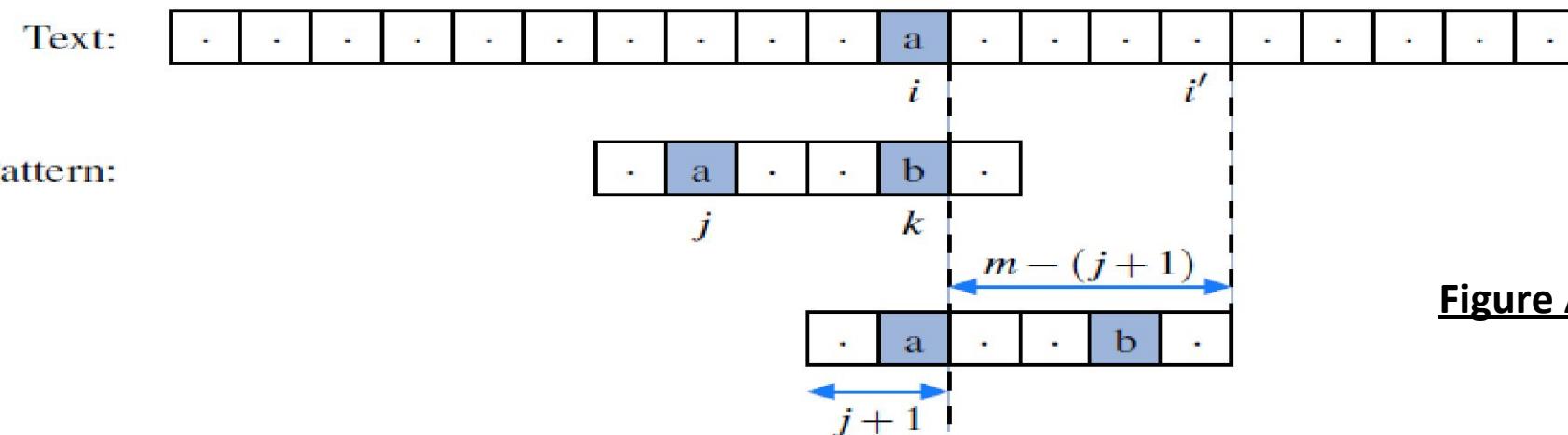


In Boyer-Moore , if a mismatch is found, and the mismatched character of the text does not occur in the pattern, we shift the entire pattern beyond that location, as originally illustrated in the following Figure

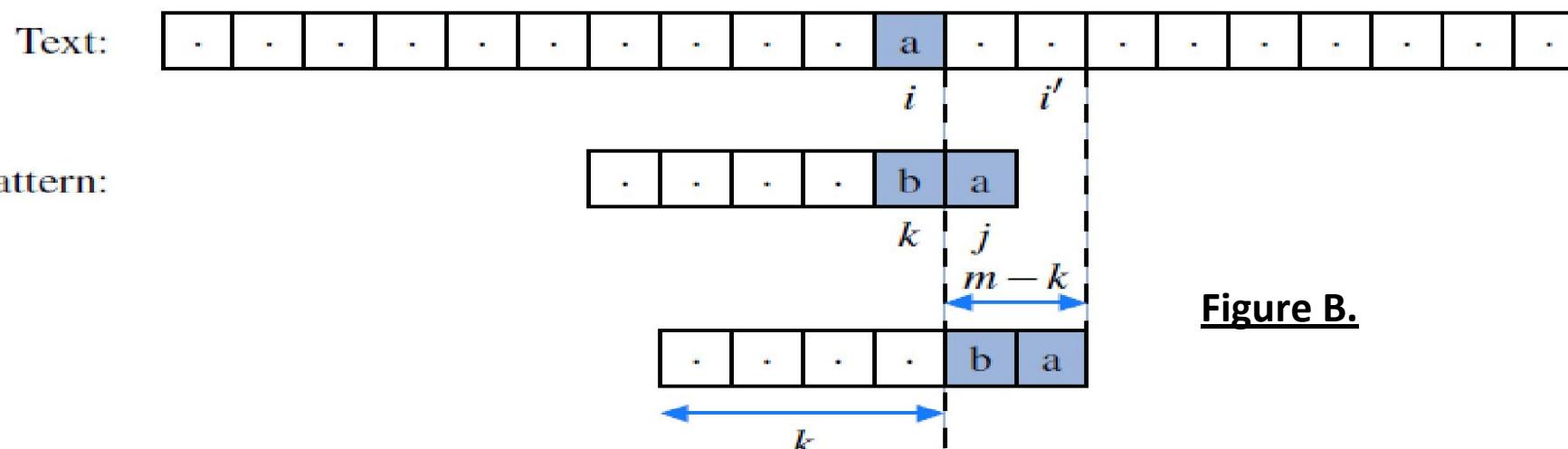


→ Here, The pattern has to be shifted by  $k-j$  positions. Implies  $i$ (index of main string) is to be incremented by  $m$  and then  $k$ (index of substring) is to be set to  $m-1$ .

In Boyer-Moore , If the mismatched character occurs elsewhere in the pattern, we must consider two possible subcases depending on whether its last occurrence is before or after the character of the pattern that was mismatched. Those two cases are illustrated in **Figure**



**Figure A.**



**Figure B.**

**Figure .** Additional rules for the character-jump heuristic of the Boyer-Moore algorithm. We let  $i$  represent the index of the mismatched character in the text,  $k$  represent the corresponding index in the pattern, and  $j$  represent the index of the last occurrence of  $\text{text}[i]$  within the pattern. We distinguish two cases: (a)  $j < k$ , in which case we shift the pattern by  $k - j$  units, and thus, index  $i$  advances by  $m - (j + 1)$  units; (b)  $j > k$ , in which case we shift the pattern by one unit, and index  $i$  advances by  $m - k$  units.

The efficiency of the Boyer-Moore algorithm relies on quickly determining where a mismatched character occurs elsewhere in the pattern. Last-Occurrence function is used for that.

## Last-Occurrence Function

- ◆ Boyer-Moore's algorithm preprocesses the pattern  $P$  and the alphabet  $\Sigma$  to build the last-occurrence function  $L$  mapping  $\Sigma$  to integers, where  $L(c)$  is defined as

- the largest index  $i$  such that  $P[i] = c$  or
- $-1$  if no such index exists

- ◆ Example:

- $\Sigma = \{a, b, c, d\}$
- $P = abacab$

Table:Last Occurrence Table for the previous example

$c$	$a$	$b$	$c$	$d$
$L(c)$	4	5	3	-1

- ◆ The last-occurrence function can be computed in time  $O(m + s)$ , where  $m$  is the size of  $P$  and  $s$  is the size of  $\Sigma$

We prefer to use a hash table to represent the last function. One possible implementation for LastOccurrence is as follows. Here Last is the HashMap with character as the key and last Occurrence as the value

```
void LastOccurrence(char[] T, char[] P)
{ for(int i=0;i<n;i++)
    Last.put(T[i], -1);
 for(int k=0;k<m;k++)
    Last.put(P[k], k);
 // System.out.println(Last);
}
```

**Example:**

Text	a   b   a   c   a   a   b   a   d   c   a   b   a   c   a   b   a   a   b   b
Pattern	a   b   a   c   a   b

Last() function result for the above example

Character	Last(Character)
a	4
b	5
c	3
d	-1

## One possible Java code for Boyer Moore

```
// One possible code Boyer Moore
import java.util.HashMap;
import java.util.Scanner;
class B_M_Pattern_Match{
int m,n;
HashMap<Character , Integer> Last;
B_M_Pattern_Match(){
    Last=new HashMap<Character, Integer>();
}
void LastOccurrence(char[] T, char[] P)
{ for(int i=0;i<n;i++)
    Last.put(T[i], -1);
for(int k=0;k<m;k++)
    Last.put(P[k], k);
// System.out.println(Last);
}
```

```
int bm(char[] T , char[] P)
{ int j,i,k;
n=T.length; m=P.length; i=m-1;k=m-1;
// System.out.println( n + " " +m);
LastOccurrence(T, P);
while(i<n){
    if(T[i]==P[k]) {
        if(k==0) return i;
        k--; i--;
    }
    else{ j=Last.get(T[i]);
        if(j==-1)
            i=i+m;
        else if(j<k) i=i+m-(j+1);
        else i=i+m-k;
        k=m-1;
    } } //while close
return -1;
}//function
```

```
public static void main(String[] args){  
    Scanner s1=new Scanner(System.in);  
    Scanner s2=new Scanner(System.in);  
    String T,P;  
    System.out.println("Enter main and substring");  
    T=s1.nextLine();  
    P=s1.next();  
    B_M_Pattern_Match Bm=new B_M_Pattern_Match();  
    int pos= Bm.bm(T.toCharArray(),P.toCharArray());  
    if(pos==-1)  
        System.out.println("Pattern not found");  
    else  
        System.out.println("Pattern found at"+ pos +"position");  
}//main close  
}//class close
```

## Performance of Boyer Moore

If using a traditional lookup table, the worst-case running time of the Boyer-Moore algorithm is  $O(nm + |S|)$ . The computation of the last function takes  $O(m + |S|)$  time. The actual search for the pattern takes  $O(nm)$  time in the worst case—the same as the brute-force algorithm. An example that achieves the worst case for Boyer-Moore is

text {aaaaaa... a}  
pattern = {baa...a}

The worst-case performance, however, is unlikely to be achieved for English text; in that case, the Boyer-Moore algorithm is often able to skip large portions of text. Experimental evidence on English text shows that the average number of comparisons done per character is 0.24 for a five-character pattern string.

# Knuth-Morris-Pratt pattern matching

Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, **but shifts** the pattern more intelligently than the brute-force algorithm.

When a mismatch occurs, it **shift** the largest prefix which is also the suffix of the sub pattern scanned till that position. The prefix which is also the suffix is called border.

# Components of KMP algorithm

## 1. Failure function

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself. The **failure function  $F(j)$**  is defined as the size of largest border of sub pattern  $P[0..j]$ .

Border length for various sub pattern is given in the next slide.

## 2. The KMP Matcher

With string 'S', pattern 'p' and Failure function F " as inputs, finds the occurrence of 'p' in 'S' and returns the position of pattern in string.

For instance consider following Substring or Pattern.

- ababaca
- Various sub patterns and their possible suffixes, prefixes and borders and their lengths are given in the following table.

Sub pattern	Possible Prefixes	Possible Suffixes	borders	Border length
a	NIL	NIL	NIL	0
ab	a	b	NIL	0
aba	a , ab	a , ba	a	1
abab	a . ab , aba	b , ab , bab	ab	2
ababa	a , ab , aba , abab	a , ba , aba , baba	aba	3
ababac	a , ab , aba , abab, ababa	c , ac , bac , abac ,babac,	NIL	0
ababaca	a , ab , aba , abab, ababa, ababac,	a, ca, aca, baca , abaca ,babaca,	a	1

Border length is nothing but number of character of pattern is to be skipped from backtracking. Failure function is used to calculate this.Which is given in the next slide

# Failure function implementation

- Void Failurefunction(char P[],int F[])

```
{i = 1 ;
F[0] = 0; j = 0;
while ( i < m )
{ if ( P[j] == P[i] )
  { F[i] = j+1;
    i++;j++;
  }
  else //P[j] != P[i]
  { if ( j > 0 )
    j = F[j-1]
    else // j == 0
    { F[i] = 0
      i++; // redundant, just to make it clear what we are looping with
    }
  }
}//WHILE CLOSE
}// FUNCTION CLOSE
```

# KMP MATCHER

- Start at LHS of string, string[0], trying to match pattern, working right.

Trying to match  $\text{string}[i] == \text{pattern}[j]$ . Given a search pattern, failure table,  $F[j]$ , showing, when there is a mismatch at pattern position  $j$ , where to reset  $j$  to. If match fails, keep  $i$  same, reset  $j$  to position  $F[j-1]$ .

- void KMPMatch(char S[], char P[])// S indicates main string and P indicates pattern**

```
{ int F[100];
FailureFunction(P, F);
i = 0;
j = 0;
while i < n
{if T[i] = P[j]
    if j = m - 1
        return i - j //match
    else{
        i = i + 1;
        j = j + 1;
    }
else  if j > 0
    j = F[j - 1];
else  i = i + 1;
}
return -1 // no match
}
```

- At each iteration of the whileloop, either *i increases by one, or* the shift amount  $i - j$  increases by at least one
- Hence, there are no more than *2n iterations of the* while-loop .Thus, KMP's algorithm runs in optimal time  $O(m + n)$ .

*Example is given in the next slides*

Example: compute F for the pattern 'p' below:

p	a	b	a	b	a	c	a
---	---	---	---	---	---	---	---

Initially:  $m = \text{length}[p] = 7$

$$F[0] = 0$$

$$j = 0$$

Step 1:  $i = 1, j=0$ , since  $P[i] \neq P[j]$

$$F[1] = 0$$

I	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0					

Step 2:  $i = 2, j= 0$ , since  $P[i]=P[j]$

$$F[2] = 1$$

i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
f	0	0	1				

Step 3:  $i = 3, j = 1$ , since  $P[i]=P[j]$

$$F[3] = 2$$

i	0	1	2	3	4	5	6
p	a	b	a	b	a	c	a
F	0	0	1	2			

Step 4:  $i = 4$ ,  $j = 2$ , since  $P[i] = P[j]$

$$F[4] = 3$$

i	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3		

Step 5:  $i = 5$ ,  $j = 3$ , since  $P[i] \neq P[j]$

$$j = F[j-1] = F[2] = 1$$

$i = 5$ ,  $j = 1$ , since  $P[i] \neq P[j]$

$$j = F[j-1] = F[0] = 0$$

Therefor  $F[5] = 0$

Step 6:  $i = 6$ ,  $j = 0$

$$F[6] = 1$$

i	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	

After iterating 6 times, the Failure function computation is complete:  
→

i	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

Illustration: given a String ‘S’ and pattern ‘p’ as follows:

S

b	a	c	b	a	b	a	b	a	c	a	c	a
---	---	---	---	---	---	---	---	---	---	---	---	---

p

a	b	a	b	a	c	a
---	---	---	---	---	---	---

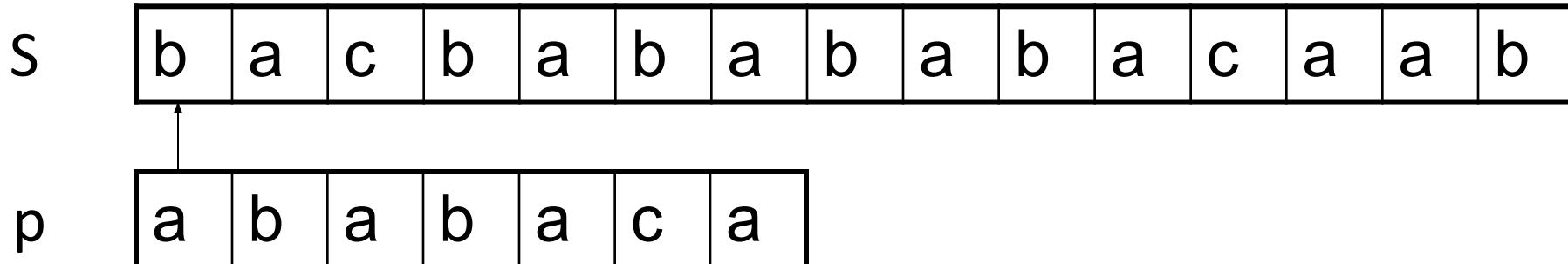
Let us execute the KMP algorithm to find whether  
‘p’ occurs in ‘S’.

*For ‘p’ the prefix function, F was computed previously and is as follows:*

i	0	1	2	3	4	5	6
P	a	b	A	b	a	c	a
F	0	0	1	2	3	0	1

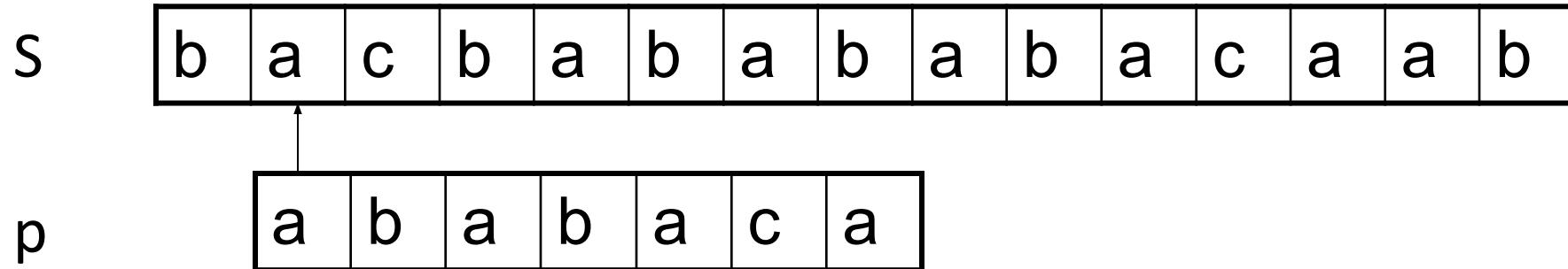
Initially:  $n = \text{size of } S = 15$ ;  
 $m = \text{size of } p = 7$

Step 1:  $i = 0, j = 0$   
comparing  $p[0]$  with  $S[0]$



$P[0]$  does not match with  $S[0]$ . 'p' will be shifted one position to the right.

Step 2:  $i = 1, j = 0$   
comparing  $p[0]$  with  $S[1]$

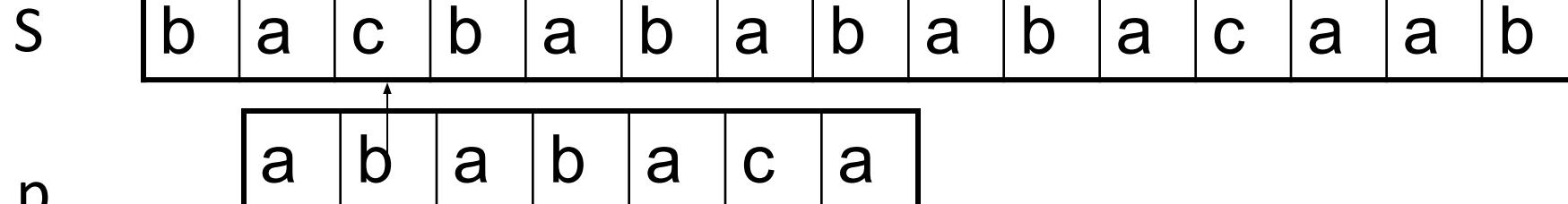


$P[0]$  matches  $S[1]$ . Since there is a match, p is not shifted.

Step 3:  $i = 2, j = 1$

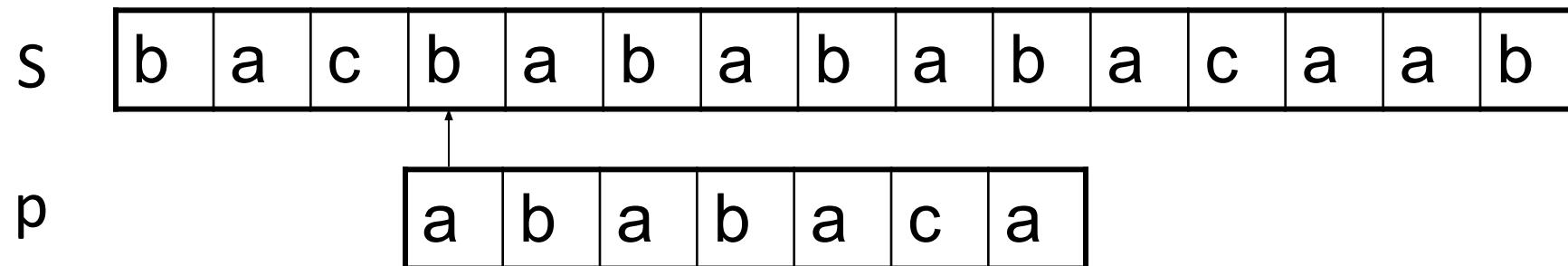
Comparing  $p[1]$  with  $S[2]$

$p[1]$  does not match with  $S[2]$



Backtracking on p set  $j=F[j-1]=F[0]=0$ , comparing  $p[0]$  and  $S[2]$  since does not match and  $j=0$  set  $i=i+1$

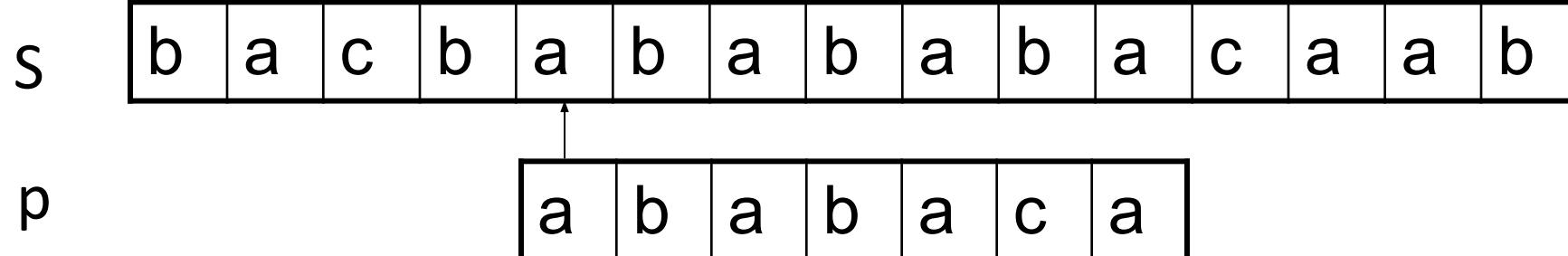
Step 4:  $i = 3, j = 0$  comparing  $p[0]$  with  $S[3]$      $p[0]$  does not match with  $S[3]$  since  $j=0$  set  $i=i+1$



Step 5:  $i = 4, j = 0$

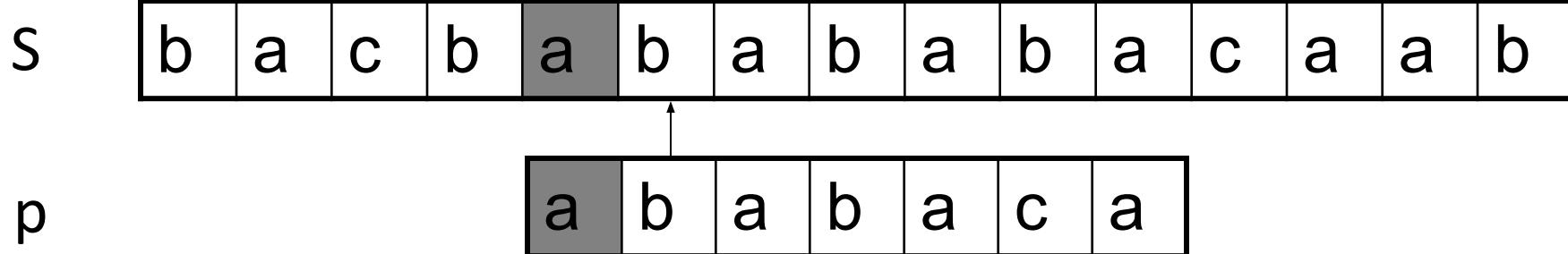
comparing  $p[0]$  with  $S[4]$

$p[0]$  matches with  $S[4]$  so set  $i=i+1, j=j+1$



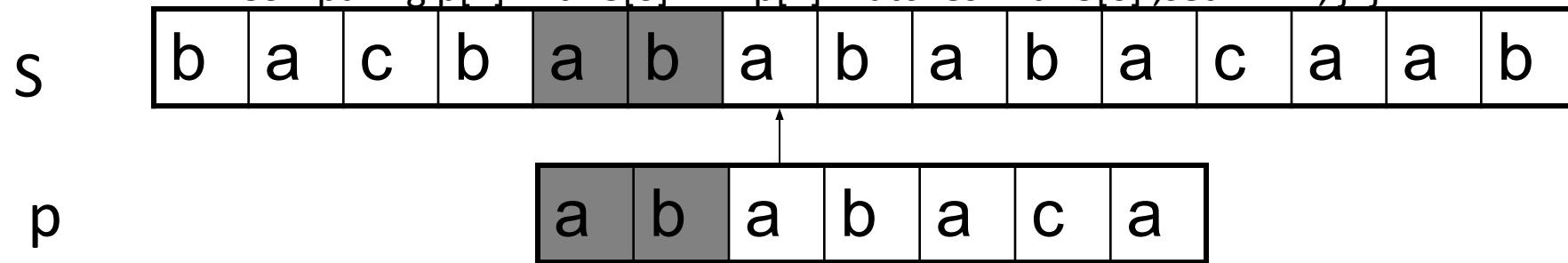
Step 6:  $i = 5$ ,  $j = 1$

Comparing  $p[1]$  with  $S[5]$   $p[1]$  matches with  $S[5]$ , set  $i=i+1$   $j=j+1$



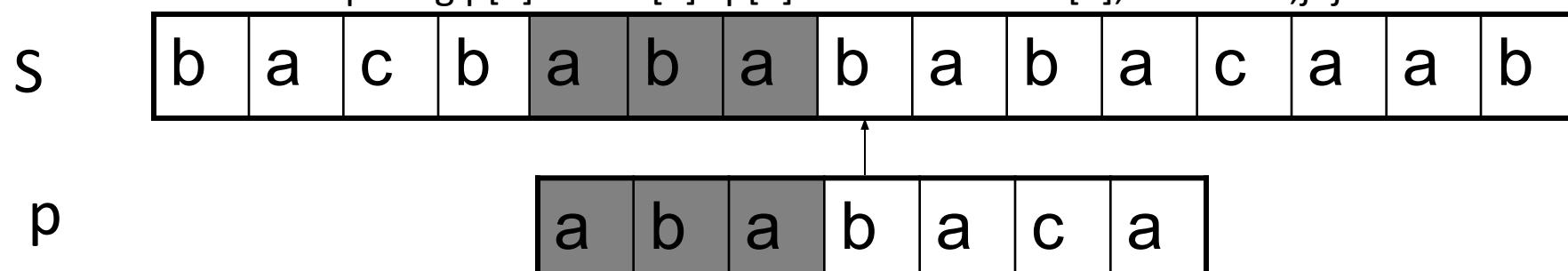
Step 7:  $i = 6$   $j = 2$

Comparing  $p[2]$  with  $S[6]$   $p[2]$  matches with  $S[6]$ , set  $i=i+1$  ,  $j=j+1$



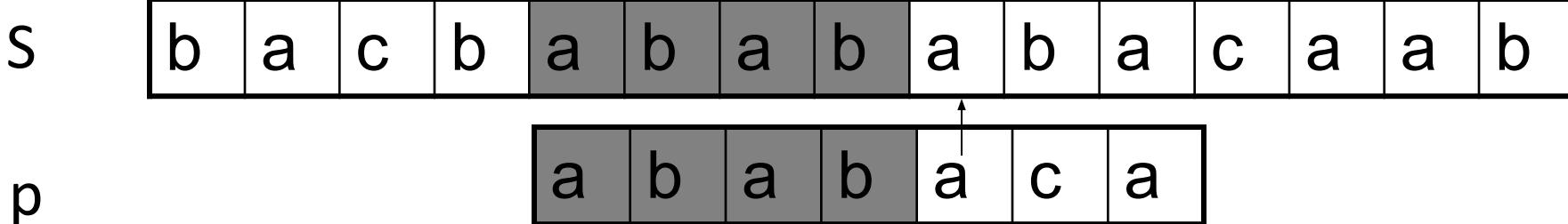
Step 8:  $i = 7$   $j = 3$

Comparing  $p[3]$  with  $S[7]$   $p[3]$  matches with  $S[7]$ , set  $i=i+1$  , $j=j+1$



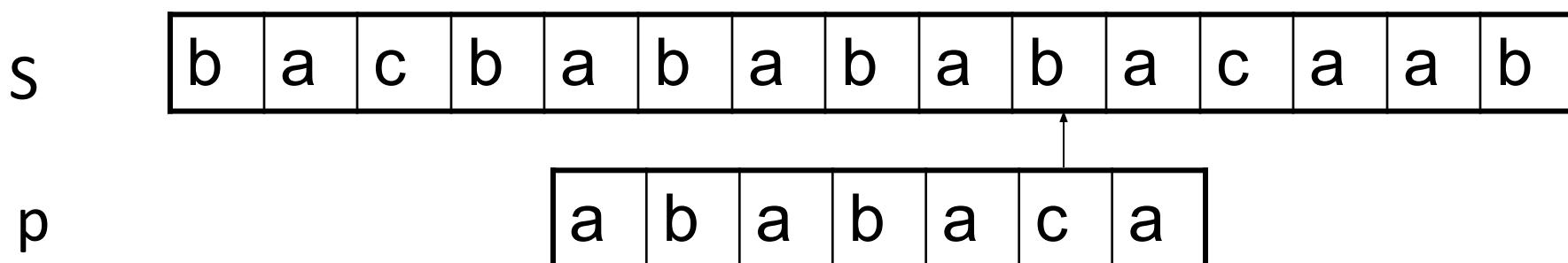
Step 9:  $i = 8, j = 4$

Comparing  $p[4]$  with  $S[8]$   $p[4]$  matches with  $S[8]$ , set  $i=i+1, j=j+1$



Step 10:  $i = 9, j = 5$

Comparing  $p[5]$  with  $S[9]$   $p[5]$  doesn't match with  $S[9]$  set  $j=F[j-1]=F[4]=3$ ,



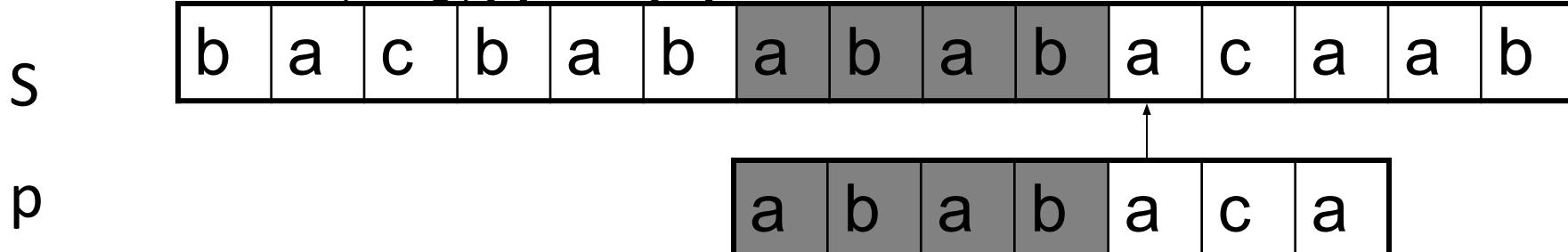
Backtracking on P set  $j=3$ , comparing  $p[3]$  with  $S[9]$  because after mismatch  $j = F[4] = 3$

$P[3]$  matches to  $S[9]$  so set  $i=i-1, j=j+1$

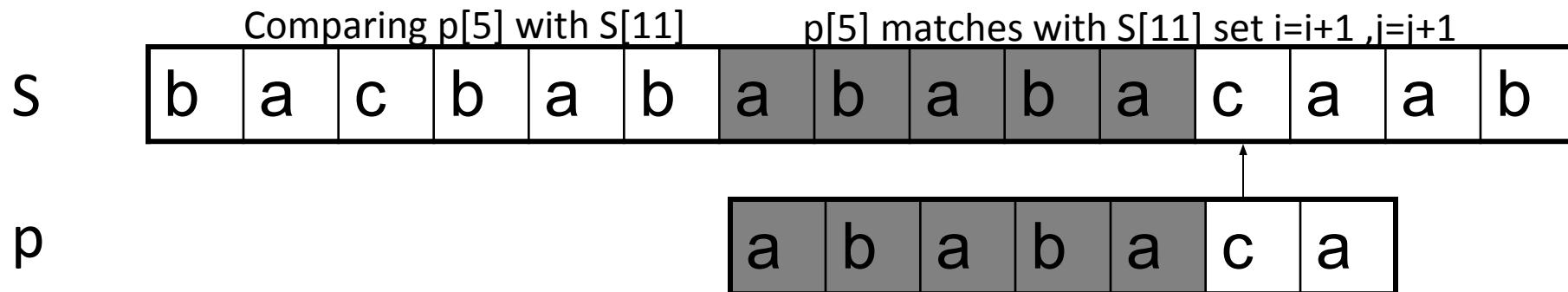
Step 11:  $i = 10, j = 4$

Comparing  $p[4]$  with  $S[10]$

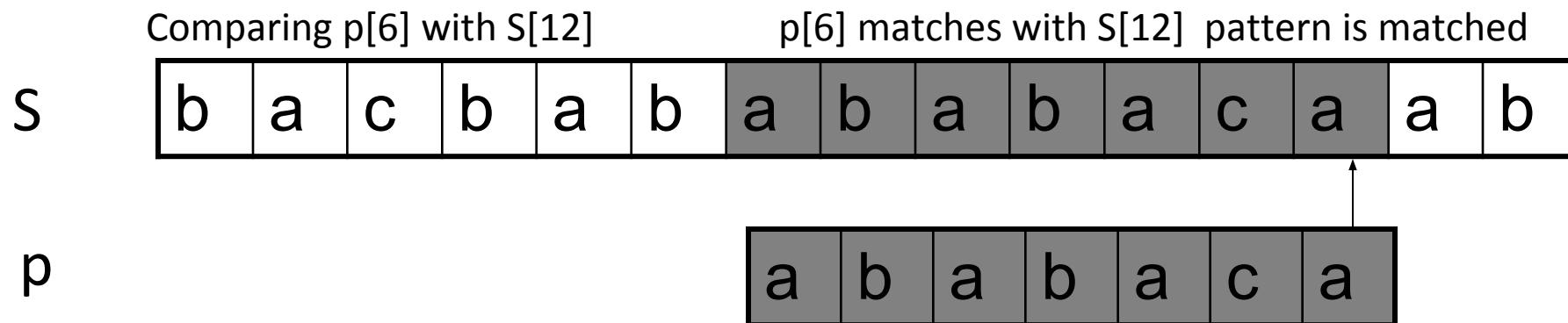
$p[4]$  matches with  $S[10]$  so set  $i=i+1, j=j+1$



Step 12: i = 11, j = 5

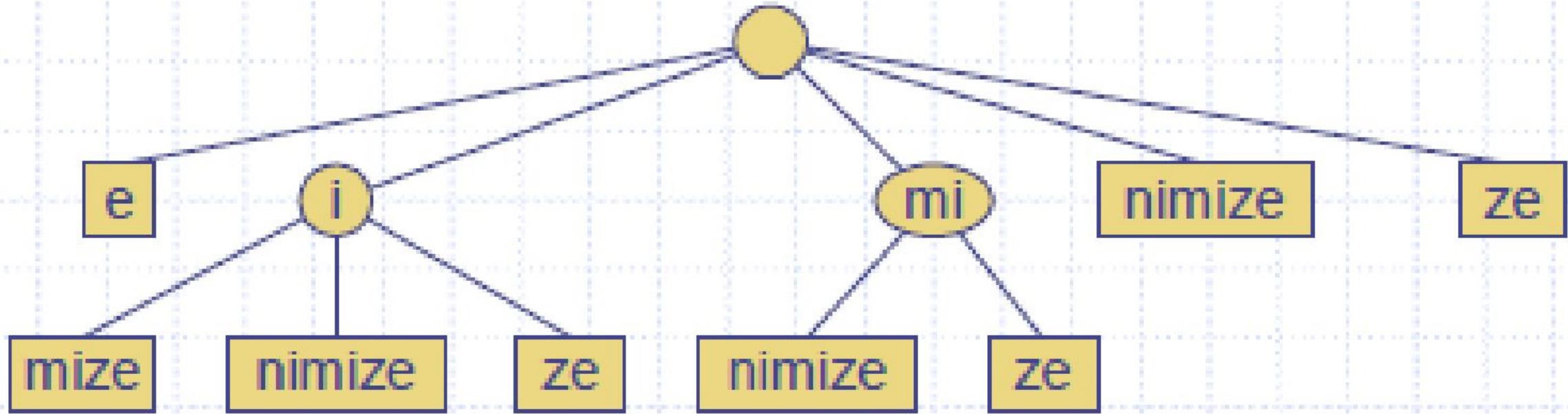


Step 13: i = 12, qj= 6



Pattern 'p' has been found to completely occur in string 'S'. Pattern position is i,j = 12-6 =6

# Tries



A ***trie*** (pronounced “try”) is a tree-based data structure for storing strings in order to support fast pattern matching. The main application for tries is in information retrieval.

This approach is suitable for applications in which many queries are performed on a fixed text, so that the initial cost of preprocessing the text is compensated by a speedup in each subsequent query (for example, a website that offers pattern matching in Shakespeare’s *Hamlet* or a search engine that offers Web pages containing the term *Hamlet*).

we can use a trie to perform a special type of pattern matching, called ***word matching***, where we want to determine whether a given pattern matches one of the words of the text exactly. Word matching differs from standard pattern matching because the pattern cannot match an arbitrary substring of the text—only one of its words.

## Standard Trie

Let  $S$  be a set of  $s$  strings from alphabet  $S$  such that no string in  $S$  is a prefix of another string. A ***standard trie*** for  $S$  is an ordered tree  $T$  with the following properties :

- Each node of  $T$ , except the root, is labeled with a character of  $S$ .
- The children of an internal node of  $T$  have distinct labels.
- $T$  has  $s$  leaves, each associated with a string of  $S$ , such that the concatenation of the labels of the nodes on the path from the root to a leaf  $v$  of  $T$  yields the string of  $S$  associated with  $v$ .

In addition to the above ,a standard trie storing a collection  $S$  of  $s$  strings of total length  $n$  from an alphabet  $S$  has the following properties:

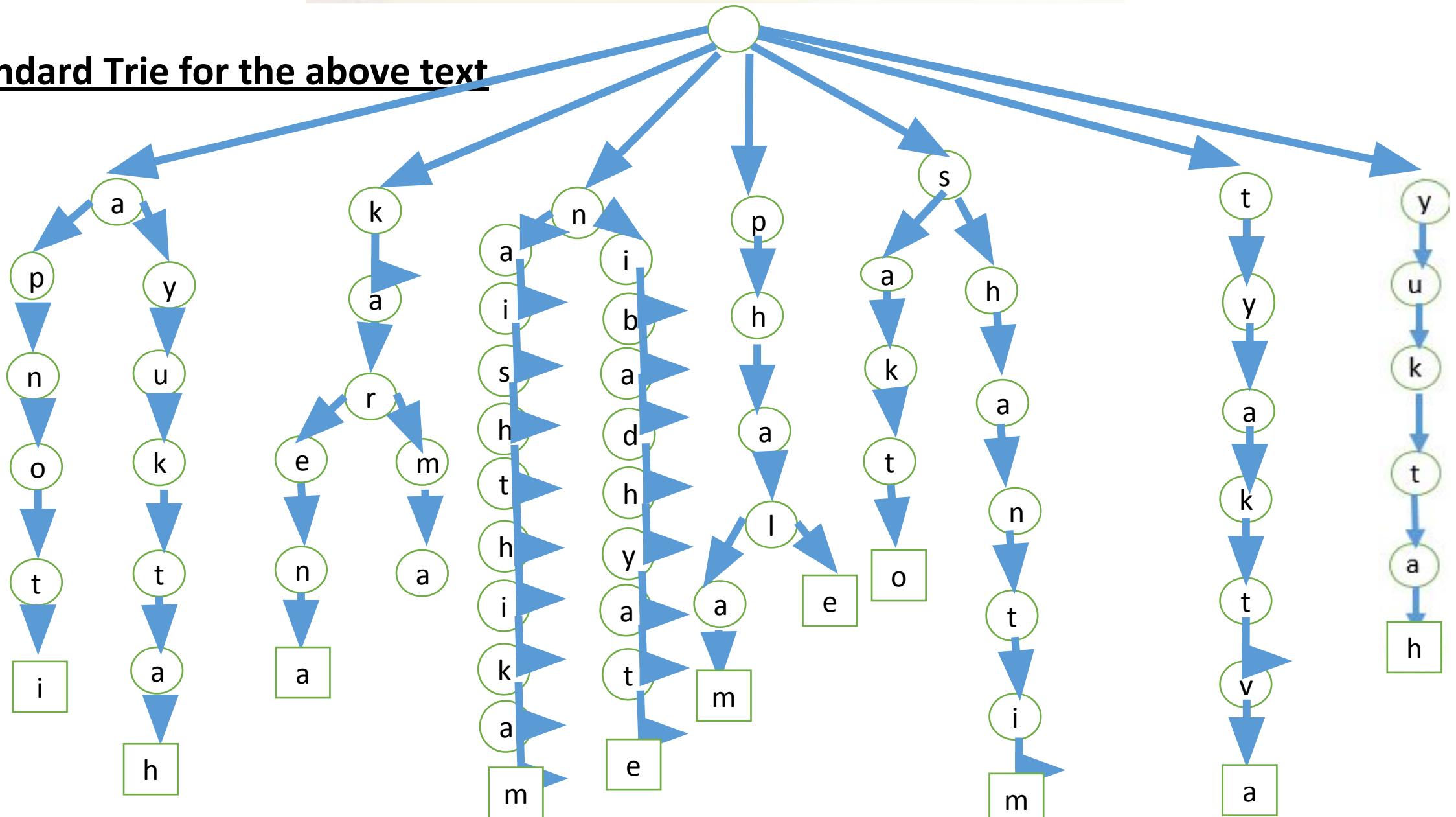
- The height of  $T$  is equal to the length of the longest string in  $S$ .
- Every internal node of  $T$  has at most  $|S|$  children.
- $T$  has  $s$  leaves.
- The number of nodes of  $T$  is at most  $n+1$ .

The worst case for the number of nodes of a trie occurs when no two strings share a common nonempty prefix; that is, except for the root, all internal nodes have one child.

Example 1:

yuktah karma-phalam tyaktvā śhāntim āpnoti naishthikīm  
ayuktah kāma-kāreṇa phale sakto nibadhyate

**Standard Trie for the above text**



## Example 2

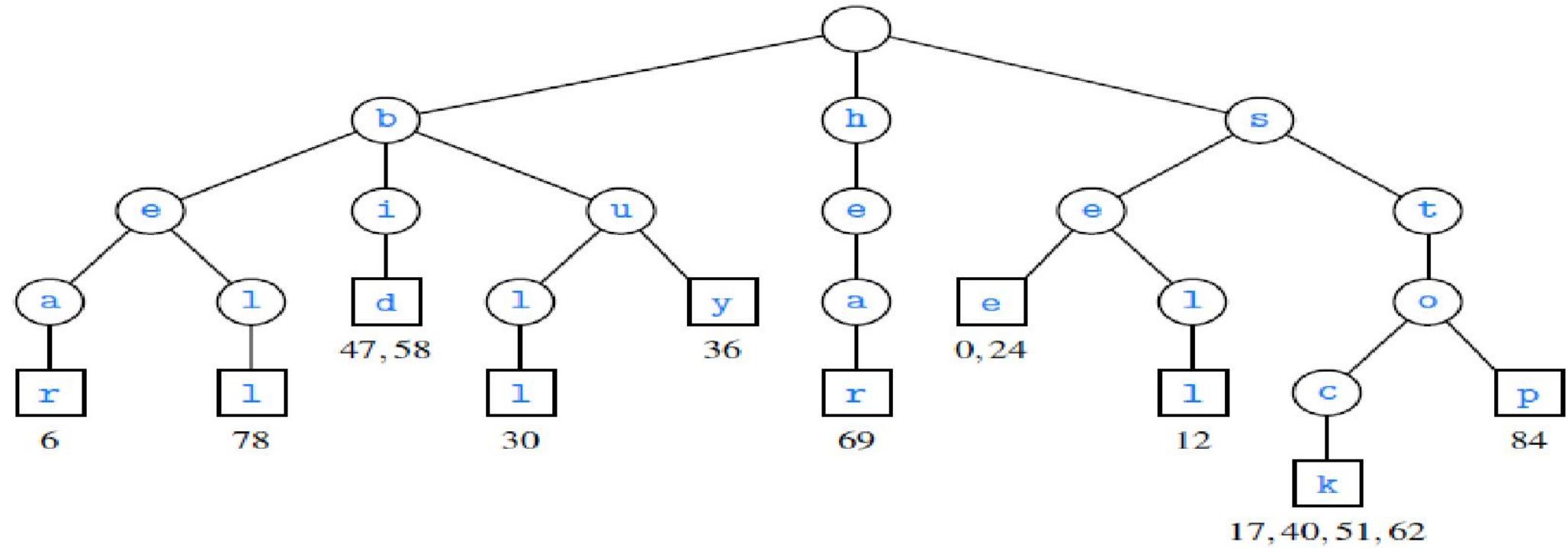
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!
23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
	s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!
46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68
	b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!	
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88			
h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!			

(a)

Figure Word matching with a standard trie:

(a) text (articles and prepositions, which are also known as **stop words**, excluded)

(b) standard trie for the words in the text, with leaves augmented with indications of the index at which the given word begins in the text. For example, the leaf for the word “stock” notes that the word begins at indices 17, 40, 51, and 62 of the text.



(b)

- ◆ A standard trie uses  $O(n)$  space and supports searches, insertions and deletions in time  $O(dm)$ , where:

$n$  total size of the strings in  $S$

$m$  size of the string parameter of the operation

$d$  size of the alphabet

## Compressed Tries

A **compressed trie** is similar to a standard trie but it ensures that each internal node in the trie has at least two children. It enforces this rule by compressing chains of single-child nodes into individual edges. Let  $T$  be a standard trie. We say that an internal node  $v$  of  $T$  is **redundant** if  $v$  has one child and is not the root. For example, the trie of Figure 13.7 has eight redundant nodes. Let us also say that a chain of  $k \geq 2$  edges,  $(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k)$ , is **redundant** if:

- $v_i$  is redundant for  $i = 1, \dots, k-1$ .
- $v_0$  and  $v_k$  are not redundant.

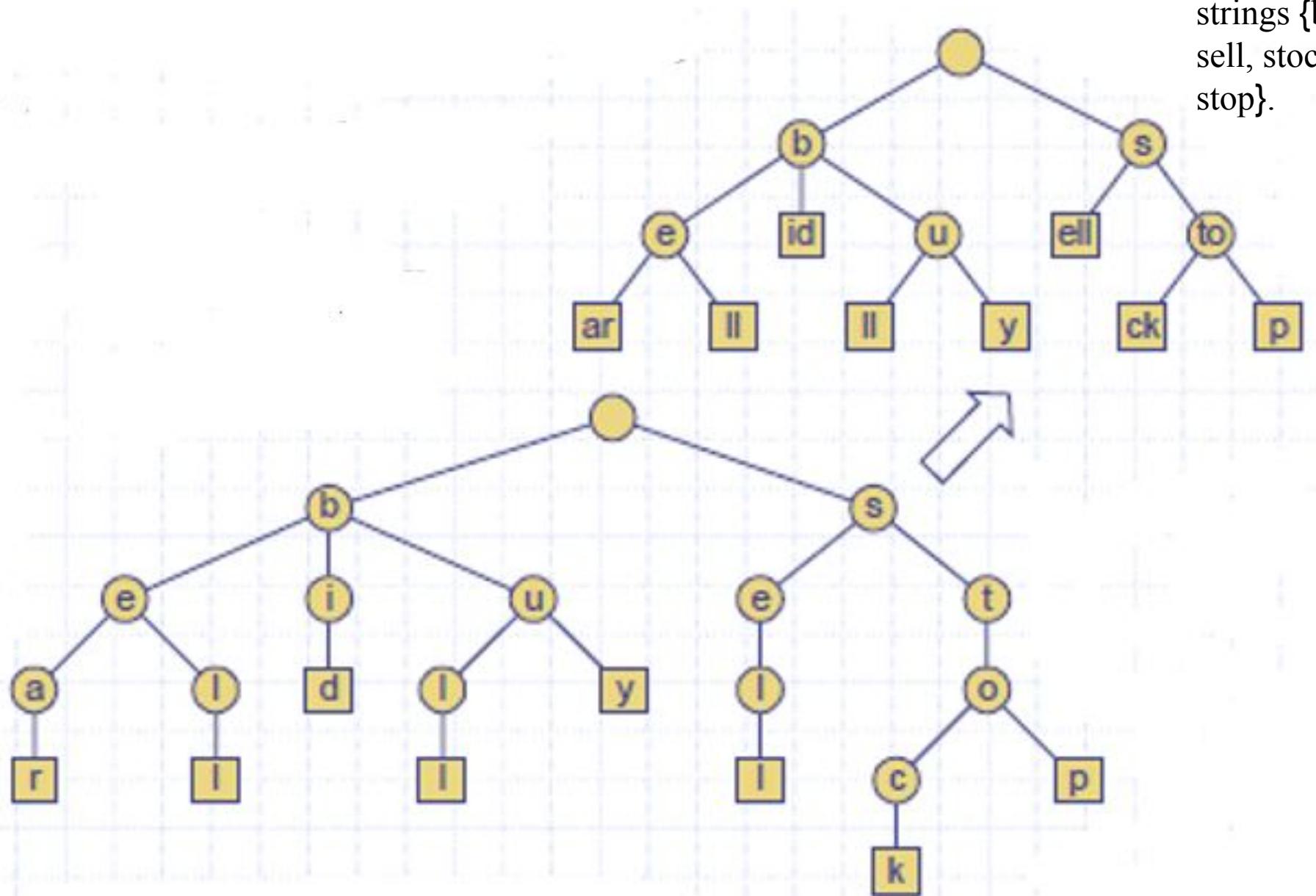
We can transform  $T$  into a compressed trie by replacing each redundant chain  $(v_0, v_1) \cdots (v_{k-1}, v_k)$  of  $k \geq 2$  edges into a single edge  $(v_0, v_k)$ , relabeling  $v_k$  with the concatenation of the labels of nodes  $v_1, \dots, v_k$ .

**compressed trie**, which is also known (for historical reasons) as the **Patricia trie**

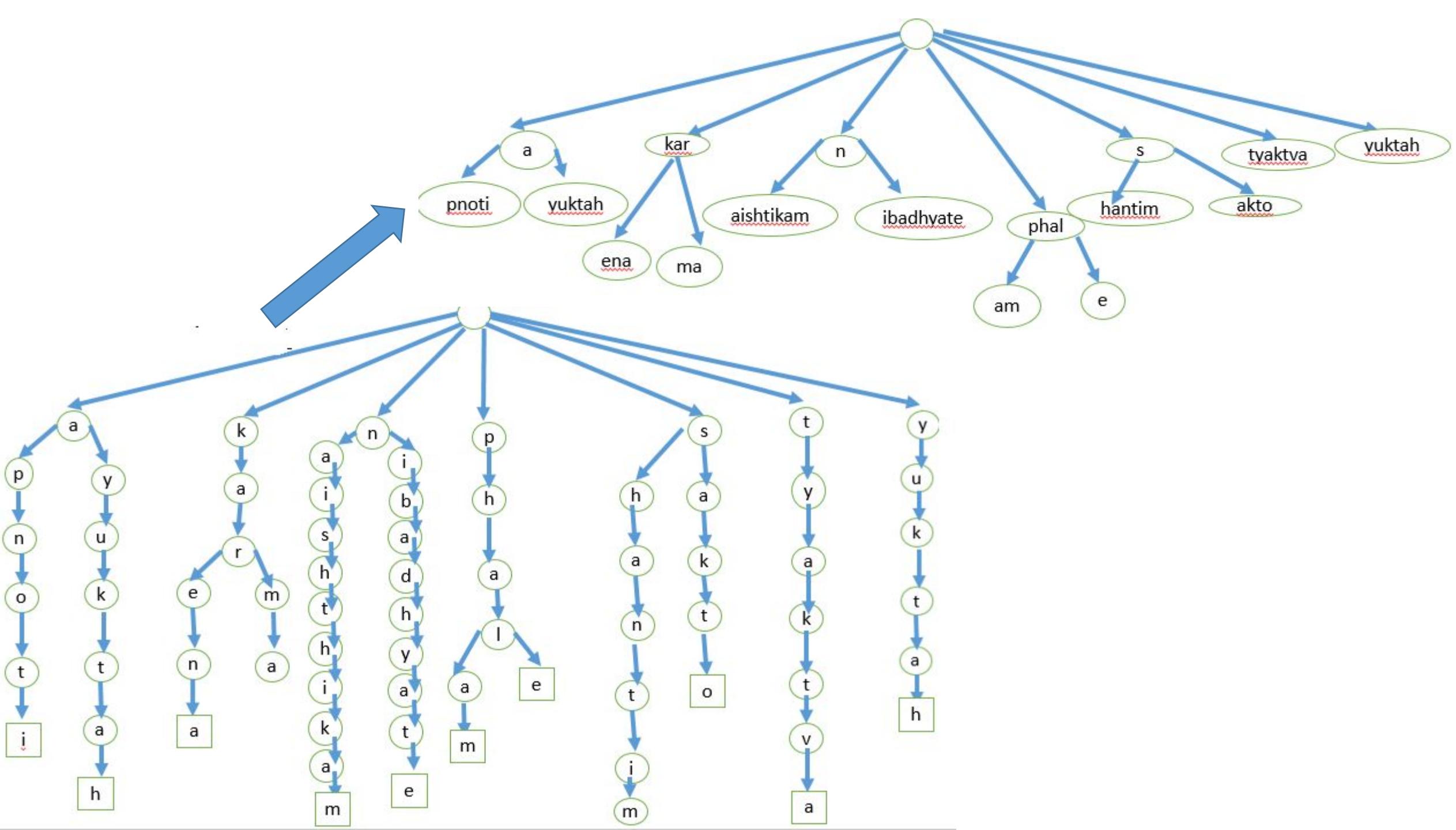
A compressed trie storing a collection  $S$  of  $s$  strings from an alphabet of size  $d$  has the following properties:

- Every internal node of  $T$  has at least two children and most  $d$  children.
- $T$  has  $s$  leaves nodes.
- The number of nodes of  $T$  is  $O(s)$ .

# Compressed Trie Example



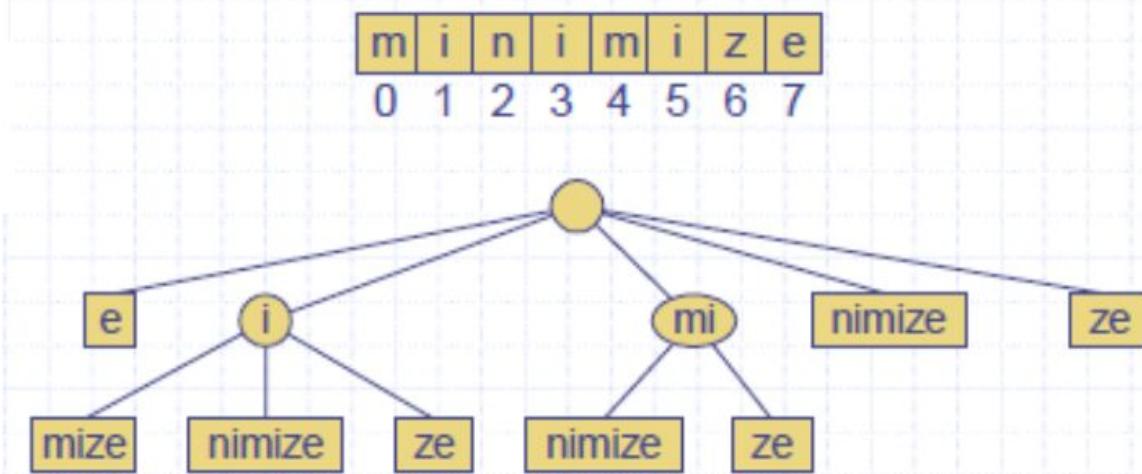
**Figure** Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.



One of the primary applications for tries is for the case when the strings in the collection  $S$  are all the suffixes of a string  $X$ . Such a trie is called the ***suffix trie*** (also known as a ***suffix tree*** or ***position tree***) of string  $X$ . For example, Following Figure shows the suffix trie for the eight suffixes of string “minimize.”

## Suffix Trie

- ◆ The suffix trie of a string  $X$  is the compressed trie of all the suffixes of  $X$



- ◆ Compact representation of the suffix trie for a string  $X$  of size  $n$  from an alphabet of size  $d$ 
  - Uses  $O(n)$  space
  - Supports arbitrary pattern matching queries in  $X$  in  $O(dm)$  time, where  $m$  is the size of the pattern

→ Using a suffix trie allows us to save space over a standard trie by using several space compression techniques, including those used for the compressed trie.

→ The compact representation of a suffix trie  $T$  for a string  $X$  of length  $n$  uses  $O(n)$  space.

→ Using a suffix trie allows us to save space over a standard trie by using several space compression techniques, including those used for the compressed trie.

→ The compact representation of a suffix trie  $T$  for a string  $X$  of length  $n$  uses  $O(n)$  space.

### Using a Suffix Trie:

The suffix trie  $T$  for a string  $X$  can be used to efficiently perform pattern-matching queries on text  $X$ . Namely, we can determine whether a pattern is a substring of  $X$  by trying to trace a path associated with  $P$  in  $T$ .  $P$  is a substring of  $X$  if and only if such a path can be traced. The search down the trie  $T$  assumes that nodes in  $T$  store some additional information, with respect to the compact representation of the suffix trie:

If node  $v$  has label  $j..k$  and  $Y$  is the string of length  $y$  associated with the path from the root to  $v$  (included), then  $X[k-y+1..k] = Y$ .

This property ensures that we can compute the start index of the pattern in the text when a match occurs in  $O(m)$  time.