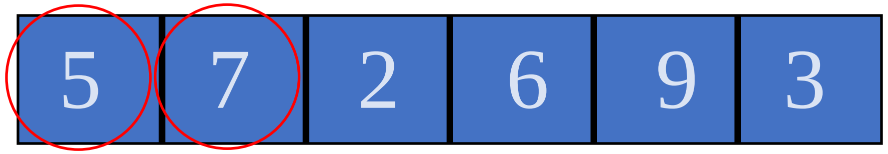
Wrapping up…

Part 1: Sorting

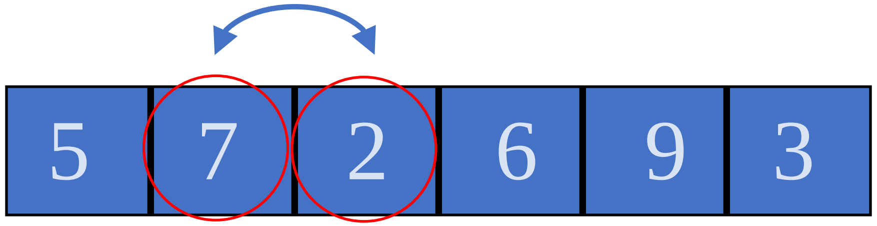
Bubble Sort

5	7	2	6	9	3

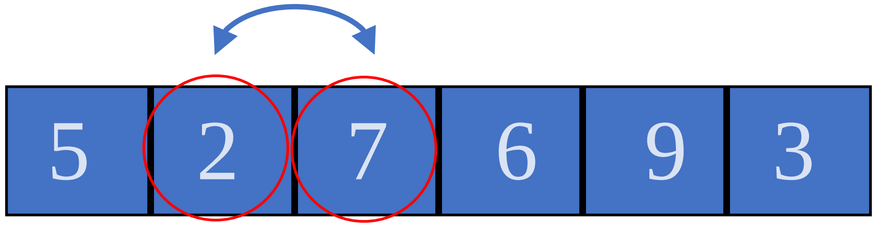
Bubble Sort



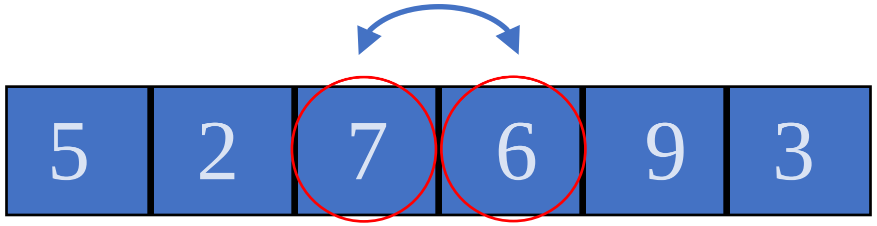
Bubble Sort



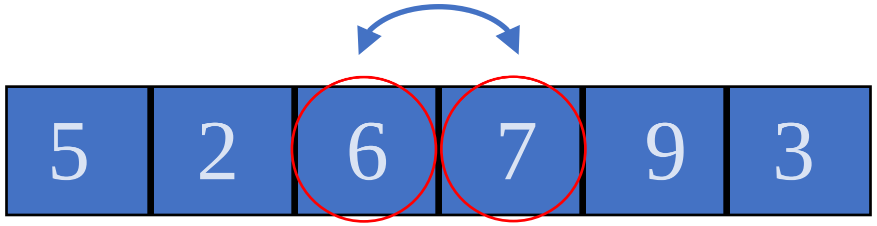
Bubble Sort



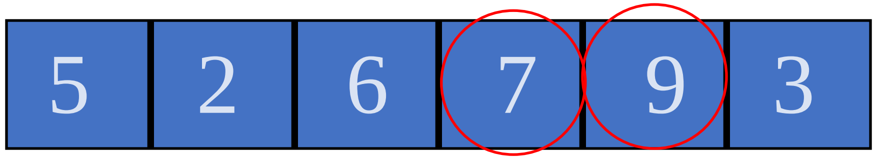
Bubble Sort



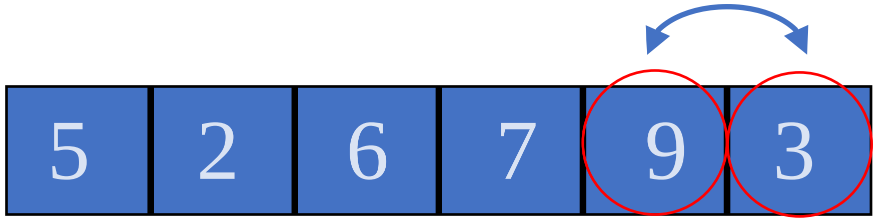
Bubble Sort



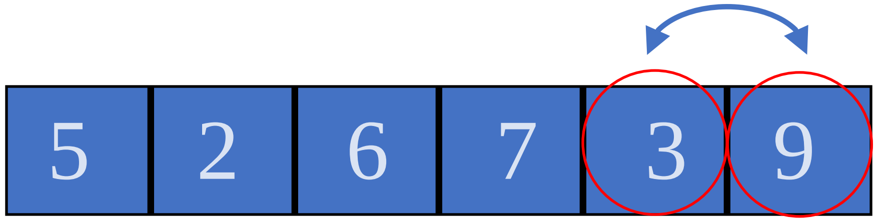
Bubble Sort



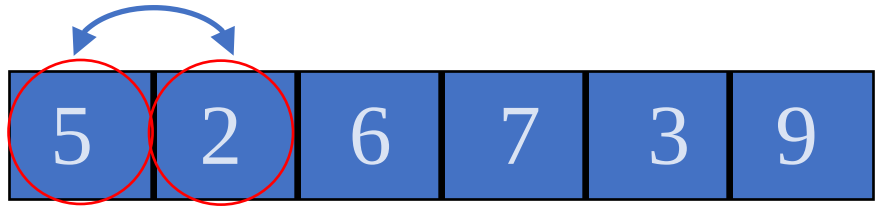
Bubble Sort



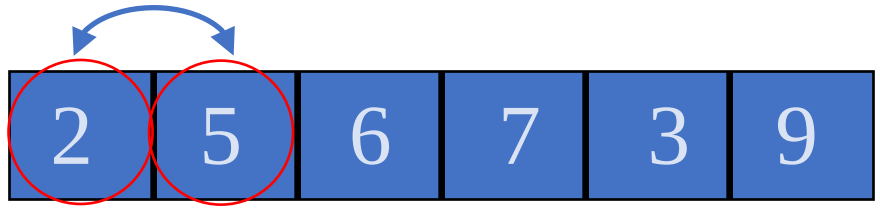
Bubble Sort



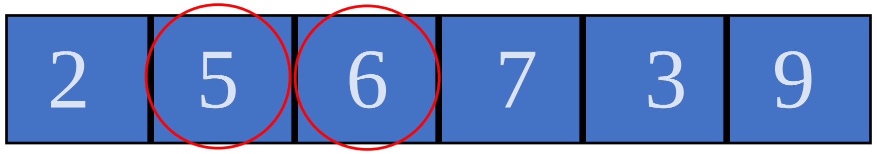
Bubble Sort



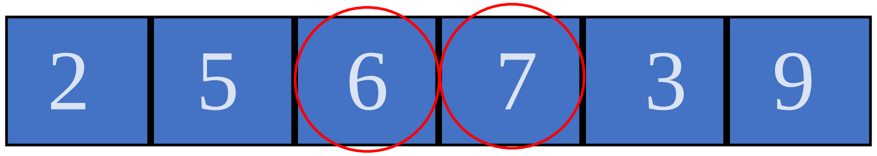
Bubble Sort



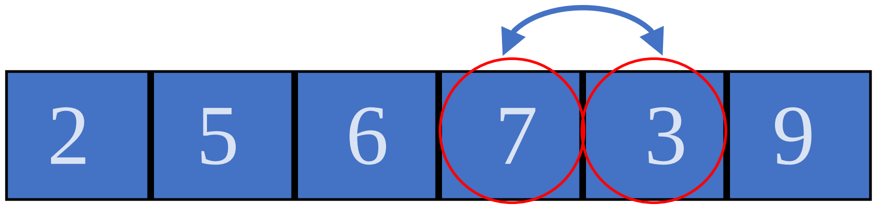
Bubble Sort



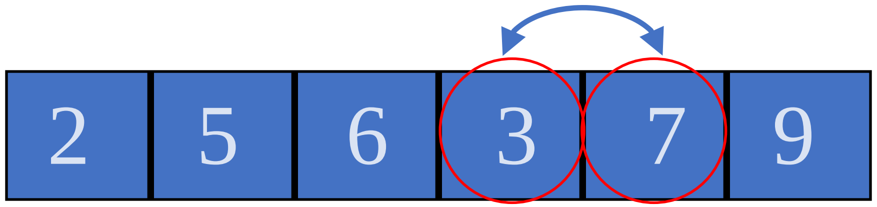
Bubble Sort



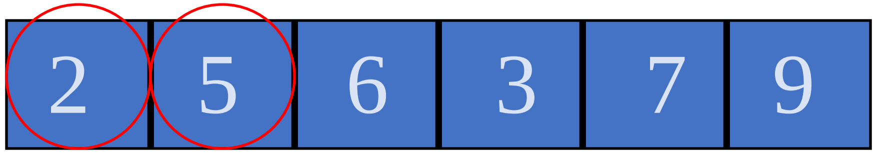
Bubble Sort



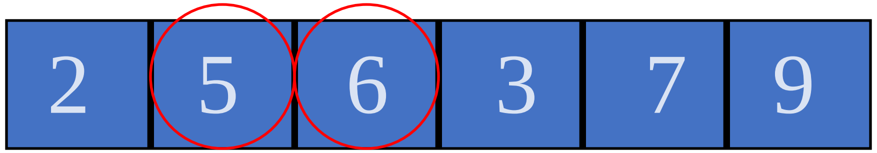
Bubble Sort



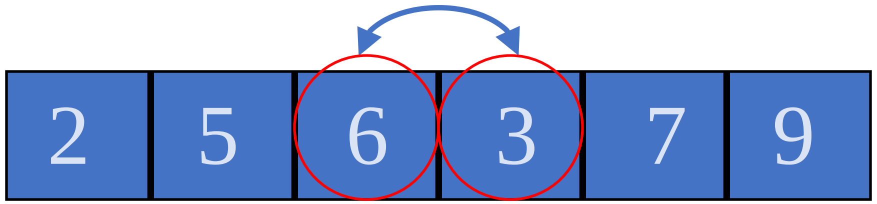
Bubble Sort



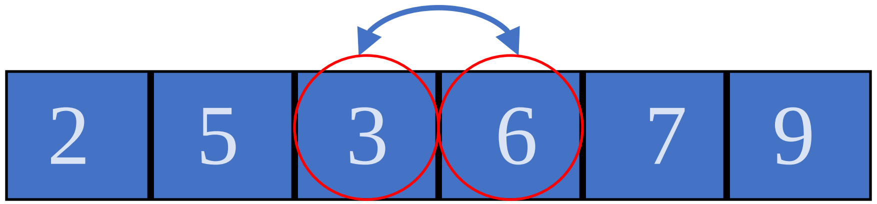
Bubble Sort



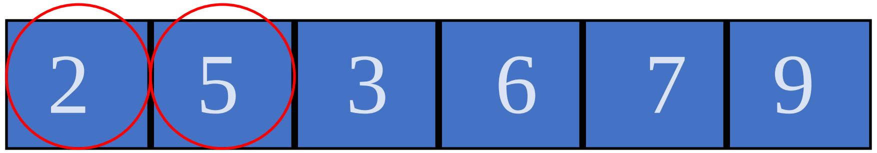
Bubble Sort



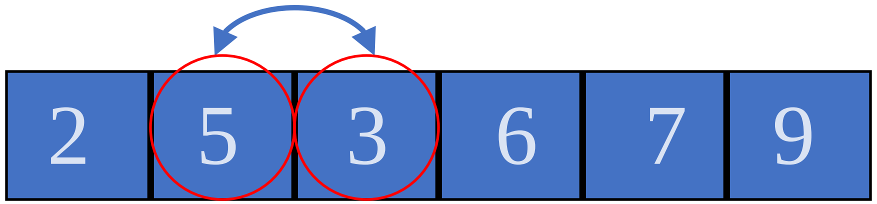
Bubble Sort



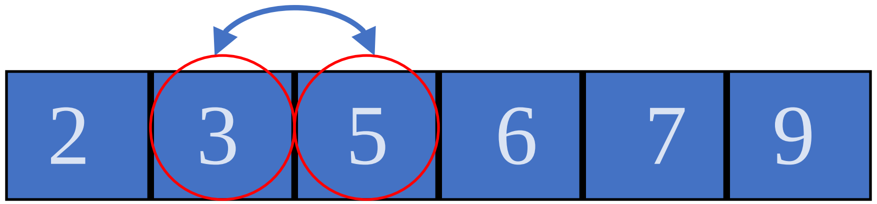
Bubble Sort



Bubble Sort



Bubble Sort



Bubble Sort

2	3	5	6	7	9

Complexity of Bubble Sort

Two loops each proportional to n, hence $T(n) = O(n^2)$

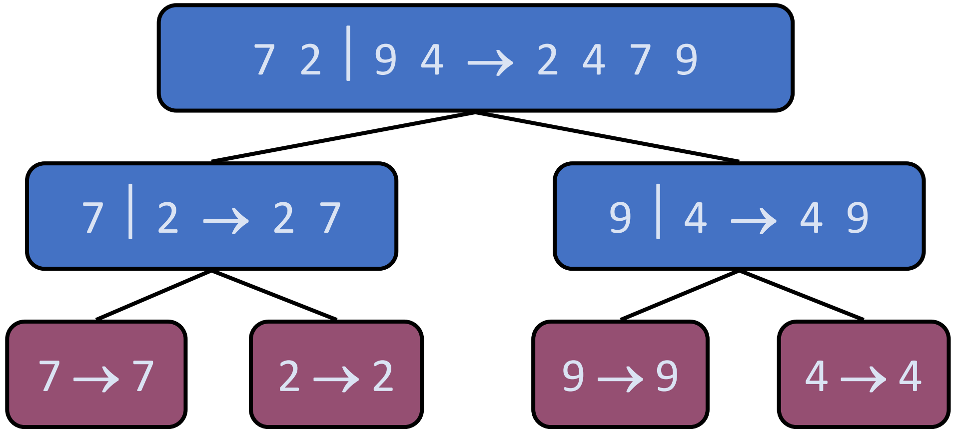
Algorithm ***bubbleSort***(***S, C***)

  Input sequence ***S***, comparator ***C***

  Output sequence ***S*** sorted according to ***C***

```
for ( i = 0; i < S.size(); i++ )
    for ( j = 1; j < S.size() – i; j++ )
        if ( S.atIndex ( j – 1 ) > S.atIndex ( j ) )
            S.swap ( j-1, j );
return(S)
```

Merge Sort



Merge Sort

- Merge sort is based on the divide-and-conquer paradigm. It consists of three steps:
  - Divide: partition input sequence $\boldsymbol{S}$ into two sequences $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ of about $\boldsymbol{n}/2$ elements each
  - Recur: recursively sort $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
  - Conquer: merge $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ into a unique sorted sequence

```
Algorithm mergeSort(S, C)
    Input sequence S, comparator C
    Output sequence S sorted
        according to C
    if S.size() > 1 {
        (S1, S2) := partition(S, S.size()/2)
        S1 := mergeSort(S1, C)
        S2 := mergeSort(S2, C)
        S := merge(S1, S2)
    }
    return(S)
```

And the complexity of mergesort…

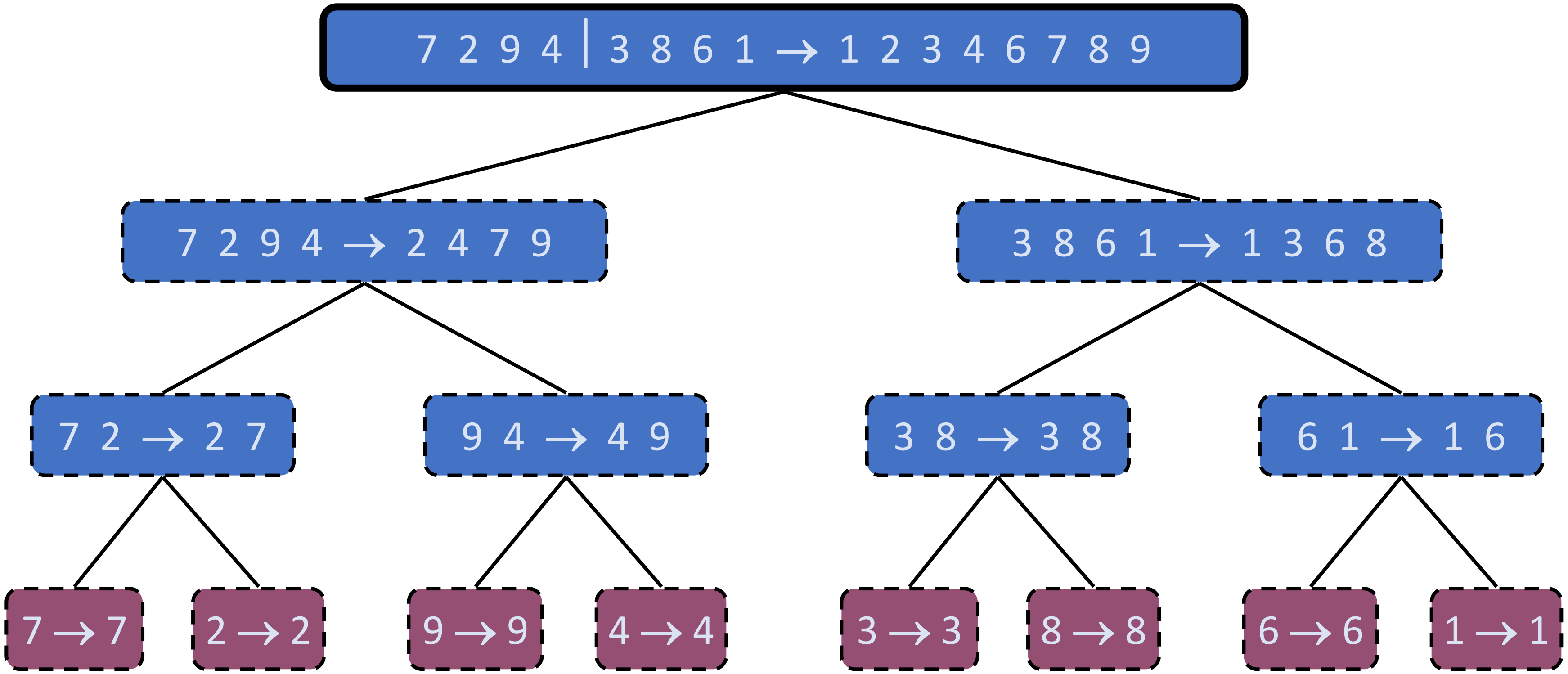
- So, the running time of Merge Sort can be expressed by the recurrence equation:

$$T(n) = 2T(n/2) + M(n)$$
$$= 2T(n/2) + O(n)$$
$$= O(n\log n)$$

Algorithm *mergeSort*(*S*, *C*)
    Input sequence ***S***, comparator ***C***
    Output sequence ***S*** sorted
        according to ***C***
    if *S.size*() > 1 {
        (S_1, S_2) := *partition*(*S*, *S.size()/2*)
        S_1 := *mergeSort*(S_1, *C*)
        S_2 := *mergeSort*(S_2, *C*)
        S := *merge*(S_1, S_2)
    }
    return(S)

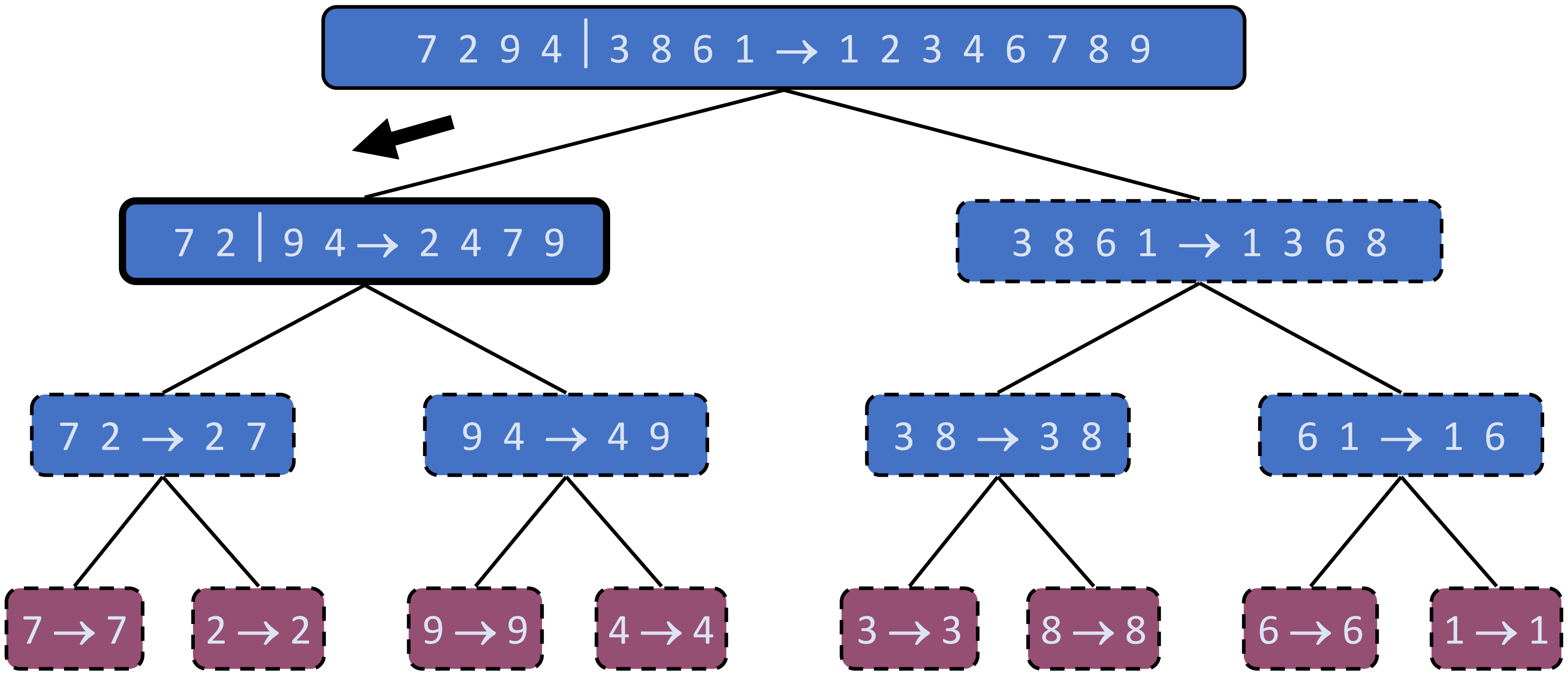
Execution Example

- Partition



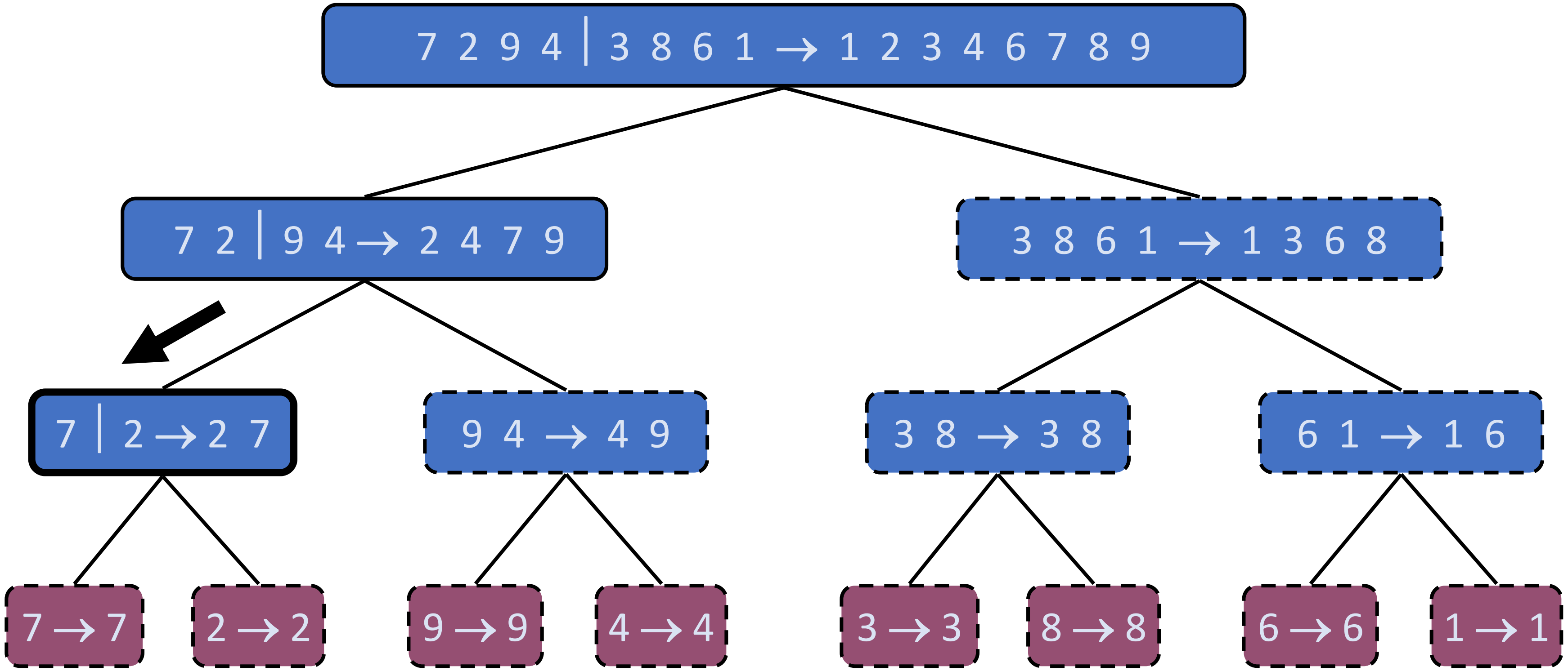
Execution Example (cont.)

- Recursive call, partition



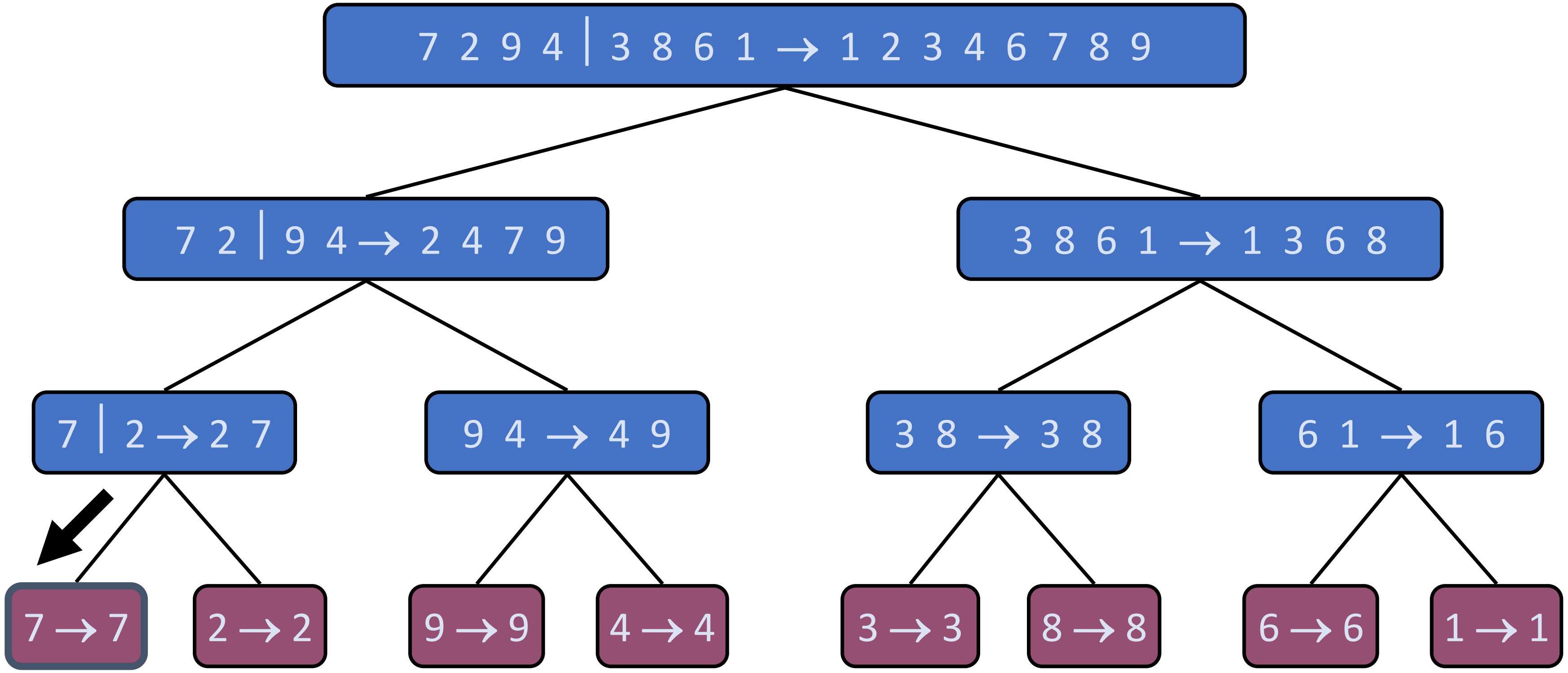
Execution Example (cont.)

- Recursive call, partition



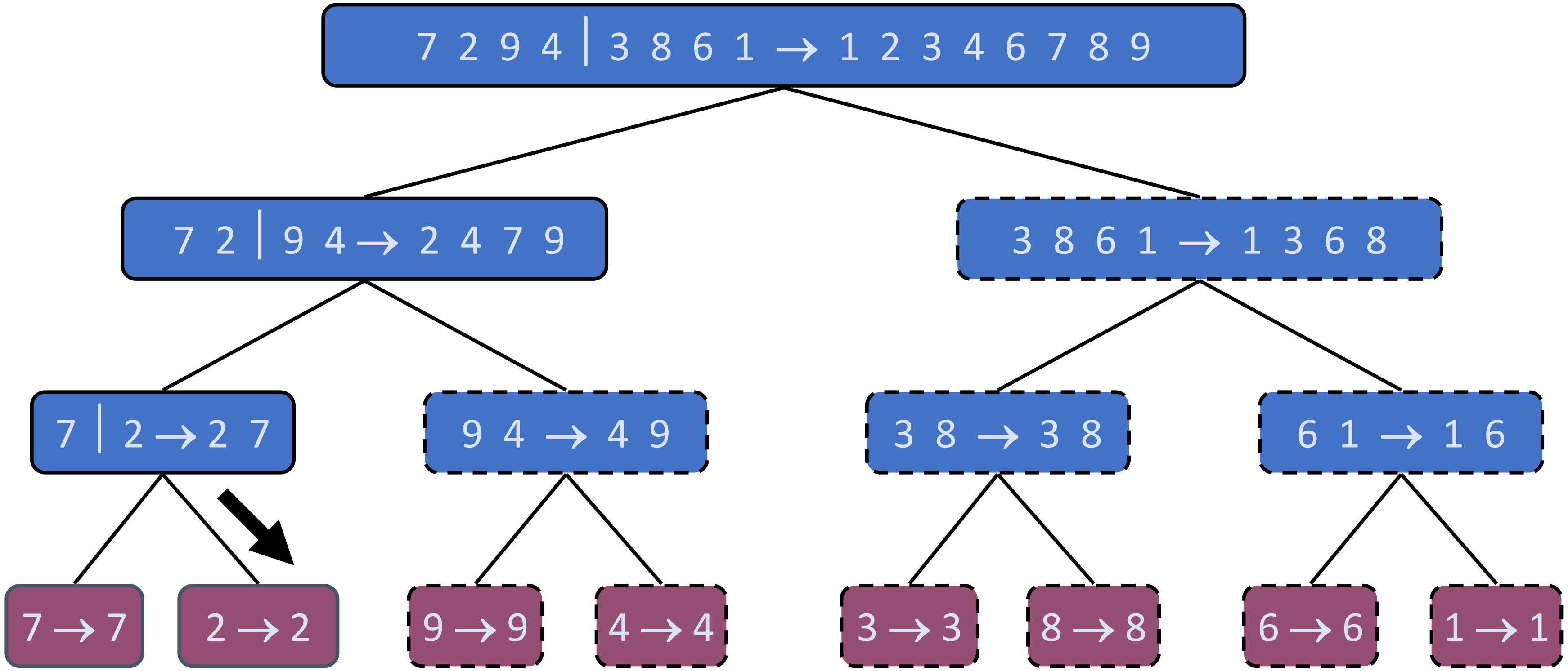
Execution Example (cont.)

- Recursive call, base case



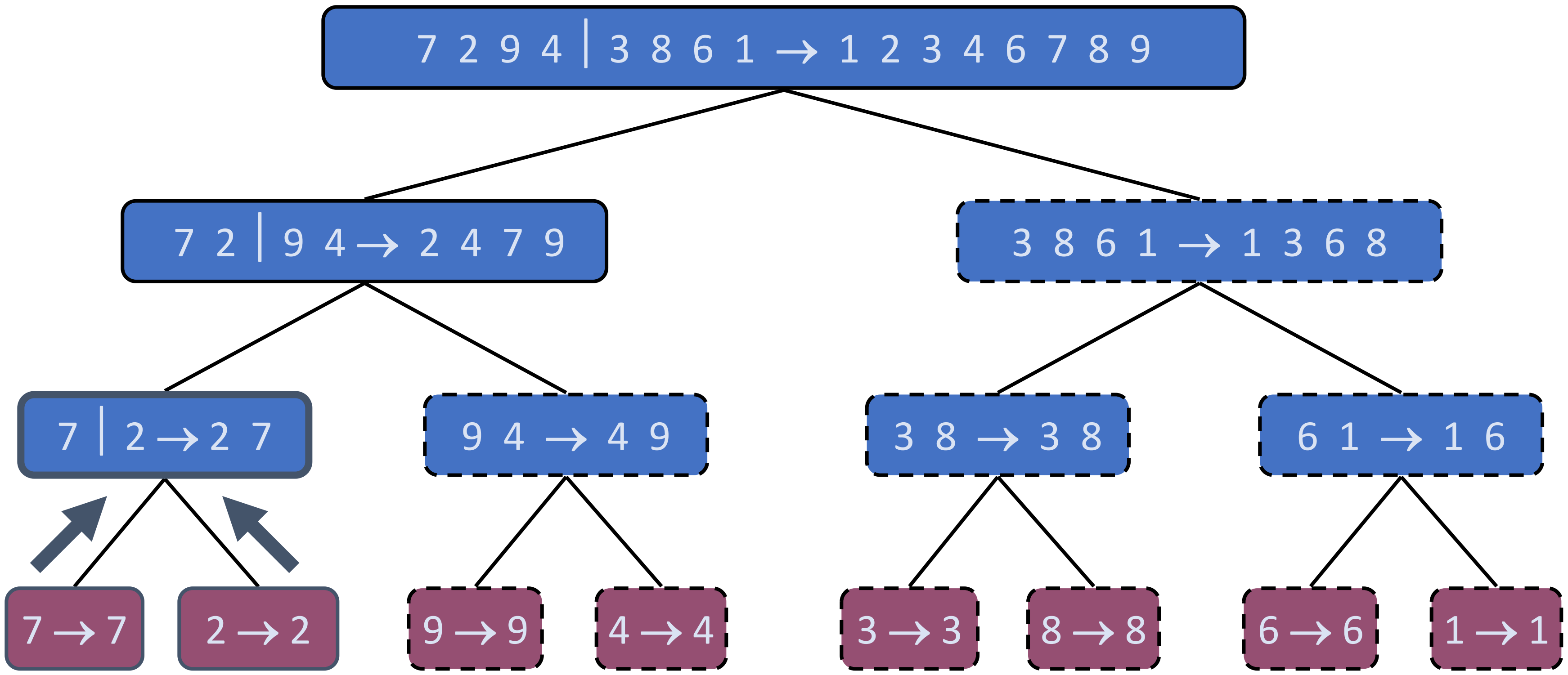
Execution Example (cont.)

- Recursive call, base case



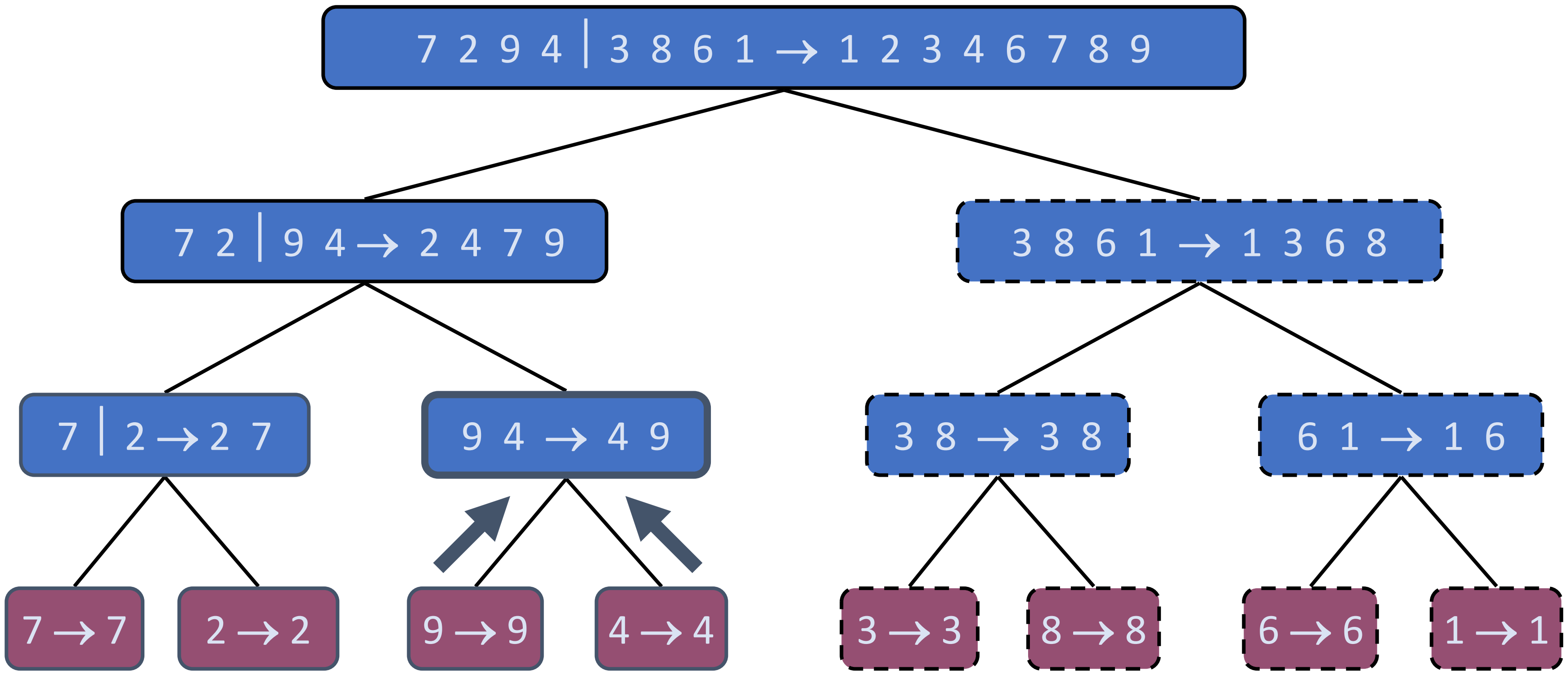
Execution Example (cont.)

- Merge



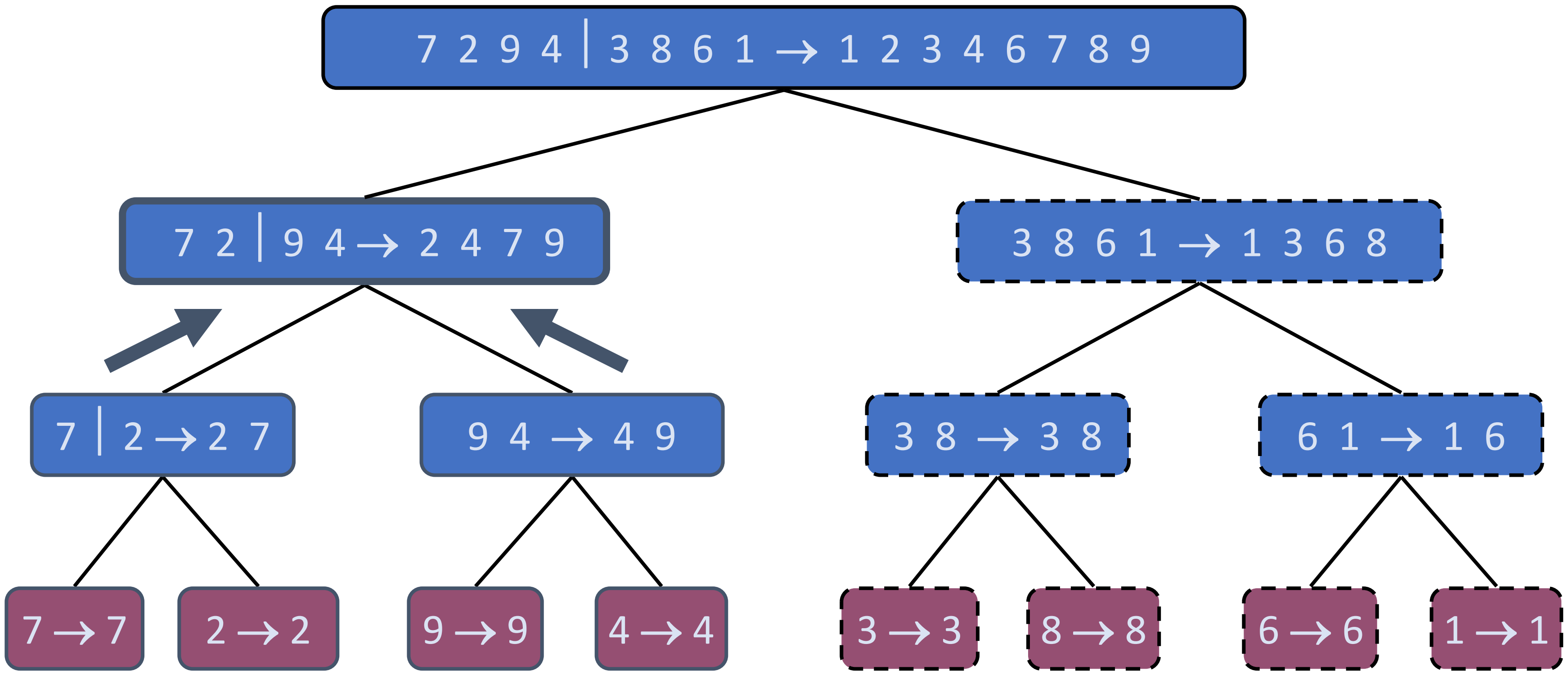
Execution Example (cont.)

- Recursive call, …, base case, merge



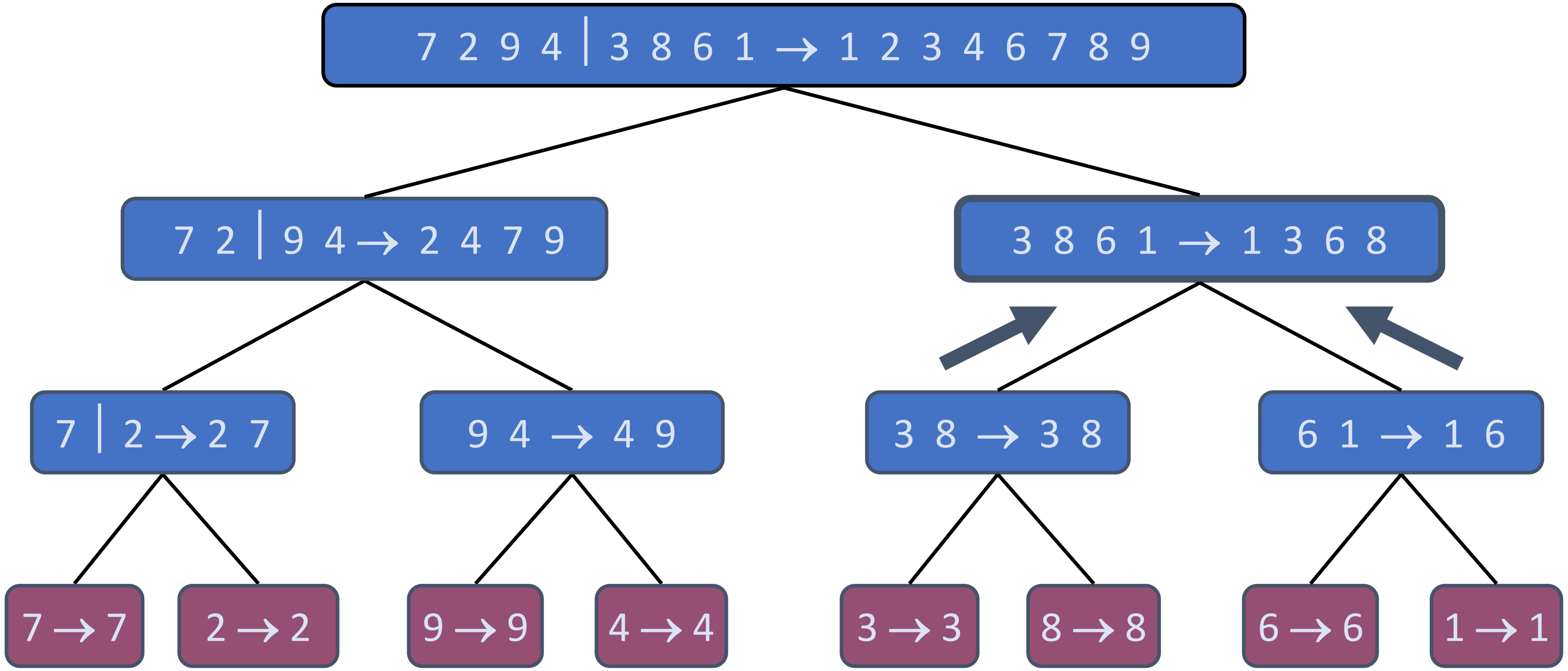
Execution Example (cont.)

- Merge



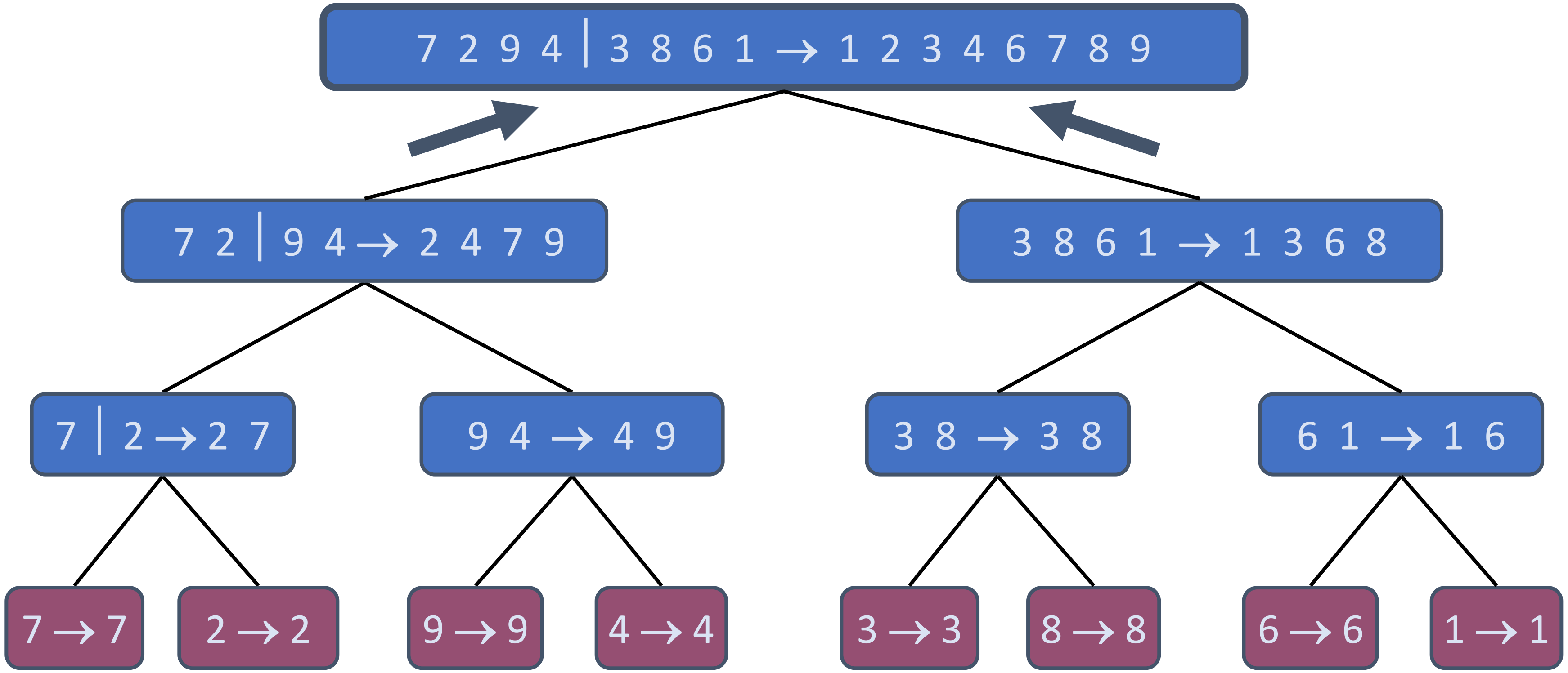
Execution Example (cont.)

- Recursive call, …, merge, merge



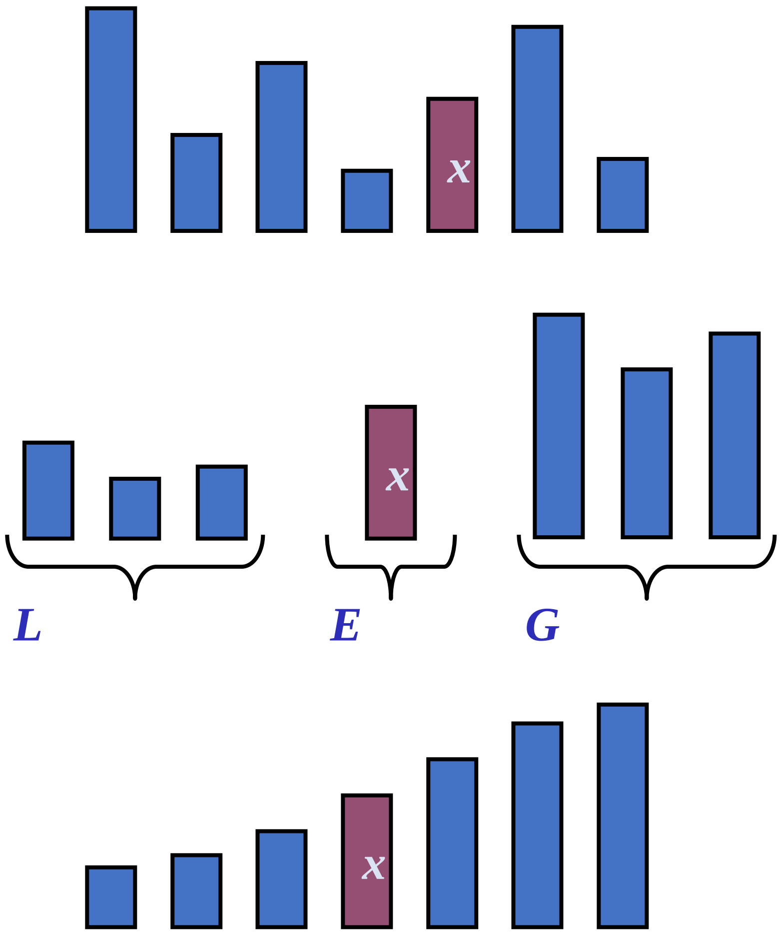
Execution Example (cont.)

- Merge



Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



Analysis of Quick Sort using Recurrence Relations

- Assumption: random pivot expected to give equal sized sublists
- The running time of Quick Sort can be expressed as:

$$T(n) = 2T(n/2) + P(n)$$

- T(n) - time to run quicksort() on an input of size n
- P(n) - time to run partition() on input of size n

Algorithm *QuickSort*(*S*, *l*, *r*)
  Input sequence ***S***, ranks ***l*** and ***r***
  Output sequence ***S*** with the elements of rank between ***l*** and ***r*** rearranged in increasing order
  if $l \geq r$
    return
  $i \leftarrow$ a random integer between ***l*** and ***r***
  $x \leftarrow$ ***S.elemAtRank***(***i***)
  (***h***, ***k***) $\leftarrow$ ***Partition***(***x***)
  QuickSort(***S***, ***l***, ***h*** − 1)
  QuickSort(***S***, ***k*** + 1, ***r***)

Summary of Sorting Algorithms (so far)

Algorithm	Time	Notes
Selection Sort	$O(n^2)$	Slow, in-place For small data sets
Insertion/Bubble Sort	$O(n^2)$ WC, AC $O(n)$ BC	Slow, in-place For small data sets
Heap Sort	$O(n\log n)$	Fast, in-place For large data sets
Quick Sort	Exp. $O(n\log n)$ AC, BC $O(n^2)$ WC	Fastest, randomized, in-place For large data sets
Merge Sort	$O(n\log n)$	Fast, sequential data access For huge data sets

Selection

The Selection Problem

- Given an integer k and n elements $x_1, x_2, \ldots, x_n$, taken from a total order, find the k^{th} smallest element in this set.
  - Also called order statistics, i^{th} order statistic is i^{th} smallest element
  - Minimum - k=1 - 1st order statistic
  - Maximum - k=n - nth order statistic
  - Median - k=n/2
  - etc

The Selection Problem

- Naïve solution - SORT!
- we can sort the set in O(n log n) time and then index the k-th element.

7 4 9 <u>6</u> 2 → 2 4 <u>6</u> 7 9

k=3

- Can we solve the selection problem faster?

The Minimum (or Maximum)

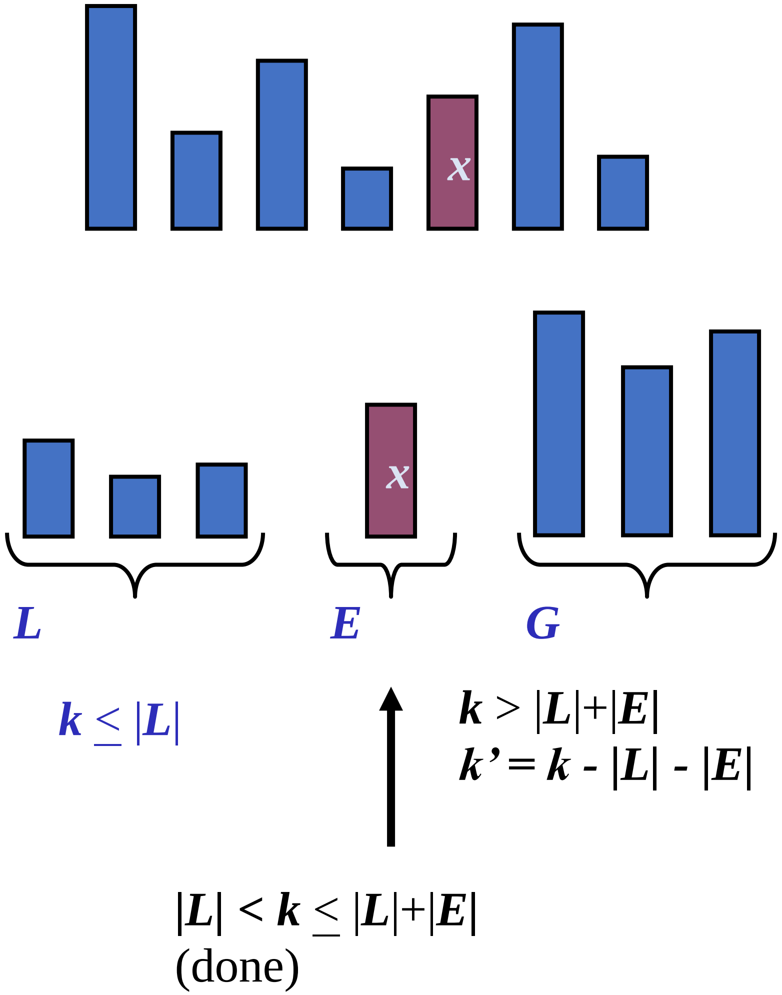
```
Minimum (A) {
    m = A[1]
    For I=2,n
        M=min(m,A[I])
    Return m
}
```

- Running Time
  - O(n)

Quick-Select

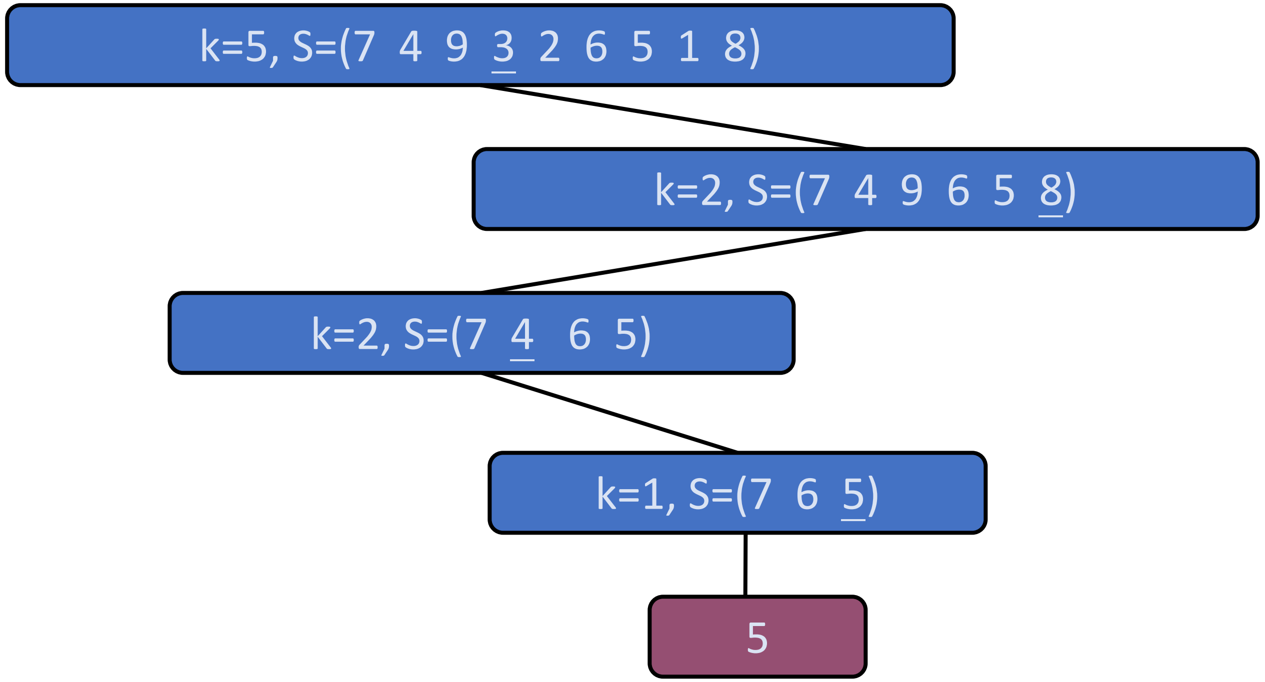
- Quick-select is a selection algorithm based on the prune-and-search paradigm:
  - Prune: pick a random element ***x*** (called pivot) and partition ***S*** into
    - ***L*** elements less than ***x***
    - ***E*** elements equal ***x***
    - ***G*** elements greater than ***x***
  - Search: depending on k, either answer is in ***E***, or we need to recur on either ***L*** or ***G***

  - Note: Partition same as Quicksort



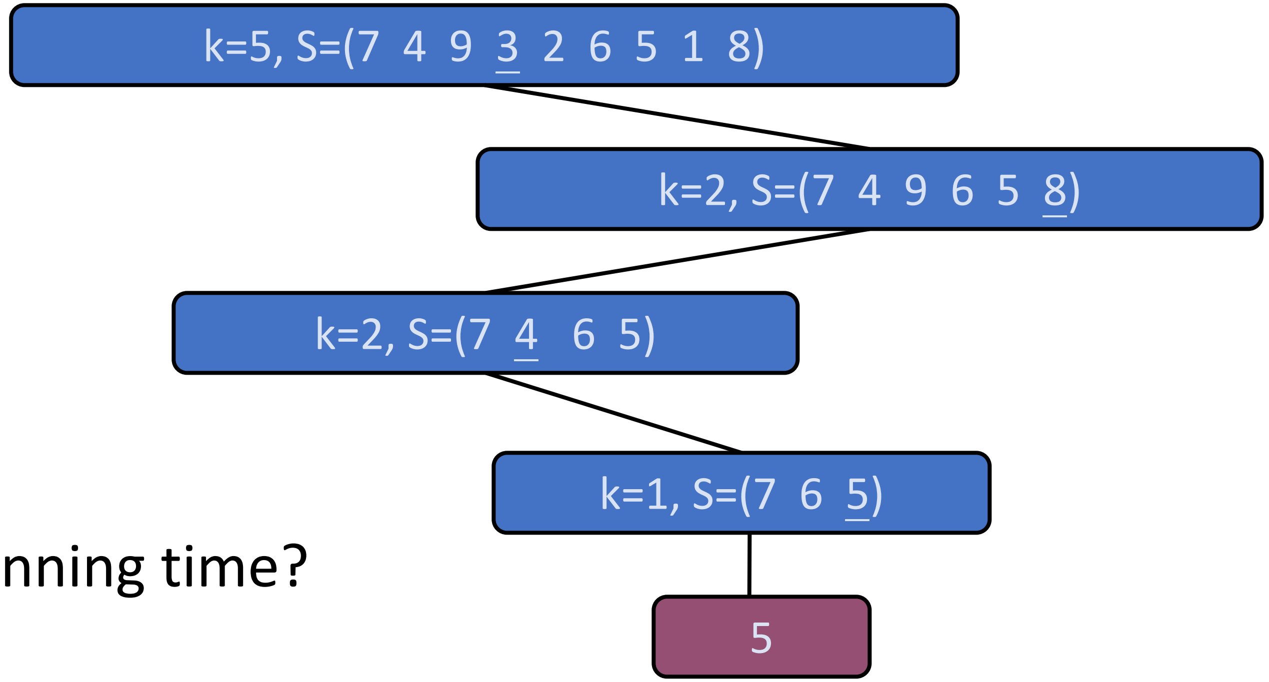
Quick-Select Visualization

- An execution of quick-select can be visualized by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence



Quick-Select Visualization

- An execution of quick-select can be visualized by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence



What is the running time?

Guaranteeing a Good Split

- We will have a good split if we can ensure that the pivot is the median element or an element close to the median.
- Hence, determining a reasonable pivot is the first step.

Choosing a Pivot

- Median-of-Medians:
  - Divide the n elements into $\lceil n/5 \rceil$ groups.
    - $\lfloor n/5 \rfloor$ groups contain 5 elements each. Last group can contain $n \bmod 5 < 5$ elements.
    - Determine the median of each of the groups.
      - Sort each group using Insertion Sort. Pick the median from the sorted list of group elements.
    - Recursively find the median x of the $\lceil n/5 \rceil$ medians.
- Recurrence for running time (of median-of-medians):
  - $T(n) = O(n) + T(\lceil n/5 \rceil)$

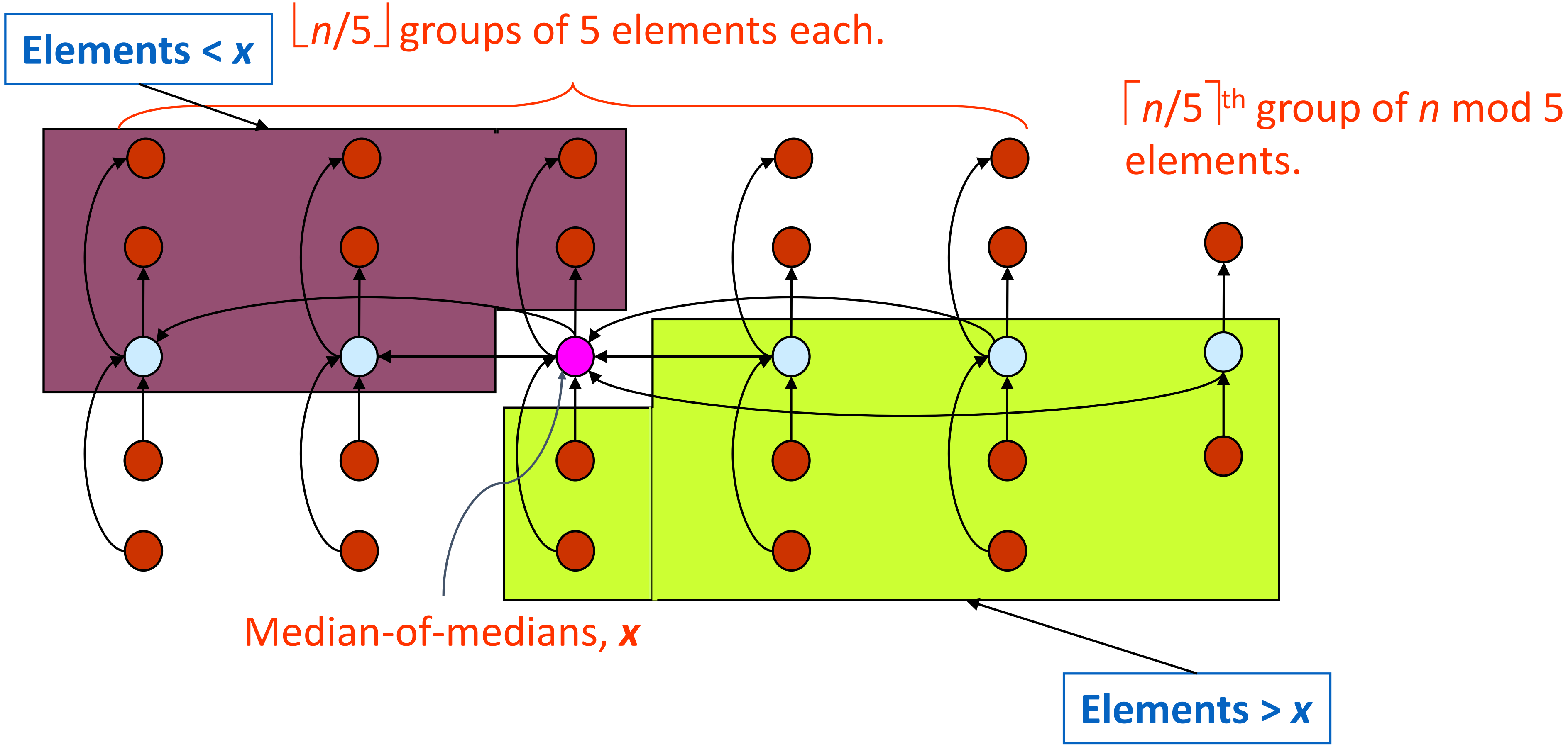
Algorithm Select

- Determine the median-of-medians x (using the procedure on the previous slide.)
- Partition the input array around x using the variant of Partition.
- Let k be the index of x that Partition returns.
- If $k = i$, then return x.
- Else if $i < k$, then apply Select recursively to $A[1..k-1]$ to find the i^{th} smallest element.
- Else if $i > k$, then apply Select recursively to $A[k+1..n]$ to find the $(i - k)^{th}$ smallest element.

(Assumption: Select operates on $A[1..n]$. For subarrays $A[p..r]$, suitably change k.)

Worst-case Split

Arrows point from larger to smaller elements.



Worst-case Split

- Assumption: Elements are distinct. Why?
- At least half of the $\lceil n/5 \rceil$ medians are greater than x.
- Thus, at least half of the $\lceil n/5 \rceil$ groups contribute 3 elements that are greater than x.
  - The last group and the group containing x may contribute fewer than 3 elements. Exclude these groups.
- Hence, the no. of elements > x is at least $3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$
- Analogously, the no. of elements < x is at least $3n/10–6$.
- Thus, in the worst case, Select is called recursively on at most $7n/10+6$ elements.

Recurrence for worst-case running time

- $T(\text{Select}) \leq T(\text{Median-of-medians}) + T(\text{Partition}) + T(\text{recursive call to select})$

- $T(n) \leq \underbrace{O(n) + T(\lceil n/5 \rceil)}_{T(\text{Median-of-medians})} + \underbrace{O(n)}_{T(\text{Partition})} + \underbrace{T(7n/10+6)}_{T(\text{recursive call})}$

  $= T(\lceil n/5 \rceil) + T(7n/10+6) + O(n)$

- Assume $T(n) \leq \Theta(1)$, for $n \leq 140$.

Solving the recurrence

- To show: $T(n) = O(n) \le cn$ for suitable c and all $n > 0$.
- Assume: $T(n) \le cn$ for suitable c and all $n \le 140$.
- Substituting the inductive hypothesis into the recurrence,
  - $T(n) \le c\lceil n/5 \rceil + c(7n/10+6)+an$

$$
\begin{aligned}
&\le cn/5 + c + 7cn/10 + 6c + an \\
&= 9cn/10 + 7c + an \\
&= cn + (-cn/10 + 7c + an) \\
&\le cn, \quad \text{if } -cn/10 + 7c + an \le 0.
\end{aligned}
$$

$-cn/10 + 7c + an \le 0$
$\equiv \; c \ge 10a(n/(n-70))$,
when $n > 70$.

For $n \ge 140$, $c \ge 20a$.

- Hence, c can be chosen for any $n = n_0 > 70$, provided it can be assumed that $T(n) = O(1)$ for $n \le n_0$.
- Thus, Select has linear-time complexity in the worst case.

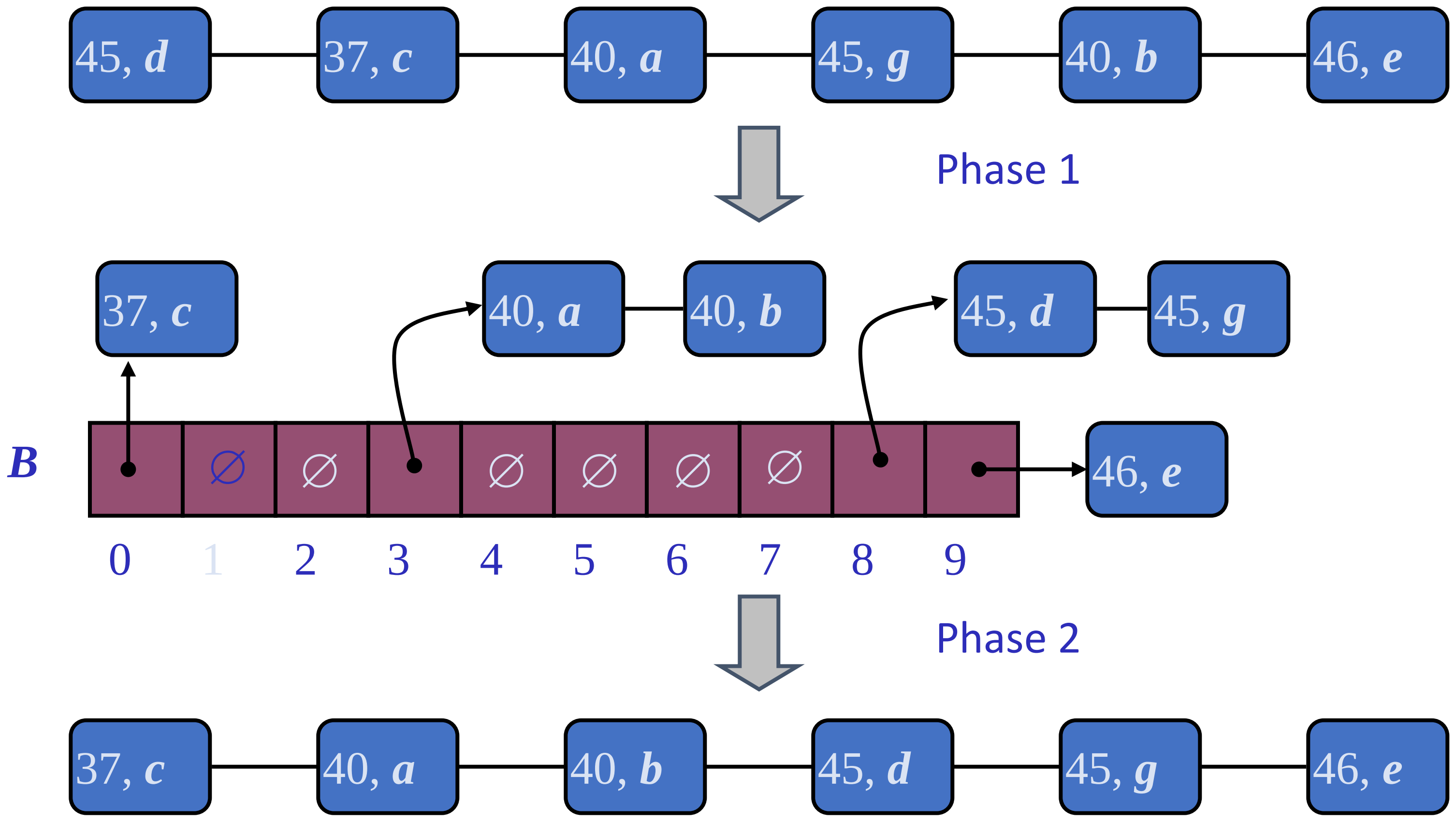
Continuing Part 1: Sorting……

Lower bound of comparison based sorting

Bucket-Sort, Radix-Sort and Counting-Sort
(Can we sort in linear time?)

Example

- Key range [37, 46] – map to buckets [0,9]
- **Solution:** using mod 10, i.e., bucket(k) = k mod 10



Bucket-Sort

- Let be S be a sequence of n (key, element) items with keys in the range $[0, N-1]$
- Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)

  Phase 1: Empty sequence S by moving each item (k, o) into its bucket $B[k]$

  Phase 2: For $i = 0, \ldots, N-1$, move the items of bucket $B[i]$ to the end of sequence S
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time

  Bucket-sort takes $O(n + N)$ time

It is a stable sort.

Algorithm *bucketSort*(S, N)
    Input sequence S of (key, element)
        items with keys in the range
        $[0, N-1]$
    Output sequence S sorted by
        increasing keys
    $B \leftarrow$ array of N empty sequences
    while $\neg S.isEmpty()$
        $f \leftarrow S.first()$
        $(k, o) \leftarrow S.remove(f)$
        $B[k].insertLast((k, o))$
    for $i \leftarrow 0$ **to** $N-1$
        while $\neg B[i].isEmpty()$
            $f \leftarrow B[i].first()$
            $(k, o) \leftarrow B[i].remove(f)$
            $S.insertLast((k, o))$

Stable Sort

- A property of sorts.
- If a sort guarantees the relative order of equal items stays the same then it is a *stable sort.*

Example:

Input: [7_1, 6, 7_2, 5, 1, 2, 7_3, -5] original data
    (Note: subscripts added for clarity)

Output: [-5, 1, 2, 5, 6, 7_1, 7_2, 7_3] sorted data
    (Result of stable sort)

Lexicographic Order

- Given a list of 3-tuples:

(7,4,6) (5,1,5) (2,4,6) (2,1,4) (5,1,6) (3,2,4)

- After sorting, the list is in lexicographical order:

(2,1,4) (2,4,6) (3,2,4) (5,1,5) (5,1,6) (7,4,6)

Lexicographic Order Formalized

- A $\boldsymbol{d}$-tuple is a sequence of $\boldsymbol{d}$ keys $(\boldsymbol{k}_1, \boldsymbol{k}_2, \ldots, \boldsymbol{k_d})$, where key $\boldsymbol{k_i}$ is said to be the $\boldsymbol{i}$-th dimension of the tuple
  - Example:
    - The Cartesian coordinates of a point in space is a 3-tuple
- The lexicographic order of two $\boldsymbol{d}$-tuples is recursively defined as follows

$$(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x_d}) < (\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y_d})$$

$$\Leftrightarrow$$

$$\boldsymbol{x}_1 < \boldsymbol{y}_1 \vee \boldsymbol{x}_1 = \boldsymbol{y}_1 \wedge (\boldsymbol{x}_2, \ldots, \boldsymbol{x_d}) < (\boldsymbol{y}_2, \ldots, \boldsymbol{y_d})$$

I.e., the tuples are compared by the first dimension, then by the second dimension, etc.

Exercise: Lexicographic Order

- Given a list of 2-tuples, we can order the tuples lexicographically by applying a stable sorting algorithm two times:

  (3,3) (1,5) (2,5) (1,2) (2,3) (1,7) (3,2) (2,2)

- Possible ways of doing it:
  1. Sort first by 1st element of tuple and then by 2nd element of tuple
  2. Sort first by 2nd element of tuple and then by 1st element of tuple

- Show the result of sorting the list using both options

Exercise: Lexicographic Order

(3,3) (1,5) (2,5) (1,2) (2,3) (1,7) (3,2) (2,2)

- Using a stable sort,
  1. Sort first by 1st element of tuple and then by 2nd element of tuple
  2. Sort first by 2nd element of tuple and then by 1st element of tuple

- Option 1:
  - 1st sort: (1,5) (1,2) (1,7) (2,5) (2,3) (2,2) (3,3) (3,2)
  - 2nd sort: (1,2) (2,2) (3,2) (2,3) (3,3) (1,5) (2,5) (1,7) - WRONG
- Option 2:
  - 1st sort: (1,2) (3,2) (2,2) (3,3) (2,3) (1,5) (2,5) (1,7)
  - 2nd sort: (1,2) (1,5) (1,7) (2,2) (2,3) (2,5) (3,2) (3,3) - CORRECT

Lexicographic-Sort

- Let C_i be the comparator that compares two tuples by their i^{th} dimension
- Let $stableSort(S, C)$ be a stable sorting algorithm that uses comparator C
- Lexicographic-sort sorts a sequence of d-tuples in lexicographic order by executing d times algorithm $stableSort$, one per dimension
- Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of $stableSort$

Algorithm *lexicographicSort*(S)

Input sequence S of d-tuples
Output sequence S sorted in lexicographic order

for $i \leftarrow d$ **downto** 1
    $stableSort(S, C_i)$

Radix-Sort

- Radix-sort is a specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension
- Radix-sort is applicable to tuples where the keys in each dimension $\boldsymbol{i}$ are integers in the range $[0, \boldsymbol{N} - 1]$
- Radix-sort runs in time $\boldsymbol{O}(\boldsymbol{d}(\boldsymbol{n} + \boldsymbol{N}))$

Algorithm ***radixSort***(S, N)
  Input sequence $\boldsymbol{S}$ of $\boldsymbol{d}$-tuples such
    that $(0, \ldots, 0) \leq (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d)$ and
    $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d) \leq (\boldsymbol{N} - 1, \ldots, \boldsymbol{N} - 1)$
    for each tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d)$ in $\boldsymbol{S}$
  Output sequence $\boldsymbol{S}$ sorted in
    lexicographic order
  for $i \leftarrow \boldsymbol{d}$ **downto** 1
    set the key $\boldsymbol{k}$ of each item
    $(\boldsymbol{k}, (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d))$ of $\boldsymbol{S}$
  to i-th dimension $\boldsymbol{x}_i$
    bucketSort(S, N)

A
1066
432
29
978
912
167
1544
533

A
1066
432
29
978
912
167
1544
533

A
432
912
533
1544
1066
167
978
29

A

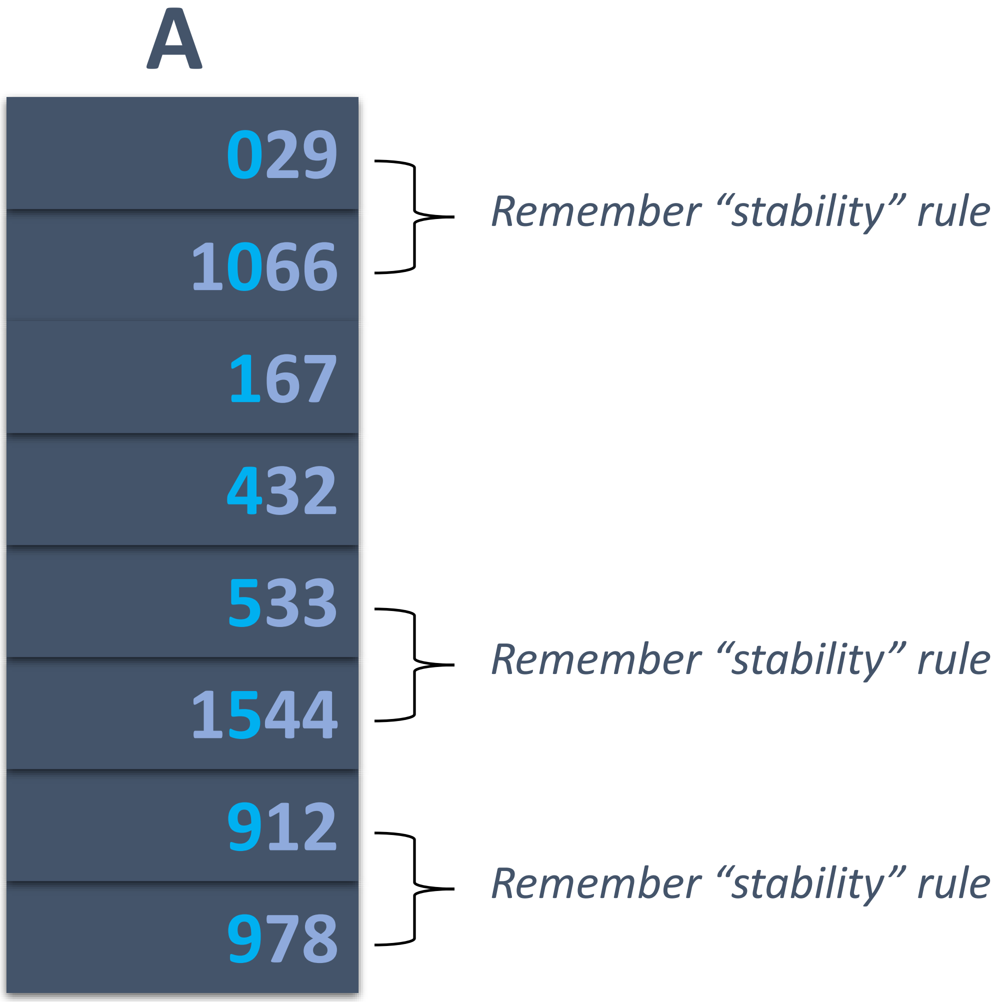
432
912
533
1544
1066
167
978
29

Remember "stability" rule

A
432
912
533
1544
1066
167
978
29


A
912
29
432
533
1544
1066
167
978
Remember “stability” rule
Remember “stability” rule

A
912
029
432
533
1544
1066
167
978


A
029
1066
167
432
533
1544
912
978
Remember “stability” rule
Remember “stability” rule
Remember “stability” rule

A
0029
1066
0167
0432
0533
1544
0912
0978

A

0029
0167
0432
0533
0912
0978
1066
1544

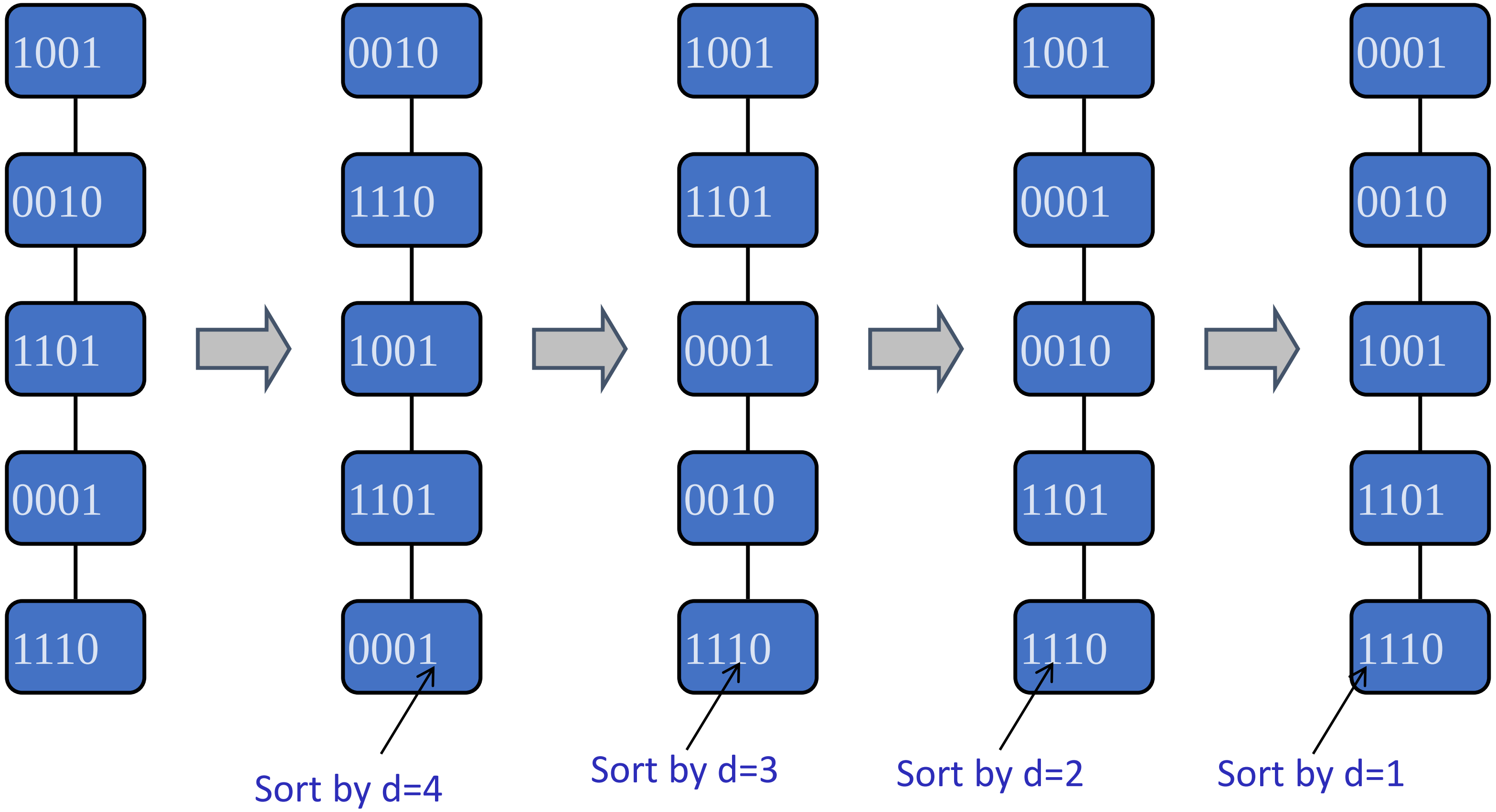
Remember “stability” rule

Remember “stability” rule

A
0029
0167
0432
0533
0912
0978
1066
1544

Example: Radix-Sort for Binary Numbers

- Sorting a sequence of 4-bit integers
  - d=4, N=2 so O(d(n+N)) = O(4(n+2)) = O(n)



Counting Sort

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

```
Example:
•  Consider the data in the range 0 to 9.
•  Input data: 1, 4, 1, 2, 7, 5, 2

Steps:
Take a count array to store the count of each unique object.
Index: 0 1 2 3 4 5 6 7 8 9
Count: 0 2 2 0 1 1 0 1 0 0
```

Part 2: Tries: Basics

How to perform pattern text matching?

Dictionary ADT for Strings

- *Dictionary* ADT for strings – stores a set of text strings:
  - *search*(x) – checks if string x is in the set
  - *insert*(x) – inserts a new string x into the set
  - *delete*(x) – deletes the string equal to x from the set of strings
- Assumptions, notation:
  - n strings, N characters in total
  - m – length of x
  - Size of the alphabet $d = |\Sigma|$

BST of Strings

A BST is a binary tree that satisfies the following properties for every node:

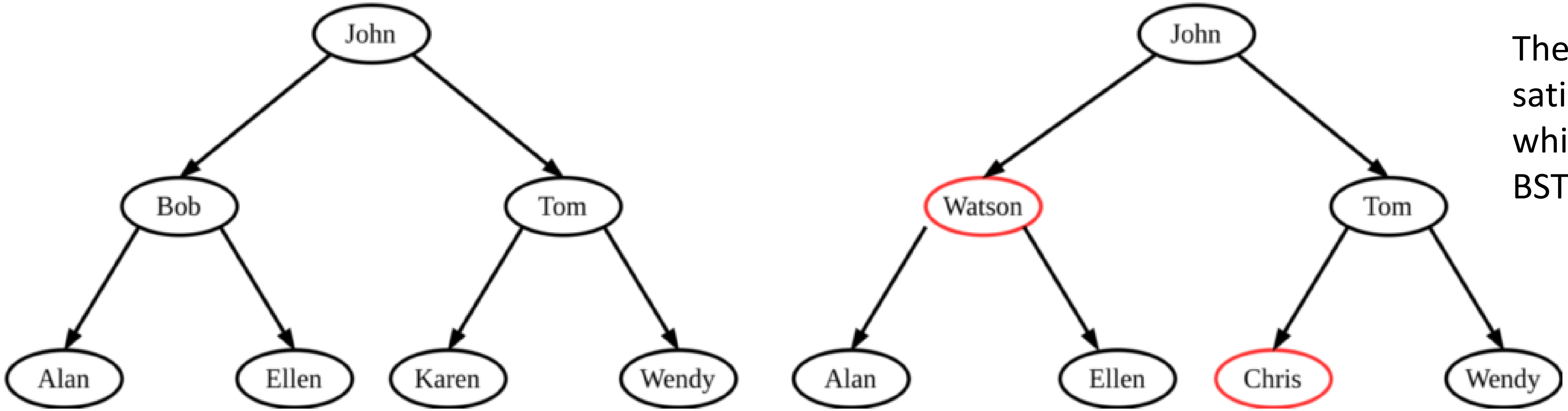
- The left subtree of the node contains only nodes with keys lesser than the node’s key
- The right subtree of the node contains only nodes with keys greater than the node’s key
- The left and right subtree each must also be a binary search tree.

Consider the case where the keys of the nodes are represented by strings and not numbers. Then, we should first define the ordering of the strings.

Lexicographic ordering is defined as the order that each string that appears in a dictionary. To determine which string is lexicographically larger we compare the corresponding characters of the two strings from left to right. The first character where the two strings differ determines which string comes first. Characters are compared using the Unicode character set and all uppercase letters come before lowercase letters.

BST of Strings

- We can, of course, use binary search trees. Some issues:
  - Keys are of varying length
  - A lot of strings share similar prefixes (beginnings) – potential for saving space
  - Let's count comparisons of characters.
    - What is the worst-case running time of searching for a string of length *m*?

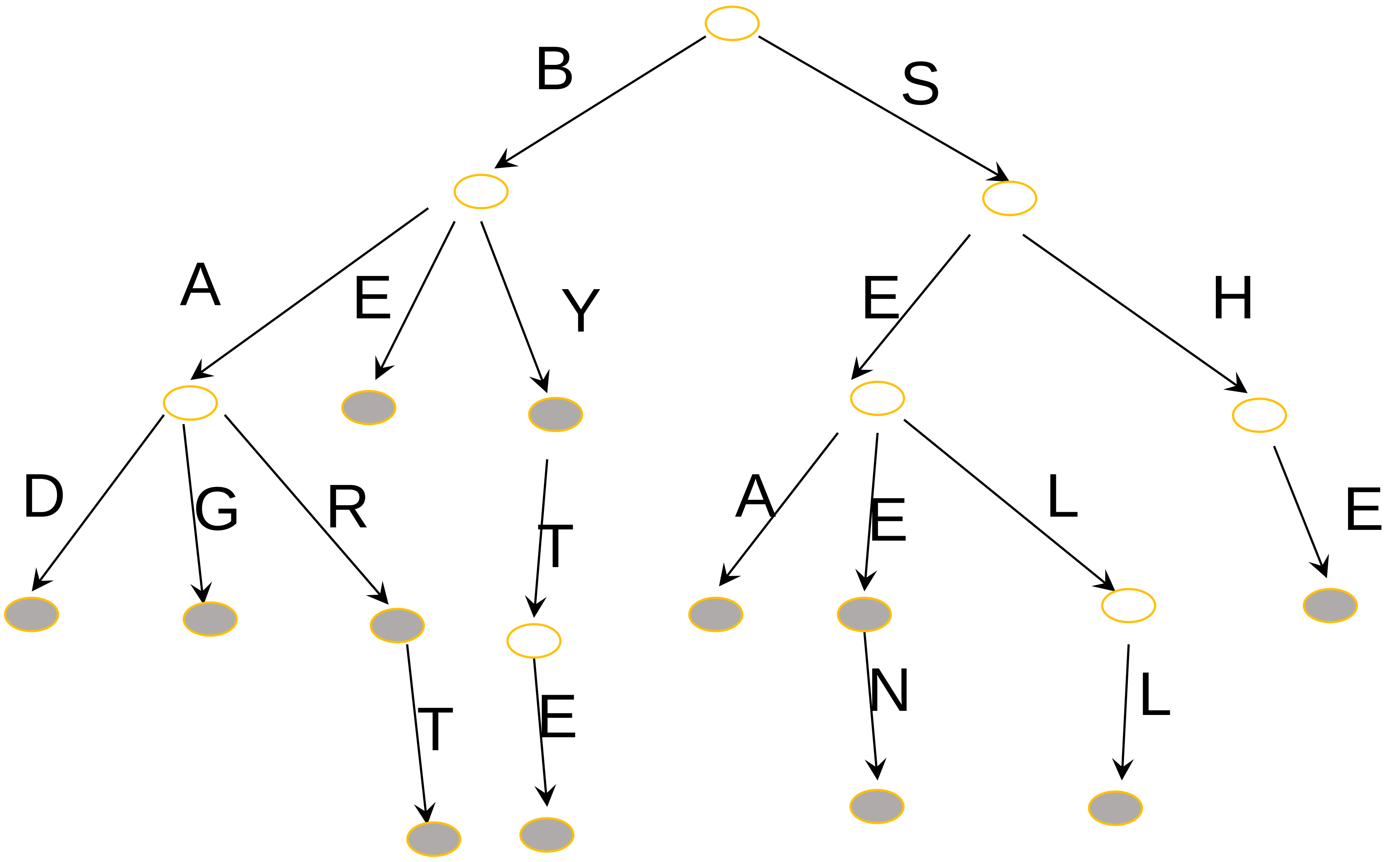


The left tree is a BST since it satisfies the above criterion while the right tree is not a BST.

Tries

- A dictionary structure, used to store a *dynamic set* of *words* that may contain *common prefixes.*
  - *Word*=string of elements from an alphabet;
  - *alphabet*=the set of all possible elements in the words
- Solution:
  - We can exploit the common prefixes of the words, and associate the words(the elements of the set) with *paths in a tree* instead of nodes of a tree
  - Solution: *Trie trees (Prefix trees, Retrieval trees)*
    - *Multipath trees*
    - *If the alphabet has N symbols, a node may have N children*

Trie Tree Example

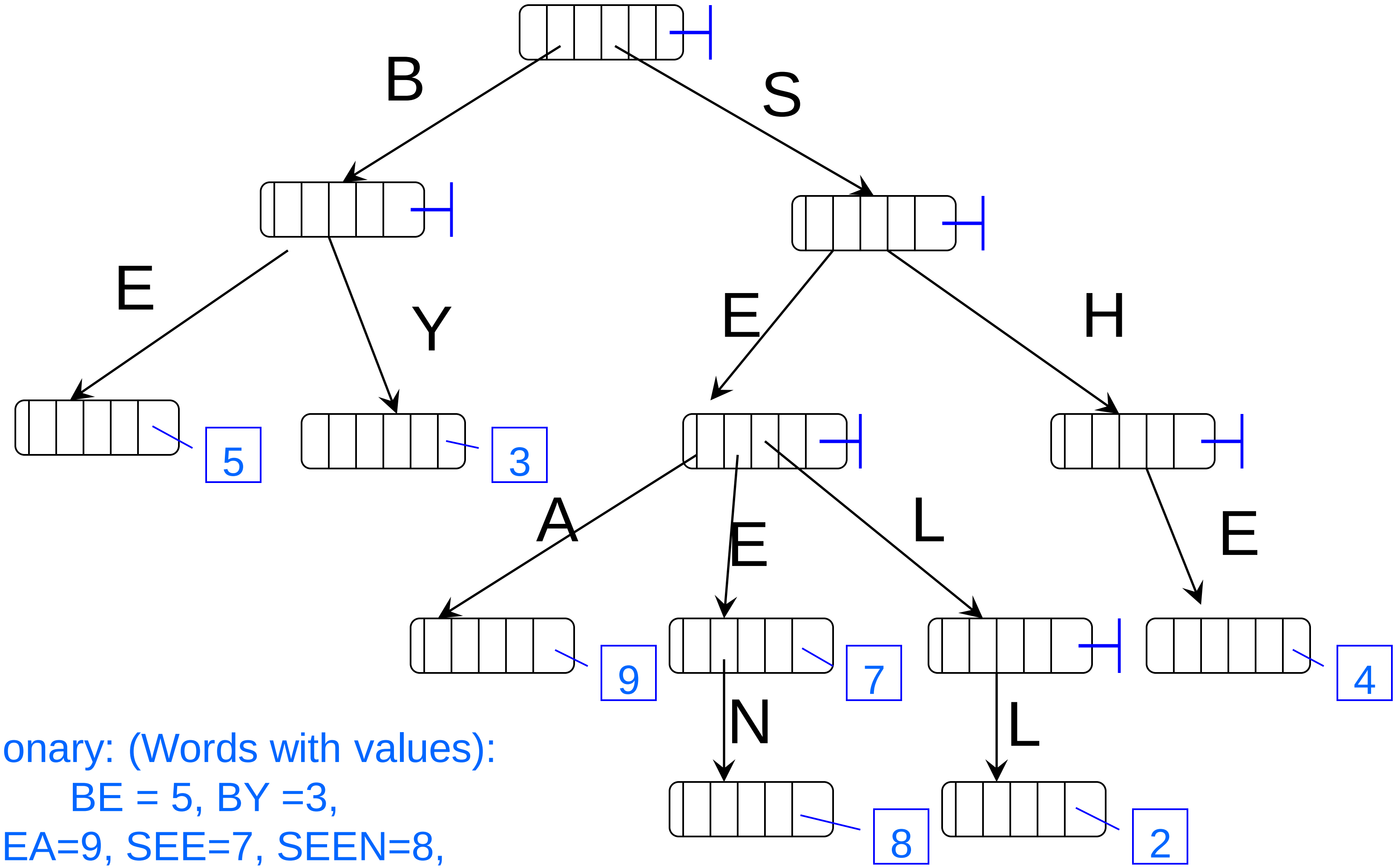


BAD
BAG
BAR
BART
BE
BY
BYTE
SEA
SEE
SEEN
SELL
SHE

Trie Trees

- A trie tree is a tree structure whose edges are labeled with the elements of the alphabet.
- Each node can have up to N children, where N is the size of the alphabet.
- Nodes correspond to strings of elements of the alphabet: each node corresponds to the sequence of elements traversed on the path from the root to that node.
- The dictionary maps a string *s* to a value *v* by storing the value *v* in the node of *s*. If a string that is a prefix of another string in the dictionary is not used, it has nil as its value.
- <u>**Possible implementation**</u>: a trie tree node structure contains an array of N links to child nodes (a link to a child node can be also nil) and a link to the current strings value (it can be also nil).

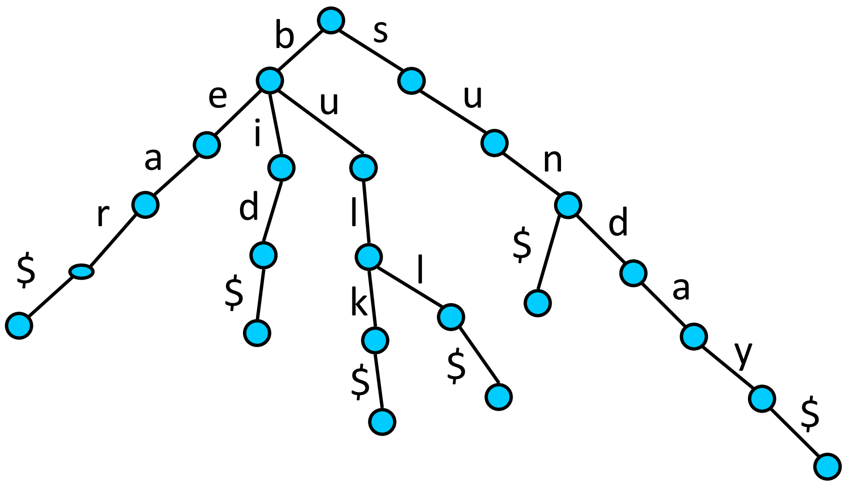
Trie Tree Example



Dictionary: (Words with values):
BE = 5, BY =3,
SEA=9, SEE=7, SEEN=8,
SELL=2, SHE=4

Tries

- *Trie* – a data structure for storing a set of strings (name from the word "re*trie*val"):
  - Let's assume, all strings end with "\$" (not in Σ)



Set of strings: {**`bear`**, **`bid`**, **`bulk`**, **`bull`**, **`sun`**, **`sunday`**}

Tries II

- Properties of a ***trie***:
  - A multi-way tree.
  - Each ***node*** has from 1 to *d* children.
  - Each ***edge*** of the tree is labeled with a character.
  - Each ***leaf*** node corresponds to the stored string, which is a concatenation of characters on a path from the root to this node.

Search and Insertion in Tries

```
Trie-Search(t, P[k..m])  //inserts string P into t
if t is leaf then return true
else if t.child(P[k])=nil then return false
     else return Trie-Search(t.child(P[k]), P[k+1..m])
```

- The search algorithm just follows the path down the tree (starting with Trie-Search(*root*, *P*[0..*m*]))

```
Trie-Insert(t, P[k..m])
if t is not leaf then  //otherwise P is already present
   if t.child(P[k])=nil then
        Create a new child of t and a "branch" starting
         with that chlid and storing P[k..m]
   else Trie-Insert(t.child(P[k]), P[k+1..m])
```

How would the delete work?

Trie Node Structure

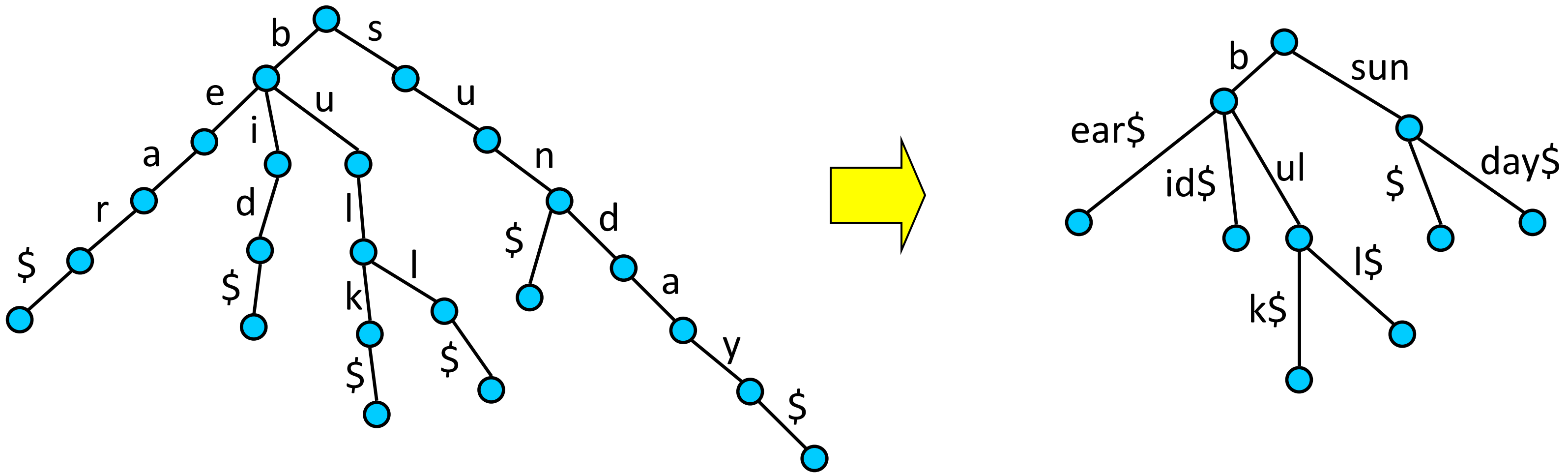
- "Implementation detail"
  - What is the node structure? = What is the complexity of the t.*child*(c) operation?:
    - An **array** of child pointers of size d: waist of space, but *child*(c) is $O(1)$
    - A **hash table** of child pointers: less waist of space, *child*(c) is expected $O(1)$
    - A **list** of child pointers: compact, but *child*(c) is $O(d)$ in the worst-case
    - A **binary search tree** of child pointers: compact and *child*(c) is $O(\lg d)$ in the worst-case

Analysis of the Trie

- Size:
  - $O(N)$ in the worst-case
- Search, insertion, and deletion (string of length m):
  - depending on the node structure: $O(dm)$, $O(m \lg d)$, $O(m)$
  - Compare with the string BST
- Observation:
  - Having chains of one-child nodes is wasteful

Compact Tries

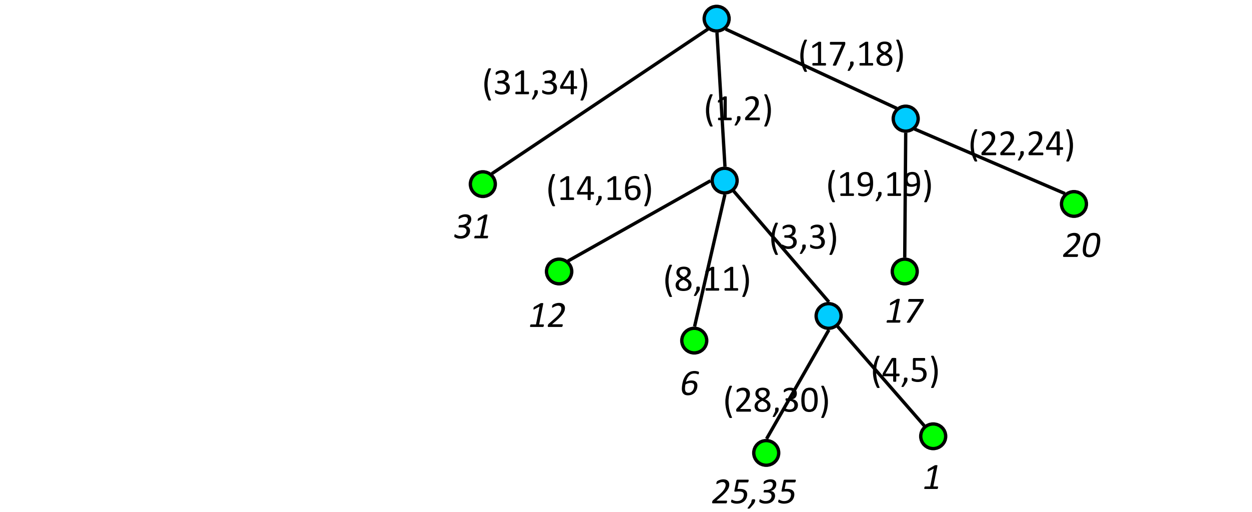
- *Compact Trie*:
  - Replace a *chain* of one-child nodes with an edge labeled with a string
  - Each non-leaf node (except root) has at least two children



Compact Tries II

- Implementation:
  - Strings are external to the structure in one array, edges are labeled with indices in the array (*from*, *to*)
- Can be used to do *word matching*: find where the given word appears in the text.
  - Use the compact trie to “store” all words in the text
  - Each child in the compact trie has a list of indices in the text where the corresponding word appears.

Word Matching with Tries



- To find a word *P*:
  - At each node, follow edge (*i*,*j*), such that *P*[*i*..*j*] = T[i..j]
  - If there is no such edge, there is no *P* in *T*, otherwise, find all starting indices of *P* when a leaf is reached

Trie Trees: Some usages

- Predictive text (autocomplete features)
- In dictionary-based compression algorithms (LZW)
- The *suffix trie* for a text supports arbitrary *pattern matching*
- The *index of a search engine* (collection of all searchable words) is stored into a compressed trie
    - Each leaf of the trie is associated with a word and has a list of pages (URLs) containing that word, called *occurrence list*
    - The trie is kept in internal memory
    - The occurrence lists are kept in external memory and are ranked by relevance
    - Boolean queries for sets of words (e.g., Java and coffee) correspond to set operations (e.g., intersection) on the occurrence lists
    - Additional *information retrieval* techniques are used, such as stop-word elimination (e.g., ignore “the” “a” “is”); stemming (e.g., identify “add” “adding” “added”); and so on…
- A router forwards packets to its neighbors using IP *prefix matching* rules. E.g., a packet with IP prefix 128.148. should be forwarded to the appropriate gateway router. The routers use tries on the alphabet 0,1 to do prefix matching.