DESIGN AND ANALYSIS OF ALGORITHMS

Surabhi Narayan

Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Transform and Conquer

Surabhi Narayan

Department of Computer Science & Engineering

Transform and Conquer:

- In Transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution.
- Then, in the second or conquering stage, it is solved.

There are three major variations

- Transformation to a simpler or more convenient instance of the same problem—we call it ***instance simplification***.
- Transformation to a different representation of the same instance—we call it ***representation change***.
- Transformation to an instance of a different problem for which an algorithm is already available—we call it ***problem reduction***.

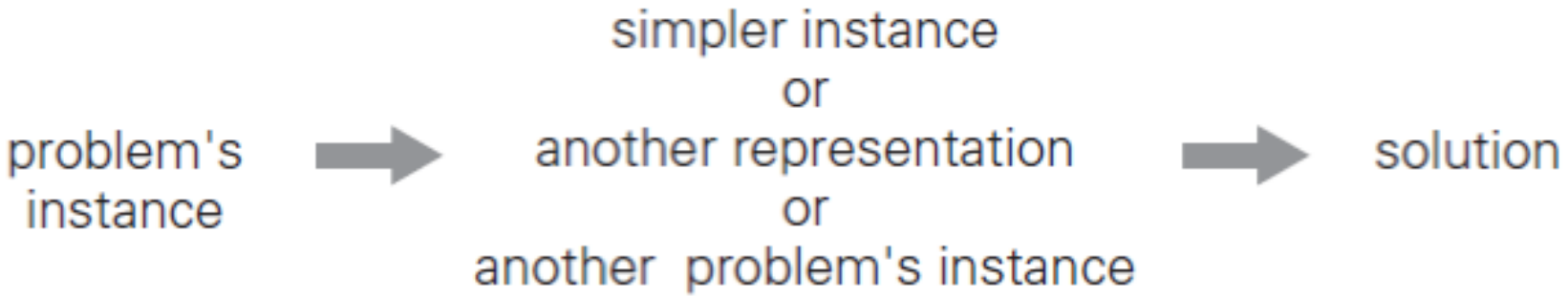


FIGURE 6.1 Transform-and-conquer strategy.

Checking element uniqueness in an array

The brute-force algorithm compared pairs of the array's elements until either two equal elements were found or no more pairs were left. Its worst-case efficiency was in (n^2).

Alternatively, we can sort the array first and then check only its consecutive elements: if the array has equal elements, a pair of them must be next to each other, and vice versa.

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

//Solves the element uniqueness problem by sorting the array first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise

sort the array A

for $i \leftarrow 0$ **to** $n-2$ **do**
if $A[i] = A[i+1]$ **return false**
return true

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n).$$

THANK YOU

Surabhi Narayan
Department of Computer Science &Engineering

surabhinarayan@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS

Surabhi Narayan

Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Heap and Heap Sort

Surabhi Narayan
Department of Computer Science & Engineering

Heap and Heap Sort

Heap is partially ordered data structure that is especially suitable for implementing priority queues.

A ***priority queue*** is a multiset of items with an orderable characteristic called an item's ***priority***, with the following operations:

- finding an item with the highest (i.e., largest) priority
- deleting an item with the highest priority
- adding a new item to the multiset

Heap and Heap Sort

DEFINITION : A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

1. The ***shape property***—the binary tree is ***essentially complete*** (or simply ***complete***), i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.

2. The ***parental dominance*** or ***heap property***—the key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)

Heap and Heap Sort

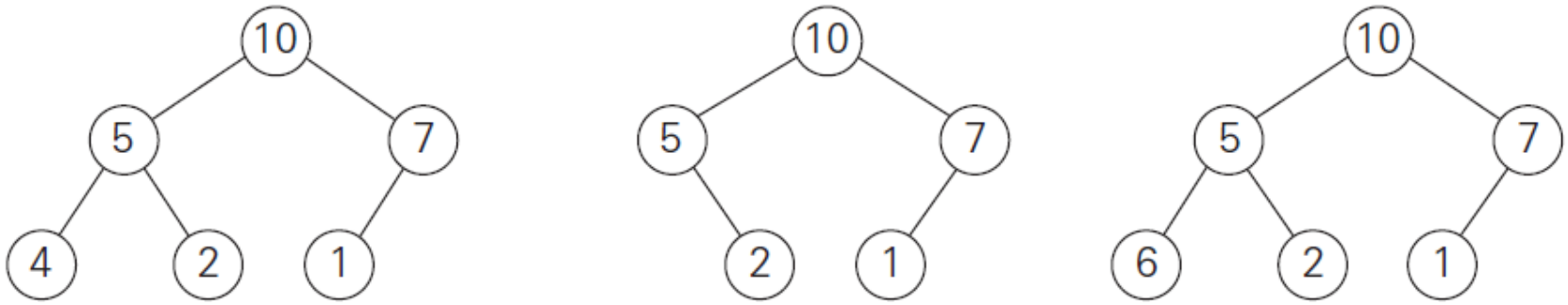


FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

Heap and Heap Sort

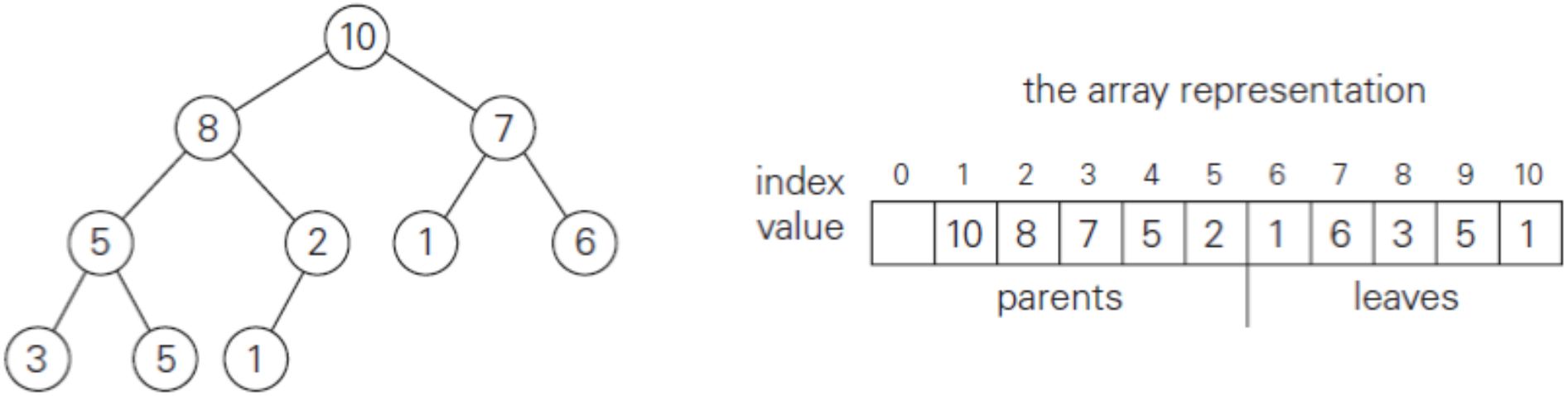


FIGURE 6.10 Heap and its array representation.

Properties of Heap

There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\log_2 n$.

2. The root of a heap always contains its largest element.

3. A node of a heap considered with all its descendants is also a heap.

4. A heap can be implemented as an array by recording its elements in the topdown, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,

a. the parental node keys will be in the first $n/2$ positions of the array, while the leaf keys will occupy the last $n/2$ positions;

b. the children of a key in the array's parental position i $(1 \le i \le n/2)$ will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i $(2 \le i \le n)$ will be in position $i/2$.

Heap and Heap Sort

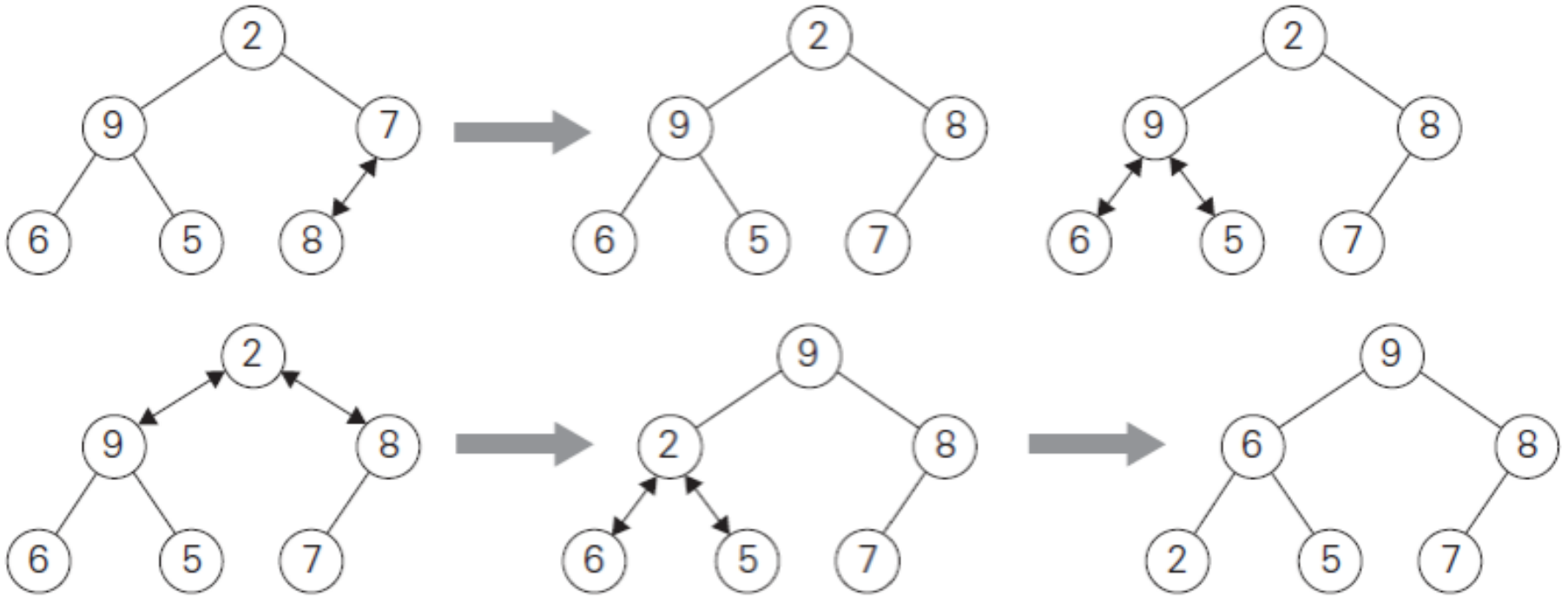


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

Heap and Heap Sort

ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n    //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];    k ← j
    H[k] ← v
```

Top Down Approach

- Insert a new key *K* into a heap by attaching a new node with key *K* in it after the last leaf of the existing heap.
- Sift *K* up to its appropriate place in the new heap as follows.
  - Compare *K* with its parent's key: if the latter is greater than or equal to *K*, stop (the structure is a heap);
  - otherwise, swap these two keys and compare *K* with its new parent.
- This swapping continues until *K* is not greater than its last parent or it reaches the root
- Insertion operation cannot require more key comparisons than the heap's height.
- The time efficiency of insertion is in $O(\log n)$.

Heap and Heap Sort

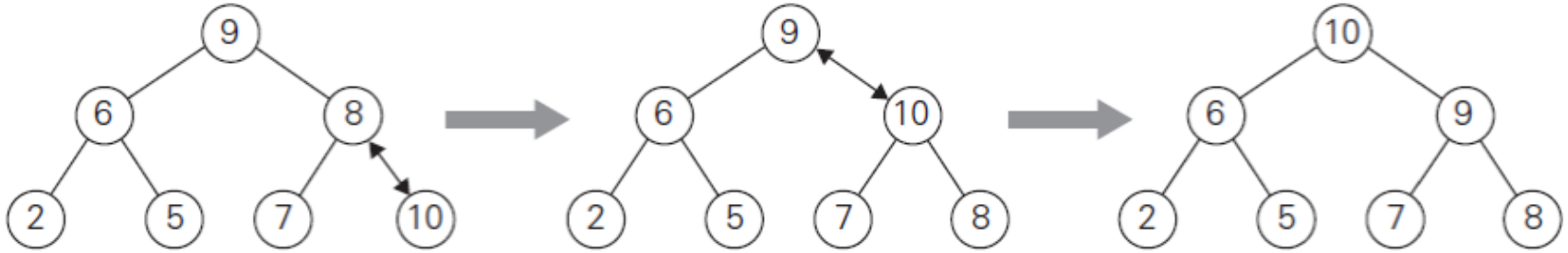


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

Heap and Heap Sort

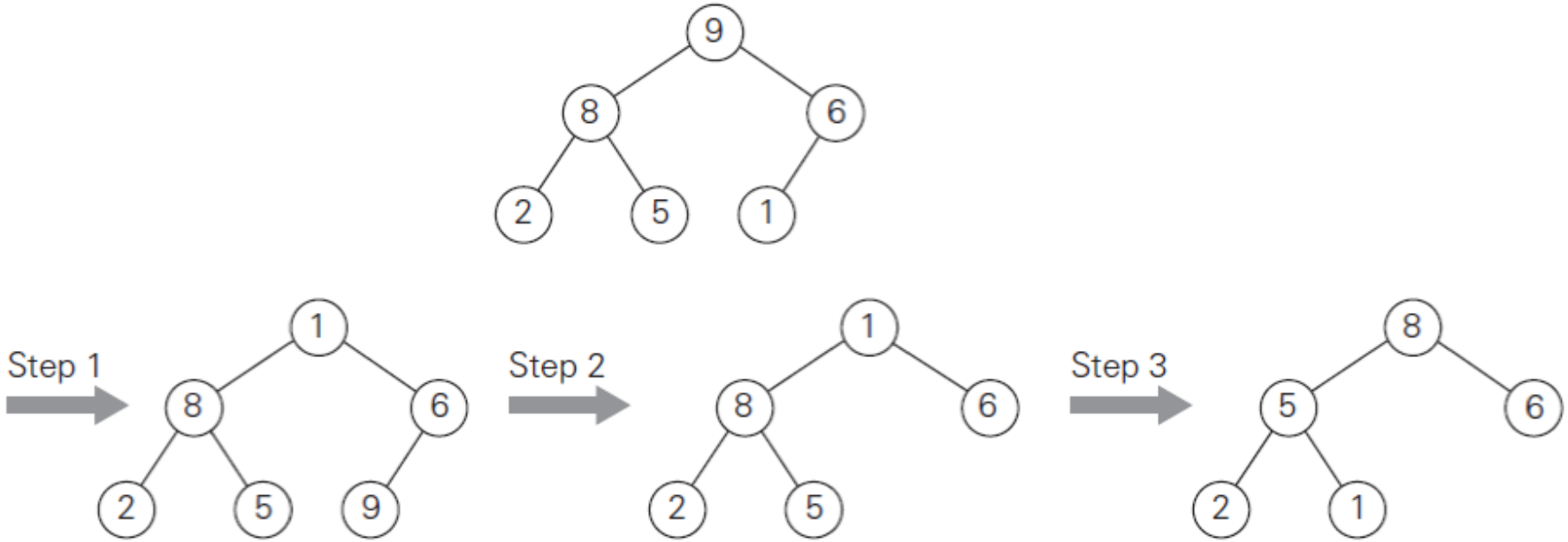


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

Maximum Key Deletion from a heap

Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

Heap Sort

Two-stage algorithm that works as follows:

Stage 1 (heap construction): Construct a heap for a given array.
Stage 2 (maximum deletions): Apply the root-deletion operation $n - 1$ times to the remaining heap.

As a result, the array elements are eliminated in decreasing order. But since under the array implementation of heaps an element being deleted is placed last, the resulting array will be exactly the original array sorted in increasing order.

Heap and Heap Sort

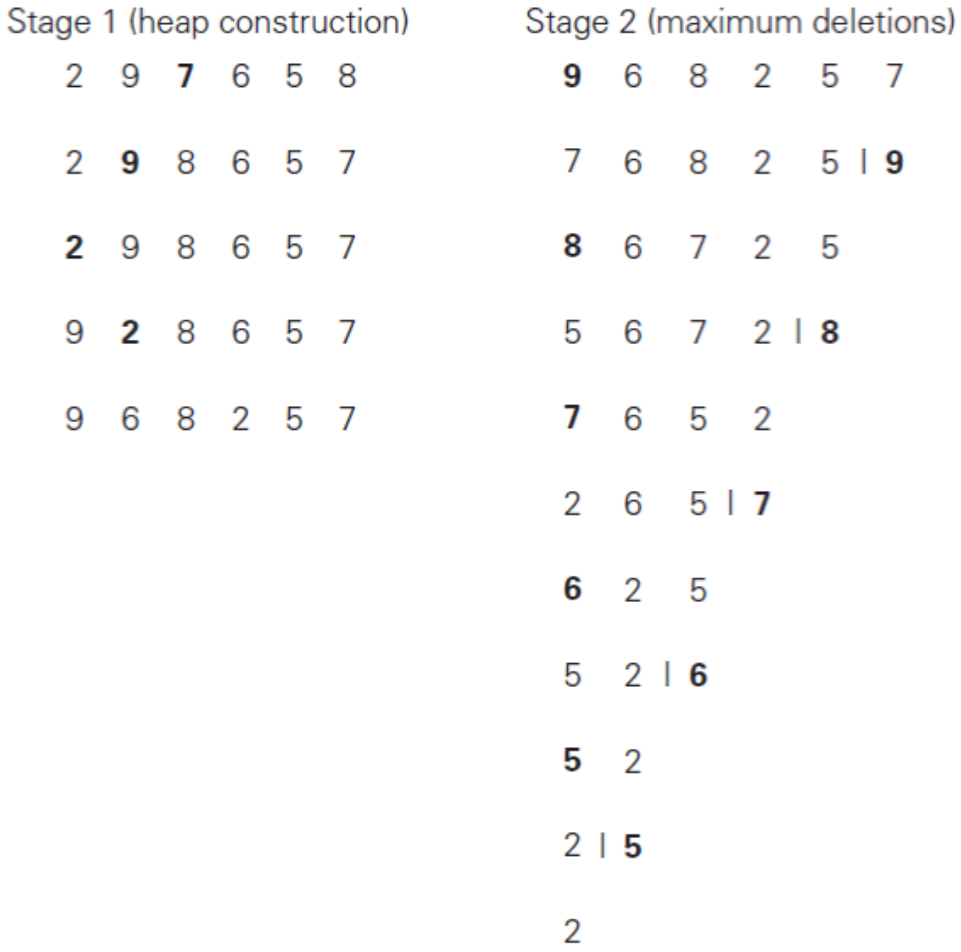


FIGURE 6.14 Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

Complexity Analysis of Heap Sort

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2 \sum_{i=1}^{n-1} \log_2 i$$

$$\le 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1) \log_2(n-1) \le 2n \log_2 n.$$

THANK YOU

Surabhi Narayan
Department of Computer Science &Engineering

surabhinarayan@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS

Surabhi Narayan

Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

RED BLACK TREE

Surabhi Narayan

Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Red Black Tree

Balanced Binary Search Tree

Revisit AVL Tree

DESIGN AND ANALYSIS OF ALGORITHMS

Red Black Tree

Balanced Binary Search Tree

Revisit AVL Tree

DESIGN AND ANALYSIS OF ALGORITHMS

Red Black Tree

Balanced Binary Search Tree

Revisit AVL Tree

Red Black Tree

- A Red Black Tree is a category of the self-balancing binary search tree.
- Created in 1972 by Rudolf Bayer who termed them **"symmetric binary B-trees**."
- A red-black tree is a Binary tree where a particular node has color as an extra attribute, either red or black.
- These colours are used to ensure that the tree remains balanced during insertions and deletions.
- Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree.

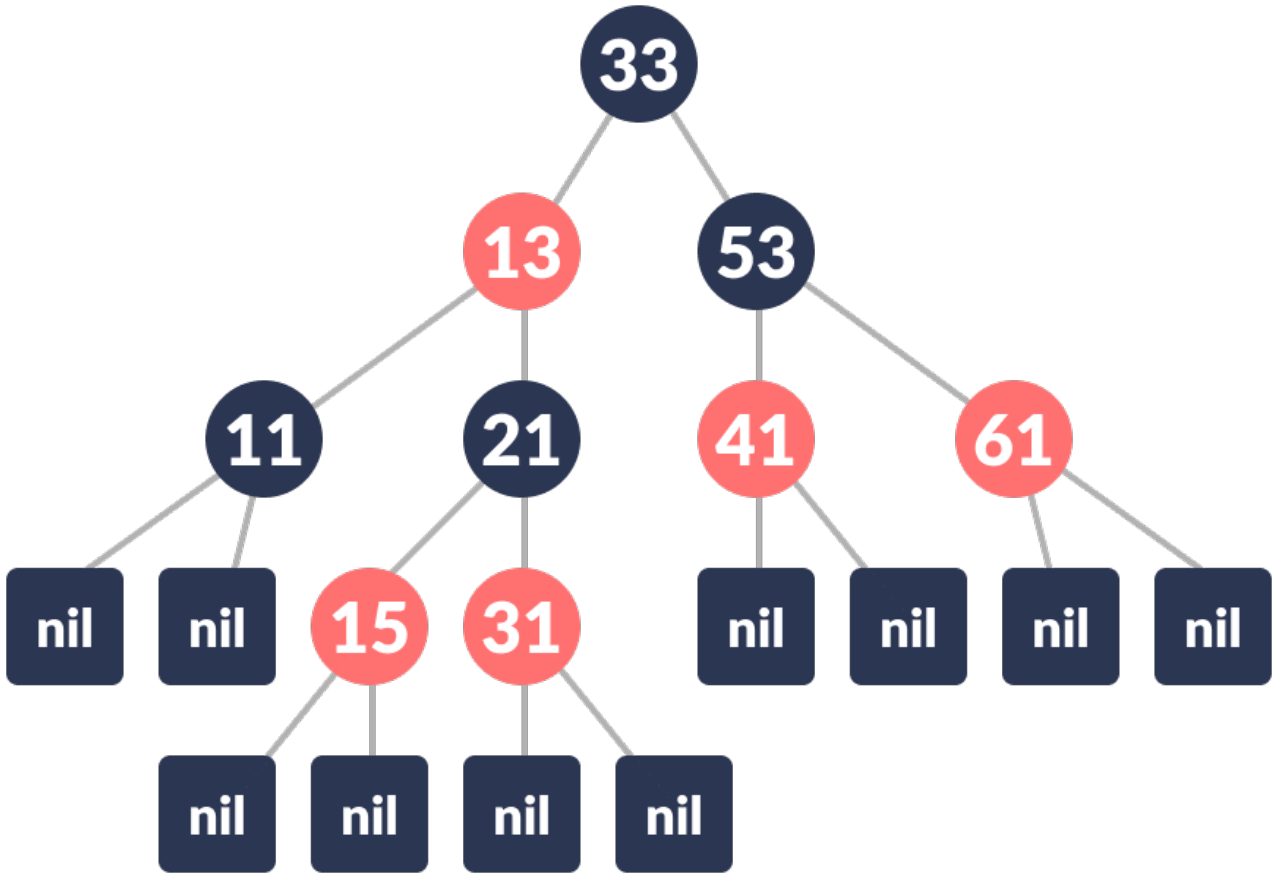
Red Black Tree

A red-black tree satisfies the following properties:

- **Red/Black Property:** Every node is colored, either red or black.
- **Root Property:** The root is black.
- **Leaf Property:** Every leaf (NIL) is black.
- **Red Property:** If a red node has children then, the children are always black.
- **Depth Property:** For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).

Red Black Tree

Red Black Tree



Each node has the following attributes:

- Color
- key
- leftChild
- rightChild
- parent (except root node)

Interesting points about Red-Black Tree:

- Black height of the red-black tree is the number of black nodes on a path from the root node to a leaf node. Leaf nodes are also counted as black nodes. So, a red-black tree of height h has black height >= h/2. *Number of nodes from a node to its farthest descendant leaf is no more than twice as the number of nodes to the nearest descendant leaf.*

- Height of a red-black tree with n nodes is $h <= 2 \log_2(n + 1)$.
- All leaves (NIL) are black.
- The black depth of a node is defined as the number of black nodes from the root to that node i.e the number of black ancestors.
- Every red-black tree is a special case of a binary tree.

Applications:

- Most of the self-balancing BST library functions like map and set in C++ (OR TreeSet and TreeMap in Java) use Red-Black Tree.
- It is used to implement CPU Scheduling Linux. Completely Fair Scheduler uses it.
- Besides they are used in the K-mean clustering algorithm for reducing time complexity.
- Moreover, MySQL also uses the Red-Black tree for indexes on tables.

Operations on a Red-Black Tree

- insert a key value (insert)
- determine whether a key value is in the tree (lookup/search)
- remove key value from the tree (delete)
- print all of the key values in sorted order (print)

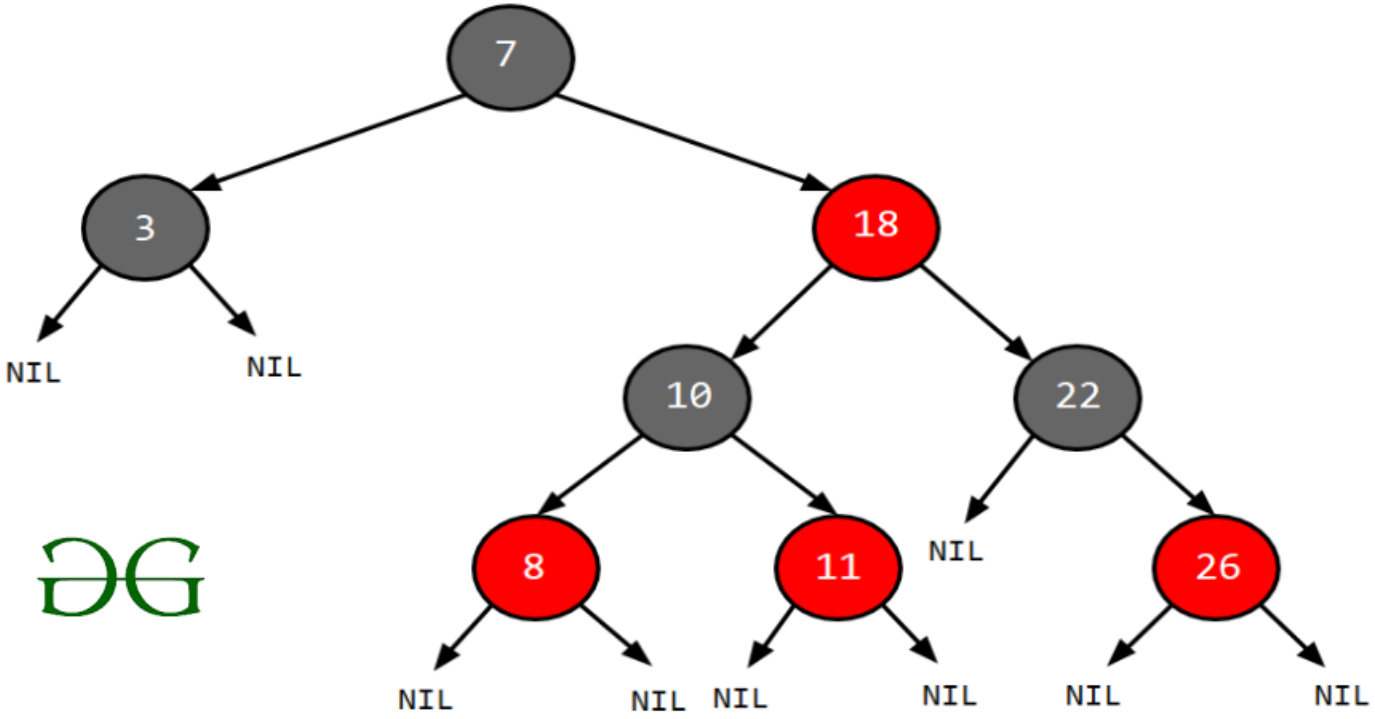
Search in Red-black Tree:

As every red-black tree is a special case of a binary tree so the searching algorithm of a red-black tree is similar to that of a binary tree.

```
searchElement (tree, val)
Step 1: If tree -> data = val OR tree = NULL
Return tree
    Else If val data
            Return searchElement (tree -> left, val)
        Else Return searchElement (tree -> right, val)
        [ End of if ]
[ End of if ]

Step 2: END
```

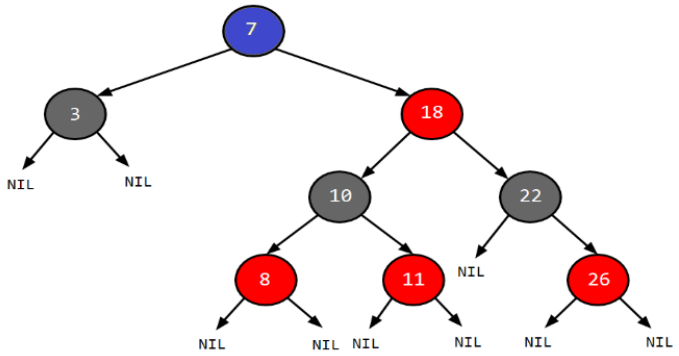
Search in Red-black Tree: search for the key 11



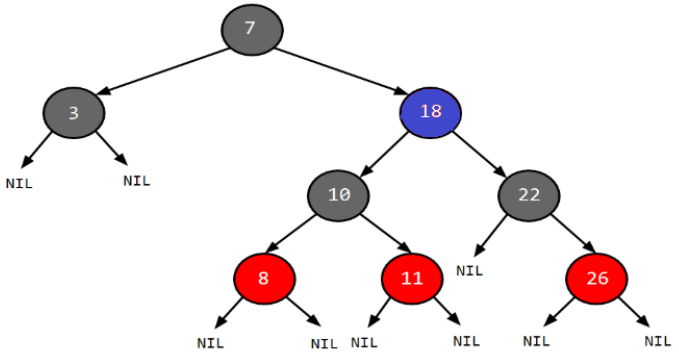
1.Start from the root.
2.Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3.If the element to search is found anywhere, return true, else return false.

Search in Red-black Tree: search for the key 11

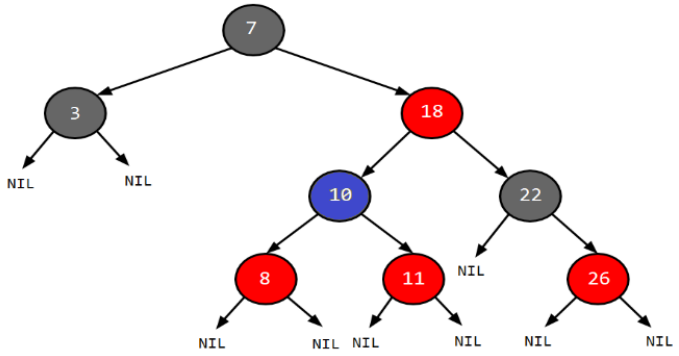
Step-1



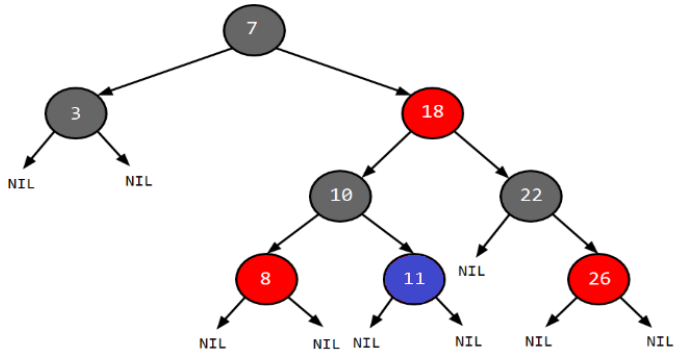
Step-2



Step-3



Step-4



Red Black Tree - Insertion

Let x be the newly inserted node.

1.Perform standard BST insertion and make the colour of newly inserted nodes as RED.

2.If x is the root, change the colour of x as BLACK (Black height of complete tree increases by 1).

3.Do the following if the color of x's parent is not BLACK **and** x is not the root.

a) If x's uncle is RED (Grandparent must have been black from property 4)

(i) Change the colour of parent and uncle as BLACK.

(ii) Colour of a grandparent as RED.

(iii) Change x = x's grandparent, repeat steps 2 and 3 for new x.

b) If x's uncle is BLACK, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**) (This is similar to AVL Tree)

(i) Left Left Case (p is left child of g and x is left child of p)

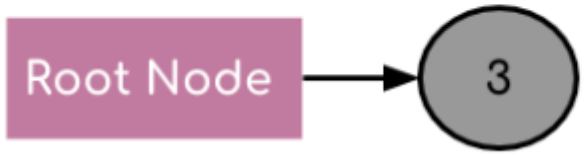
(ii) Left Right Case (p is left child of g and x is the right child of p)

(iii) Right Right Case (Mirror of case i)

(iv) Right Left Case (Mirror of case ii)

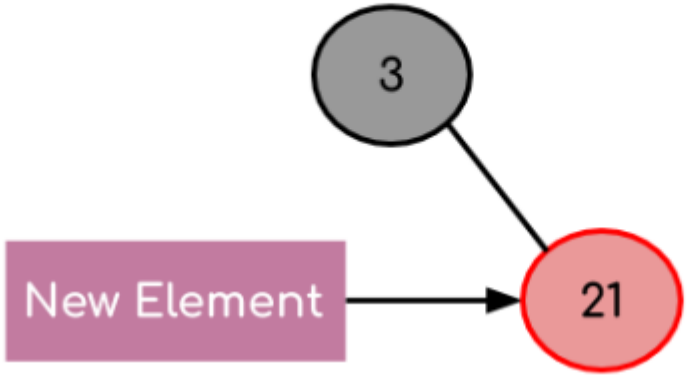
Create a red-black tree with elements 3, 21, 32 and 15.

Step 1: Inserting element 3 inside the tree.



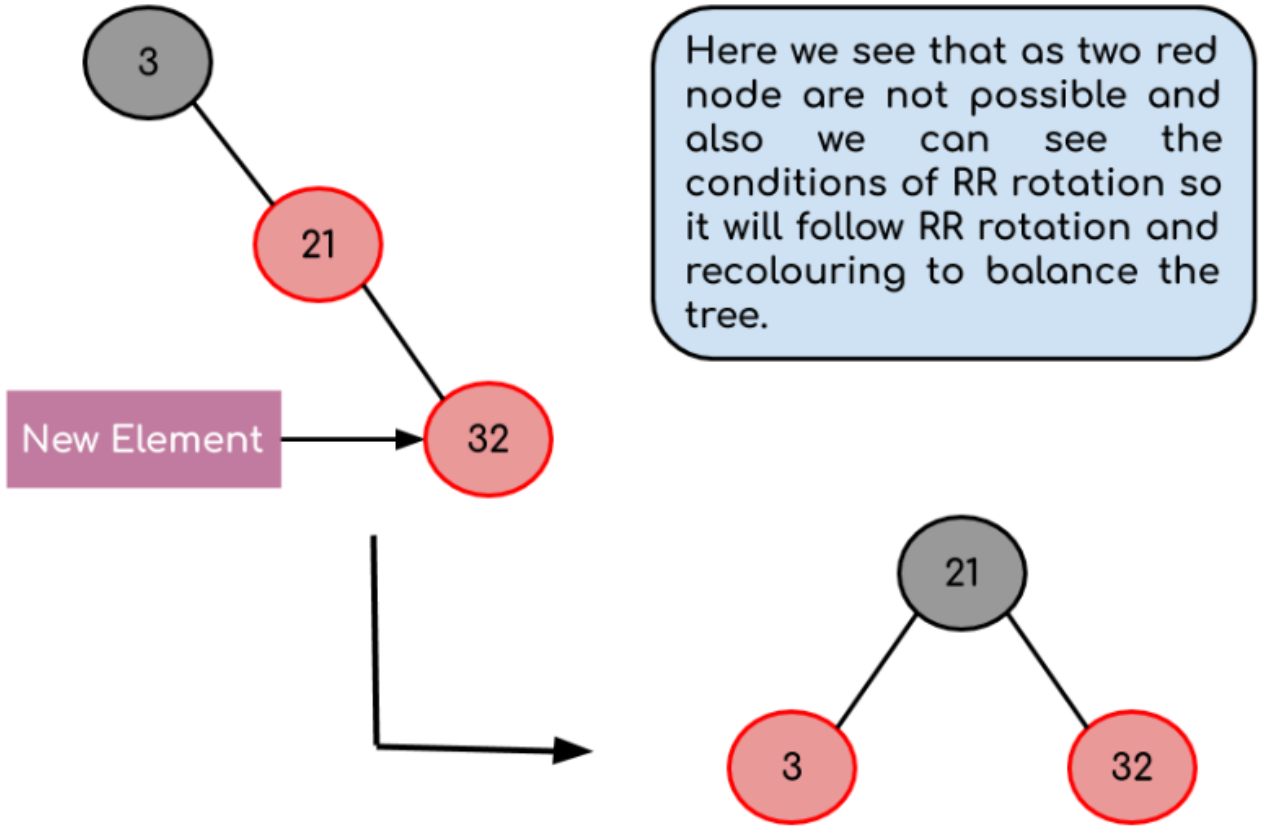
Create a red-black tree with elements 3, 21, 32 and 15.

Step 2: Inserting element 21 inside the tree.



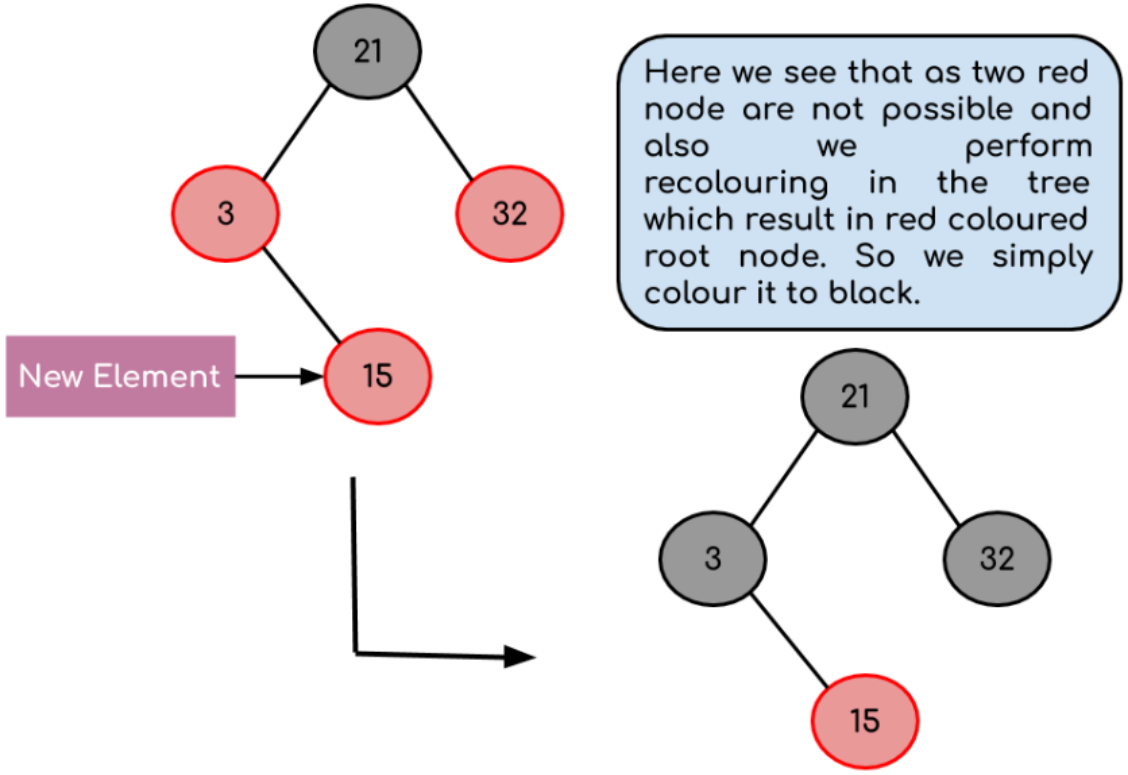
Create a red-black tree with elements 3, 21, 32 and 15.

Step 3: Inserting element 32 inside the tree.



Create a red-black tree with elements 3, 21, 32 and 15.

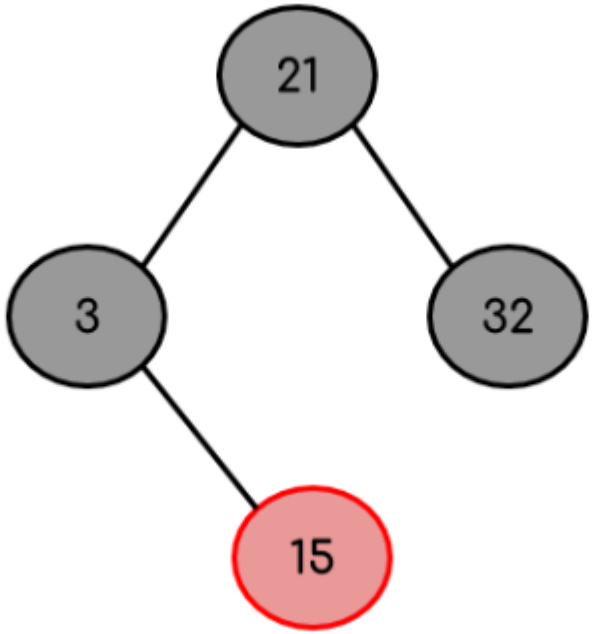
Step 4: Inserting element 17 inside the tree.



Red Black Tree - Insertion

PES UNIVERSITY ONLINE

Create a red-black tree with elements 3, 21, 32 and 15.



Red Black Tree - Insertion

Create a red-black tree with elements 1,2,3,4,5,6,7,8,9,10,11,12,13.

THANK YOU

Surabhi Narayan
Department of Computer Science &Engineering

surabhinarayan@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS

Surabhi Narayan

Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

2-3 Trees

Surabhi Narayan

Department of Computer Science & Engineering

2-3 Tree

2-3 tree is a tree that can have nodes of two kinds: 2-nodes and 3-nodes.

- A ***2-node*** contains a single key*K* and has two children:
  the left child serves as the root of a subtree whose keys are less than *K*, and the right child serves as the root of a subtree whose keys are greater than *K*.

- A ***3-node*** contains two ordered keys *K*1 and *K*2 (*K*1<*K*2*)* and has three children.

- The leftmost child serves as the root of a subtree with keys less than *K*1, the middle child serves as the root of a subtree with keys between *K*1 and *K*2, and the rightmost child serves as the root of a subtree with keys greater than *K*2
- The last requirement of the 2-3 tree is that all its leaves must be on the same level. In other words, a 2-3 tree is always perfectly height-balanced: the length of a path from the root to a leaf is the same for every leaf.

2-3 Tree

Here are the properties of a 2-3 tree:

- each node has either one value or two value
- a node with one value is either a leaf node or has exactly two children (non-null). Values in left subtree < value in node < values in right subtree
- a node with two values is either a leaf node or has exactly three children (non-null). Values in left subtree < first value in node < values in middle subtree < second value in node < value in right subtree.
- all leaf nodes are at the same level of the tree

2-3 Tree

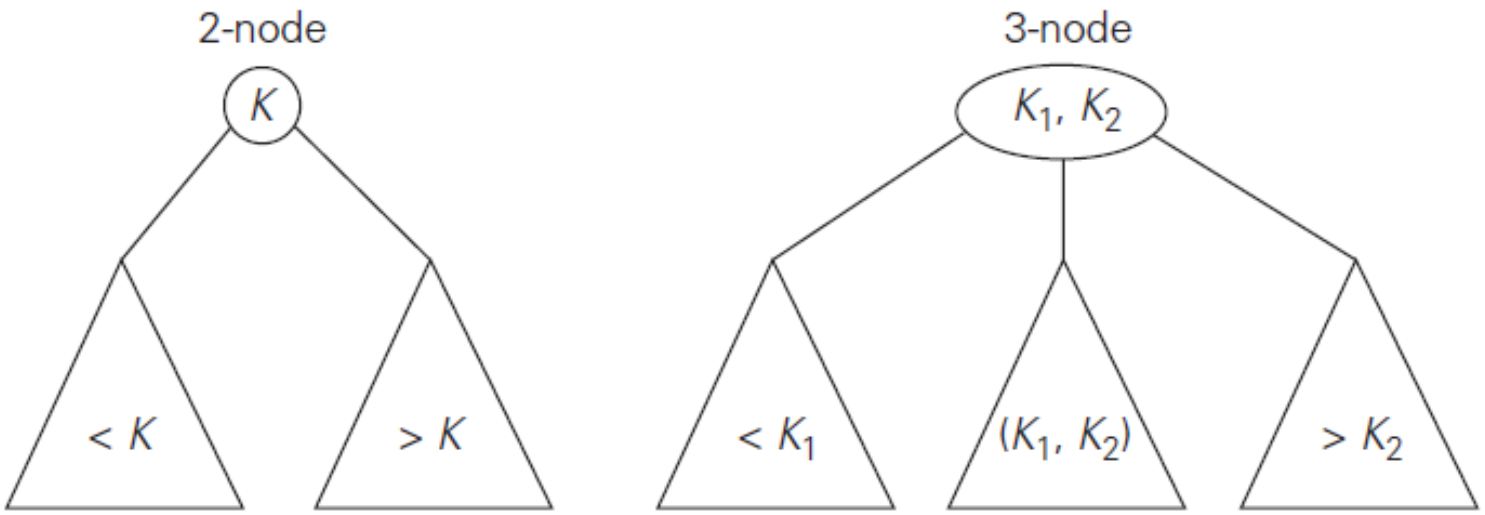


FIGURE 6.7 Two kinds of nodes of a 2-3 tree.

2-3 Tree

Search: To search a key **K** in given 2-3 tree **T**, we follow the following procedure:

Base cases:

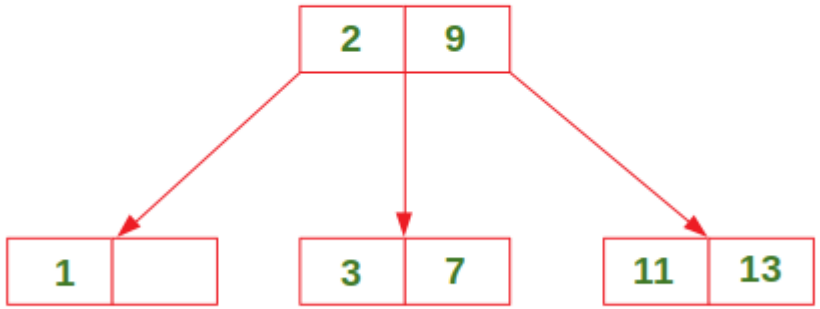
1. If **T** is empty, return False (key cannot be found in the tree).
2. If current node contains data value which is equal to **K**, return True.
3. If we reach the leaf-node and it doesn't contain the required key value **K**, return False.

Recursive Calls:

1. If **K** < currentNode.leftVal, we explore the left subtree of the current node.
2. Else if currentNode.leftVal < **K** < currentNode.rightVal, we explore the middle subtree of the current node.
3. Else if **K** > currentNode.rightVal, we explore the right subtree of the current node.

2-3 Tree

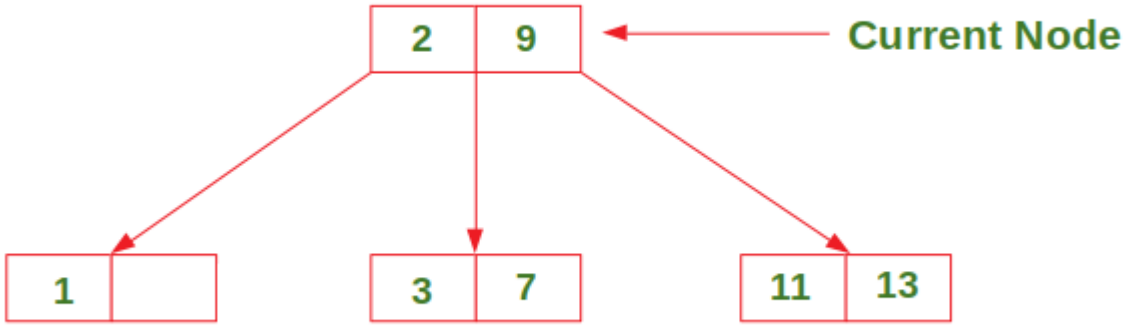
Search 5 in the following 2-3 Tree:



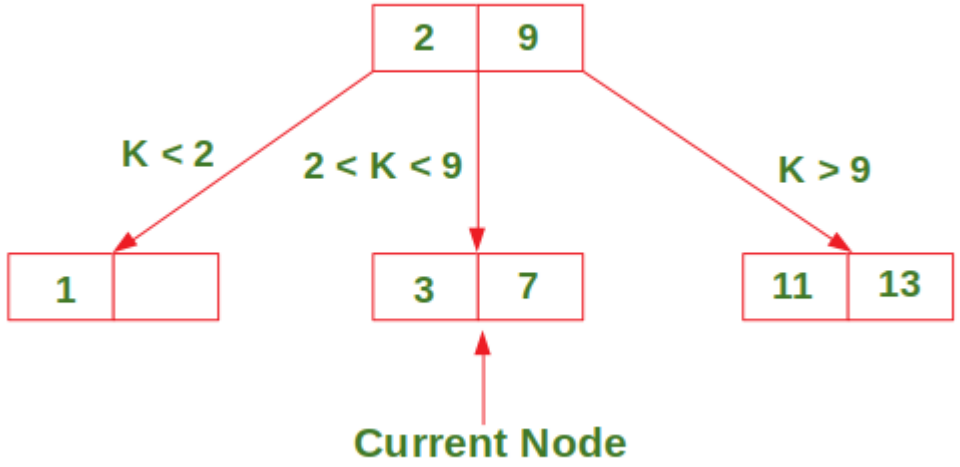
2-3 Tree

PES UNIVERSITY ONLINE

Search 5

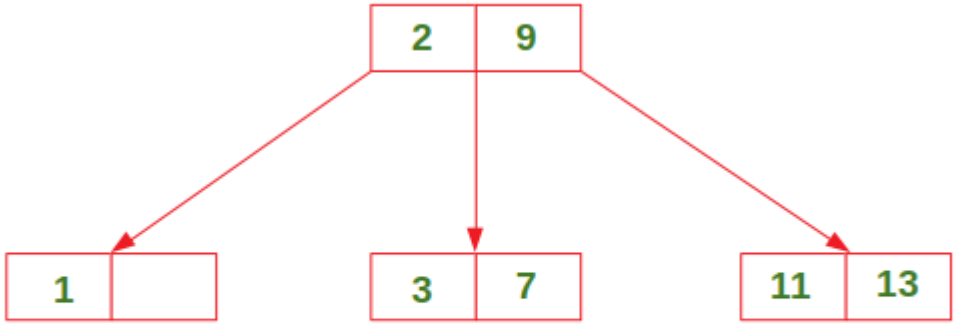


Search 5



2-3 Tree

Search 5



5 Not Found. Return False

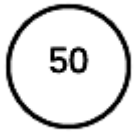
Insertion

The insertion algorithm into a two-three tree is quite different from the insertion algorithm into a binary search tree. In a two-three tree, the algorithm will be as follows:

- If the tree is empty, create a node and put value into the node
- Otherwise find the leaf node where the value belongs.
- If the leaf node has only one value, put the new value into the node
- If the leaf node has more than two values, split the node and promote the median of the three values to parent.
- If the parent then has three values, continue to split and promote, forming a new root node if necessary

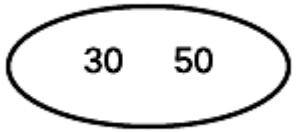
2-3 Tree

Insert 50



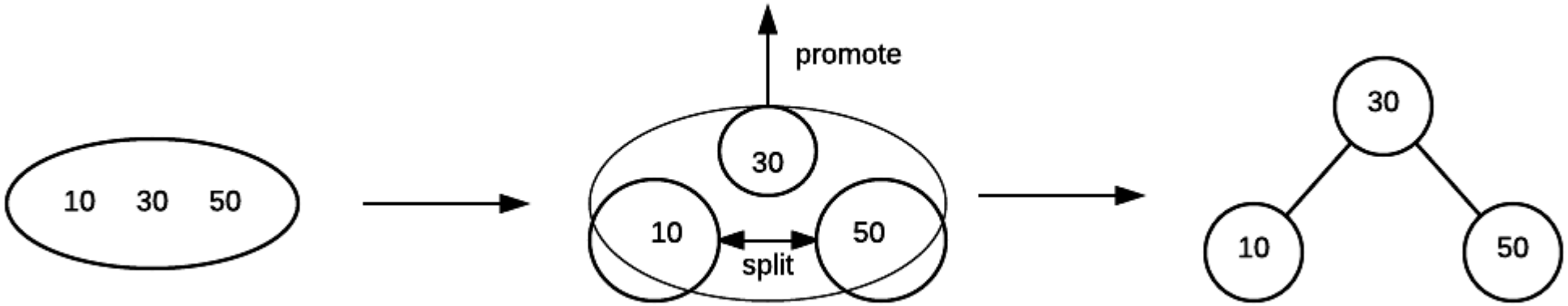
2-3 Tree

Insert 30



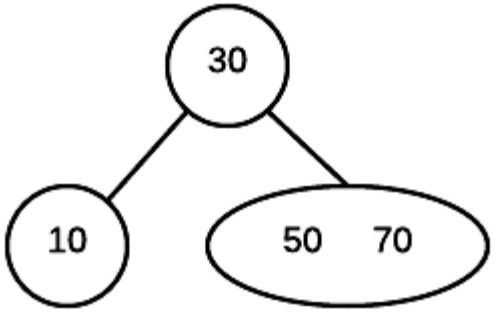
2-3 Tree

Insert 10



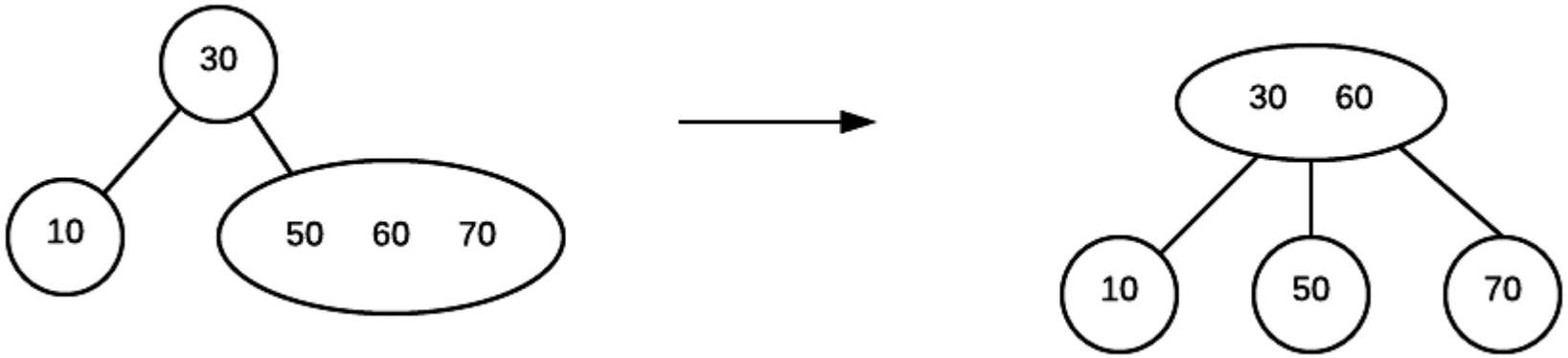
2-3 Tree

Insert 70



2-3 Tree

Insert 60



DESIGN AND ANALYSIS OF ALGORITHMS

2-3 Tree

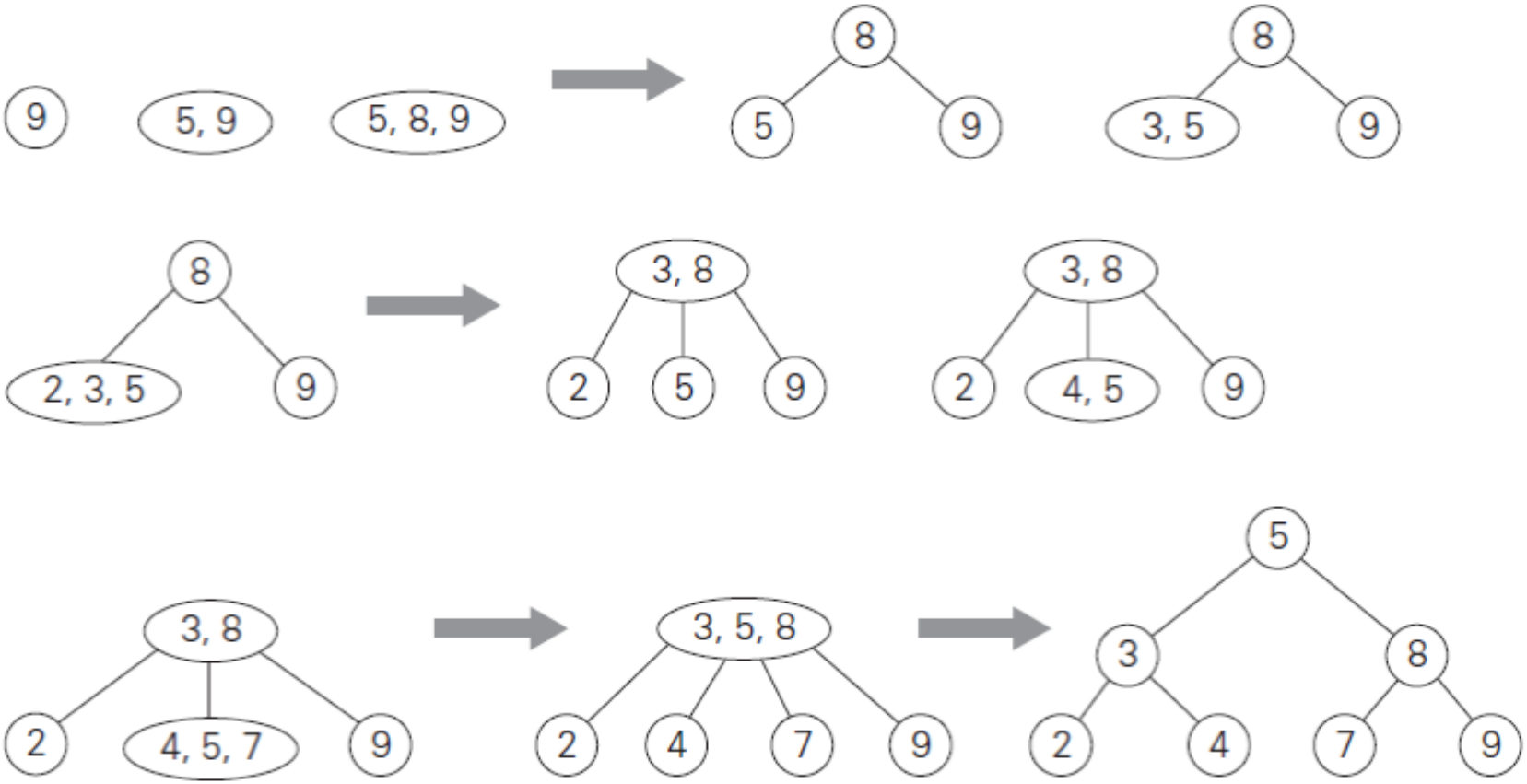


FIGURE 6.8 Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

Time Complexity

The property of being perfectly balanced, enables the **2-3 Tree** operations of insert, delete and search to have a time complexity of O(log (n)).

THANK YOU

Surabhi Narayan
Department of Computer Science &Engineering

surabhinarayan@pes.edu

DESIGN AND ANALYSIS OF ALGORITHMS

Surabhi Narayan

Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

B Trees

Surabhi Narayan

Department of Computer Science & Engineering

B Tree

- Self-balancing search tree
- Each node can contain more than one key
- Can have more than two children.
- All data records (or record keys) are stored at the leaves, in increasing order of the keys each parental node contains $n - 1$ ordered keys.

It is also known as a height-balanced m-way tree.

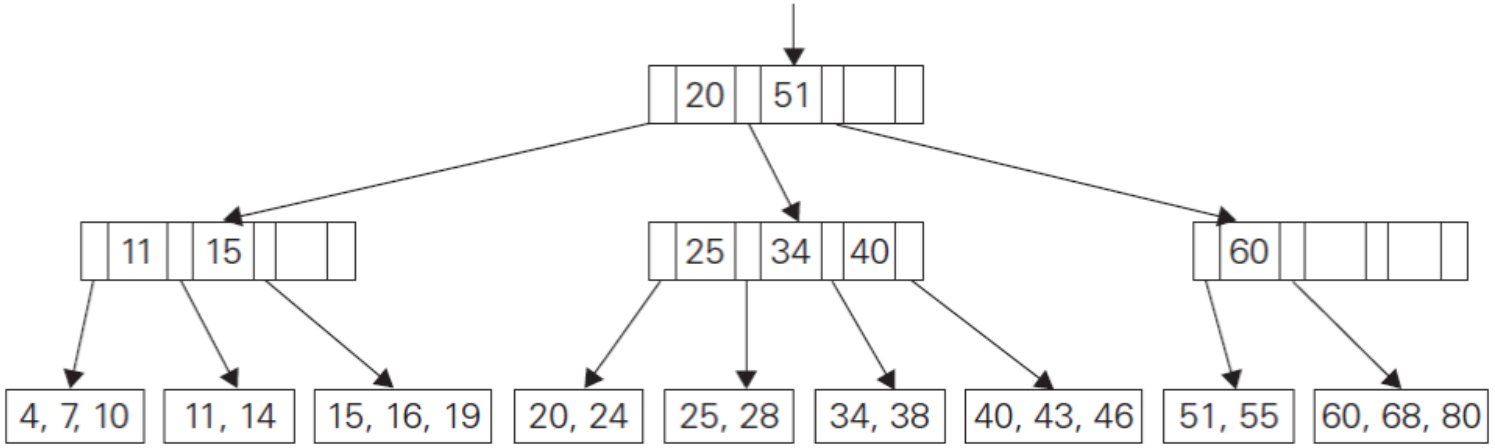


FIGURE 7.8 Example of a B-tree of order 4.

B Tree

Structural properties:

- All leaves are at the same level.
- For each node x, the keys are stored in increasing order.
- If n is the order of the tree, each internal node can contain at most n - 1 keys along with a pointer to each child.
- Each node except root can have at most n children and at least n/2 children.
- All leaves have the same depth (i.e. height-h of the tree).
- The root has at least 2 children and contains a minimum of 1 key.
- If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2$, $h \geq \log_t (n+1)/2$
- B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).

Operations on a B Tree:

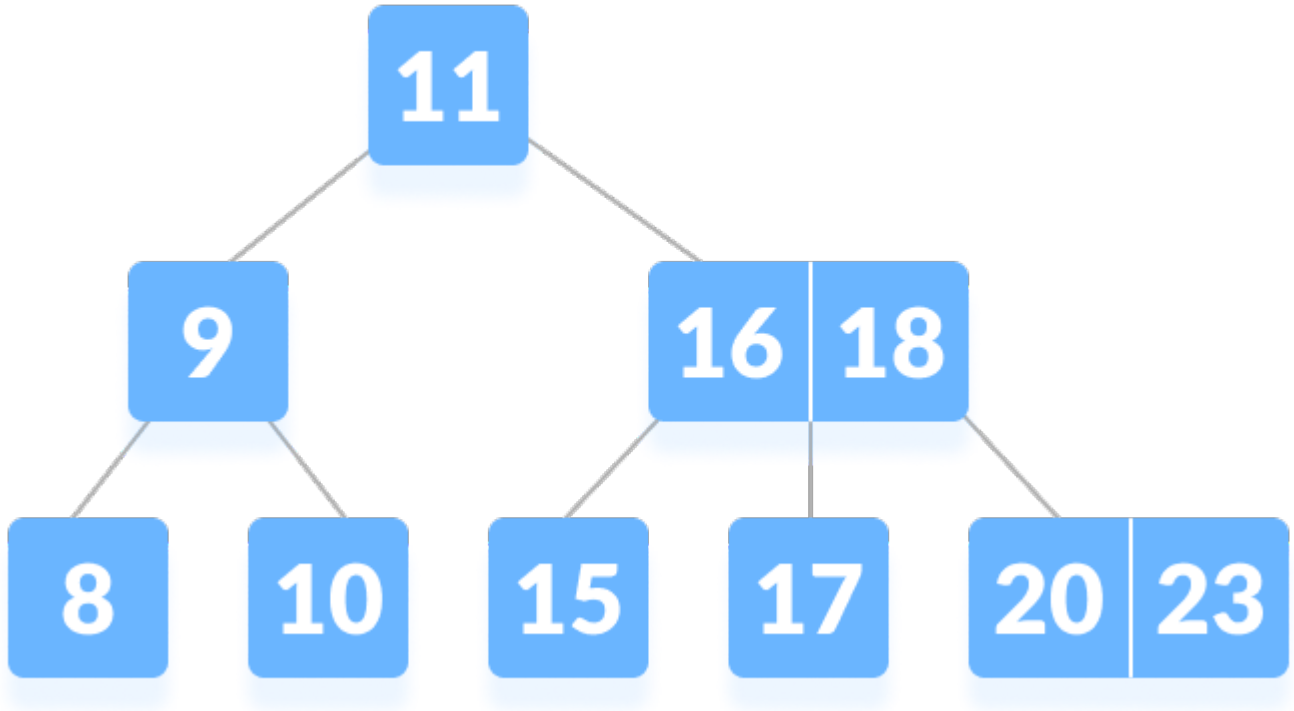
- Searching
- Insertion
- Deletion

Search

1. Starting from the root node, compare k with the first key of the node.
2. If k = the first key of the node, return the node and the index
3. If k.leaf = true, return NULL, i.e. not found
4. If k < the first key of the root node, search the left child of this key recursively.
5. If there is more than one key in the current node and k > the first key compare k with the next key in the node.
6. If k < next key search the left child of this key (ie. k lies in between the first and the second keys). Else, search the right child of the key
7. Repeat steps 1 to 4 until the leaf is reached.

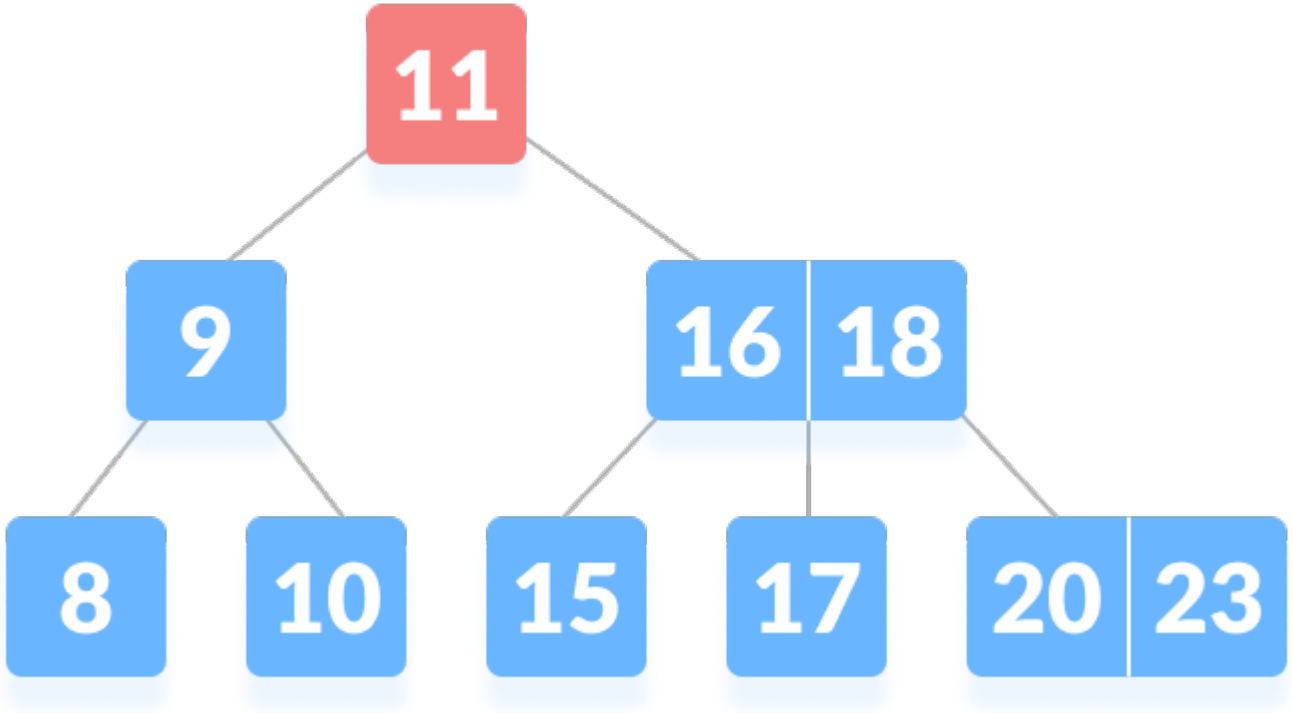
B Tree

Let us search key , k = 17



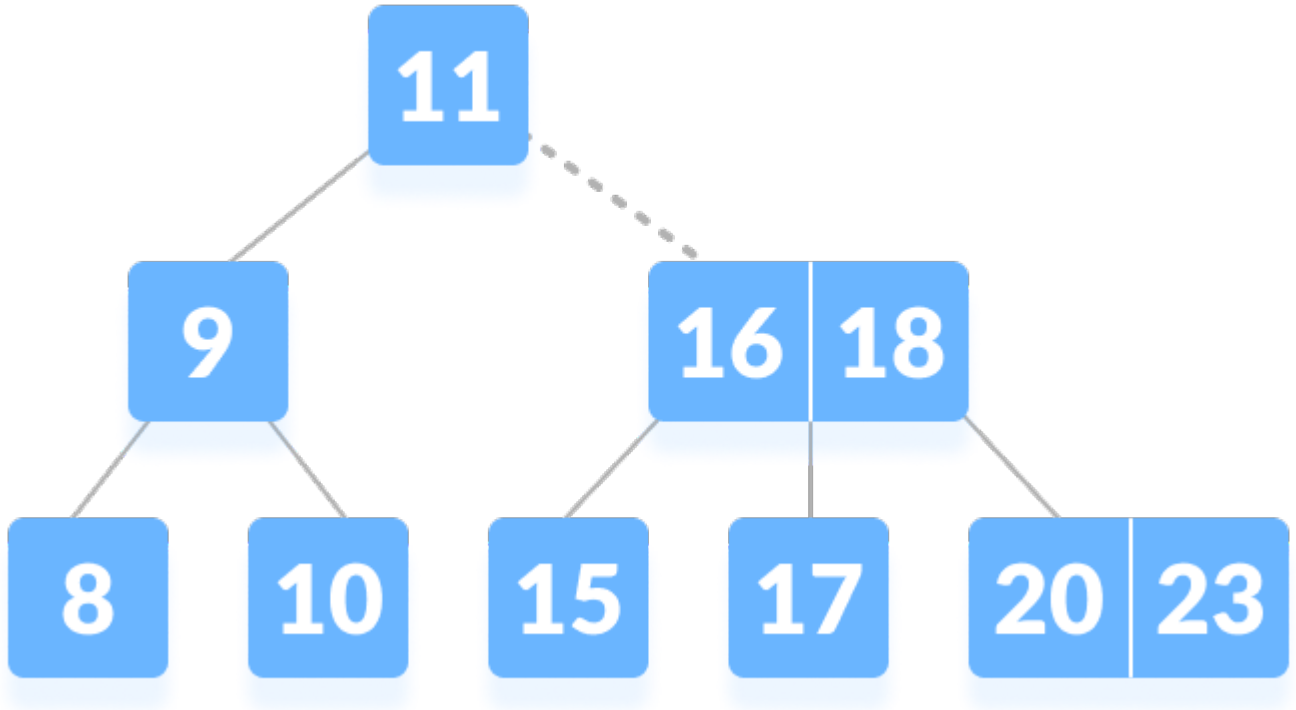
B Tree

K is not found in the root so, compare it with the root key



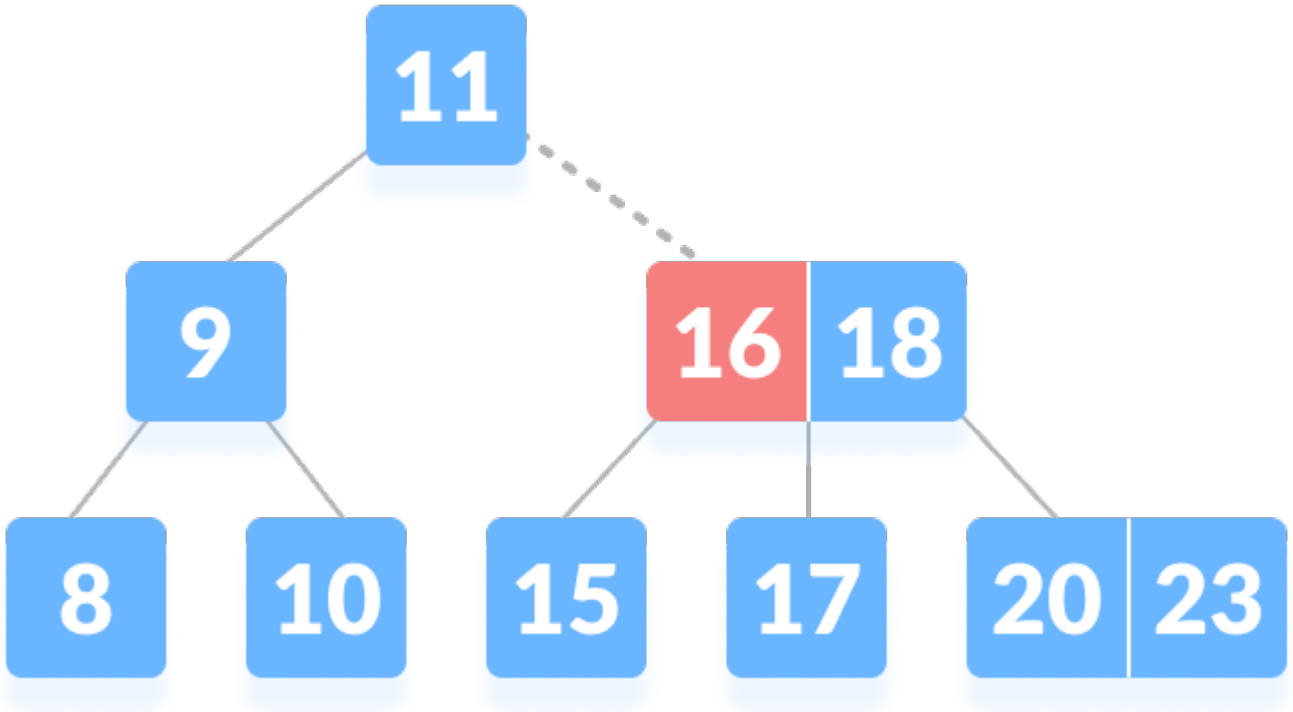
B Tree

Since k > 11 go to the right child of the root node.



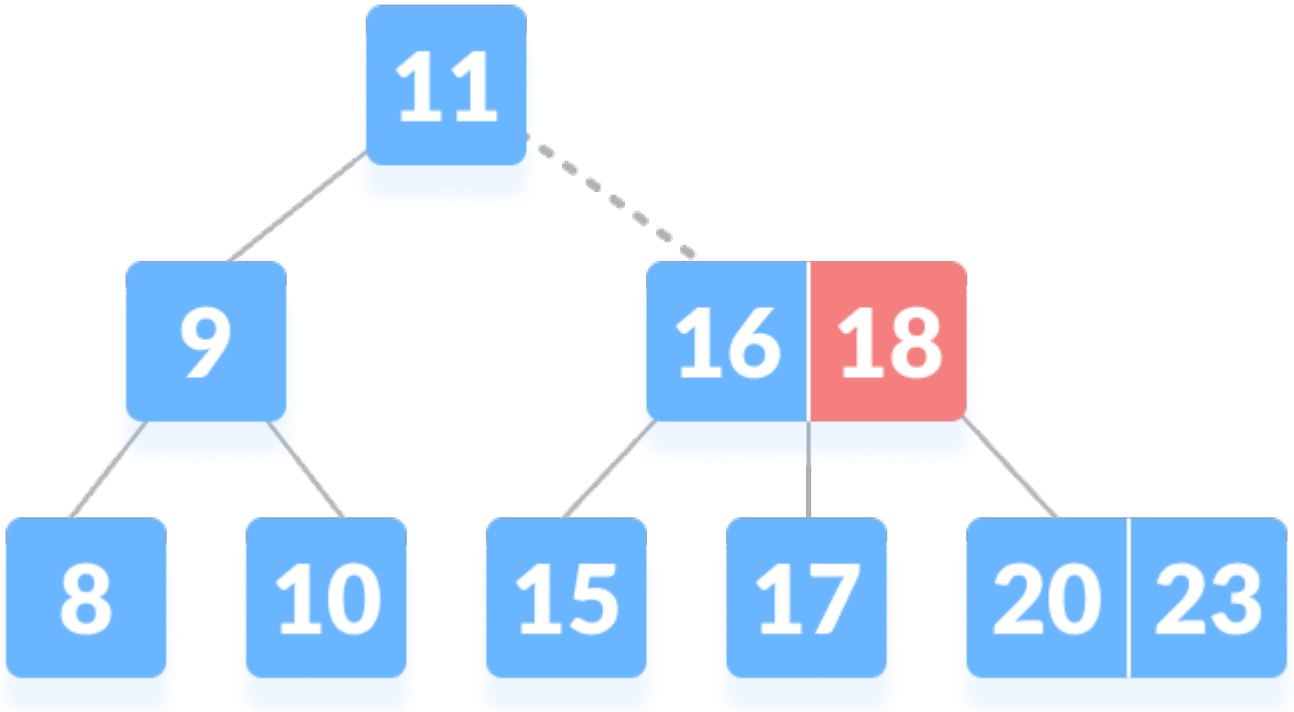
B Tree

Compare k with 16. Since k > 16, compare k with the next key 18



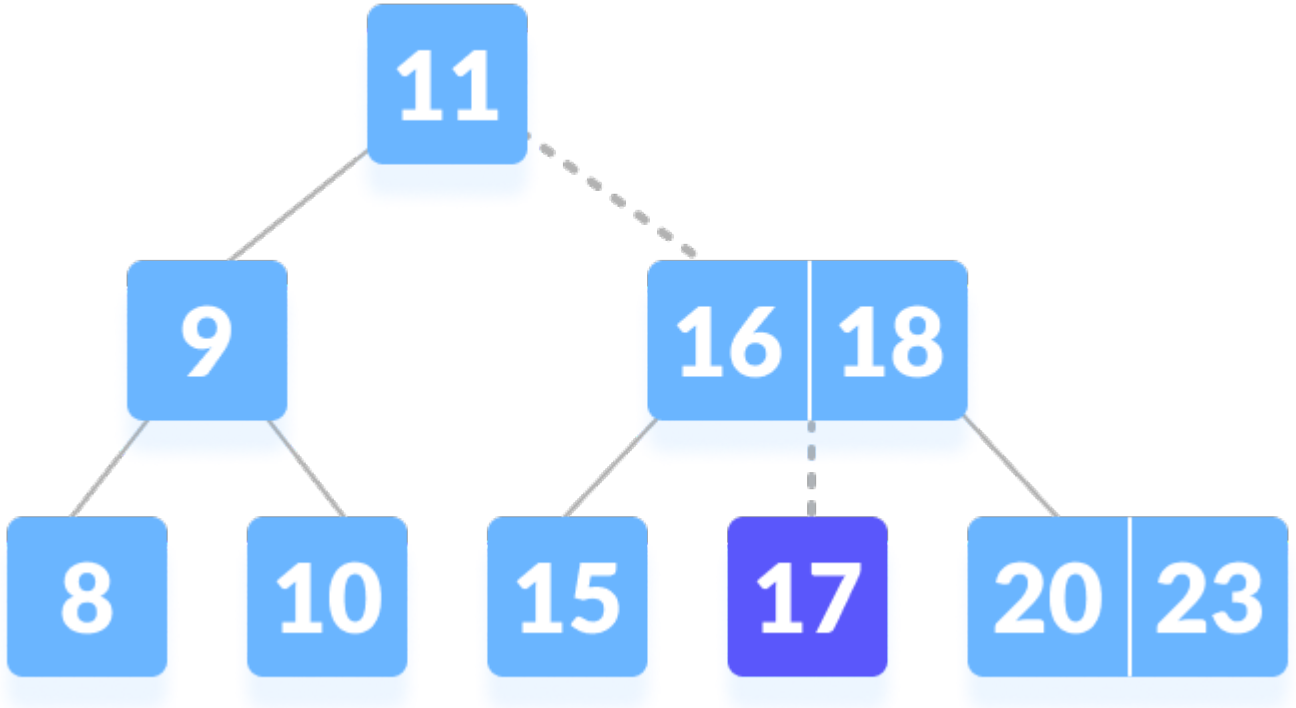
B Tree

Since k < 18, k lies between 16 and 18. Search in the right child of 16 or the left child of 18.



B Tree

k is found



Insertion into a B-tree

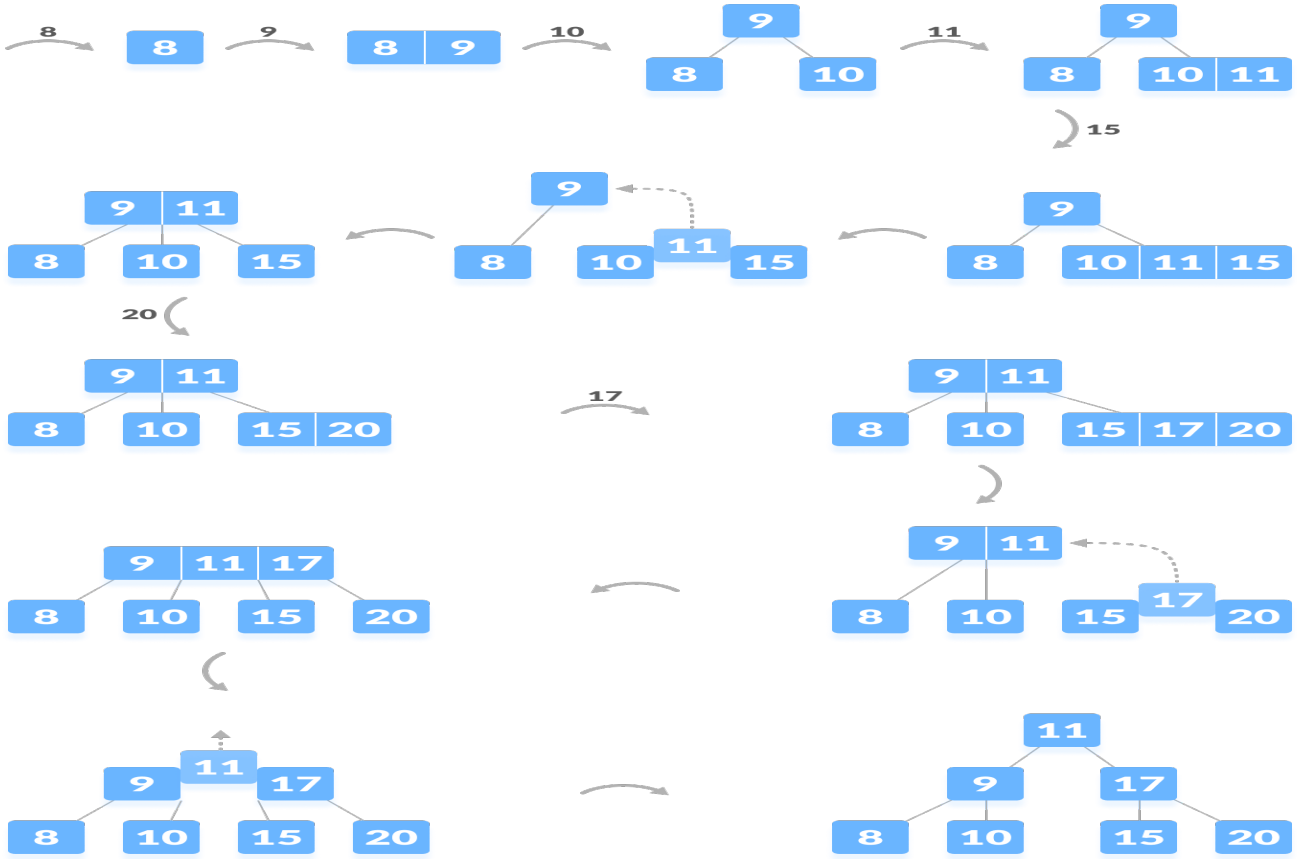
Inserting an element on a B-tree consists of two events: **searching the appropriate node** to insert the element and **splitting the node** if required.Insertion operation always takes place in the bottom-up approach.

Insertion Operation

1.Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2.If the leaf node contain less than m-1 keys then insert the element in the increasing order.

3. Else, if the leaf node contains m-1 keys, then follow the following steps.
   - Insert the new element in the increasing order of elements.
   - Split the node into the two nodes at the median.
   - Push the median element upto its parent node.
   - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

B Tree



Insertion Example

Let us understand the insertion operation with the illustrations below.
The elements to be inserted are 8, 9, 10, 11, 15, 16, 17, 18, 20, 23.

B Tree

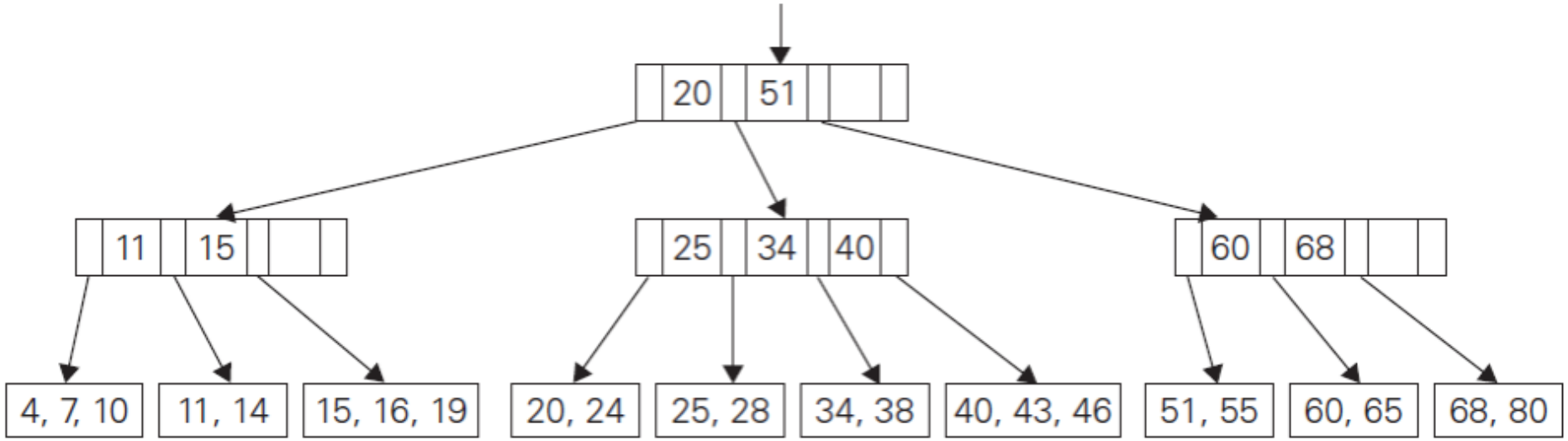


FIGURE 7.9 B-tree obtained after inserting 65 into the B-tree in Figure 7.8.

THANK YOU

Surabhi Narayan
Department of Computer Science &Engineering

surabhinarayan@pes.edu

PES UNIVERSITY

Text Book: Introduction to the Design and Analysis of Algorithms
Author: Anany Levitin 2 nd Edition

Unit-4

1. Space and Time Trade-Offs

Consider, as an example, the problem of computing values of a function at many points in its domain. If it is time that is at a premium, we can precompute the function's values and store them in a table. This is exactly what human computers had to do before the advent of electronic computers, in the process burdening libraries with thick volumes of mathematical tables. Though such tables have lost much of their appeal with the widespread use of electronic computers, the underlying idea has proven to be quite useful in the development of several important algorithms for other problems. In somewhat more general terms, the idea is to preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward. This approach is called input ***enhancement***.

The other type of technique that exploits space-for-time trade-offs simply uses extra space to facilitate faster and/or more flexible access to the data and this approach is called ***prestructuring.*** There is one more algorithm design technique related to the space-for-time trade-off idea: ***dynamic programming***. This strategy is based on recording solutions to overlapping subproblems of a given problem in a table from which a solution to the problem in question is then obtained.

I. **Input Enhancement**
   - Preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward.
   - Eg: Counting methods of sorting, Horspool's algorithm, Boyer-Moore's algorithm.

Sorting by Counting

1. **Comparison Counting Sorting**
   - For each element of the list, count the total number of elements smaller than this element.
   - These numbers will indicate the positions of the elements in the sorted list.
2. **Distribution Counting Sorting**
   - Suppose the elements of the list to be sorted belong to a finite set (aka domain).
   - Count the frequency of each element of the set in the list to be sorted.

- Scan the set in order of sorting and print each element of the set according to its frequency, which will be the required sorted list.

1 Comparison Counting Sorting

- For each element of the list, count the total number of elements smaller than the element.
- These numbers indicates the positions (0-based) of the elements in the sorted list.

Eg: **62 31 84 96 19 47**

lesser elements: **3 1 4 5 0 2**

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

```
//Sorts an array by comparison counting
//Input: An array A[0..n − 1] of orderable elements
//Output: Array S[0..n − 1] of A's elements sorted
for i ← 0 to n − 1 do Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do S[Count[i]] ← A[i]
return S
```

Example of sorting by comparison counting

Array $A[0..5]$		62	31	84	96	19	47
Initially	$Count[]$	0	0	0	0	0	0
After pass $i = 0$	$Count[]$	3	0	1	1	0	0
After pass $i = 1$	$Count[]$		1	2	2	0	1
After pass $i = 2$	$Count[]$			4	3	0	1
After pass $i = 3$	$Count[]$				5	0	1
After pass $i = 4$	$Count[]$					0	2
Final state	$Count[]$	3	1	4	5	0	2
Array $S[0..5]$		19	31	47	62	84	96

The time efficiency of this algorithm:

It should be quadratic because the algorithm considers all the different pairs of an n-element array. More formally, the number of times its basic operation, the comparison $A[i]<A[j]$, is executed is equal to the sum we have encountered several times already:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}$$

Thus, the algorithm makes the same number of key comparisons as selection sort and in addition uses a linear amount of extra space. On the positive side, the algorithm makes the minimum number of key moves possible, placing each of them directly in their final position in a sorted array.

Design and Analysis of Algorithms
Unit -4

Bharathi R
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Unit 4: Space and Time Tradeoffs

Distribution Counting Sort

Slides from **Anany Levitin**

Bharathi R
Department of Computer Science & Engineering

1.Input enhancement

a)Sorting by Counting

1. **Comparison Counting Sorting**
   I. For each element of the list, count the total number of elements smaller than this element.
   II. These numbers will indicate the positions of the elements in the sorted list.

2. **Distribution Counting Sorting**
   I. Suppose the elements of the list to be sorted belong to a finite set (aka domain).
   II. Count the frequency of each element of the set in the list to be sorted.
   III. Scan the set in order of sorting and print each element of the set according to its frequency, which will be the required sorted list.

1. Input Enhancement

→ Sorting by Counting →→ ii)Distribution Counting Sorting

- A sorting method in which, with the help of some **associated information** of the elements , the elements can be placed in an array at their relative positions.

- The required information which is used to place the elements at proper positions is **accumulated sum of frequencies** which is also called as distribution in statistics.

- Hence this method is called as **Distribution counting method** for sorting.

1. Input Enhancement

→ Sorting by Counting →→ ii)Distribution Counting Sorting

Consider sorting the array whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting. The frequency and distribution arrays are as follows:

13	11	12	13	12	12

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

11	12	12	12	13	13

Design and Analysis of Algorithms

Example of sorting by distribution counting.

13	11	12	13	12	12

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

11	12	12	12	13	13

The distribution values being decremented are shown in bold.

	$D[0..2]$		
$A[5] = 12$	1	**4**	6
$A[4] = 12$	1	**3**	6
$A[3] = 13$	1	2	**6**
$A[2] = 12$	1	**2**	5
$A[1] = 11$	**1**	1	5
$A[0] = 13$	0	1	**5**

$S[0..5]$					
			12		
		12			
					13
	12				
11					
				13	

Algorithm for Distribution Counting

ALGORITHM *DistributionCountingSort*$(A[0..n-1], l, u)$

//Sorts an array of integers from a limited range by distribution counting
//Input: An array $A[0..n-1]$ of integers between l and u $(l \le u)$
//Output: Array $S[0..n-1]$ of A's elements sorted in nondecreasing order
for $j \leftarrow 0$ **to** $u - l$ **do** $D[j] \leftarrow 0$ //initialize frequencies
for $i \leftarrow 0$ **to** $n - 1$ **do** $D[A[i] - l] \leftarrow D[A[i] - l] + 1$ //compute frequencies
for $j \leftarrow 1$ **to** $u - l$ **do** $D[j] \leftarrow D[j-1] + D[j]$ //reuse for distribution
for $i \leftarrow n - 1$ **downto** 0 **do**
    $j \leftarrow A[i] - l$
    $S[D[j] - 1] \leftarrow A[i]$
    $D[j] \leftarrow D[j] - 1$
return S

Distribution Counting Sorting

1. This is a better time-efficiency class than that of the most efficient sorting algorithms—mergesort, quicksort, and heapsort—we have encountered.

2. It is important to remember, however, that this efficiency is obtained by exploiting the specific nature of inputs for which sorting by distribution counting works, in addition to trading space for time.

References

"Introduction to the Design and Analysis of Algorithms", Anany Levitin, Pearson Education, Delhi (Indian Version), 3rd edition, 2012. Chapter- 7

THANK YOU

Bharathi R

Department of Computer Science & Engineering

rbharathi@pes.edu

Design and Analysis of Algorithms
Unit -4

Bharathi R
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Unit 4: Space and Time Tradeoffs

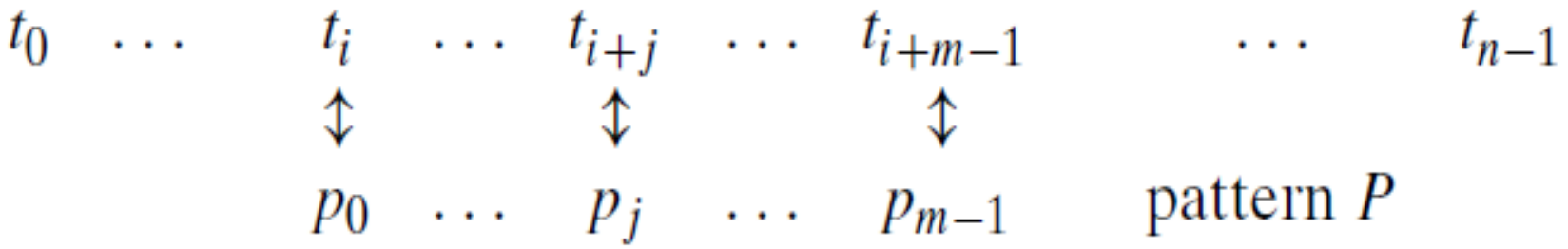
Input Enhancement in String Matching

Bharathi R

Department of Computer Science & Engineering

String Matching

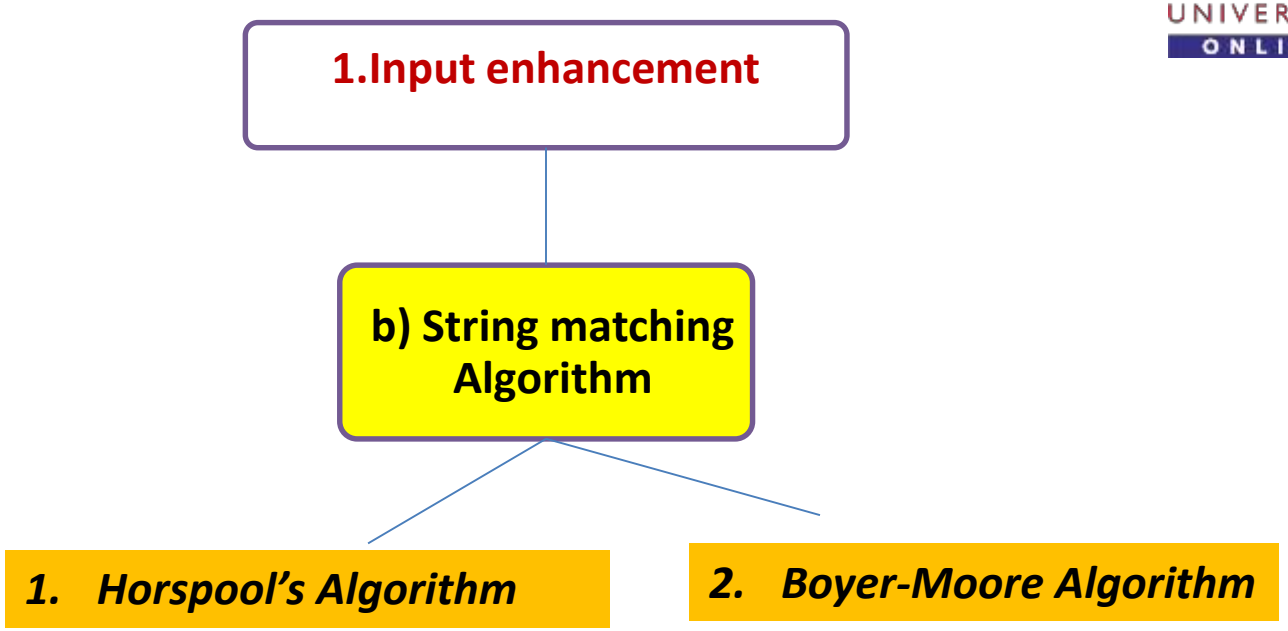
String matching requires finding an occurrence of a given string of *m* characters called the ***pattern*** in a longer string of *n* characters called the ***text***.



Input Enhancement in String Matching

Input-enhancement idea:

Preprocess the pattern to get some information about it, store this information in a table, and then use this information during an actual search for the pattern in a given text.



String matching by Brute force

pattern: a string of *m* characters to search for
text**:** a (long) string of *n* characters to search in

Brute force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until either all characters are found to match (successful search) or a mismatch is detected

Step 3 While a mismatch is detected and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Time complexity (worst-case): O(mn)

Design and Analysis of Algorithms

String matching by preprocessing

Several string searching algorithms are based on the input enhancement idea of preprocessing the **pattern**

1. **Knuth-Morris-Pratt (KMP)** algorithm preprocesses pattern left to right to get useful information for later searching

2. **Boyer -Moore algorithm** preprocesses pattern right to left and store information into two tables

3. **Horspool's algorithm** simplifies the Boyer-Moore algorithm by using just one table

Horspool's Algorithm

A simplified version of Boyer-Moore algorithm:

Preprocesses pattern to generate a "shift table" that determines how much to shift the pattern when a mismatch occurs

Always makes a shift based on the text's character **c** aligned with the last compared (mismatched) character in the pattern according to the shift table's entry for ***c***

How far to shift?

Look at first (rightmost) character in text that was compared:

1. The character is not in the pattern

```
.....c..... (c not in pattern)
     ≠
BAOBAB
```

2. The character is in the pattern (but not the rightmost)

```
.....O..... (O occurs once in pattern)
BAOBAB

.....A..... (A occurs twice in pattern)
BAOBAB
```

3. The rightmost characters do match

```
.....B.....
BAOBAB
```

Design and Analysis of Algorithms

Shift Table

Shift sizes can be precomputed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases}$$

By scanning pattern before search begins and stored in a table called ***shift table***. After the shift, the right end of pattern is *t(c)* positions to the right of the last compared character in text.

Shift table is indexed by text and pattern alphabet Eg: for **BAOBAB**:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6

Shift Table

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases}$$

ALGORITHM *ShiftTable*$(P[0..m-1])$

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: $Table[0..size-1]$ indexed by the alphabet's characters and
// filled with shift sizes computed by formula (7.1)
for $i \leftarrow 0$ **to** $size-1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

Horspool Matching

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//        or −1 if there are no matches
ShiftTable(P[0..m − 1])        //generate Table of shifts
i ← m − 1                      //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                      //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```

Design and Analysis of Algorithms

Horspool Matching -Example

$s_0 \quad \ldots \quad c \quad \ldots \quad s_{n-1}$

B A R B E R

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

B A R B E R

B A R B E R

B A R B E R

B A R B E R

Design and Analysis of Algorithms

Example of Horspool's algorithm

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

```
BARD LOVED BANANAS
BAOBAB
      BAOBAB
           BAOBAB
                   BAOBAB  (unsuccessful search)
```

References

"Introduction to the Design and Analysis of Algorithms", Anany Levitin, Pearson Education, Delhi (Indian Version), 3rd edition, 2012. Chapter- 7

THANK YOU

Bharathi R

Department of Computer Science & Engineering

rbharathi@pes.edu

Design and Analysis of Algorithms
Unit -4

Bharathi R
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Unit 4: Space and Time Tradeoffs

Input Enhancement in String Matching- The Boyer-Moore Algorithm

Bharathi R

Department of Computer Science & Engineering

The Boyer-Moore algorithm

Based on the same two ideas:

comparing pattern characters to text from right to left

precomputing shift sizes in **two** tables

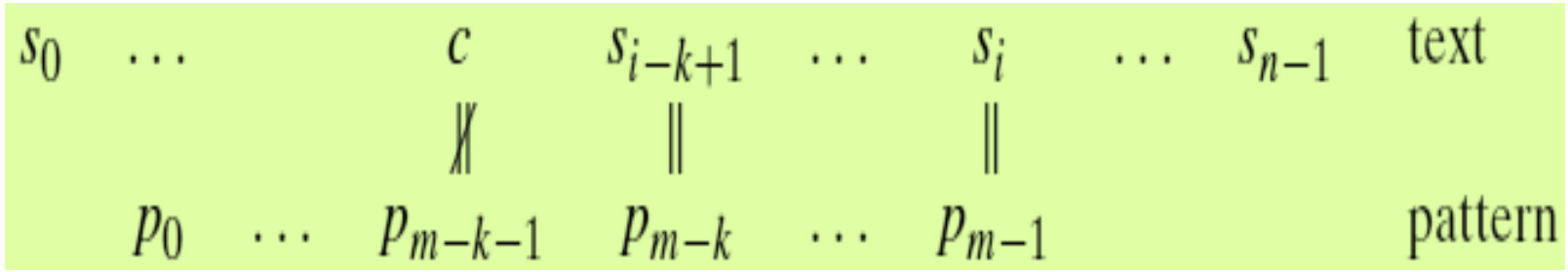
1. ***bad-symbol table*** indicates how much to shift based on text's character causing a mismatch

2. ***good-suffix table*** indicates how much to shift based on matched part (suffix) of the pattern (taking advantage of the periodic structure of the pattern)

Design and Analysis of Algorithms

Bad-symbol shift in Boyer-Moore algorithm

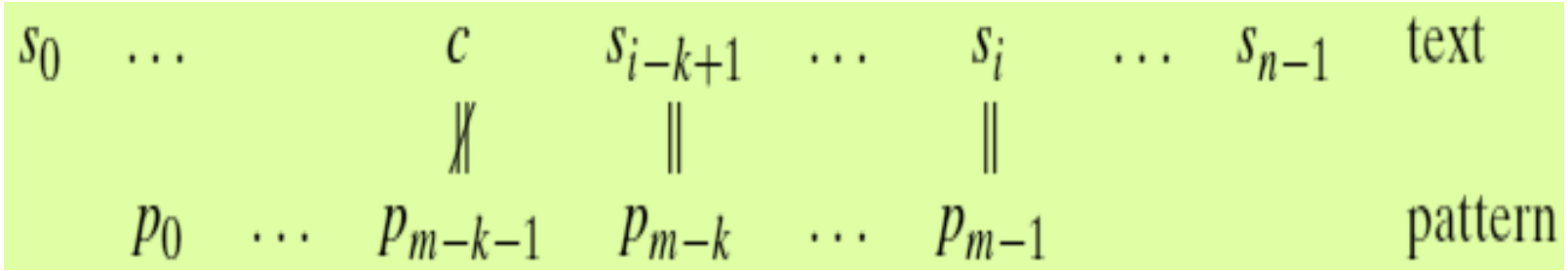
1. If the rightmost character of the pattern doesn't match, BM algorithm acts as Horspool's
2. If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern's characters match or a mismatch on text's character c is encountered after $k > 0$ matches



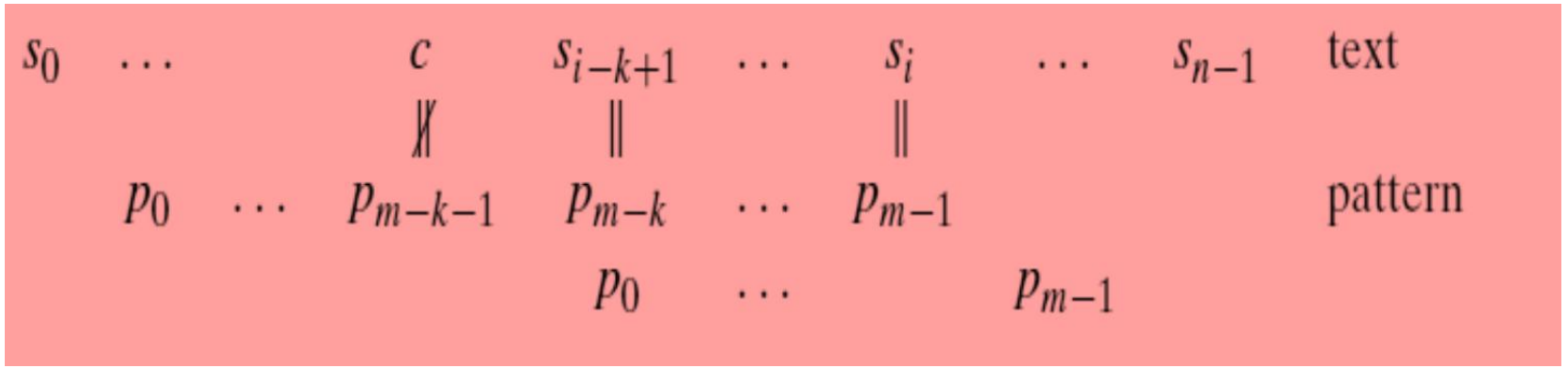
In this situation, the Boyer-Moore algorithm determines the shift size by considering two quantities. The first one is guided by the text's character c that caused a mismatch with its counterpart in the pattern. Accordingly, it is called the ***bad symbol shift***.

Design and Analysis of Algorithms

Bad-symbol shift in Boyer-Moore algorithm



If *c* is not in the pattern, we shift the pattern to just pass this *c* in the text. Conveniently, the size of this shift can be computed by the formula *t1(c)* − *k* where *t1(c)* is the entry in the precomputed table used by Horspool's algorithm and *k* is the number of matched characters:



Bad-symbol shift in Boyer-Moore algorithm- Example

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

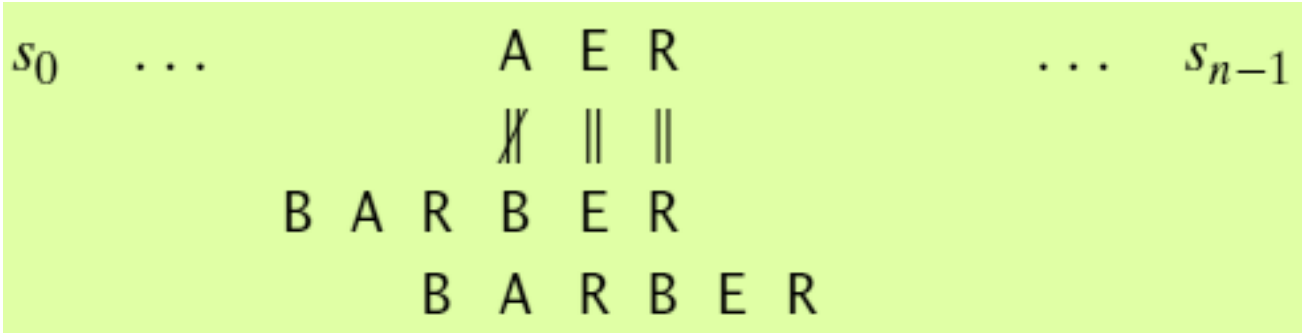
For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter S in the text, we can shift the pattern by $t_1(\text{S}) - 2 = 6 - 2 = 4$ positions:

```
s0  . . .          S  E  R                    . . .  sn−1
                   ✗  ‖  ‖
          B  A  R  B  E  R
                      B  A  R  B  E  R
```

Bad-symbol shift in Boyer-Moore algorithm- Example

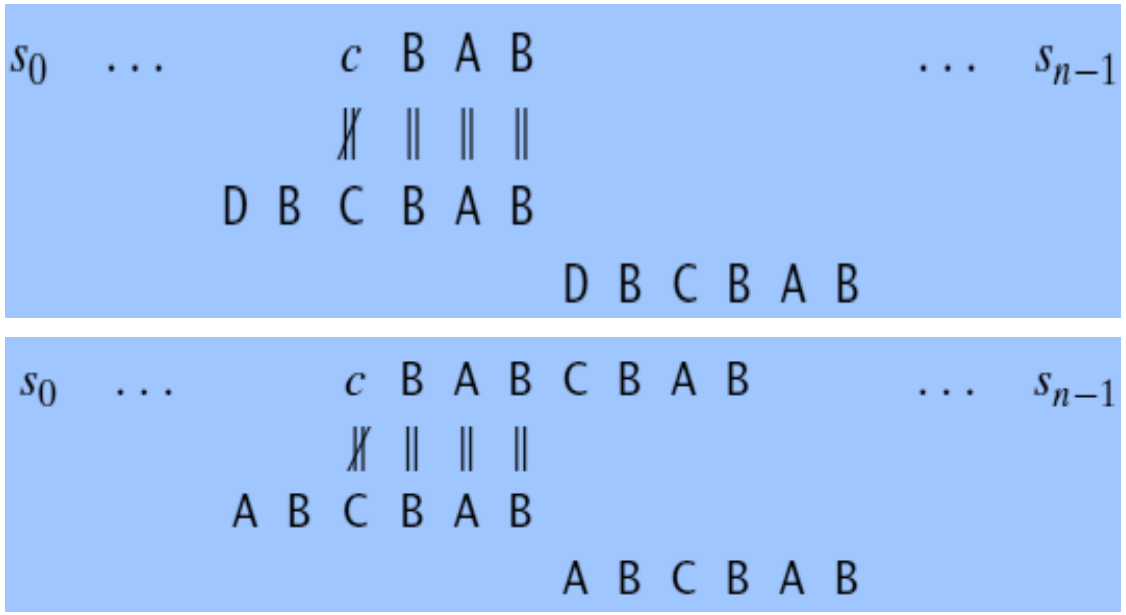
character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The same formula can also be used when the mismatching character c of the text occurs in the pattern, provided $t_1(c) - k > 0$. For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter A, we can shift the pattern by $t_1(\text{A}) - 2 = 4 - 2 = 2$ positions:



$$d_1 = \max\{t_1(c) - k, 1\}$$

Good-suffix shift in Boyer-Moore algorithm and when to use?



Erroneous shift will happen.
To avoid that a shift based on matched suffix of the pattern.
Called "Good Suffix shift". It is denoted by d_2.

Good-suffix shift in Boyer-Moore algorithm

$d_2(k)$ = Distance between matched suffix of size k and its rightmost occurrence in the pattern that is not preceded by the same character as the suffix

k	pattern	d_2
1	ABC$\overline{B}$AB	2
2	$\overline{AB}$CBAB	4
3	$\overline{AB}$CBAB	4
4	$\overline{AB}$CBAB	4
5	$\overline{AB}$CBAB	4

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$

Good-suffix shift in Boyer-Moore algorithm

- Good-suffix shift d_2 is applied after $0 < k < m$ last characters were matched

- $d_2(k)$ = the distance between (the last letter of) the matched suffix of size k and (the last letter of) its rightmost occurrence in the pattern that is not preceded by the same character preceding the suffix

  Example: `CABABA` $d_2(1) = 4$

- If there is no such occurrence, match the longest part (tail) of the k-character suffix with corresponding prefix;
  if there are no such suffix-prefix matches, $d_2(k) = m$

  Example: `WOWWOW` $d_2(2) = 5$, $d_2(3) = 3$, $d_2(4) = 3$, $d_2(5) = 3$

Boyer-Moore algorithm

Step 1 Fill in the bad-symbol shift table
Step 2 Fill in the good-suffix shift table
Step 3 Align the pattern against the beginning of the text
Step 4 Repeat until a matching substring is found or text ends:
Compare the corresponding characters right to left.
If no characters match, retrieve entry $t_1(c)$ from the bad-symbol table for the text's character c causing the mismatch and shift the pattern to the right by $t_1(c)$.
If $0 < k < m$ characters are matched, retrieve entry $t_1(c)$ from the bad-symbol table for the text's character c causing the mismatch and entry $d_2(k)$ from the good-suffix table and shift the pattern to the right by

$$d = \max \{d_1, d_2\}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

Design and Analysis of Algorithms

Example of Boyer-Moore alg. application

BAOBAB

c	A	B	C	D	. . .	O	. . .	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

k	pattern	d_2
1	BAO$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{B}}$AOB$\underline{\text{AB}}$	5
3	$\overline{\text{B}}$AO$\underline{\text{BAB}}$	5
4	$\overline{\text{B}}$A$\underline{\text{OBAB}}$	5
5	$\overline{\text{B}}$A$\underline{\text{OBAB}}$	5

B E S S _ K N E W _ A B O U T _ B A O B A B S

B A O B A B

$d_1 = t_1(K) - 0 = 6$

B A O B A B

$d_1 = t_1(\_) - 2 = 4$

$d_2 = 5$

$d = \max\{4, 5\} = 5$

B A O B A B

$d_1 = t_1(\_) - 1 = 5$

$d_2 = 2$

$d = \max\{5, 2\} = 5$

B A O B A B

Analysis

- The worst-case efficiency of the Boyer-Moore algorithm is known to be linear.
- Though this algorithm runs very fast, especially on large alphabets (relative to the length of the pattern),many people prefer its simplified versions, such as Horspool's algorithm, when dealing with natural-language–like strings.

References

Chapter 7 ,Introduction to The Design and Analysis of Algorithms by Anany Levitin

THANK YOU

Bharathi R

Department of Computer Science & Engineering

rbharathi@pes.edu

Design and Analysis of Algorithms
Unit -4

Bharathi R
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Unit 4: Greedy Technique

Bharathi R

Department of Computer Science & Engineering

The Greedy Approach

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step, the choice made must be:

-*feasible*: it has to satisfy the problem's constraints

-*locally optimal*: it has to be the best local choice among all feasible choices available on that step

- *irrevocable*: once decision was made, it cannot be changed on subsequent steps of the algorithm

Examples of Greedy Algorithms:

- Coin-change problem
- Minimum Spanning Tree (MST)
  - Prim's Algorithm
  - Kruskal's Algorithm
- Single-source shortest paths
  - Dijkstra's Algorithm
- Huffman codes

Coin Change Problem

A greedy algorithm to find the minimum number of coins for making the change of a given amount of money. Usually, this problem is referred to as the change-making problem.

Design and Analysis of Algorithms
Change Making Problem

- In the change-making problem, we're provided with an array, D = { d1, d2, d3,.....dm} of m distinct coin denominations.
- Now we need to find an array(subset) s having minimum number of coins that add up to a given amount of money **n**, provided that there exists a viable solution.
- Let's consider a real-life example for a better understanding of the change-making problem.
- Let's assume that we're working at a cash counter and have an infinite supply of **D ={1,2,5,10}** valued coins.
- A person buys things worth **Rs. 72** and gives a **Rs. 100** bill. How does the cashier give change for **Rs. 28**?

Change Making Problem

Change-making problem:

How can a given amount of money be made with the least number of coins of given denominations?

Example:

Change for Rs. **28**

Option	Choosen Coins
28-10 = 18	10
18-10 =8	10 10
8-5 =3	10 10 5
3-2 = 1	10 10 5 2
1-1 =0	10 10 5 2 1

Design and Analysis of Algorithms

Change Making Problem

Iteration 1
D ={1,2,5,10}
n = 28 and i=4

Select D[3] =10 as 28>= 10
Decrease n by 10 (n= 18)
Add 10 to set S = {10}

Iteration 2
D ={1,2,5,10}
n = 28 and i=3

Select D[3] =10 as 18>= 10
Decrease n by 10 (n= 8)
Add 10 to set S = {10, 10}

Iteration 3
D ={1,2,5,10}
n = 8 and i=2

Select D[2] =5 as 8>= 5
Decrease n by 5 (n= 3)
Add 5 to set S = {10, 10,5}

Iteration 4
D ={1,2,5,10}
n = 3 and i=1

Select D[1] =2 as 3>= 2
Decrease n by 2 (n= 1)
Add 2 to set S = {10, 10, 5,2}

Iteration 5
D ={1,2,5,10}
n = 1 and i=0

Select D[0] =1 as 1>= 1
Decrease n by 1 (n= 0)
Add 1 to set S = {10, 10, 5,2,1}

Summary

The ***greedy technique*** suggests constructing a solution to an optimization problem through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.

On each step, the choice made must be ***feasible***, ***locally optimal***, and ***irrevocable***.

Design and Analysis of Algorithms

Text Books

Chapter 9 ,Introduction to The Design and Analysis of Algorithms by Anany Levitin

THANK YOU

Bharathi R

Department of Computer Science & Engineering

rbharathi@pes.edu

Design and Analysis of Algorithms
Unit -4

Bharathi R
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Unit 4: Greedy Technique

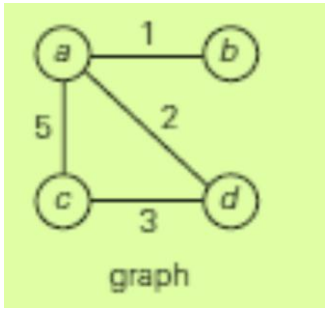
Prim's algorithm

Bharathi R

Department of Computer Science & Engineering

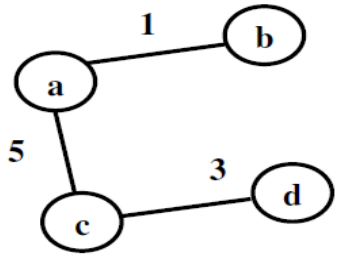
Minimum Spanning Tree : DEFINITIONs

A *spanning tree* of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph.

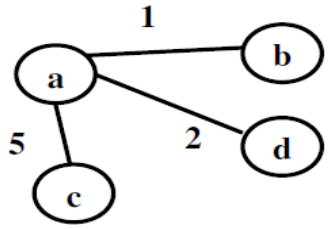


Example

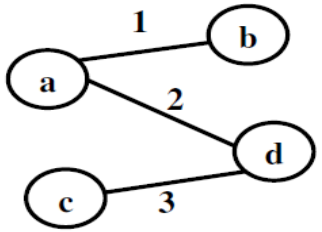
The spanning trees for the above graph are as follows:



Weight (T_1) = 9

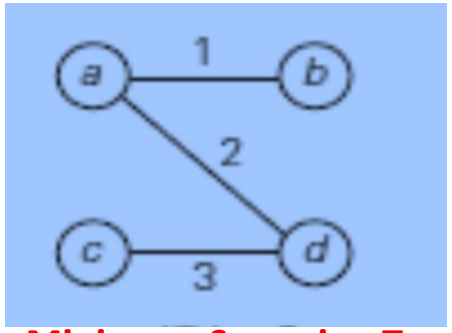


Weight (T_2) = 8



Weight (T_3) = 6

Minimum Spanning Tree (MST) of a weighted, connected graph G is a spanning tree of G with minimum total weight.



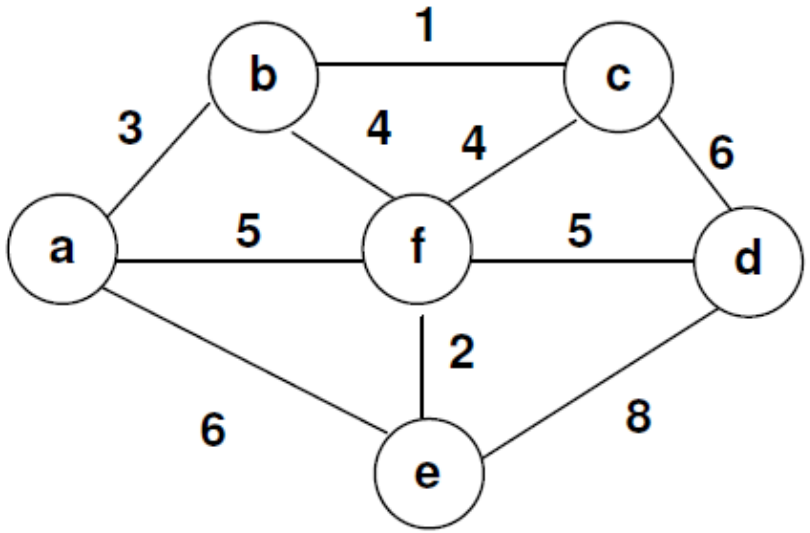
Minimum Spanning Tree

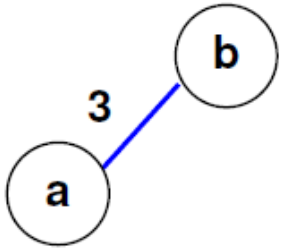
Prim's Algorithm : Greedy Approach

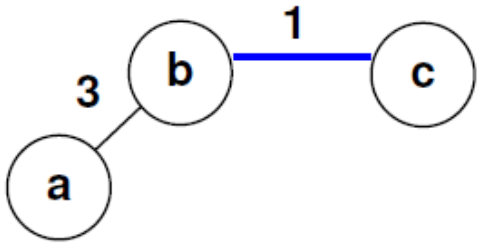
- Start with a tree, $\mathbf{T_1}$, consisting of one vertex (V).
- Adjacent vertices of the vertex in $\mathbf{T_1}$ are "fringe" vertices of $\mathbf{T_1}$.
- For i = 1 to |V|-1 do
  - Construct $\mathbf{T_i}$ from $\mathbf{T_{i-1}}$ by adding the fringe vertex with the minimum weight edge from the set. The vertex is removed from the set of fringe vertices.
  - Add the adjacent vertices of the vertex to the set of fringe vertices which are not in $\mathbf{T_i}$.
  - Remove vertices from the set of fringe vertices where the new vertex is one of the terminal vertex of the edge.
- Return $\mathbf{T_n}$ which is a minimum spanning tree.

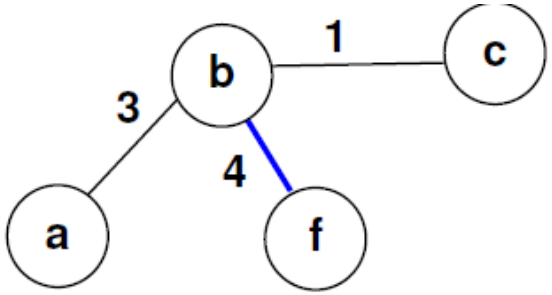
Design and Analysis of Algorithms

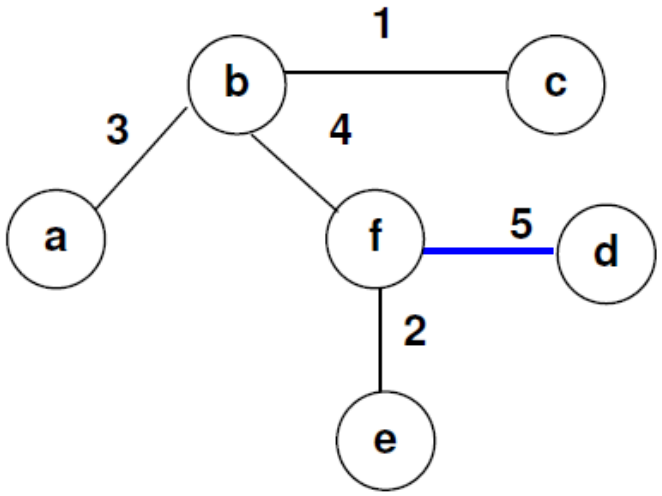
Prim's Algorithm

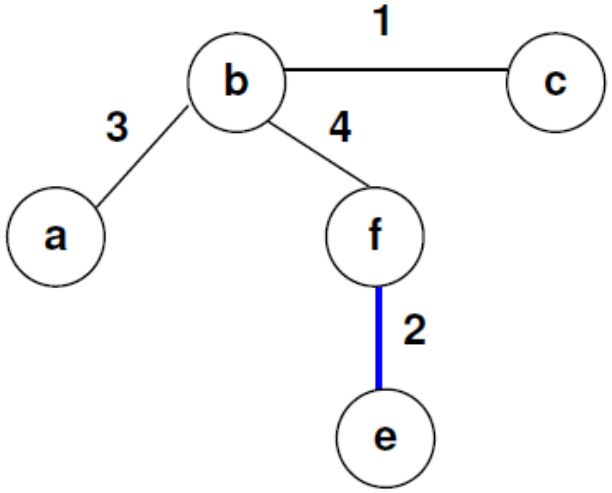




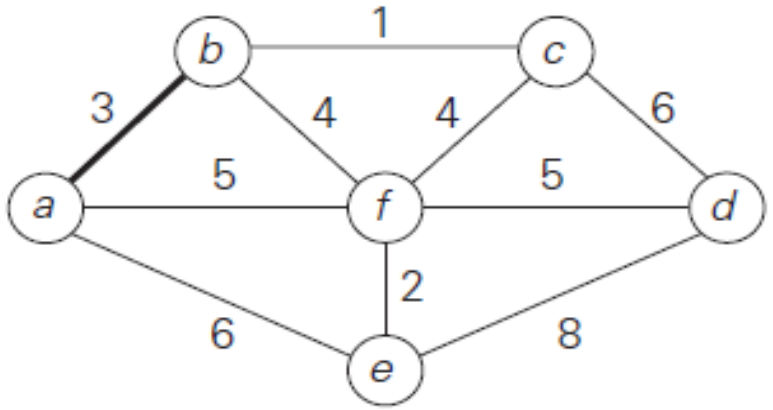
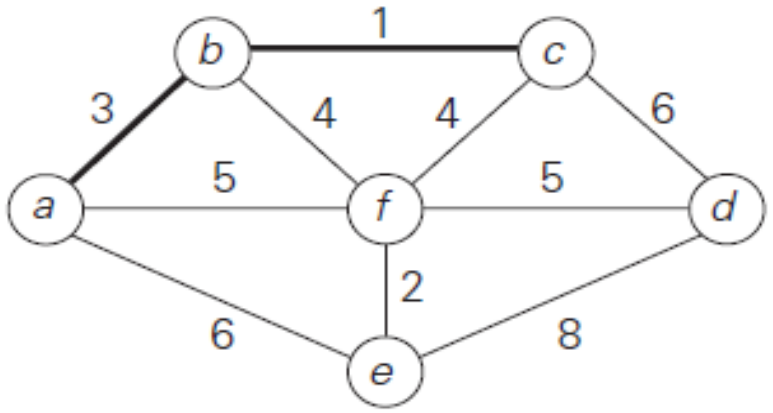




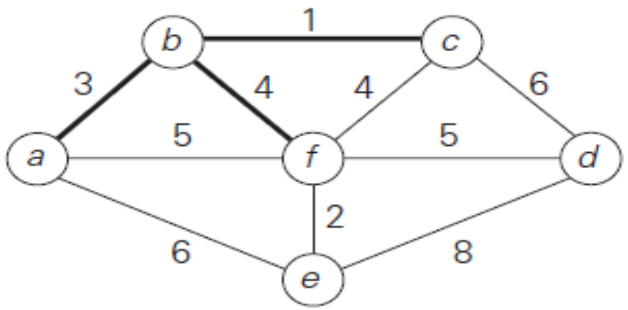
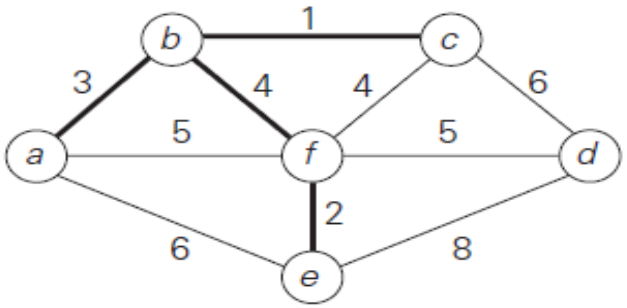
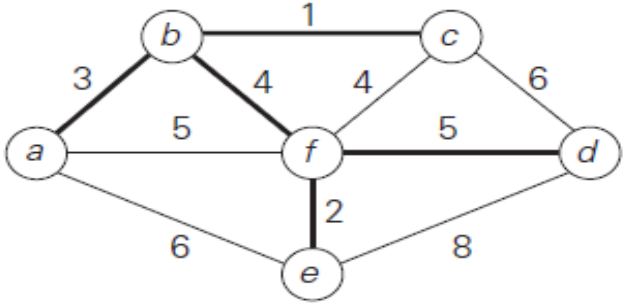




Prim's Algorithm

Tree vertices	Remaining vertices	Illustration
a(–, –)	**b(a, 3)** c(–, ∞) d(–, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(–, ∞) e(a, 6) f(b, 4)	

Prim's Algorithm

Tree vertices	Remaining vertices	Illustration
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

Prim's Algorithm

ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T and u is in $V - V_T$

After we have identified a vertex u^* to be added to the tree, we need to perform two operations:

Move u^* from the set $V - V_T$ to the set of tree vertices V_T.

For each remaining vertex u in $V - V_T$ that is connected to u^* by a shorter edge than the u's current distance label, update its labels by u^* and the weight of the edge between u^* and u, respectively.

How efficient is Prim's Algorithm

- The answer depends on the data structures chosen for the graph itself and for the priority queue of the set $V - V_T$ whose vertex priorities are the distances to the nearest tree vertices.

- If a graph is represented by its weight matrix and the priority queue is implemented as an unordered array, the algorithm's running time will be in $\Theta(|V|^2)$

- If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in $O(|E| \log |V|)$.

Summary

Prim's algorithm is very similar to Kruskal's: whereas Kruskal's "grows" a forest of trees, Prim's algorithm grows a single tree until it becomes the minimum spanning tree.

Both algorithms use the greedy approach - they add the cheapest edge that will not cause a cycle. But rather than choosing the cheapest edge that will connect *any* pair of trees together, Prim's algorithm only adds edges that join nodes to the existing tree.

(In this respect, Prim's algorithm is very similar to Dijkstra's algorithm for finding shortest paths.)

Design and Analysis of Algorithms

Text Books

Chapter 9 ,Introduction to The Design and Analysis of Algorithms by Anany Levitin

THANK YOU

Bharathi R

Department of Computer Science & Engineering

rbharathi@pes.edu

Design and Analysis of Algorithms
Unit -4

Bharathi R
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Unit 4: Greedy Technique

Bharathi R

Department of Computer Science & Engineering

Minimum Spanning Tree : Kruskal's Algorithm

This algorithm was described by **Joseph Kruskal** in 1956.

Prim's algorithm and Kruskal's algorithm solve the same problem (**Minimum Spanning Tree**) by applying the **greedy approach** in two different ways, and both of them always yield an optimal solution.

Minimum Spanning Tree : Kruskal's Algorithm

Kruskal's algorithm initially places all the nodes of the original graph isolated from each other, to form a forest of single node trees, and then gradually merges these trees, combining at each iteration any two of all the trees with some edge of the original graph.

Before the execution of the algorithm, all edges are sorted by weight (in non-decreasing order).

Then begins the process of unification: pick all edges from the first to the last (in sorted order), and if the ends of the currently picked edge belong to different subtrees, these subtrees are combined, and the edge is added to the answer.

After iterating through all the edges, all the vertices will belong to the same sub-tree, and we will get the answer.

Design and Analysis of Algorithms

Kruskal's Algorithm

Let G = {V,E} be weighted connected graph

The vertices are numbered 0 through $n - 1$.

V = {0,1,….n-1} a nd E =[e_1,e_2,….e_n}

Sort the edges in non-decreasing order of their weights.

•Initially, trees of the forest are the vertices (no edges).

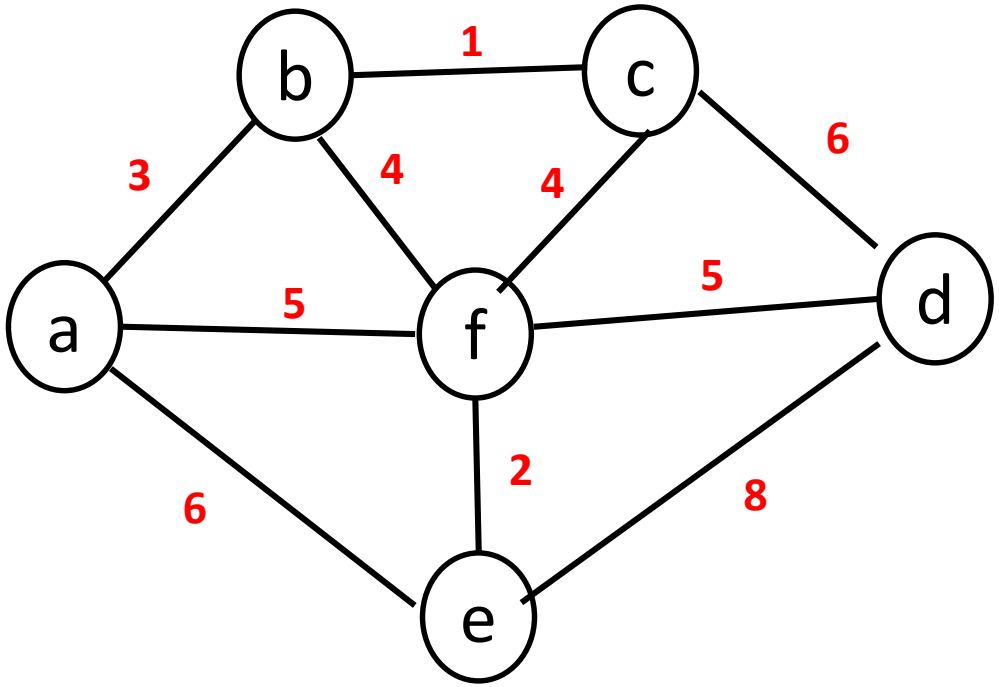
•Start with an empty minimum spanning tree and mark each edge as unvisited,

•While there are edges not yet visited or while the minimum spanning tree does not have $n - 1$ edges:

- Find the edge with minimum weight that has not yet been visited.
- Mark that edge as visited.
- If adding that edge does not create a cycle (that is, the two vertices are not already connected), add that edge to the minimum spanning tree.

Kruskal's Algorithm

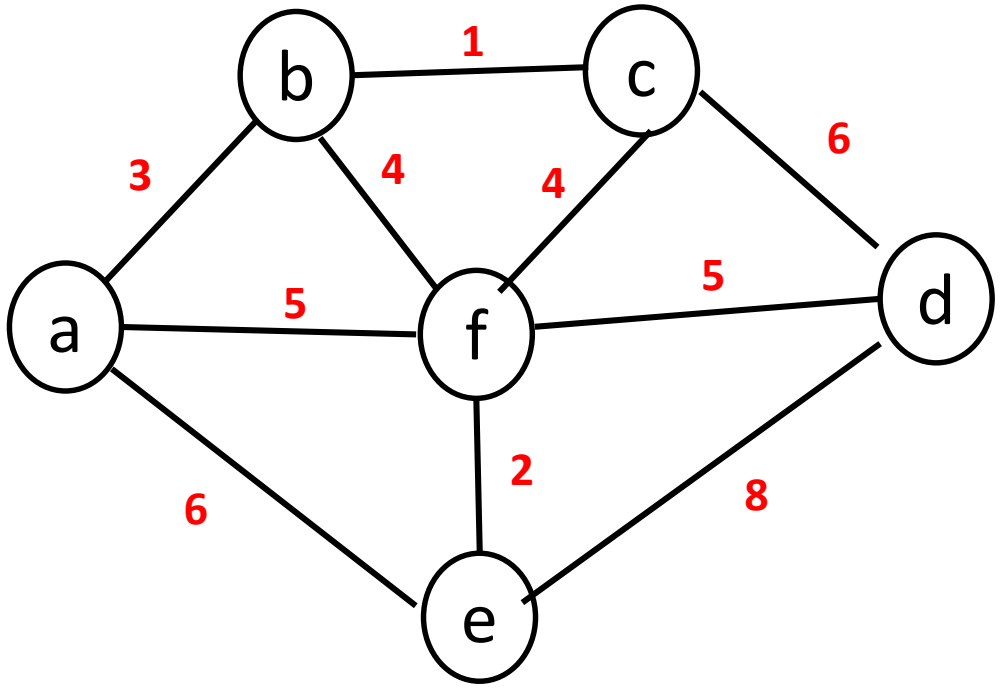
Given : Graph <V,E>



Design and Analysis of Algorithms

Kruskal's Algorithm

Given : Graph<V,E>



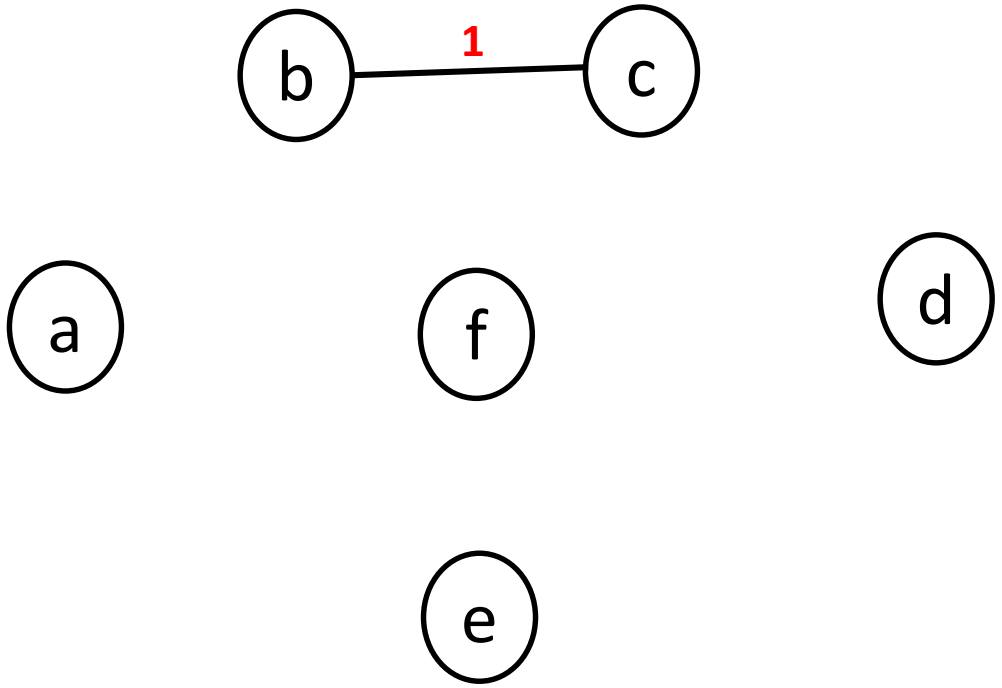
Sorted list of edges in non decreasing order

Edge(u,v)	Weight
bc	1
ef	2
ab	3
bf	4
cf	4
af	5
df	5
ae	6
cd	6
de	8

Design and Analysis of Algorithms

Kruskal's Algorithm

Given : Graph<V,E>



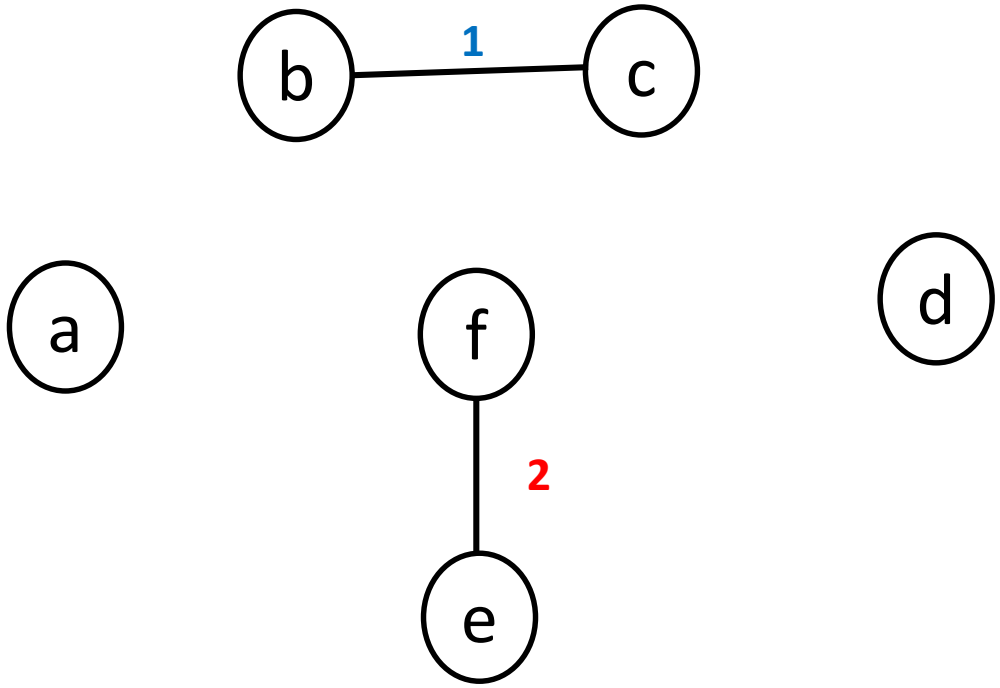
Sorted list of edges in non decreasing order

Edge(u,v)	Weight
bc	**1**
ef	2
ab	3
bf	4
cf	4
af	5
df	5
ae	6
cd	6
de	8

Design and Analysis of Algorithms

Kruskal's Algorithm

Given : Graph<V,E>



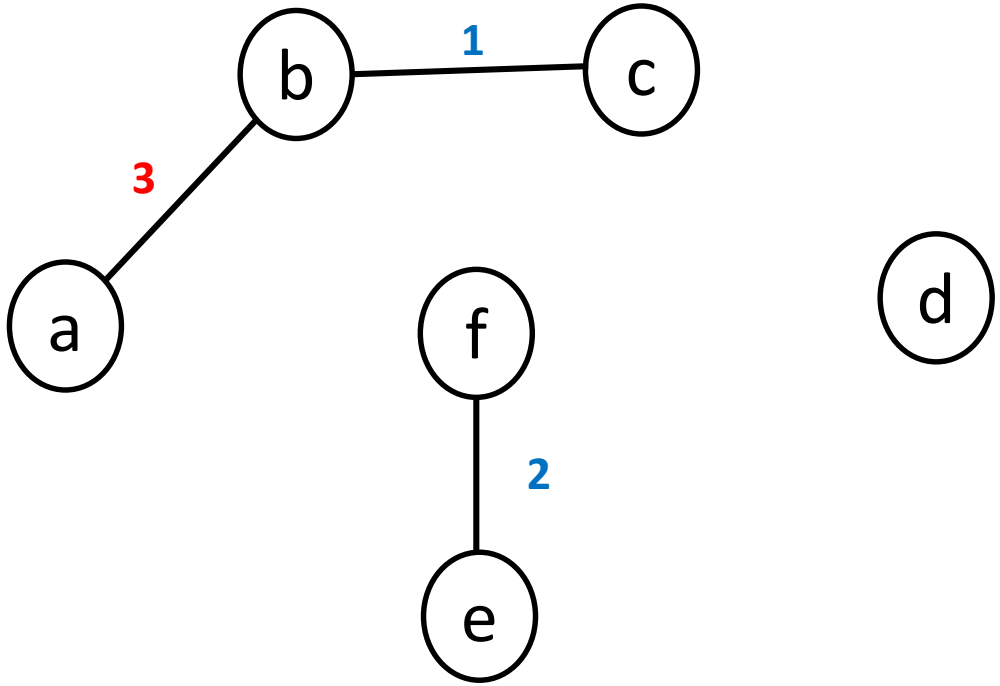
Sorted list of edges in non decreasing order

Edge(u,v)	Weight
bc	1
ef	2
ab	3
bf	4
cf	4
af	5
df	5
ae	6
cd	6
de	8

Design and Analysis of Algorithms

Kruskal's Algorithm

Given : Graph<V,E>



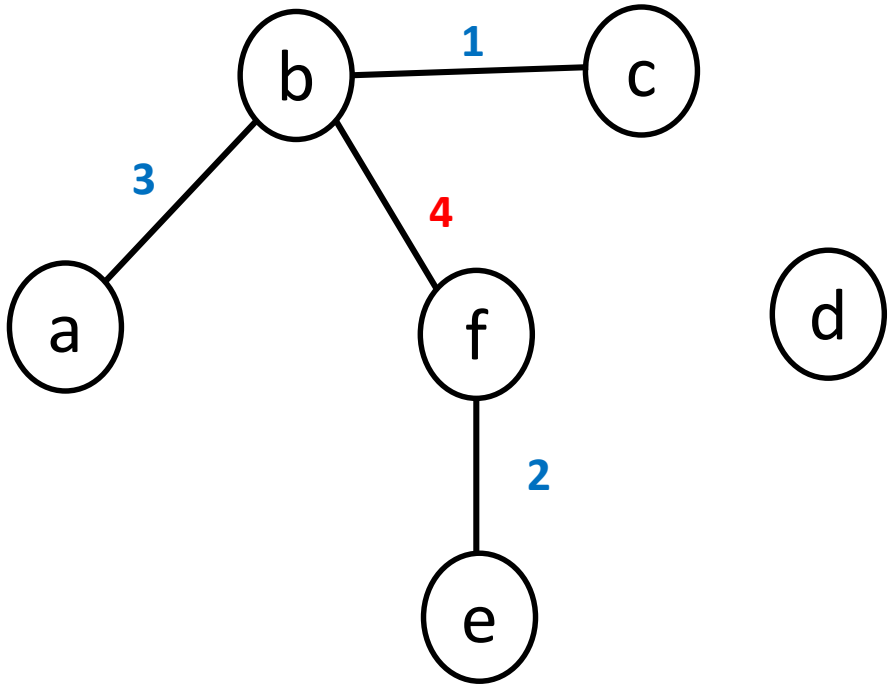
Sorted list of edges in non decreasing order

Edge(u,v)	Weight
bc	1
ef	2
ab	3
bf	4
cf	4
af	5
df	5
ae	6
cd	6
de	8

Design and Analysis of Algorithms

Kruskal's Algorithm

Given : Graph<V,E>



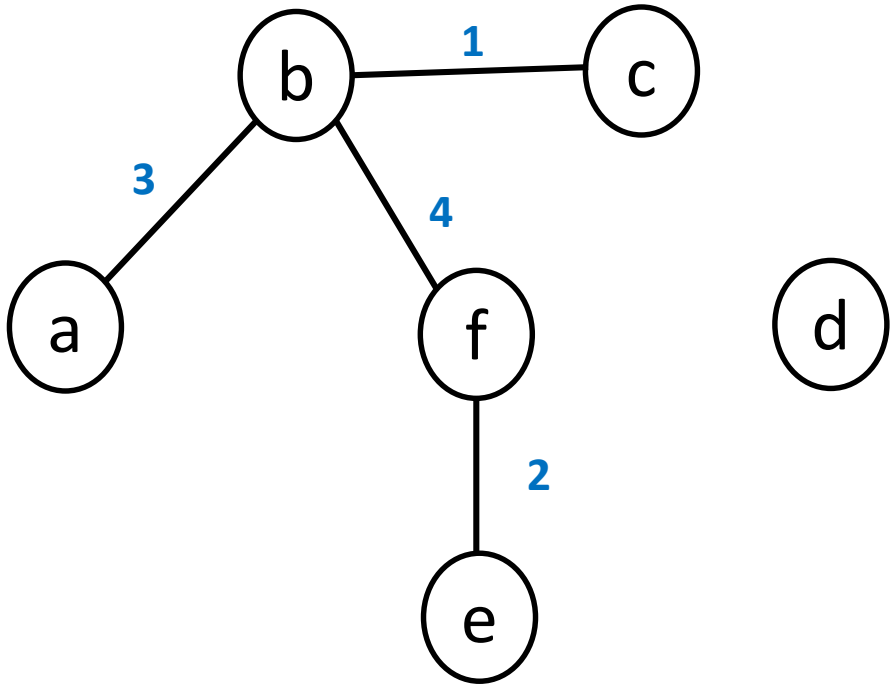
Sorted list of edges in non decreasing order

Edge(u,v)	Weight
bc	1
ef	2
ab	3
bf	4
cf	4
af	5
df	5
ae	6
cd	6
de	8

Design and Analysis of Algorithms

Kruskal's Algorithm

Given : Graph<V,E>



Sorted list of Edges in non decreasing Order

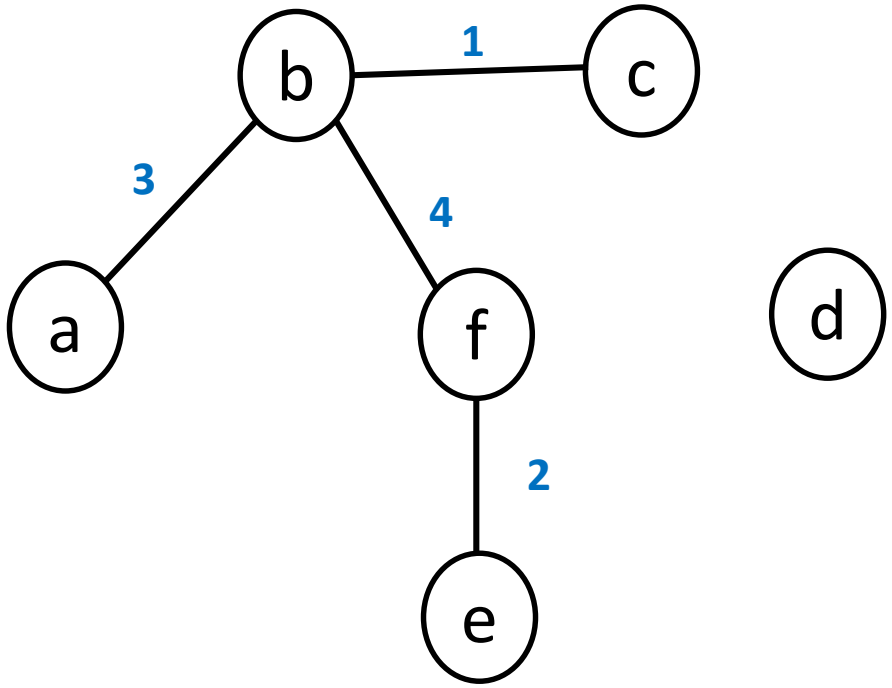
Edge(u,v)	Weight
bc	1
ef	2
ab	3
bf	4
cf	4 x
af	5
df	5
ae	6
cd	6
de	8

Cyclic

Design and Analysis of Algorithms

Kruskal's Algorithm

Given : Graph<V,E>



Sorted list of edges in non decreasing order

Edge(u,v)	Weight
bc	1
ef	2
ab	3
bf	4
cf	4
af	5
df	5
ae	6
cd	6
de	8

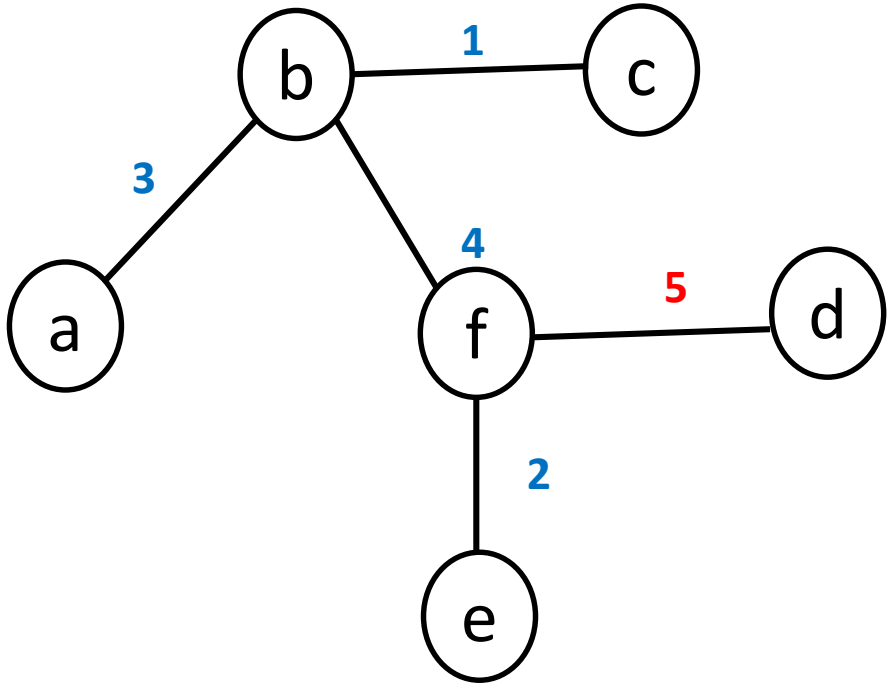
Cyclic

X

Design and Analysis of Algorithms

Kruskal's Algorithm

Given : Graph<V,E>



Sorted list of edges in non decreasing order

Edge(u,v)	Weight
bc	1
ef	2
ab	3
bf	4
cf	4
af	5
df	5
ae	6
cd	6
de	8

Acyclic

Before formalizing the above idea, lets quickly review the disjoint-set data structure.

1. *makeset*(***v***): Create a new set whose only member is pointed to by **v**. Note that for this operation v must already be in a set.
2. *find*(***v***): Returns a pointer to the set containing *v*.
3. *union*(**u,v**): Unites the dynamic sets that contain *u* and *v* into a new set that is union of these two sets.

For example, let ***S* = {1, 2, 3, 4, 5, 6}.**
Then ***makeset(i)*** creates the set ***{i}*** and applying this operation six times initializes the structure to the collection of six singleton sets:

{1}, {2}, {3}, {4}, {5}, {6}.

Performing ***union(1, 4)*** and ***union(5, 2)*** yields

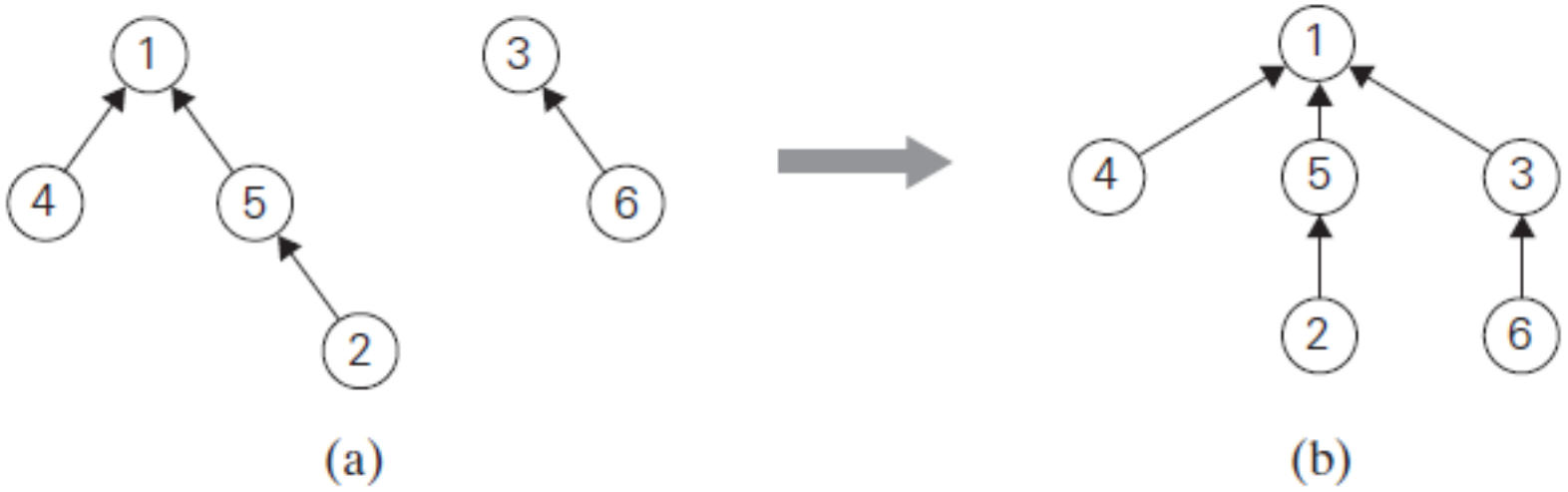
{1, 4}, {5, 2}, {3}, {6},

and, if followed by ***union(4, 5)*** and then by ***union(3, 6)***, we end up with the disjoint subsets

{1, 4, 5, 2}, {3, 6}.

(a) Forest representation of subsets {1, 4, 5, 2} and {3, 6}

(b) Result of *union*(5, 6).

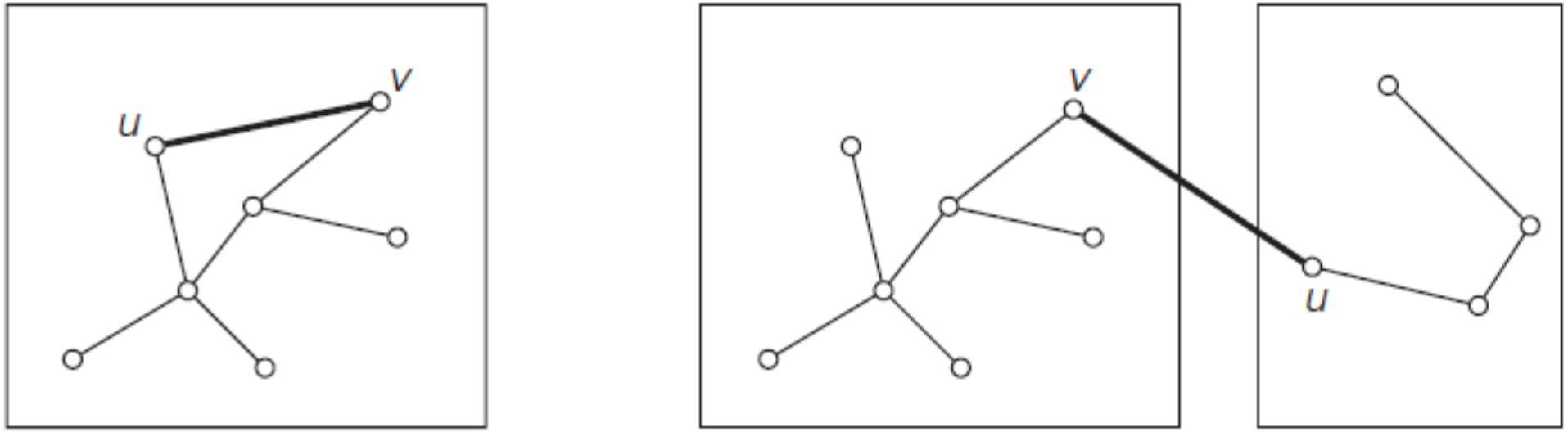


Kruskal's Algorithm

ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

Kruskal's Algorithm



With an efficient Sorting algorithm and an Union-Find algorithm.

Efficiency: $\Theta(|E| \log |E|)$

Design and Analysis of Algorithms

Text Book

Chapter 9 ,Introduction to The Design and Analysis of Algorithms by Anany Levitin

THANK YOU

Bharathi R

Department of Computer Science & Engineering

rbharathi@pes.edu

Design and Analysis of Algorithms
Unit -4

Bharathi R
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Unit 4: Greedy Technique

Dijkstra's Algorithm

Bharathi R

Department of Computer Science & Engineering

Dijkstra's Algorithm

Single Source Shortest Paths Problem: Given a weighted connected (directed) graph G, find shortest paths from source vertex ***s*** to each of the other vertices

Dijkstra's algorithm: Similar to Prim's MST algorithm, with a different way of computing numerical labels: Among vertices not already in the tree, it finds vertex ***u*** with the smallest sum

$$d_v + w(v,u)$$

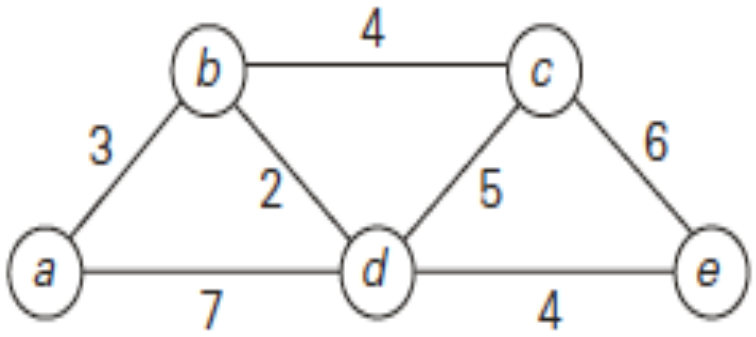
where

v is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree rooted at ***s***)

d_v is the length of the shortest path from source ***s*** to ***v***

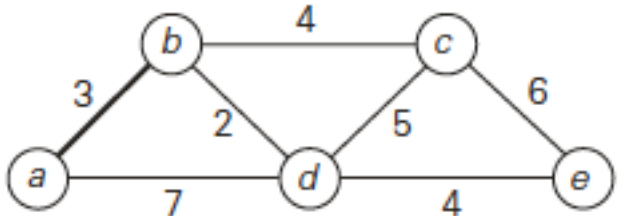
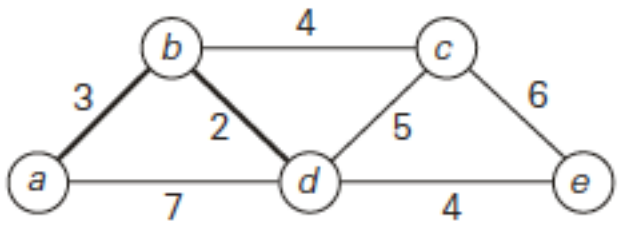
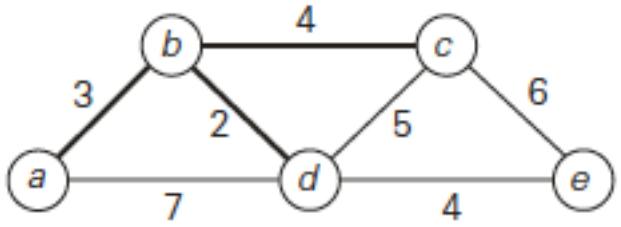
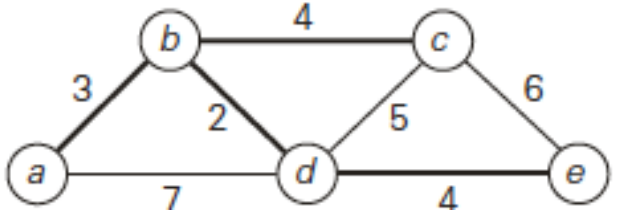
$w(v,u)$ is the length (weight) of edge from ***v*** to ***u***

Example



Design and Analysis of Algorithms

Dijkstra's Algorithm-Example

Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

The next closest vertex is shown in bold.

Design and Analysis of Algorithms

Dijkstra's Algorithm

ALGORITHM *Dijkstra*(G, s)

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
//        and its vertex s
//Output: The length d_v of a shortest path from s to v
//         and its penultimate vertex p_v for every vertex v in V
Initialize(Q)  //initialize priority queue to empty
for every vertex v in V
    d_v ← ∞;  p_v ← null
    Insert(Q, v, d_v)  //initialize vertex priority in the priority queue
d_s ← 0;  Decrease(Q, s, d_s)  //update priority of s with d_s
V_T ← ∅
for i ← 0 to |V| − 1 do
    u* ← DeleteMin(Q)  //delete the minimum priority element
    V_T ← V_T ∪ {u*}
    for every vertex u in V − V_T that is adjacent to u* do
        if d_u* + w(u*, u) < d_u
            d_u ← d_u* + w(u*, u);  p_u ← u*
            Decrease(Q, u, d_u)
```

Dijkstra's Algorithm- Time Efficiency

The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself.

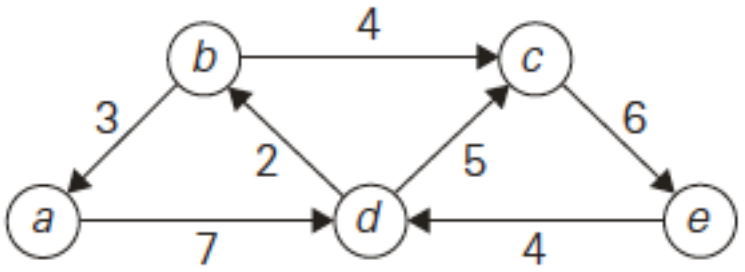
- **$O(|V|^2)$** for graphs represented by weight matrix and array implementation of priority queue
- **$O(|E|\log|V|)$** for graphs represented by adj. lists and min-heap implementation of priority queue

Notes on Dijkstra's Algorithm

- Correctness can be proven by induction on the number of vertices.
- Doesn't work for graphs with negative weights (whereas Floyd's algorithm does, as long as there is no negative cycle).
- Applicable to both undirected and directed graphs
- Don't mix up Dijkstra's algorithm with Prim's algorithm!

Exercises

Solve the following instances of the single-source shortest-paths problem with vertex a as the source:



References

Chapter-9 Greedy Technique

Introduction to the Design & Analysis of Algorithms- Anany Levitin

THANK YOU

Bharathi R

Department of Computer Science & Engineering

rbharathi@pes.edu

Design and Analysis of Algorithms
Unit -4

Bharathi R
Department of Computer Science & Engineering

DESIGN AND ANALYSIS OF ALGORITHMS

Unit 4: Greedy Technique- Huffman Trees

Bharathi R

Department of Computer Science & Engineering

Huffman Trees

The Huffman trees are constructed for encoding a given text of n characters.

While encoding a given text, each character is associated with some of bits called the ***codeword***.

- **Fixed length encoding**: Assigns to each character a bit string of the same length.
- **Variable length encoding**: Assigns codewords of different lengths to different characters.
- **Prefix free code**: In Prefix free code, no codeword is a prefix of a codeword of another character.

Binary prefix code :

- The characters are associated with the leaves of a binary tree.
- All left edges are labeled as 0, and right edges as 1.
- Codeword of a character is obtained by recording the labels on the simple path from the root to the character's leaf.
- Since, there is no simple path to a leaf that continues to another leaf, no codeword can be a prefix of another codeword.

About Huffman Algorithm:

- Invented by David A Huffman in 1951.
- Constructs binary prefix code tree.
- Huffman's algorithm achieves data compression by finding the best variable length binary encoding scheme for the symbols that occur in the file to be compressed. Huffman coding uses frequencies of the symbols in the string to build a variable rate prefix code
  - Each symbol is mapped to a binary string
  - More frequent symbols have shorter codes
  - No code is a prefix of another code (prefix free code)
- Huffman Codes for data compression achieves 20-90% Compression.

Huffman Algorithm:

Input: Alphabet and frequency of each symbol in the text.

Step 1: Initialize n one-node trees (forest) and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's weight. (Generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2: Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight. Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

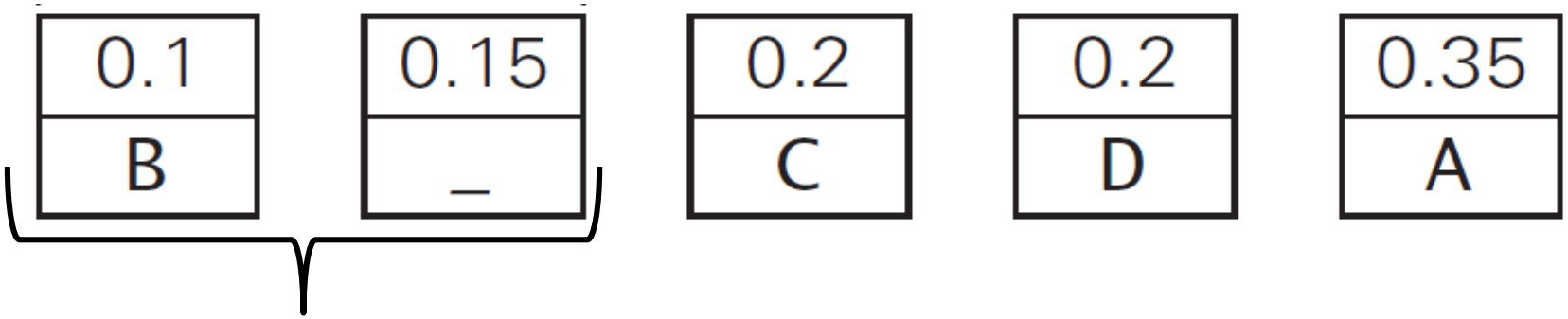
Design and Analysis of Algorithms

Huffman Algorithm:

EXAMPLE

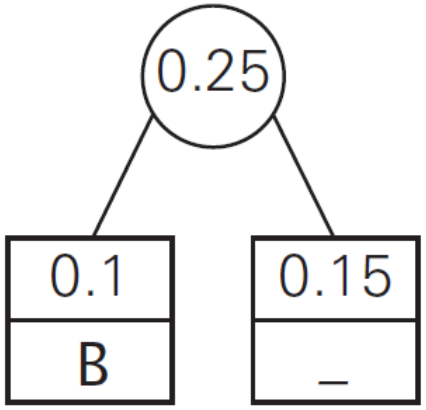
symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

1.First arrange the characters in ascending order of their probabilities

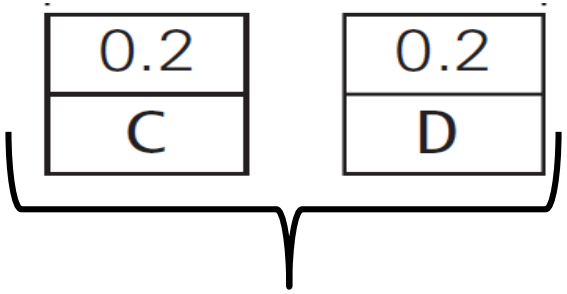


Combine these two nodes and form a single node

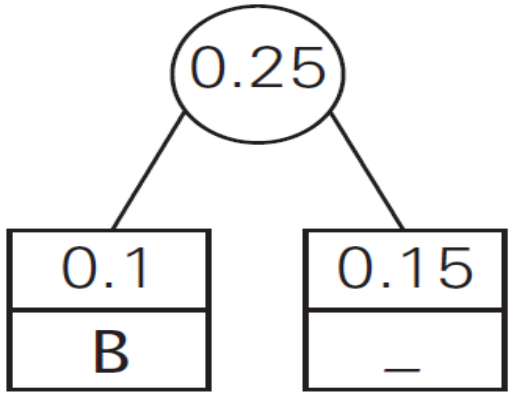
Huffman Algorithm:

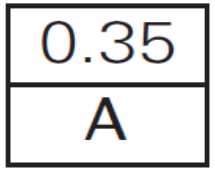


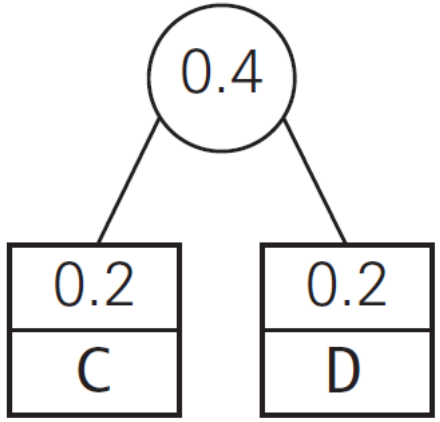
2.Again arrange probabilities in ascending order



Combine these nodes



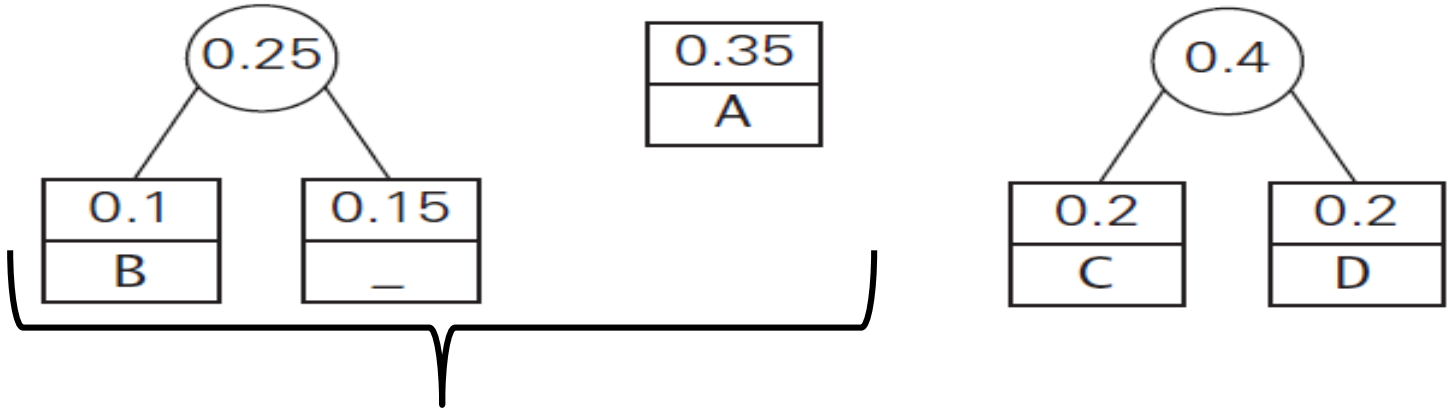




Design and Analysis of Algorithms

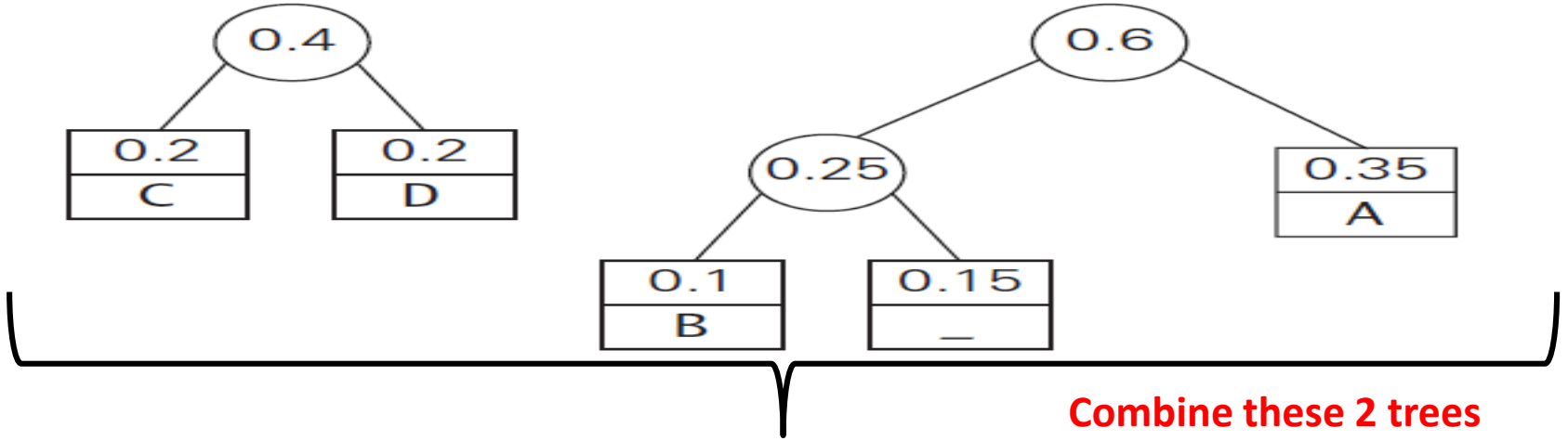
Huffman Algorithm:

3.Again arrange probabilities in ascending order



Combine these 2 trees

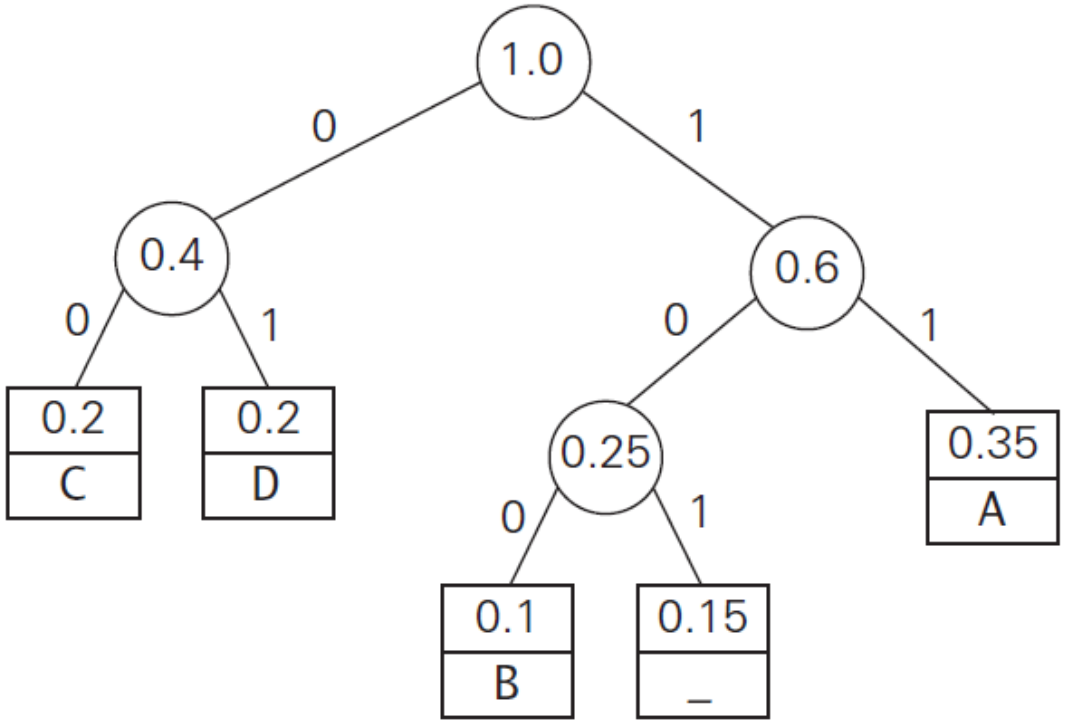
4. Arrange probabilities in ascending order



Combine these 2 trees

Design and Analysis of Algorithms

Huffman Tree



DAD is encoded as 011101, and
10011011011101 is decoded as BAD_AD

The resulting codewords are as follows

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Compression Ratio

Compression Ratio : A standard measure of a compression algorithm's efficiency.

Character	A	B	C	D	-
probability	0.4	0.1	0.2	0.15	0.15

Character	A	B	C	D	-
probability	0.4	0.1	0.2	0.15	0.15
Code word	0	100	111	101	110

Compute compression ratio:

Bits per character = Codeword length * Frequency

= (1 * 0.4) + (3 * 0.1) + (3 * 0.2) + (3 * 0.15) + (3 * 0.15)

= 2.20

Compression ratio is = (3 – 2.20)/ 3 . 100% = 26.6%

Features of Huffman's Encoding

- Huffman's encoding is one of the most important file-compression methods.
- In addition to its simplicity and versatility, it yields an optimal, i.e., minimal-length, encoding (provided the frequencies of symbol occurrences are independent and known in advance).
- The simplest version of Huffman compression calls, in fact, for a preliminary scanning of a given text to count the frequencies of symbol occurrences in it. Then these frequencies are used to construct a Huffman coding tree and encode the text.

Design and Analysis of Algorithms

Text Books

Chapter-9 Greedy Technique
Introduction to the Design & Analysis of Algorithms- Anany Levitin

THANK YOU

Bharathi R

Department of Computer Science & Engineering

rbharathi@pes.edu