Heap and Heap Sort

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.)

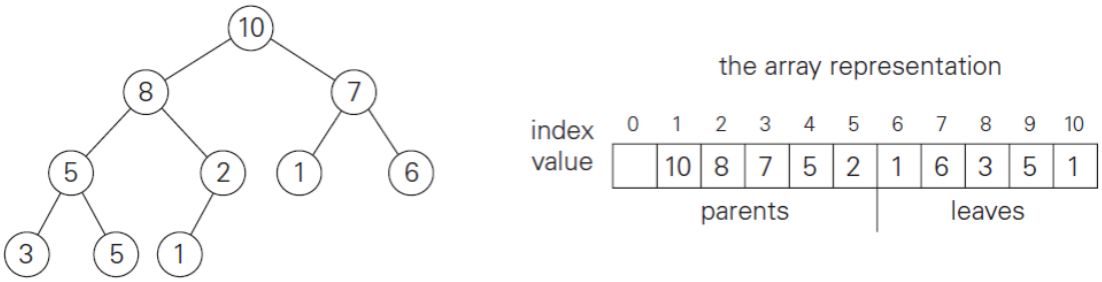


FIGURE 6.10 Heap and its array representation.

properties of heaps

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$.
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 top-down, 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 $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions;
   b. the children of a key in the array's parental position i $(1 \le i \le \lfloor n/2 \rfloor)$ 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 $\lfloor i/2 \rfloor$.

Thus, we could also define a heap as an array $H[1..n]$ in which every element in position i in the first half of the array is greater than or equal to the elements in positions $2i$ and $2i + 1$, i.e.,

$$H[i] \ge \max\{H[2i], H[2i + 1]\} \quad \text{for } i = 1, \ldots, \lfloor n/2 \rfloor.$$

Bottom Up Construction

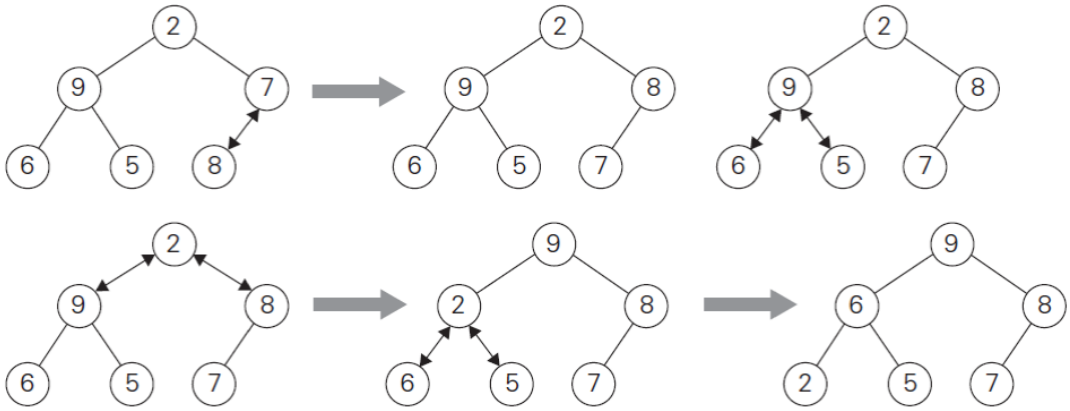


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

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
```

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

Top Down Construction

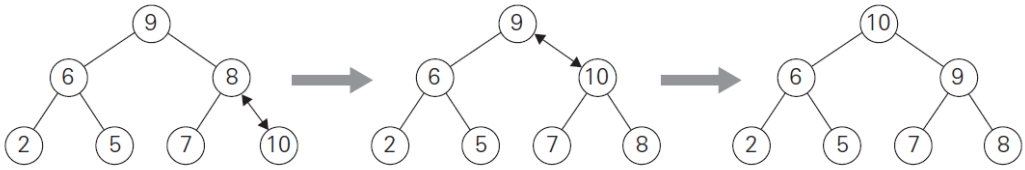


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

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.

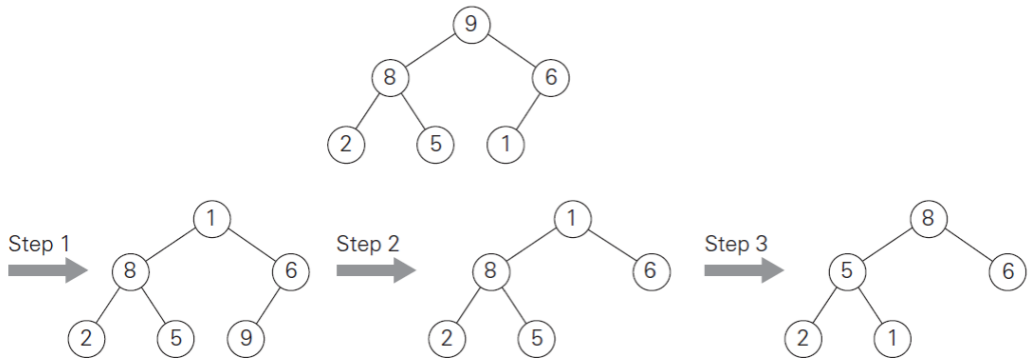


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

Heapsort

Now we can describe ***heapsort***—an interesting sorting algorithm discovered by J. W. J. Williams [Wil64]. This is a 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.

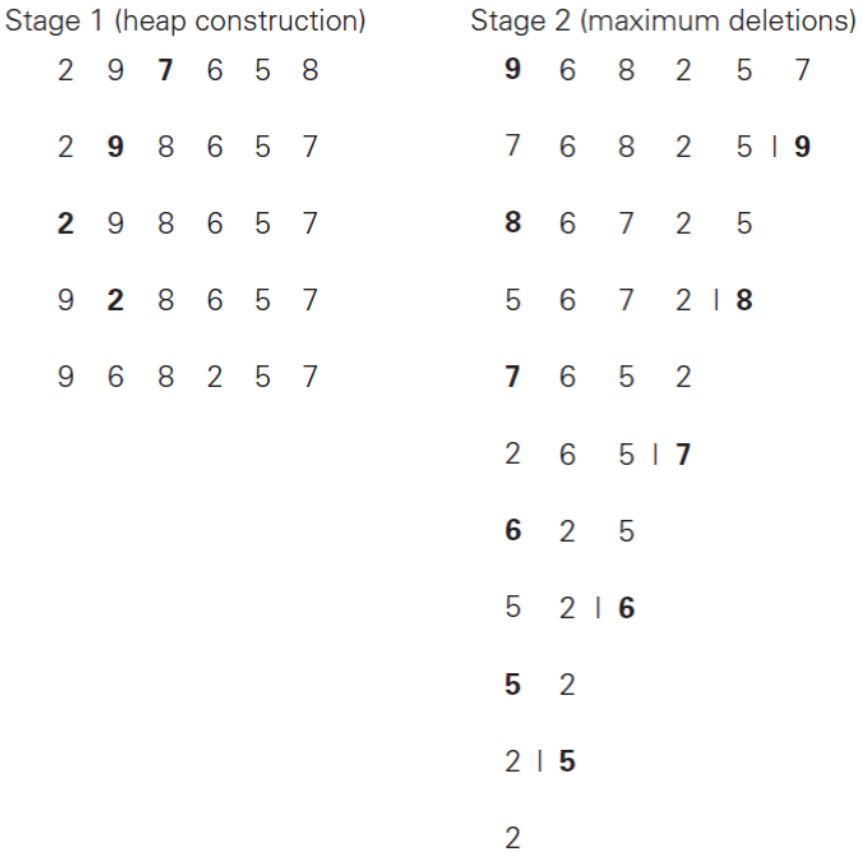


FIGURE 6.14 Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

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

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

Red Black Tree

Introduction:
A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (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. This tree was invented in 1972 by Rudolf Bayer.

It must be noted that as each node requires only 1 bit of space to store the colour information, these types of trees show identical memory footprint to the classic (uncoloured) binary search tree.

Rules That Every Red-Black Tree Follows:

1. Every node has a colour either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.

Comparison with AVL Tree:
The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

Interesting points about Red-Black Tree:

1. 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.
2. Height of a red-black tree with n nodes is $h <= 2 \log_2(n + 1)$.

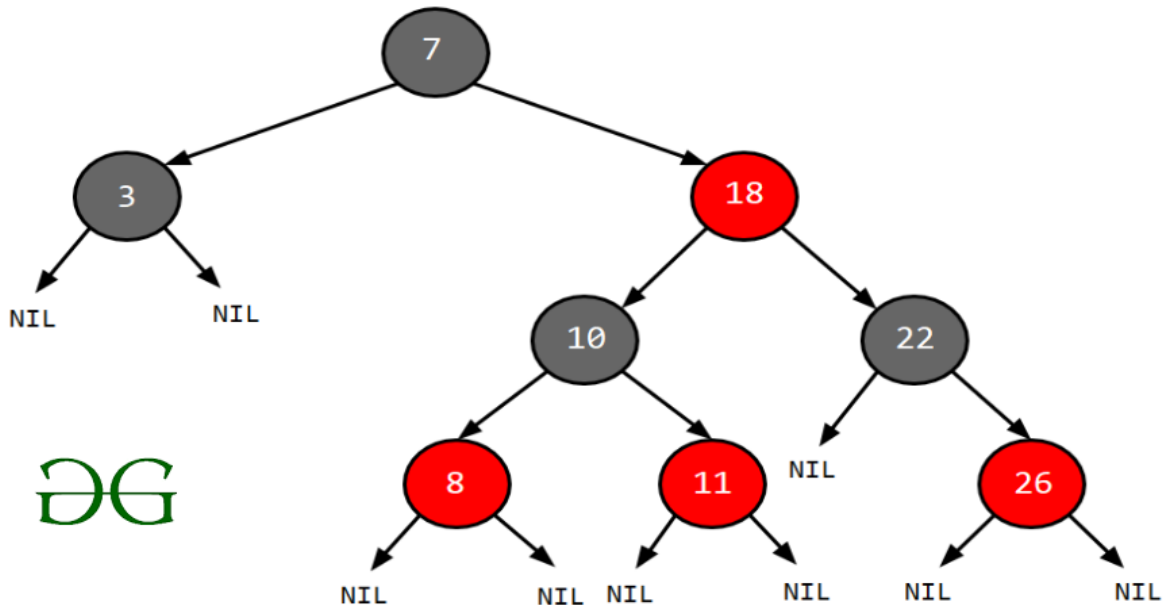
3. All leaves (NIL) are black.
4. The black depth of a node is defined as the number of black nodes from the root to that node i.e the number of black ancestors.
5. Every red-black tree is a special case of a binary tree.

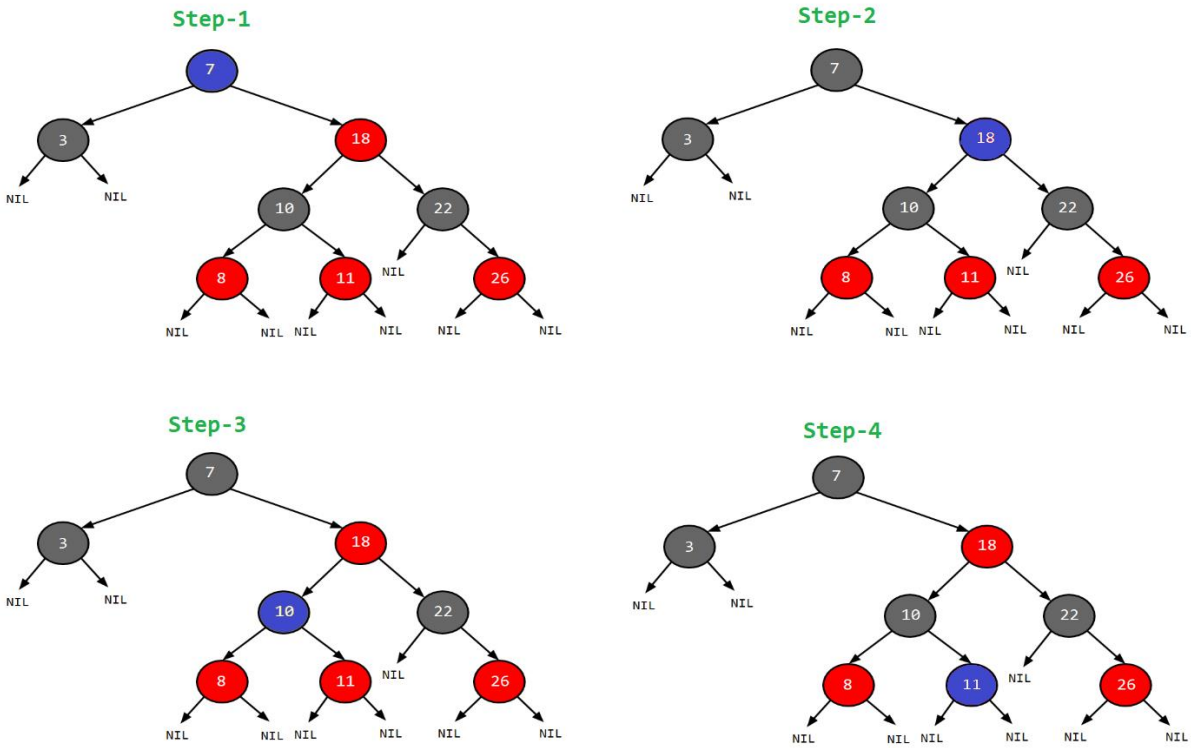
Search Operation in Red-black Tree:
As every red-black tree is a special case of a binary tree so the searching algorithm of a red-black tree is similar to that of a binary tree.

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





Applications:

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

Insertion in Red Black Tree:

Algorithm:

Let x be the newly inserted node.

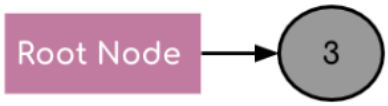
1. Perform standard BST insertion and make the colour of newly inserted nodes as RED.
2. If x is the root, change the colour of x as BLACK (Black height of complete tree increases by 1).
3. Do the following if the color of x's parent is not BLACK **and** x is not the root.

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

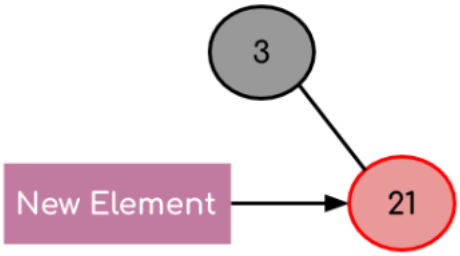
   4). (i) Change the colour of parent and uncle as BLACK.

(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)

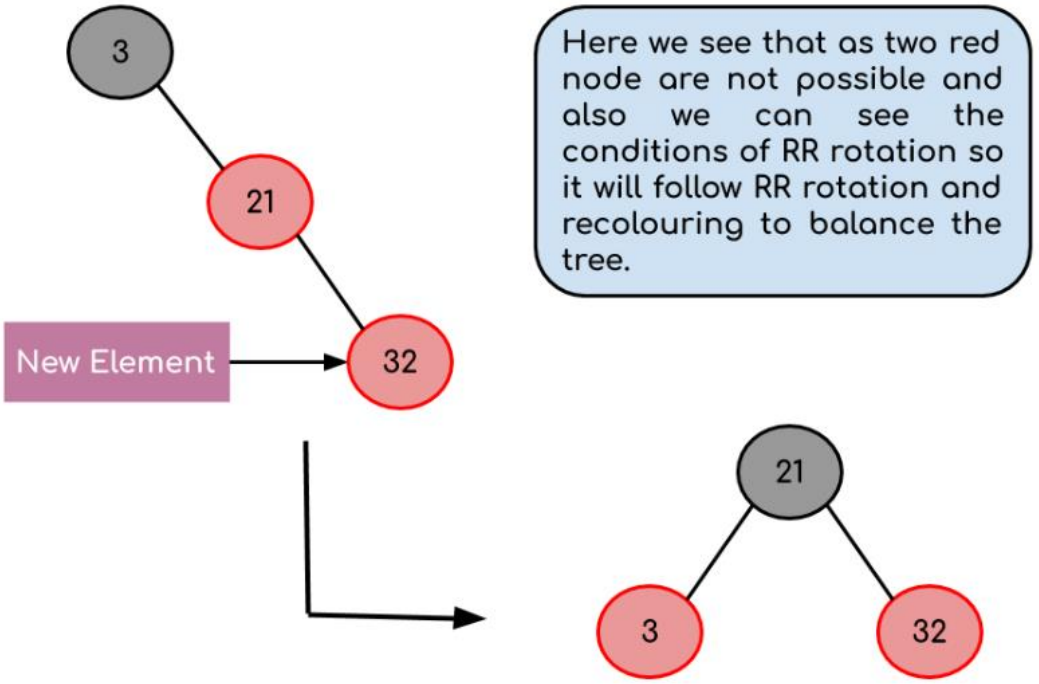
Step 1: Inserting element 3 inside the tree.



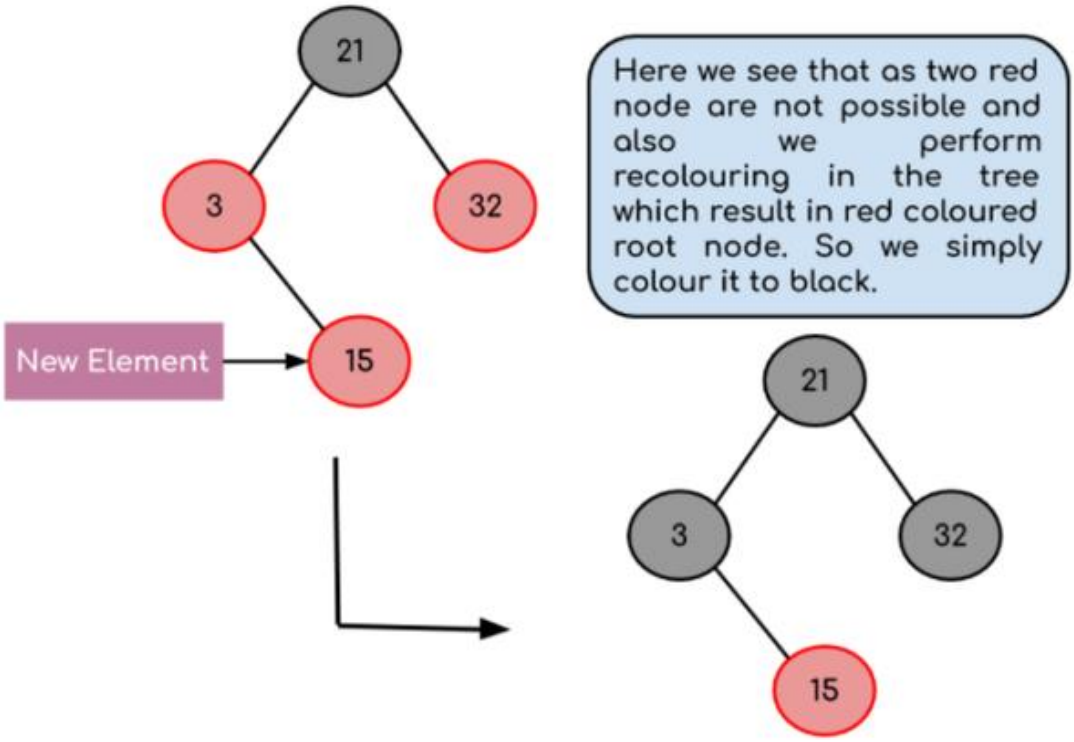
Step 2: Inserting element 21 inside the tree.



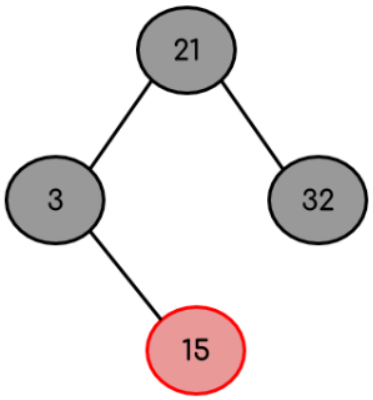
Step 3: Inserting element 32 inside the tree.



Step 4: Inserting element 15 inside the tree.



Final Red Black Tree:



2-3 Tree

a **2–3 tree** is a tree data structure, where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements. A 2-3 tree is a B-tree of order 3. Nodes on the outside of the tree (leaf nodes) have no children and one or two data elements. 2−3 trees were invented by John Hopcroft in 1970

2–3 trees are required to be balanced, meaning that each leaf is at the same level. It follows that each right, center, and left subtree of a node contains the same or close to the same amount of data.

We say that an internal node is a **2-node** if it has *one* data element and *two* children.

We say that an internal node is a **3-node** if it has *two* data elements and *three* children.

We say that *T* is a **2–3 tree** if and only if one of the following statements hold:

- *T* is empty. In other words, *T* does not have any nodes.
- *T* is a 2-node with data element *a*. If *T* has left child *p* and right child *q*, then
    - *p* and *q* are 2–3 trees of the same height
    - *a* is greater than each element in *p*; and
    - *a* is less than or equal to each data element in *q*.
- *T* is a 3-node with data elements *a* and *b*, where $a < b$. If *T* has left child *p*, middle child *q*, and right child *r*, then
    - *p*, *q*, and *r* are 2–3 trees of equal height;
    - *a* is greater than each data element in *p* and less than or equal to each data element in *q*; and
    - *b* is greater than each data element in *q* and less than or equal to each data element in *r*.

Properties

- Every internal node is a 2-node or a 3-node.
- All leaves are at the same level.
- All data is kept in sorted order.

Searching

Searching for an item in a 2–3 tree is similar to searching for an item in a binary search tree. Since the data elements in each node are ordered, a search function will be directed to the correct subtree and eventually to the correct node which contains the item.

1. Let *T* be a 2–3 tree and *d* be the data element we want to find. If *T* is empty, then *d* is not in *T* and we're done.
2. Let *t* be the root of *T*.
3. Suppose *t* is a leaf.
    - If *d* is not in *t*, then *d* is not in *T*. Otherwise, *d* is in *T*. We need no further steps and we're done.

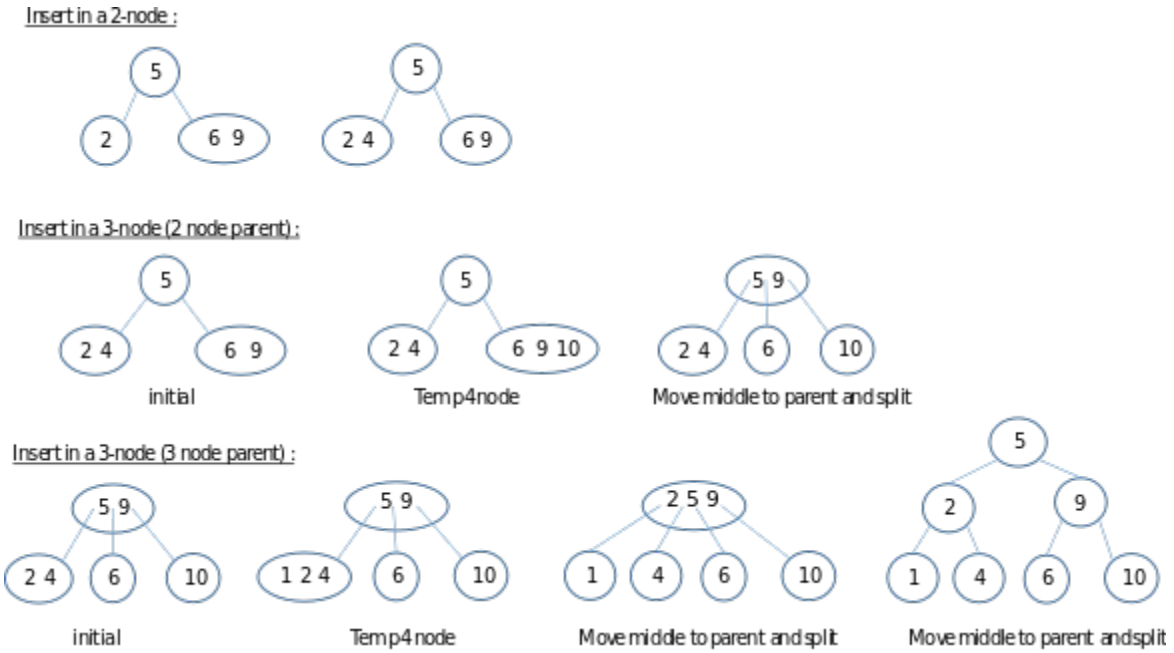
4. Suppose *t* is a 2-node with left child *p* and right child *q*. Let *a* be the data element in *t*. There are three cases:
    - If *d* is equal to *a*, then we've found *d* in *T* and we're done.
    - If d<a, then set *T* to *p*, which by definition is a 2–3 tree, and go back to step 2.
    - If d<a, then set *T* to *q* and go back to step 2.
5. Suppose *t* is a 3-node with left child *p*, middle child *q*, and right child *r*. Let *a* and *b* be the two data elements of *t*, where a<b. There are four cases:
    - If *d* is equal to *a* or *b*, then *d* is in *T* and we're done.
    - If d<a, then set *T* to *p* and go back to step 2.
    - If d<a, then set *T* to *q* and go back to step 2.
    - If d<a, then set *T* to *r* and go back to step 2.

Insertion

Insertion maintains the balanced property of the tree.[5]

To insert into a 2-node, the new key is added to the 2-node in the appropriate order.

To insert into a 3-node, more work may be required depending on the location of the 3-node. If the tree consists only of a 3-node, the node is split into three 2-nodes with the appropriate keys and children.



The delete operation

Deleting key k is similar to inserting: there is a special case when T is just a single (leaf) node containing k (T is made empty); otherwise, the parent of the node to be deleted is found, then the tree is fixed up if necessary so that it is still a 2-3 tree.

Once node n (the parent of the node to be deleted) is found, there are two cases, depending on how many children n has:

case 1: n has 3 children

- Remove the child with value k, then fix n.leftMax, n.middleMax, and n's ancestors' leftMax and middleMax fields if necessary.

case 2: n has only 2 children

- If n is the root of the tree, then remove the node containing k. Replace the root node with the other child (so the final tree is just a single leaf node).
- If n has a left or right sibling with 3 kids, then:
    - remove the node containing k
    - "steal" one of the sibling's children
    - fix n.leftMax, n.middleMax, and the leftMax and middleMax fields of n's sibling and ancestors as needed.
- If n's sibling(s) have only 2 children, then:
    - remove the node containing k
    - make n's remaining child a child of n's sibling
    - fix leftMax and middleMax fields of n's sibling as needed
    - remove n as a child of its parent, using essentially the same two cases (depending on how many children n's parent has) as those just discussed

The time for delete is similar to insert; the worst case involves one traversal down the tree to find n, and another "traversal" up the tree, fixing leftMax and middleMax fields along the way (the traversal up is really actions that happen after the recursive call to delete has finished).

B Tree

B Tree is a Self-balancing search tree that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree generalizes the binary search tree, allowing for nodes with more than two children. Unlike other self-balancing binary search trees, the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as disks. It is commonly used in databases and file systems. B-trees were invented by Rudolf Bayer and Edward M.McCreight while working at Boeing Research Labs, for the purpose of efficiently managing index pages for large random access files. The basic assumption was that indexes would be so voluminous that only small chunks of the tree could fit in main memory.

A B-tree of order m is a tree which satisfies the following properties:

1. Every node has at most m children.
2. Every non-leaf node (except root) has at least $\lceil m/2 \rceil$ child nodes.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with k children contains $k - 1$ keys.
5. All leaves appear in the same level and carry no information.

Each internal node's keys act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys: a_1 and a_2. All values in the leftmost subtree will be less than a_1, all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2.

Internal nodes

Internal nodes are all nodes except for leaf nodes and the root node. They are usually represented as an ordered set of elements and child pointers. Every internal node contains a **maximum** of U children and a **minimum** of L children. Thus, the number of elements is always 1 less than the number of child pointers (the number of elements is between $L-1$ and $U-1$). U must be either $2L$ or $2L-1$; therefore each internal node is at least half full. The relationship between U and L implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there's room to push one element up into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

The root node

The root node's number of children has the same upper limit as internal nodes, but has no lower limit. For example, when there are fewer than $L-1$ elements in the entire tree, the root will be the only node in the tree with no children at all.

Leaf nodes

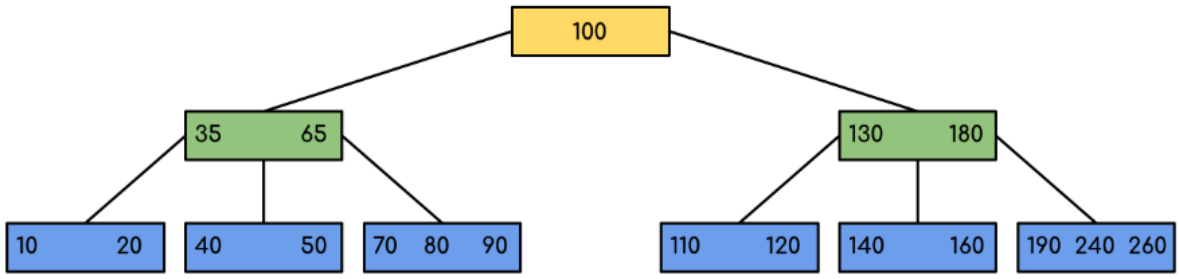
In Knuth's terminology, leaf nodes do not carry any information. The internal nodes that are one level above the leaves are what would be called "leaves" by other authors: these nodes only store keys (at most m-1, and at least m/2-1 if they are not the root) and pointers to nodes carrying no information.

A B-tree of depth n+1 can hold about U times as many items as a B-tree of depth n, but the cost of search, insert, and delete operations grows with the depth of the tree. As with any balanced tree, the cost grows much more slowly than the number of elements.

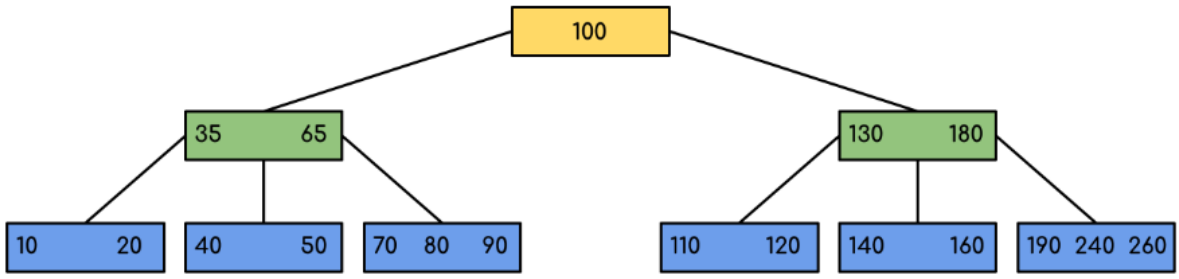
Some balanced trees store values only at leaf nodes, and use different kinds of nodes for leaf nodes and internal nodes. B-trees keep values in every node in the tree except leaf nodes.

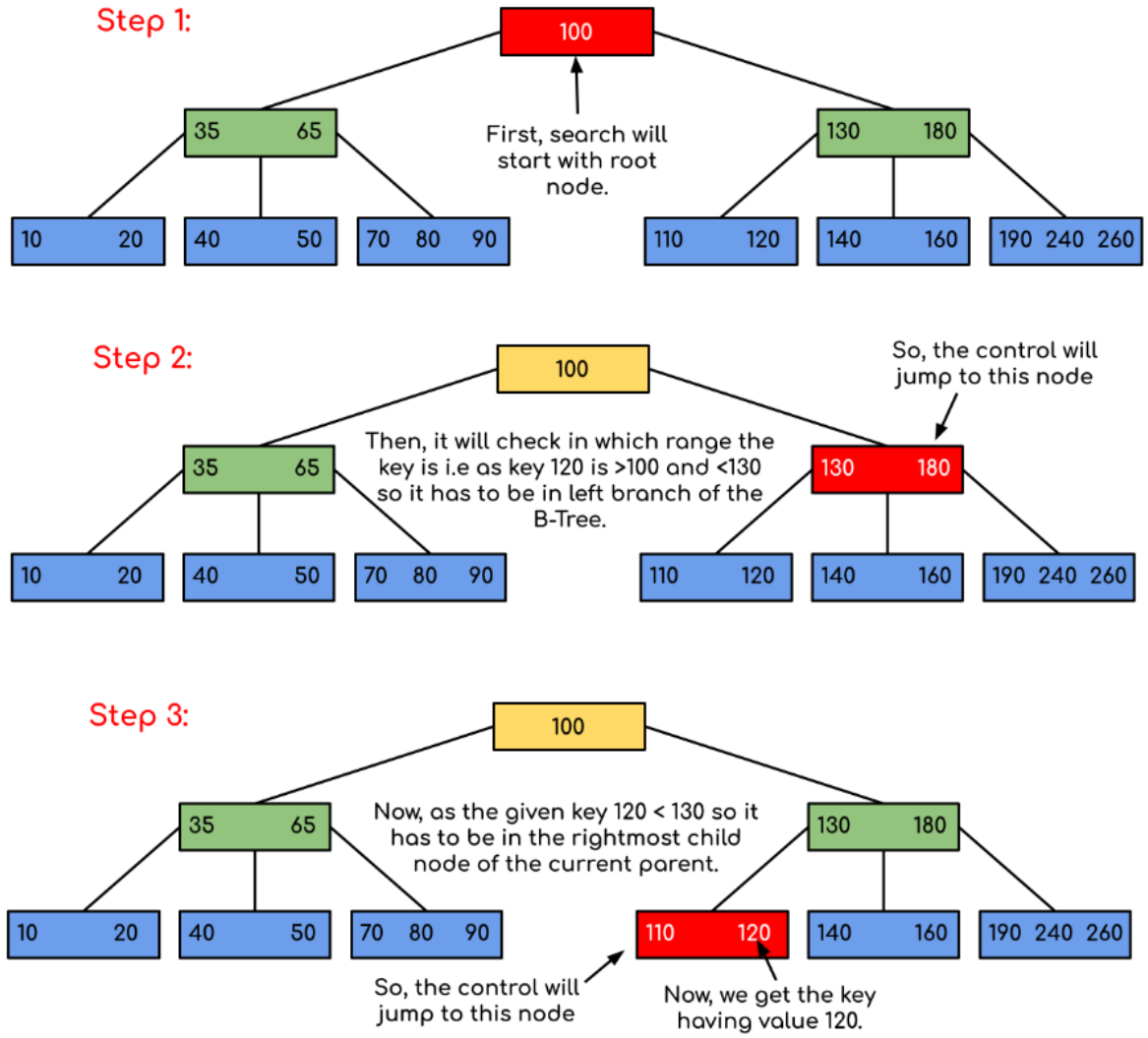
Properties of B-Tree:

1. All leaves are at the same level.
2. A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.
3. Every node except root must contain at least (ceiling)([t-1]/2) keys. The root may contain minimum 1 key.
4. All nodes (including root) may contain at most t – 1 keys.
5. Number of children of a node is equal to the number of keys in it plus 1.
6. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
7. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).



Searching 120 in the given B-Tree





Insertion

1. Initialize x as root.

2) While x is not leaf, do following

..**a)** Find the child of x that is going to be traversed next. Let the child be y.

..**b)** If y is not full, change x to point to y.

..**c)** If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. Else second part of y. When we split y, we move a key from y to its parent x.

3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

Let us understand the algorithm with an example tree of minimum degree 't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.

Initially root is NULL. Let us first insert 10.

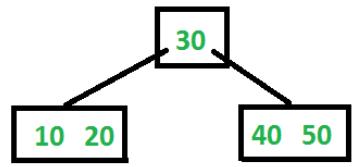
Insert 10

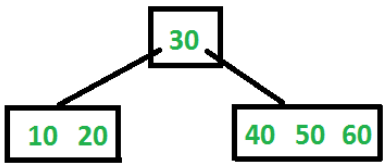
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because the maximum number of keys a node can accommodate is 2*t – 1 which is 5.

Insert 20, 30, 40 and 50

Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.

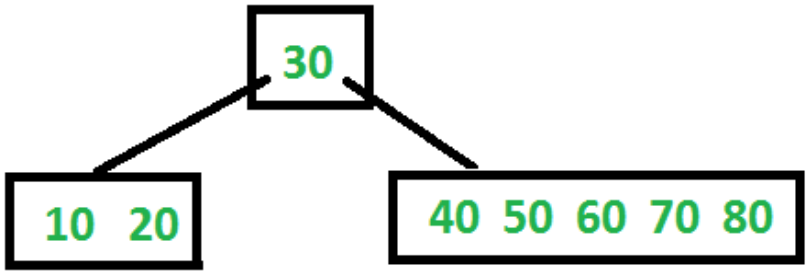
Insert 60



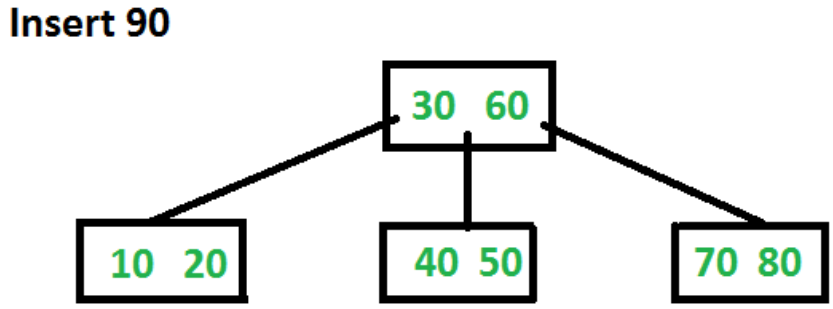


Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.



Delete Operation in B-Tree

Deletion from a B-tree is more complicated than insertion, because we can delete a key from any node-not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children.

The deletion procedure deletes the key k from the subtree rooted at x. This procedure guarantees that whenever it calls itself recursively on a node x, the number of keys in x is at least the minimum degree t . Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b then we delete x, and x's only child x.c1 becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

1.

3. If the key k is not present in internal node x, determine the root x.c(i) of the appropriate subtree that must contain k, if k is in the tree at all. If x.c(i) has only t-1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

a) If x.c(i) has only t-1 keys but has an immediate sibling with at least t keys, give x.c(i) an extra key by moving a key from x down into x.c(i), moving a key from x.c(i) 's immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into x.c(i).

b) If x.c(i) and both of x.c(i)'s immediate siblings have t-1 keys, merge x.c(i) with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

If the key k is in node x and x is a leaf, delete the key k from x.

2. If the key k is in node x and x is an internal node, do the following.

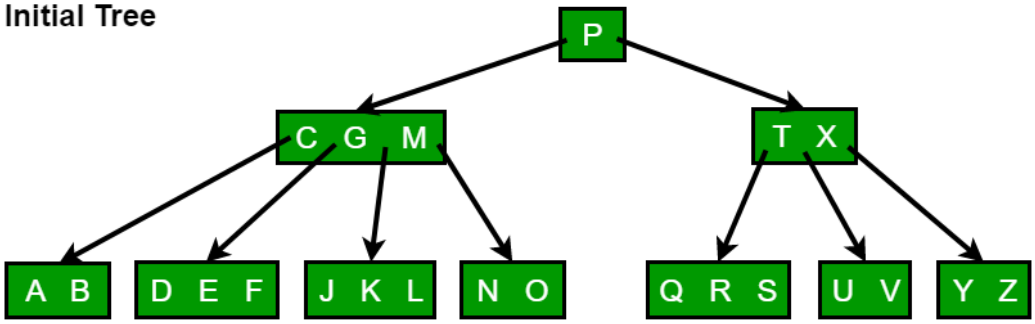
a) If the child y that precedes k in node x has at least t keys, then find the predecessor k0 of k in the sub-tree rooted at y. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)

b) If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k0 of k in the subtree rooted at z. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)

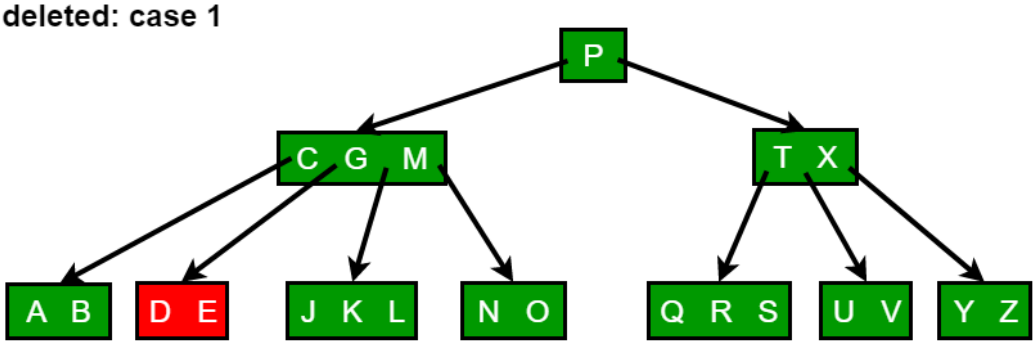
c) Otherwise, if both y and z have only t-1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 keys. Then free z and recursively delete k from y.

Since most of the keys in a B-tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor

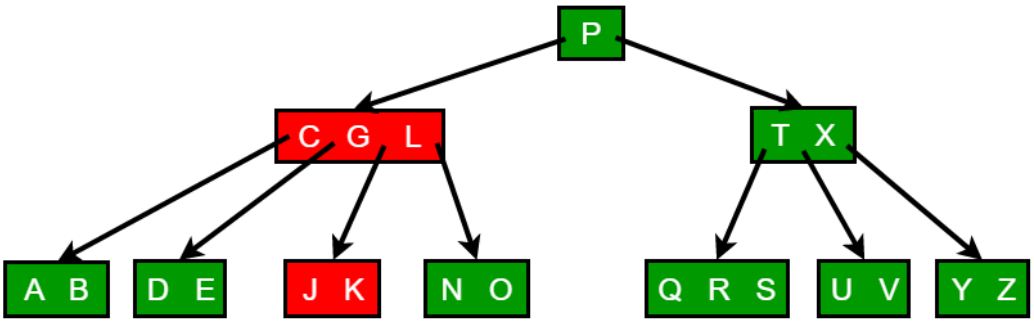
(a) Initial Tree



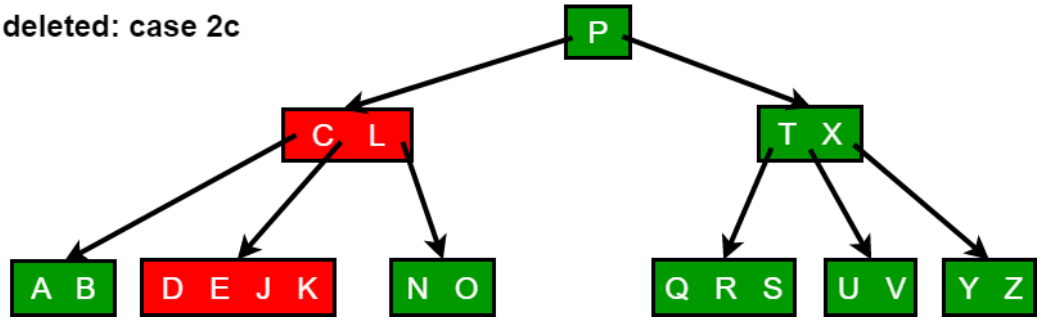
(b) F deleted: case 1



(c) M deleted: case 2a



(d) G deleted: case 2c



PES UNIVERSITY

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

Unit-4

2. Distribution Counting Sort

1. **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.

Consider sorting the array

13 11 12 13 12 12

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:

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

Note that the distribution values indicate the proper positions for the last occurrences of their elements in the final sorted array. If we index array positions from 0 to $n - 1$, the distribution values must be reduced by 1 to get corresponding element positions.

It is more convenient to process the input array right to left. For the example, the last element is 12, and, since its distribution value is 4, place this 12 in position 4 − 1= 3 of the array *S* that will hold the sorted list. Then decrease the 12's distribution value by 1 and proceed to the next (from the right) element in the given array. The entire processing of this example is depicted in Figure below,

	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 *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 ≤ u)
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for j ← 0 to u − l do D[j] ← 0                          //initialize frequencies
for i ← 0 to n − 1 do D[A[i] − l] ← D[A[i] − l] + 1     //compute frequencies
for j ← 1 to u − l do D[j] ← D[j − 1] + D[j]            //reuse for distribution
for i ← n − 1 downto 0 do
    j ← A[i] − l
    S[D[j] − 1] ← A[i]
    D[j] ← D[j] − 1
return S
```

Assuming that the range of array values is fixed, this is obviously a linear algorithm because it makes just two consecutive passes through its input array *A*. This is a better time-efficiency class than that of the most efficient sorting algorithms like mergesort, quicksort, and heapsort. 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.

PES UNIVERSITY

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

Unit-4

3. Input Enhancement in String Matching – Horspool's algorithm

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

Several faster algorithms have been discovered. Most of them exploit the 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. This is exactly the idea behind the two best known algorithms of this type: the Knuth-Morris-Pratt algorithm and the Boyer-Moore algorithm. The principal difference between these two algorithms lies in the way they compare characters of a pattern with their counterparts in a text: the Knuth-Morris-Pratt algorithm does it left to right, whereas the Boyer-Moore algorithm does it right to left.

Although the underlying idea of the Boyer-Moore algorithm is simple, its actual implementation in a working method is less so. Therefore, we start our discussion with a simplified version of the Boyer-Moore algorithm suggested by R. Horspool . In addition to being simpler, Horspool's algorithm is not necessarily less efficient than the Boyer-Moore algorithm on random strings.

Horspool's Algorithm

Consider, as an example, searching for the pattern BARBER in some text:

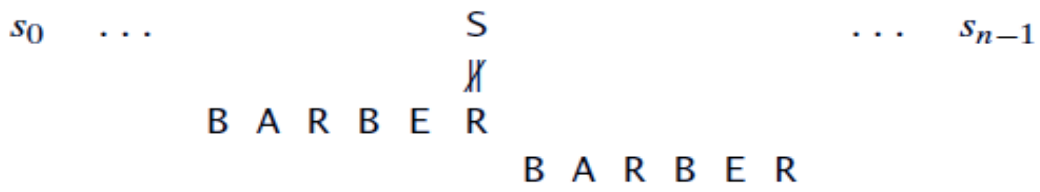
$s_0 \ldots c \ldots s_{n-1}$
B A R B E R

Starting with the last **R** of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text. If all the pattern's characters match successfully, a matching substring is found. Then the search can be either stopped altogether or continued if another occurrence of the same pattern is desired. If a mismatch occurs, we need to shift the pattern to the right.

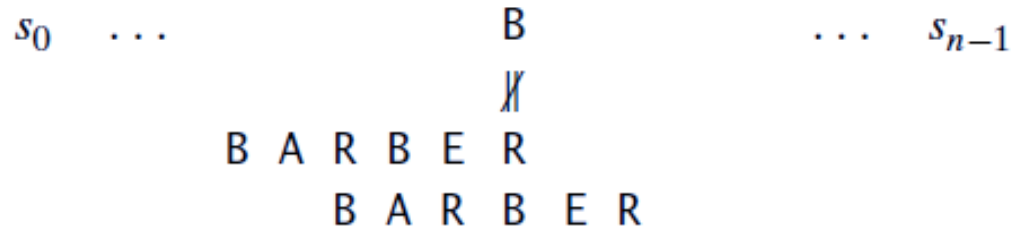
Clearly, we would like to make as large a shift as possible without risking the possibility of missing a matching substring in the text. Horspool's algorithm determines the size of such a shift by looking at the character c of the text that is aligned against the last character of the pattern. This is the case even if character c itself matches its counterpart in the pattern.

In general, the following four possibilities can occur.

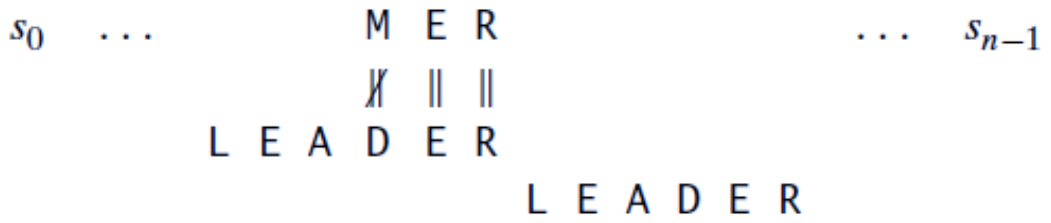
Case 1 If there are no c's in the pattern—e.g., **c** is letter **S** in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character **c** that is known not to be in the pattern):



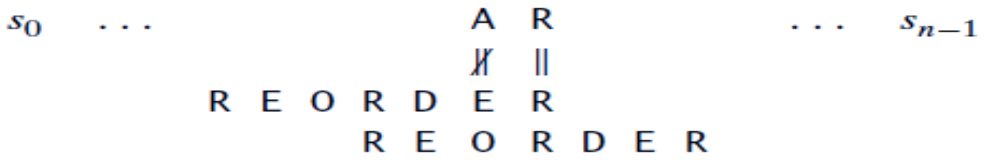
Case 2 If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter **B** in our example—the shift should align the rightmost occurrence of **c** in the pattern with the **c** in the text:



Case 3: If **c** happens to be the last character in the pattern but there are no c's among its other **m − 1** characters—e.g., **c** is letter **R** in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length m:



Case 4: Finally, if **c** happens to be the last character in the pattern and there are other c's among its first **m − 1** characters—e.g., **c** is letter **R** in our example—the situation is similar to that of Case 2 and the rightmost occurrence of **c** among the first **m − 1** characters in the pattern should be aligned with the text's **c**:



Formula To compute Shift Table

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \quad \textbf{(7.1)}$$

For example, for the pattern **BARBER,** all the table's entries will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively. Here is a simple algorithm for computing the shift table entries. Initialize all the entries to the pattern's length **m** and scan the pattern left to right repeating the following step **m − 1** times: for the **j^{th}** character of the pattern **(0 ≤ j ≤ m − 2),**

overwrite its entry in the table with **m − 1− j**, which is the character's distance to the last character of the pattern. Note that since the algorithm scans the pattern from left to right, the last overwrite will happen for the character's rightmost occurrence exactly as we would like it to be.

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's algorithm

Step 1 For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched (then stop) or a mismatching pair is encountered. In the latter case, retrieve the entry t (c) from the c's column of the shift table where c is the text's character currently aligned against the last character of the pattern, and shift the pattern by t (c) characters to the right along the text.

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
```

EXAMPLE : As an example of a complete application of Horspool's algorithm, consider searching for the **pattern BARBER** in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

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 actual search in a particular text proceeds as follows:

```
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
          B A R B E R                 B A R B E R
```

PES UNIVERSITY

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

Unit-4

4. Input Enhancement in String Matching – Boyer Moore algorithm

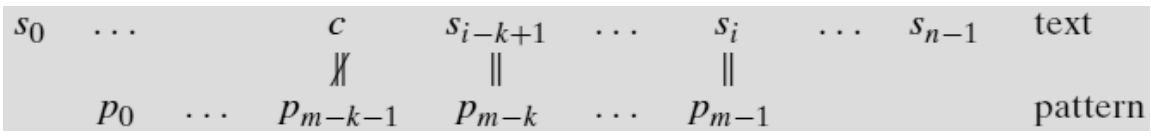
String matching involves finding an occurrence of a given string of *m* characters called the ***pattern*** in a longer string of *n* characters called the ***text***.

Boyer Moore algorithm compares pattern characters to text from right to left precomputing shift sizes in **two** tables

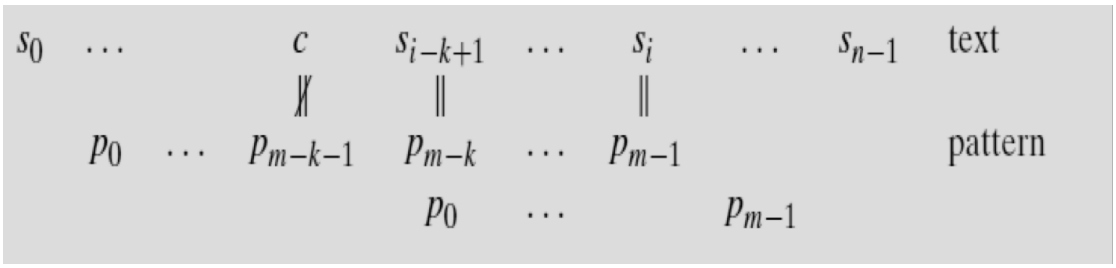
1. ***bad-symbol table*** indicates how much to shift based on text's character causing a mismatch
2. ***good-suffix table*** indicates how much to shift based on matched part (suffix) of the pattern (taking advantage of the periodic structure of the pattern)

Bad-symbol shift Table:

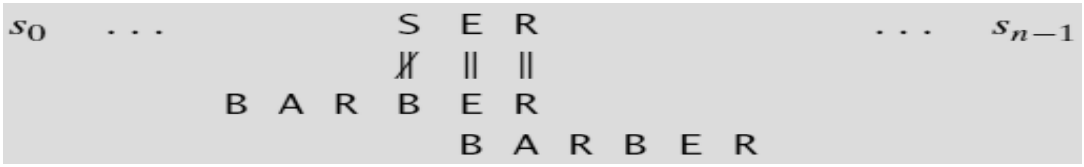
1. If the rightmost character of the pattern doesn't match, BM algorithm acts as Horspool's
2. If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern's characters match or a mismatch on text's character *c* is encountered after $k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



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 $t_1(c) - k$ where $t_1(c)$ is the entry in the precomputed table used by **Horspool's algorithm** and ***k*** is the **number of matched characters**:



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 $t_1(S) - 2 = 6 - 2 = 4$ positions:



If **$t_1(c) - k \leq 0$**, we obviously do not want to shift the pattern by 0 or a negative number of positions. Rather, we can fall back on the brute-force thinking and simply shift the pattern by one position to the right.
To summarize, the bad-symbol shift d1 is computed by the Boyer-Moore algorithm either as **$t_1(c) - k$** if this quantity is positive and as 1 if it is negative or zero. This can be expressed by the following compact formula:

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

Good-suffix shift:

The second type of shift is guided by a successful match of the last k > 0 characters of the pattern.We refer to the ending portion of the pattern as its suffix of size k and denote it suff (k). Accordingly, we call this type of shift the good-suffix shift.

$d_2(k)$ = 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$\overline{\text{B}}$AB	2
2	$\overline{\text{AB}}$CBAB	4
3	$\overline{\text{A}}$BC$\overline{\text{B}}$AB	4
4	$\overline{\text{AB}}$CBAB	4
5	$\overline{\text{AB}}$CBAB	4

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

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

Boyer-Moore algorithm :

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described earlier.

Step 3 Align the pattern against the beginning of the text.

Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by the formula

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

Shifting by the maximum of the two available shifts when $k > 0$ is quite logical. The two shifts are based on the observations—the first one about a text's mismatched character, and the second one about a matched group of the pattern's rightmost characters—that imply that shifting by less than d_1 and d_2 characters, respectively, cannot lead to aligning the pattern with a matching substring in the text. Since we are interested in

shifting the pattern as far as possible without missing a possible matching substring, we take the maximum of these two numbers.

Example

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}}$$\underline{\text{AOBAB}}$	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

When searching for the first occurrence of the pattern, the worst-case efficiency of the Boyer-Moore algorithm is known to be linear.

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

REFERENCE BOOK: "Fundamentals of Computer Algorithms", Horowitz, Sahni, Rajasekaran, Universities Press, 2/e, 2007

Unit-4

5. Greedy Approach

Greedy Approach is a general design technique and it is applicable to optimization problems only. 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—and this is the central point of this technique—the choice made must be:

- **feasible**, i.e., it has to satisfy the problem's constraints
- **locally optimal**, i.e., it has to be the best local choice among all feasible choices available on that step
- **irrevocable**, i.e., once made, it cannot be changed on subsequent steps of the algorithm

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

CONTROL ABSTRACTION

```
Algorithm Greedy (a, n)
// a(1 : n) contains the 'n' inputs
{
solution :={} ;  // initialize the solution to empty
        for i:=1 to n do
```

```
                    {
                    x := select (a);
                    // initialize the solution to empty
                    if feasible (solution, x) then
                    solution := Union (Solution, x);
                    }

    return solution;

    }
```

Examples of Greedy Algorithms:

1. Coin-change problem
2. Minimum Spanning Tree (MST)
    a. Prim's Algorithm
    b. Kruskal's Algorithm
3. Single-source shortest paths
    a. Dijkstra's Algorithm
4. Huffman codes

1. 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.

- 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, 50,100} valued coins.
- **A person buys things worth Rs. 72 and gives a Rs. 100 bill. How does the cashier give change for Rs. 28?**

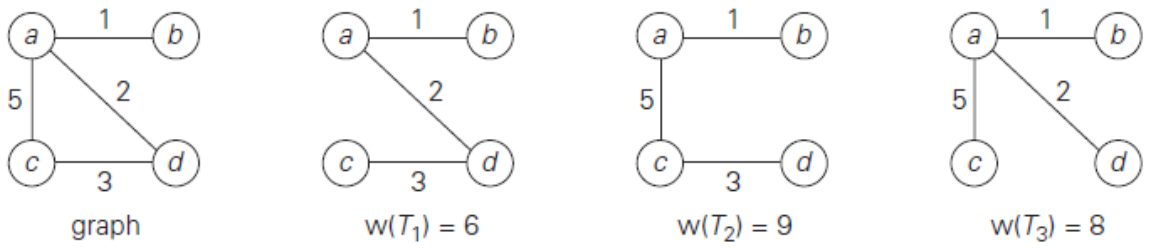
Option	**Choosen Coins**
28-10 = 18	**10**
18-10 =8	**10,10**
8-5 =3	**10,10,5**
3-2 = 1	**10,10,5,2**
1-1 =0	**10,10,5,2,1**

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

Unit-4

5. Prim's Algorithm

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. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.



Graph and its spanning trees, with T_1 being the minimum spanning tree.

Applications:

1. The MST has applications in many practical situations: given *n* points, connect them in the cheapest possible way so that there will be a path between every pair of points.

2. It has direct applications to the design of all kinds of networks—including communication, computer, transportation, and electrical—by providing the cheapest way to achieve connectivity. It identifies clusters of points in data sets.
3. It has been used for classification purposes in archeology, biology, sociology, and other sciences.
4. It is also helpful for constructing approximate solutions to more difficult problems such the traveling salesman problem

Prim's Algorithm

- Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees.
- The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set *V* of the graph's vertices.
- On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. (By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight. Ties can be broken arbitrarily.)
- The algorithm stops after all the graph's vertices have been included in the tree being constructed.
- Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is $n - 1$, where n is the number of vertices in the graph.
- The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.

Pseudocode

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

Time Complexity

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 $V - VT$ whose vertex priorities are the distances to the nearest tree vertices.

In particular, 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 $\vartheta(|V|2)$. Indeed, on each of the $|V| - 1$ iterations, the array implementing the priority queue is traversed to find and delete the minimum and then to update, if necessary, the priorities of the remaining vertices.

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 |).***

This is because the algorithm performs **$|V| - 1$** deletions of the smallest element and makes **$|E|$** verifications and, possibly, changes of an element's priority in a min-heap of size not exceeding **$|V|$.** Each of these operations, as noted earlier, is a ***O(log |V |)*** operation. Hence, the running time of this implementation of Prim's algorithm is in

$$(|V| - 1 + |E|)O(\log |V|) = O(|E| \log |V|)$$

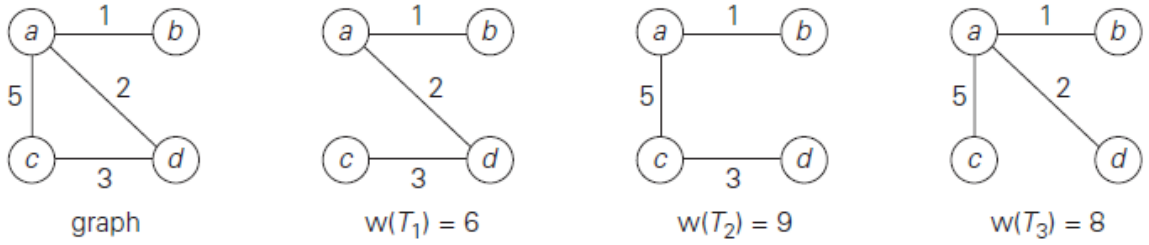
because, in a connected graph, $|V| - 1 \leq |E|$.

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

Unit-4

7. Kruskal's's Algorithm

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. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.



Graph and its spanning trees, with T1 being the minimum spanning tree.

Applications:

1. The MST has applications in many practical situations: given *n* points, connect them in the cheapest possible way so that there will be a path between every pair of points.

2. It has direct applications to the design of all kinds of networks— including communication, computer, transportation, and electrical—by providing the cheapest way to achieve connectivity. It identifies clusters of points in data sets.
3. It has been used for classification purposes in archeology, biology, sociology, and other sciences.
4. It is also helpful for constructing approximate solutions to more difficult problems such the traveling salesman problem

Kruskal's algorithm

Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph $G = \langle V, E \rangle$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest. The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

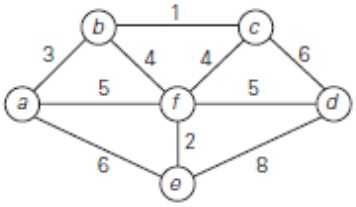
Algorithm

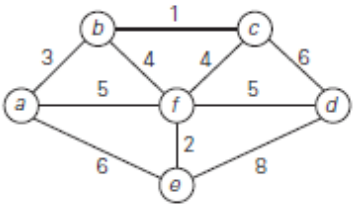
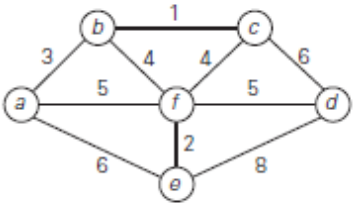
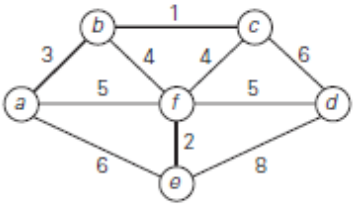
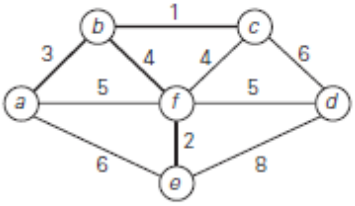
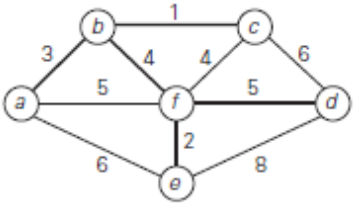
ALGORITHM *Kruskal(G)*

```
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = ⟨V, E⟩
//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}) ≤ ··· ≤ w(e_{i_|E|})
E_T ← ∅;  ecounter ← 0        //initialize the set of tree edges and its size
k ← 0                         //initialize the number of processed edges
while ecounter < |V| − 1 do
    k ← k + 1
    if E_T ∪ {e_{i_k}} is acyclic
        E_T ← E_T ∪ {e_{i_k}};  ecounter ← ecounter + 1
return E_T
```

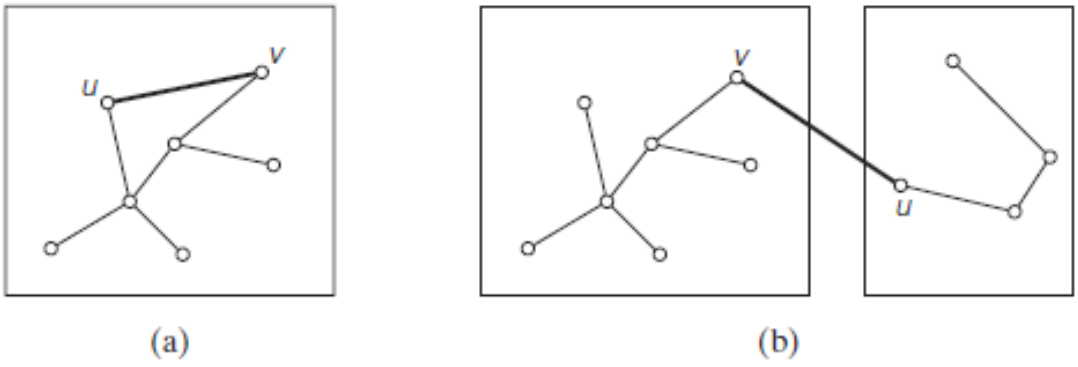
We can consider the algorithm's operations as a progression through a series of forests containing *all* the vertices of a given graph and *some* of its edges. The initial forest consists of $|V|$ trivial trees, each comprising a single vertex of the graph. The final forest consists of a single tree, which is a minimum spanning tree of the graph. On each iteration, the algorithm takes the next edge *(u, v)* from the sorted list of the graph's edges, finds the trees containing the vertices *u* and *v*, and, if these trees are not the same, unites them in a larger tree by adding the edge *(u, v)*.

Example:



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

Fortunately, there are efficient algorithms for doing so, including the crucial check for whether two vertices belong to the same tree. They are called ***unionfind*** algorithms. With an efficient union-find algorithm, the running time of Kruskal's algorithm will be dominated by the time needed for sorting the edge weights of a given graph. **Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in *O(|E| log |E|).***



GURE 9.6 New edge connecting two vertices may (a) or may not (b) create a cycle.

Disjoint Subsets and Union-Find Algorithms

Kruskal's algorithm is one of a number of applications that require a dynamic partition of some *n* element set *S* into a collection of disjoint subsets *S*1, *S*2, . . . , *Sk.*After being initialized as a collection of *n* one-element subsets, each containing a different element of *S*, the collection is subjected to a sequence of intermixed union and find operations.

An abstract data type of a collection of disjoint subsets of a finite set with the following operations:

makeset(x): creates a one-element set {*x*}. It is assumed that this operation can be applied to each of the elements of set *S* only once.

find(x): returns a subset containing *x*.

union(x, y): constructs the union of the disjoint subsets *Sx* and *Sy* containing *x* and *y,* respectively, and adds it to the collection to replace *Sx* and *Sy,* which are deleted from it.

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}.

Text Book: Introduction to the Design and Analysis of Algorithms

Author: Anany Levitin 2 nd Edition

Unit-4

8. Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source.

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

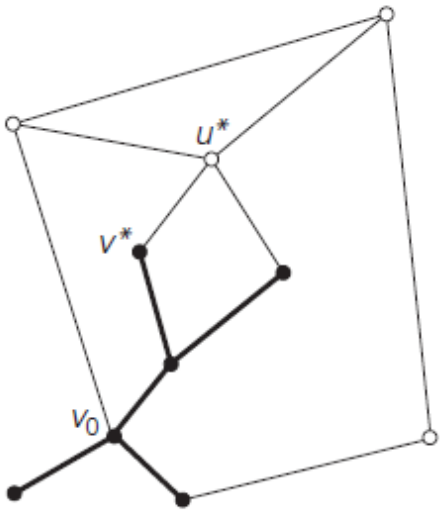
v is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree rooted at *s*)

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*

First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. In general, before its i^{th} iteration commences, the algorithm has already identified the shortest paths to $i - 1$ other vertices nearest

to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree T_i of the given graph.



0 Idea of Dijkstra's algorithm. The subtree of the shortest paths already found is shown in bold. The next nearest to the source v_0 vertex, u^*, is selected by comparing the lengths of the subtree's paths increased by the distances to vertices adjacent to the subtree's vertices.

Since all the edge weights are nonnegative, the next vertex nearest to the source can be found among the vertices adjacent to the vertices of *Ti* . The set of vertices adjacent to the vertices in *Ti* can be referred to as "fringe vertices"; they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source. To identify the *i*th nearest vertex, the algorithm computes, for every fringe vertex *u*, the sum of the distance to the nearest tree vertex *v* (given by the weight of the edge *(v, u)*) and the length *dv* of the shortest path from the source to *v* (previously determined by the algorithm) and then selects the vertex with the smallest such sum. The fact that it suffices to compare the lengths of such special paths is the central insight of Dijkstra's algorithm.

To facilitate the algorithm's operations, we label each vertex with two labels. The numeric label *d* indicates the length of the shortest path from the source to this vertex found by the algorithm so far; when a vertex is added to the tree, *d* indicates the length of the shortest path from the source to that vertex. The other label indicates the name of the next-to-last vertex on such a path, i.e., the parent of the vertex in the tree being constructed. (It can be left unspecified for the source *s* and vertices that are adjacent to none of the current tree vertices.) With such labeling, finding the next nearest vertex $u*$ becomes a simple task of finding a fringe vertex with the smallest *d* value. Ties can be broken arbitrarily. Afte rwe have identified a vertex $u*$ to be added to the tree, we need to perform two operations:

- Move u^* from the fringe to the set of tree vertices.
- For each remaining fringe vertex u that is connected to u^* by an edge of weight $w(u^*, u)$ such that $d_{u^*} + w(u^*, u) < d_u$, update the labels of u by u^* and $d_{u^*} + w(u^*, u)$, respectively.

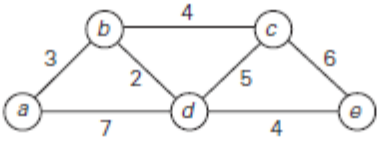
ALGORITHM *Dijkstra*(G, s)

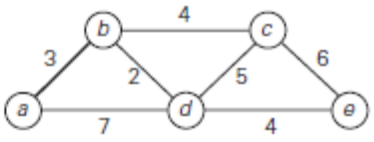
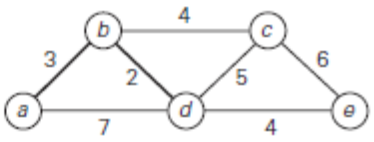
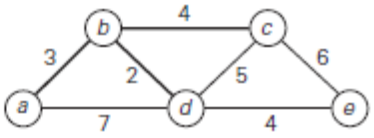
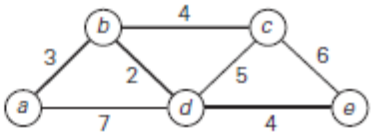
```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
//        and its vertex s
//Output: The length d_v of a shortest path from s to v
//         and its penultimate vertex p_v for every vertex v in V
Initialize(Q)  //initialize 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)
```

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. It is $\theta(|V|^2)$ for graphs represented by their weight matrix and the priority queue implemented as an unordered array. For graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is in $O(|E| \log |V|)$.

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 shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from a to b :	$a - b$	of length 3
from a to d :	$a - b - d$	of length 5
from a to c :	$a - b - c$	of length 7
from a to e :	$a - b - d - e$	of length 9

Text Book: Introduction to the Design and Analysis of Algorithms

Author: Anany Levitin 2 nd Edition

Unit-4

9. Huffman Coding

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.
- 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.

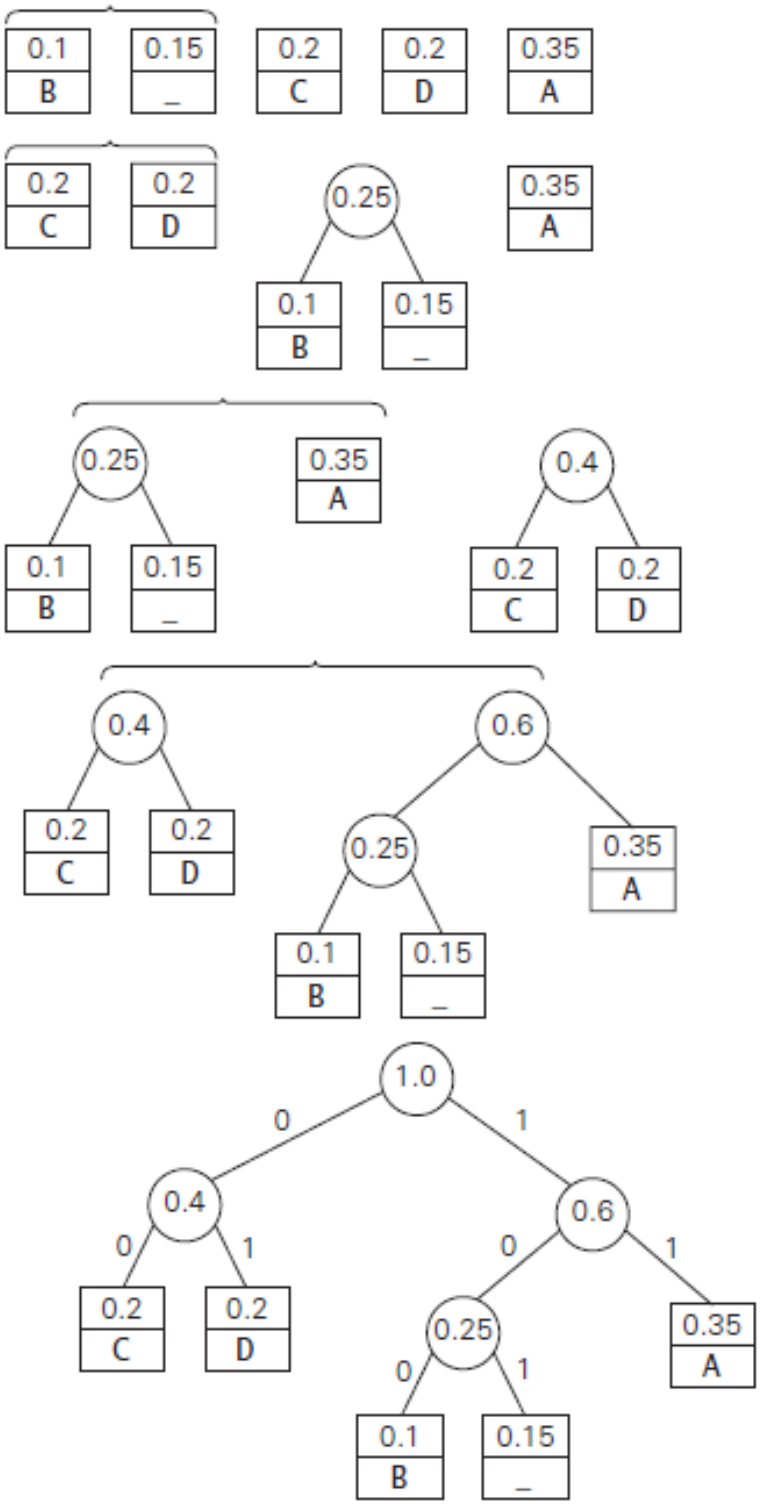
Step 1: Initialize n one-node trees (forest) and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's weight. (Generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

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.

A tree constructed by the above algorithm is called a ***Huffman tree***.
It defines—in the manner described above—a ***Huffman code***.

EXAMPLE Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

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


0.1
B
0.15
_
0.2
C
0.2
D
0.35
A
0.25
0.4
0.6
1.0
0
1

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

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

With the occurrence frequencies given and the code word lengths obtained, the average number of bits per symbol in this code is

2 . 0.35 + 3 . 0.1+ 2 . 0.2 + 2 . 0.2 + 3 . 0.15 = 2.25.

Had we used a fixed-length encoding for the same alphabet, we would have to use at least 3 bits per each symbol. Thus, for this toy example, Huffman's code achieves the ***compression ratio***—a standard measure of a compression algorithm's effectiveness—of *(*3− 2.25*)/*3 . 100%= 25%. In other words, Huffman's encoding of the text will use 25% less memory than its fixed-length 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.