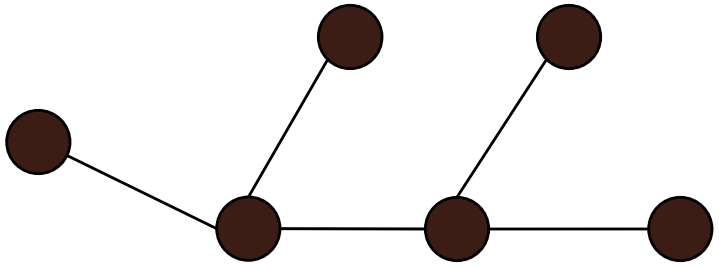
TREE ALGORITHMS

By: Tamal Chakraborty

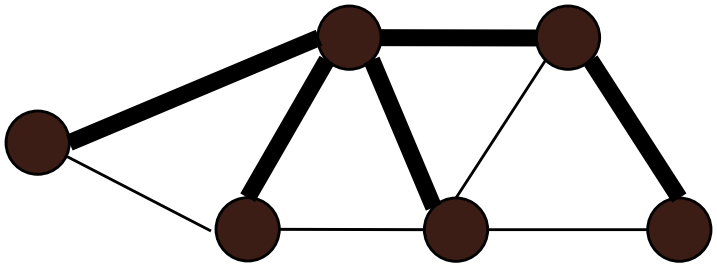
Trees

- A tree is a connected graph without any circuits (i.e. a minimally connected graph).



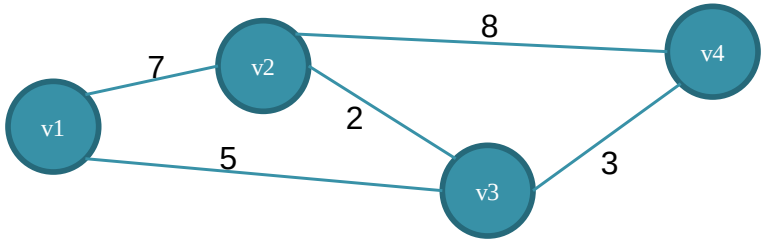
Spanning Tree

- A tree T is said to be a spanning tree of a connected graph G if T is a sub-graph of G and T contains all vertices of G.



Minimal Spanning Tree

- The weight of a spanning tree T of G is defined as the sum of the weights of all the branches of T.
- A spanning tree with the smallest weight in a weighted graph is called a minimal spanning tree.
- Given a weighted graph G, we have to find a minimal spanning tree of the graph.



Minimal Spanning tree

Greedy Strategy

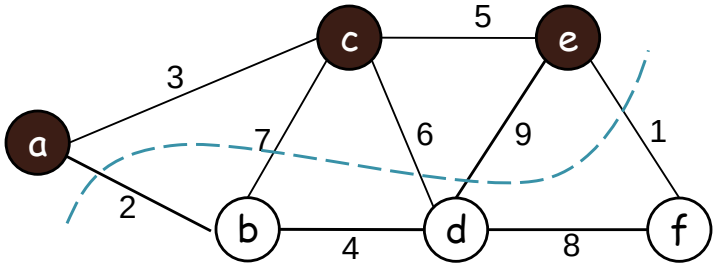
Let A be a sub-set of the minimal spanning tree

At each step determine an edge (u, v) such that, if we add (u, v) to A, A remains a subset of the minimal spanning tree.

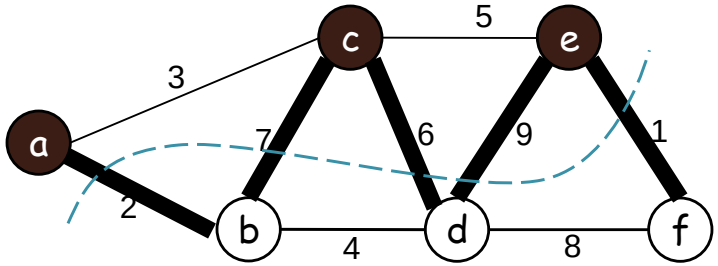
The edge (u, v) is called a safe edge for A.

Minimal Spanning Tree

- A **Cut** (S, V – S) of a graph G(V, E) is a partition of V.

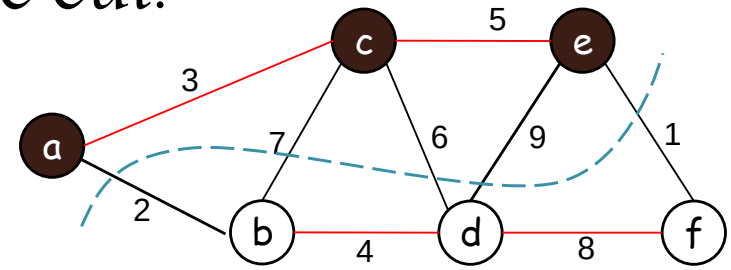


- An edge (u, v) **crosses** the cut if one of its end-points is in S and the other end point is in V – S.

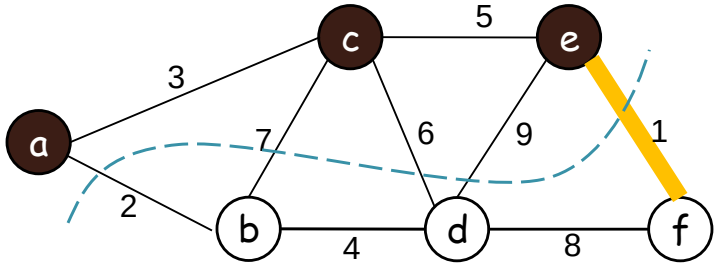


Minimal Spanning Tree

- A Cut respects a set A of edges, if no edge in A crosses the cut.



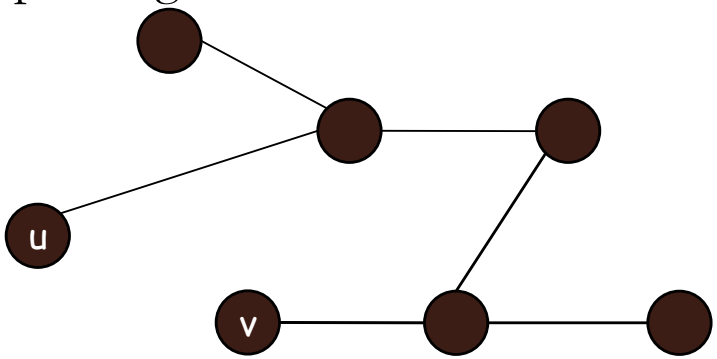
- An edge (u, v) is a light edge crossing a cut if its weight is minimum of any edge crossing the cut.



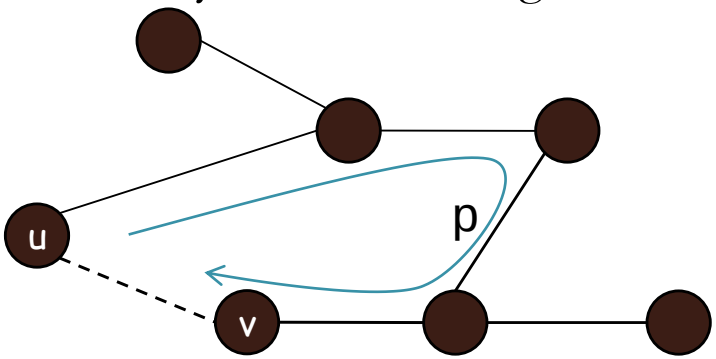
Minimal Spanning Tree

Let (u, v) be a minimum weight edge in a graph G, prove that (u, v) belongs to a minimal spanning tree of G.

- Let T be a minimal spanning tree of G that does not contain (u, v)



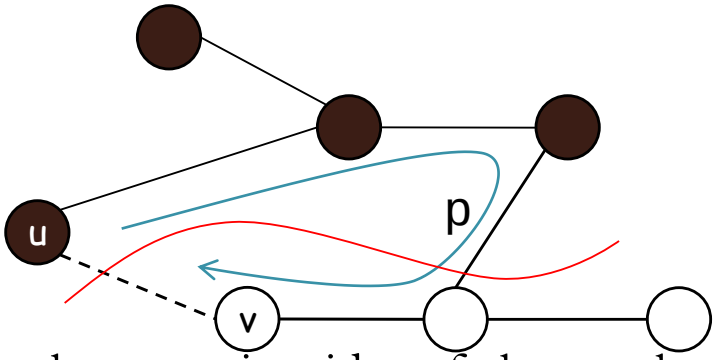
- Then the edge (u, v) forms a cycle with the edges on the path p from u to v.



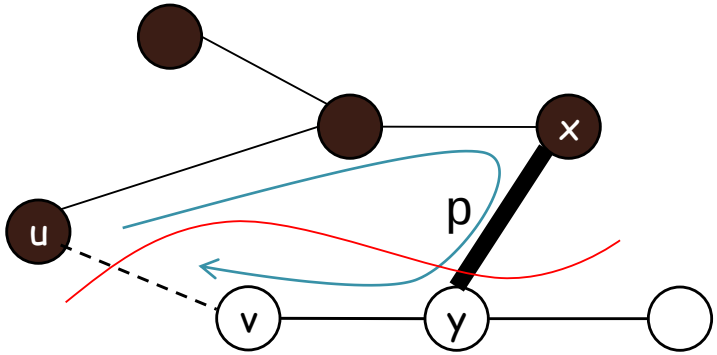
Minimal Spanning Tree

Let (u, v) be a minimum weight edge in a graph G, prove that (u, v) belongs to a minimal spanning tree of G.

- Let there be a cut of the graph, such that (u, v) crosses the cut.



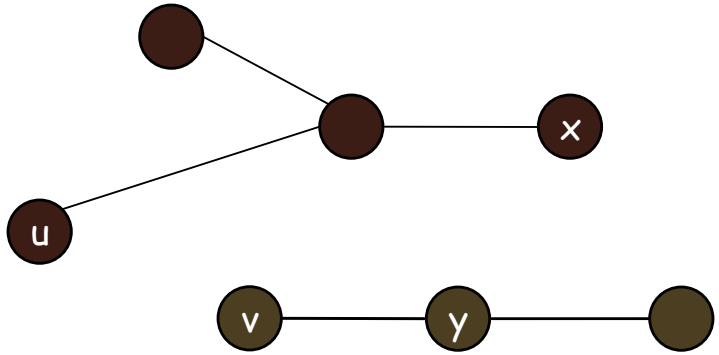
- Since (u, v) are on the opposite sides of the cut there must be at least another edge (x, y) in the graph that also crosses the cut.



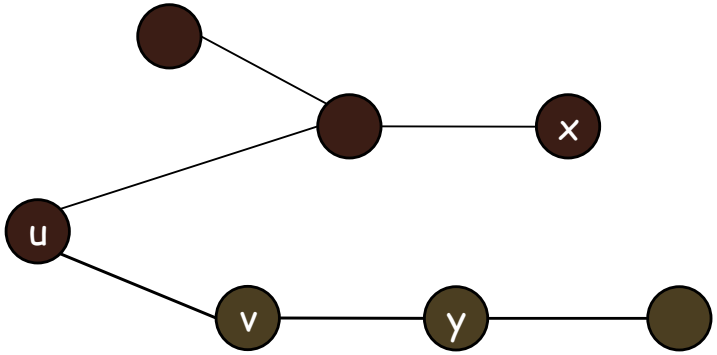
Minimal Spanning Tree

Let (u, v) be a minimum weight edge in a graph G, prove that (u, v) belongs to a minimal spanning tree of G.

- Since (x, y) is in the unique path from u to v in T, removing (x, y) breaks T into two components.



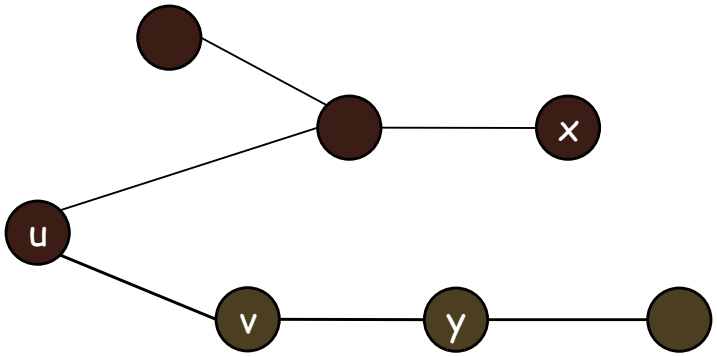
- Adding (u, v) reconnects them to form a new spanning tree T1 = T – {(x, y)} + {(u, v)}



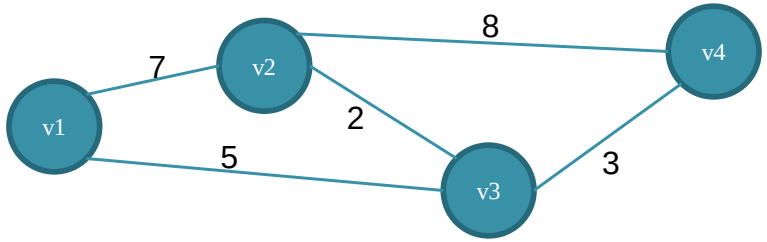
Minimal Spanning Tree

Let (u, v) be a minimum weight edge in a graph G, prove that (u, v) belongs to a minimal spanning tree of G.

- Since (u, v) is a minimum weight edge in G, weight of (u, v) must be less than or equal to weight of (x, y)
- Hence spanning tree T1 = T – {(x, y)} + {(u, v)} must have equal or less weight than T.
- Thus T1 is a minimal spanning tree of G.



Kruskal's Algorithm



1. List all the edges of the graph G in order of non-decreasing weight.

   (v2, v3), (v3, v4), (v1, v3), (v1, v2), (v2, v4)

2. Select a smallest edge from G

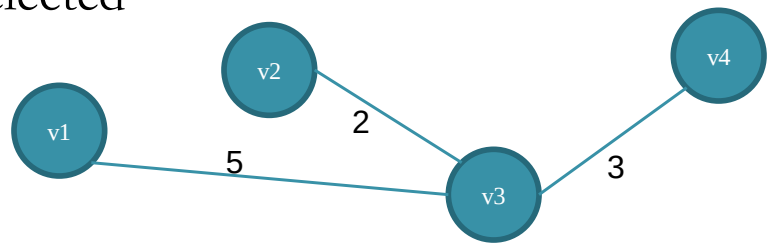
   (v2, v3)

3. From the remaining edges select the smallest edge which doesn't make a circuit with the previously selected edges.

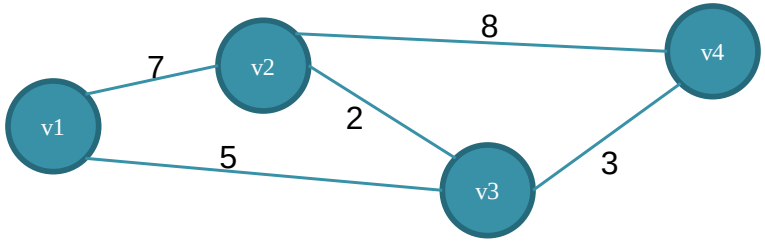
   (v2, v3), (v3, v4)

4. Continue until (n – 1) edges have been selected

   (v2, v3), (v3, v4), (v1, v3)



Prim's Algorithm



$$\begin{pmatrix} 0 & 7 & 5 & \infty \\ 7 & 0 & 2 & 8 \\ 5 & 2 & 0 & 3 \\ \infty & 8 & 3 & 0 \end{pmatrix}$$

1. Start with vertex v1 and connect to the vertex which has the smallest entry in row 1, say v_k.

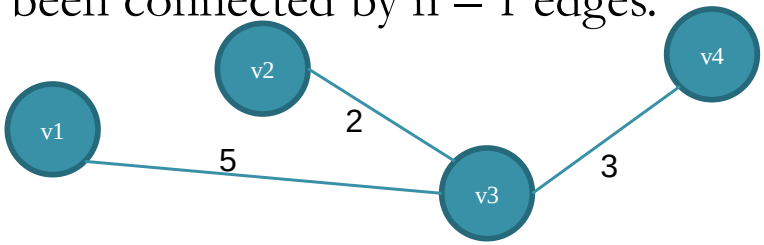
   (v1, v3)

2. Now consider (v1, v_k) as one sub-graph and connect this sub-graph to a vertex other than v1 and v_k that has the smallest entry among all entries in rows 1 and k, say v_j.

   (v1, v3), (v3, v2)

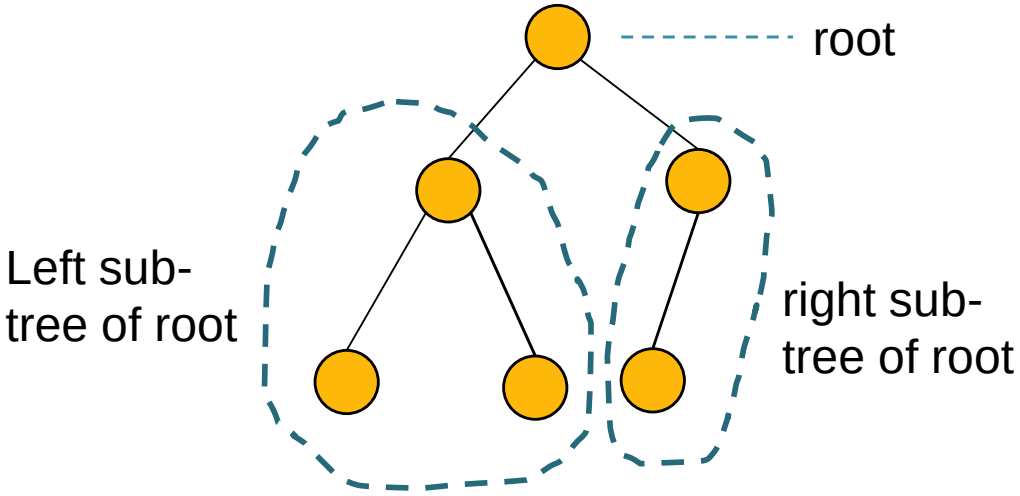
3. Now regard the tree with vertices v1, v_k and v_j as one sub-graph and continue this process until n vertices have been connected by n – 1 edges.

   (v1, v3), (v3, v2), (v3, v4)

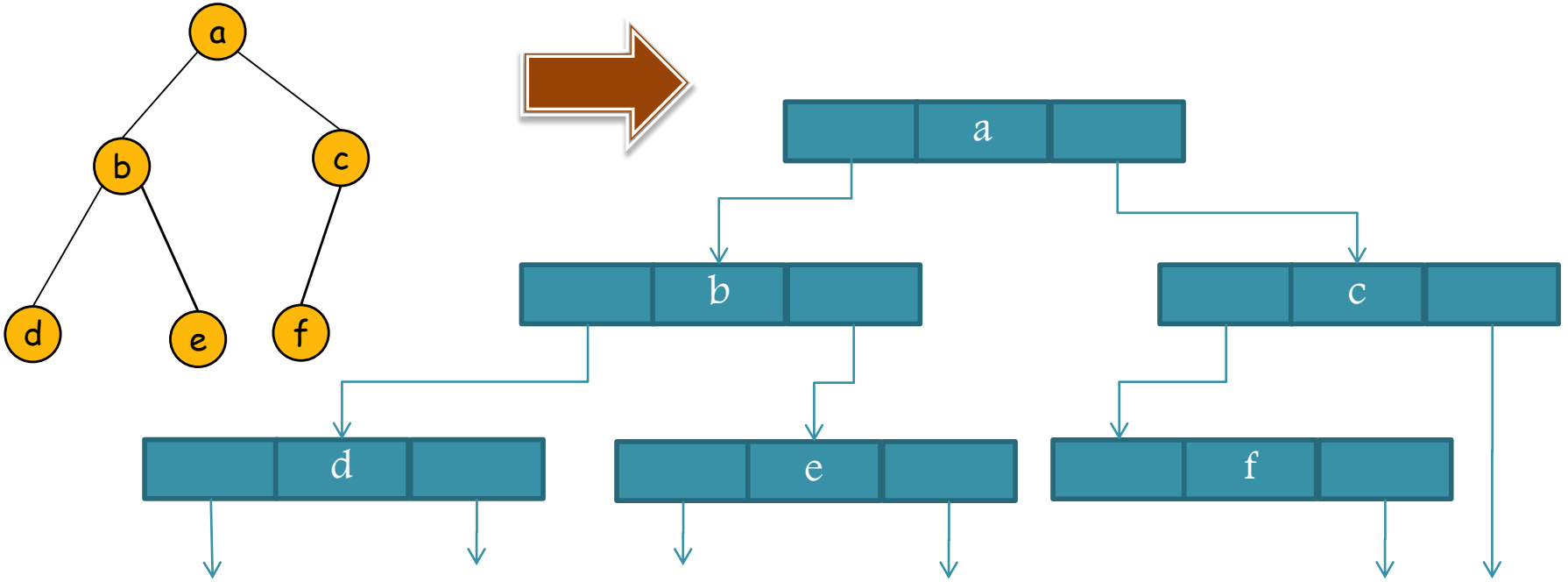


Binary Tree

- A binary tree t is a finite (may be empty) set of elements.
- A non-empty binary tree has a root element.
- The remaining elements (if any) can be partitioned into two binary trees, which are called the left and right sub-trees of t.

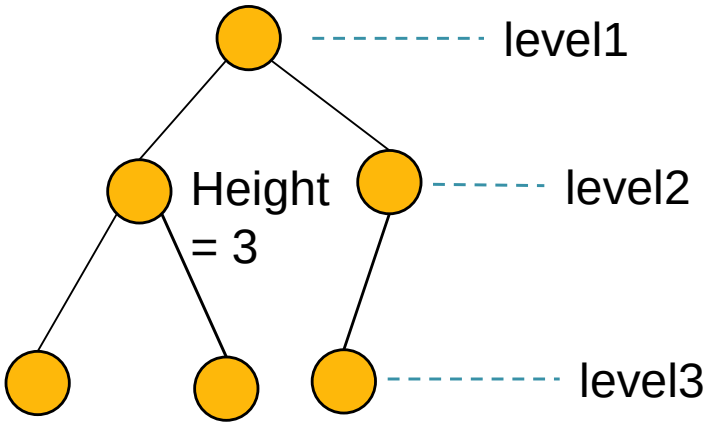


Implementation of Binary Tree by Linked List



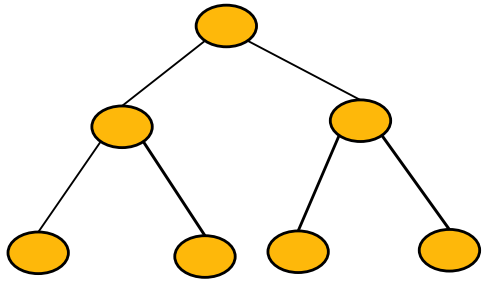
Terminology

- Level
  - By definition root is at level 1
  - Children of root are at level 2
  - Their children are at level 3 and so on…
- Height
  - The height or depth of a binary tree is the number of levels in it

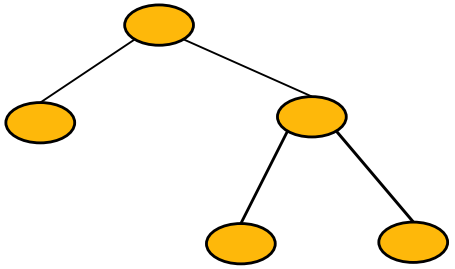


Properties of Binary Tree

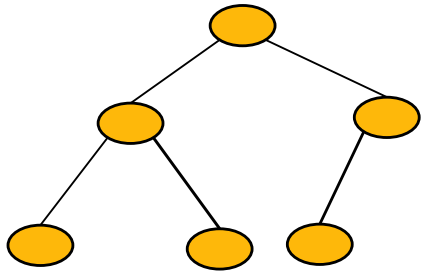
- A binary tree with n elements has exactly $n - 1$ edges.
- A binary tree of height h ($h \geq 0$) has at least h and at most $2^h - 1$ elements in it.
- The height of a binary tree that contains n ($n \geq 0$) elements is at most n and at least $\lg(n + 1)$
- A binary tree of height h that has exactly $2^h - 1$ elements is called a **full binary tree**.
- If every non-leaf node in a binary tree has non-empty left and right sub-trees, the tree is called a **strictly binary tree**.
- A binary tree is called a **complete binary tree**, if all its levels, except possibly the last have the maximum number of possible elements, and if all elements at the last level appear as far left as possible.



Full binary tree



Strictly binary tree



Complete binary tree

Binary Tree Traversal

- Pre-order
  - Visit(root)
  - Pre-order(root→left-child)
  - Pre-order(root→right-child)

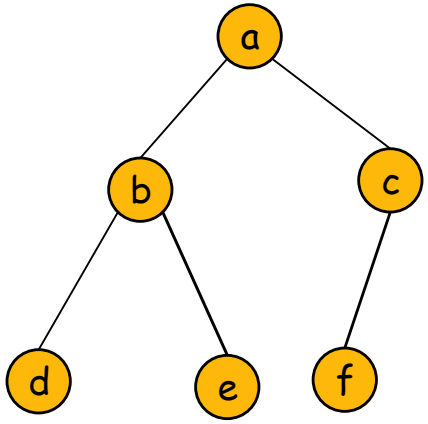
a b d e c f

- In-order
  - In-order(root→left-child)
  - Visit(root)
  - In-order(root→right-child)

d b e a f c

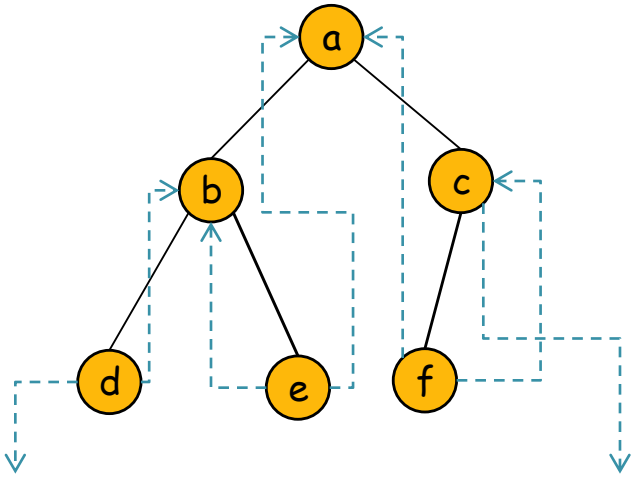
- Post-order
  - Post-order(root→left-child)
  - Post-order(root→right-child)
  - Visit(root)

d e b f c a



Threaded Binary Tree

- In a threaded binary tree if left-child of node n is empty, then left-child of n is set to point the in-order predecessor of n.
- Similarly if right-child of n is empty, then right-child of n is set to point the in-order successor of n.



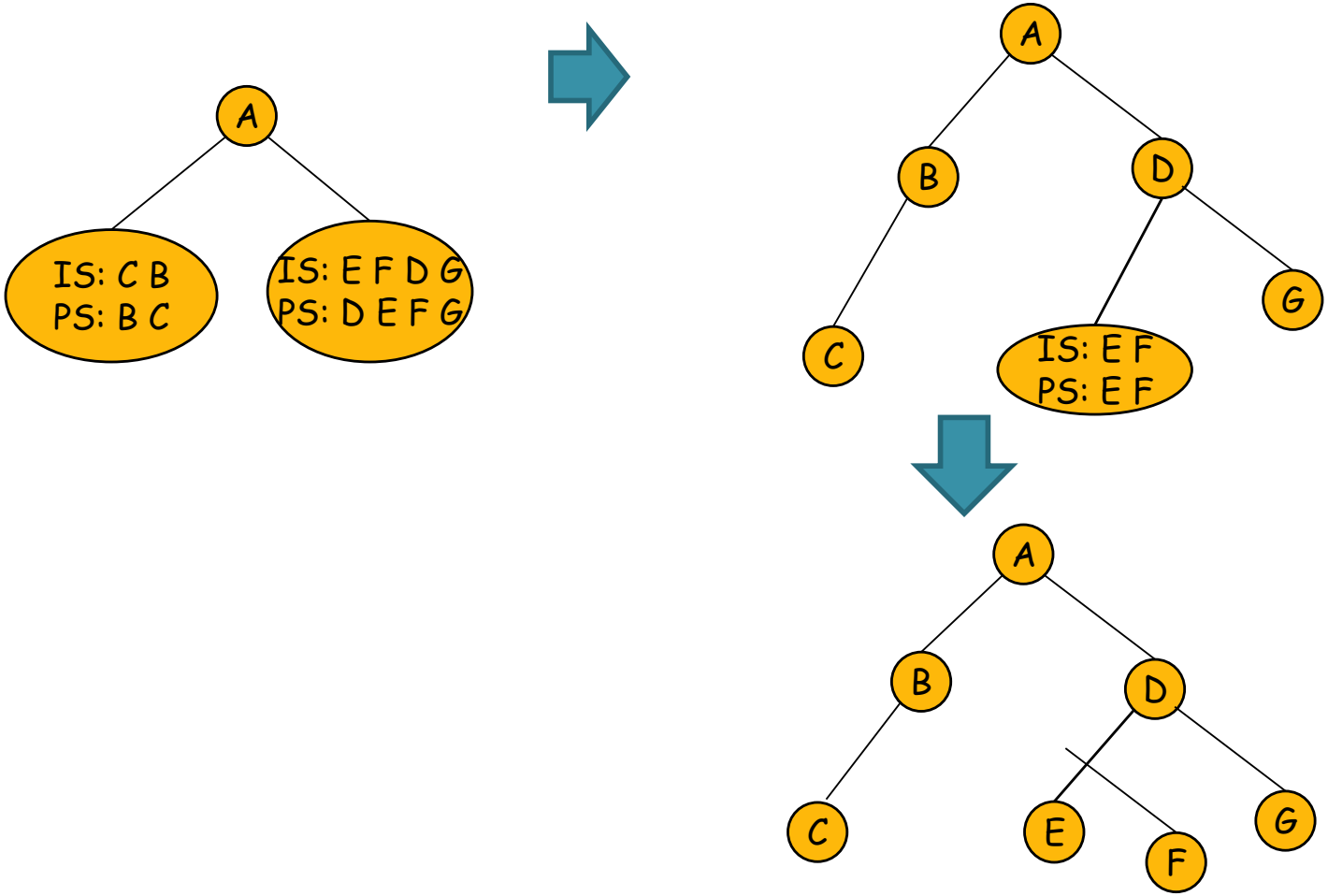
In-order sequence: d b e a f c

Reconstruction of binary tree from in-order & pre-order

- In-order: C B A E F D G
- Pre-order: A B C D E F G

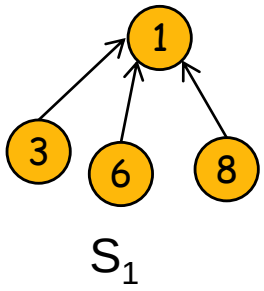
- The first node visited in pre-order traversal of a binary tree is the root (A).
- Nodes to the left of A in the given in-order sequence belong to the left sub-tree and nodes to the right of A belong to the right sub-tree.

Reconstruction of binary tree from in-order & pre-order

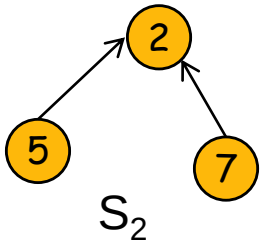


Disjoint Sets

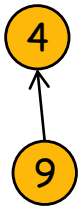
- Say, we have disjoint sets of numbers 1, 2, 3, …, n
- i.e. if S_i & S_j ($i \neq j$) are two sets, then there is no element that is in both S_i & S_j
- For example, say there are three sets $S_1 = \{1, 3, 6, 8\}$, $S_2 = \{2, 5, 7\}$, $S_3 = \{4, 9\}$
- We can represent the sets as trees, with the exception that instead of parent nodes pointing to children nodes, the children nodes point to the parent node.



S_1



S_2



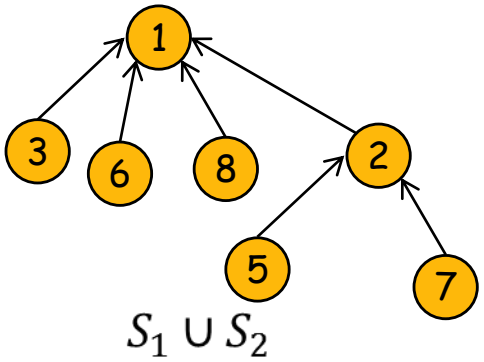
S_3

Disjoint Set Operations

- Union
  - If S_i and S_j are two disjoint sets then their union $S_i \cup S_j$ are all elements x such that x is in S_i or S_j. So $S_1 \cup S_2$ in our example = {1, 3, 6, 8, 2, 5, 7}
- Find
  - Given an element i, we need to find the set that contains i. Thus 3 is in S_1 and 9 is in S_3

Union & Find

- To obtain union of S_i and S_j we simply make one of the trees a sub-tree of the other.
- Thus to obtain union of two disjoint sets we have to set the parent field of one of the roots to the other root.



- For simplicity let us ignore the set names and represent the sets by the roots of the trees representing them.
- The operation Find(i) now becomes determining the root of tree containing element i.

Array representation of disjoint sets

- Since the set elements are numbered 1 … n, we can represent the tree nodes using an array p[1 … n].
- The i^{th} element of this array represents the tree nodes that contain element i.
- This array element gives the parent of the corresponding node.
- For root nodes the value of parent is set to –1.
- Thus, we can represent the sets $S_1 = \{1, 3, 6, 8\}$, $S_2 = \{2, 5, 7\}$, $S_3 = \{4, 9\}$ by the following array:

i	1	2	3	4	5	6	7	8	9
p	-1	-1	1	-1	2	1	2	1	4

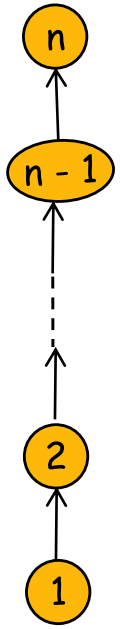
Algorithms for union & find

```
Union (i, j) { /* union of two trees with roots at i & j */
        p[i] = j;
}

Find(i) {
        while (p[i] ≥ 0) { i = p[i]; }
        return i;
}
```

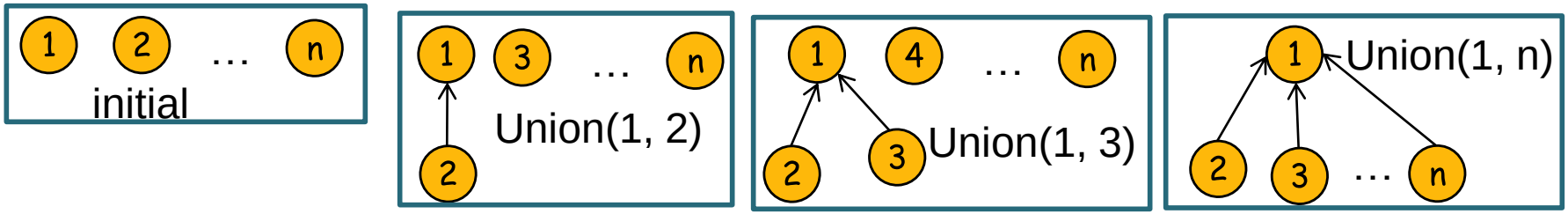
Degenerate Tree

- Say we have n sets, each having one element {1}, {2}, …, {n}.
- Now if we perform the following sequence of union operations
  - Union(1, 2), Union(2, 3), …, Union(n – 1, n)
- We will get a degenerate tree as shown.
- Since algorithm union runs in constant time, n – 1 unions will be processed in O(n) time.
- But now if we perform the following sequence of find operations on this degenerate tree
  - Find(1), Find(2), …, Find(n)
- Time needed to perform a find at level i if the tree would be O(i).
- Hence total time needed for all the finds is:
  - $O(\sum_{i=1}^{n} i) = O(n^2)$



Weighting rule for union

- We can improve the performance of our Union and Find algorithms by avoiding the creation of degenerate trees. To accomplish this we will use the weighting rule for Union(i, j)
- Weighting rule for Union(i, j)
  - If the number of nodes in the tree with root i is less than the number of nodes in the tree with root j, then make j the parent of i, else make i the parent of j.
- When we use the weighting rule of union to perform the set of unions given before, we obtain the following:



Implementation of weighting rule

- To implement the weighting the weighting rule we need to know how many nodes are present in a tree.
- To achieve this we maintain a count field in the root of every tree.
- If r is a root node the count(r) equals number of nodes in that tree.
- Since all nodes other than roots have a positive number in the p field, we maintain the count in the p fields of the root as a negative number.
- Thus, we can represent the sets $S_1 = \{1, 3, 6, 8\}$, $S_2 = \{2, 5, 7\}$, $S_3 = \{4, 9\}$ by the following array:

i	1	2	3	4	5	6	7	8	9
p	-4	-3	1	-2	2	1	2	1	4

Algorithm weightedUnion

```
wightedUnion(i, j) {
        int tmp = p[i] + p[j];
        if (p[i] > p[j]) { p[i] = j; p[j] = tmp; }
        else { p[j] = i; p[i] = tmp; }
}
```

- The weightedUnion algorithm is a constant time operation.

Find operation on weightedUnion Trees

- Say we start with n sets, each having one element. Let T be a tree of m nodes created as a result of sequence of unions each performed using weightedUnion. The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$
- Proof by induction:

This is true for $m = 1$

Say, this is true for all i, $1 \leq i \leq m - 1$

Let's prove it for $i = m$

Let T be a tree with m nodes formed by weightedUnion

Let us consider the last union operation Union(k, j)

Let a be the number of nodes in tree j, and $m - a$ be the nodes in tree k

If $a < m/2$ then height of T is same as that of k, i.e.

$$h \leq \lfloor \log_2 m - a \rfloor + 1 \leq \lfloor \log_2 m \rfloor + 1$$

Else if $a = m/2$, height of T is one more than that of j, i.e.

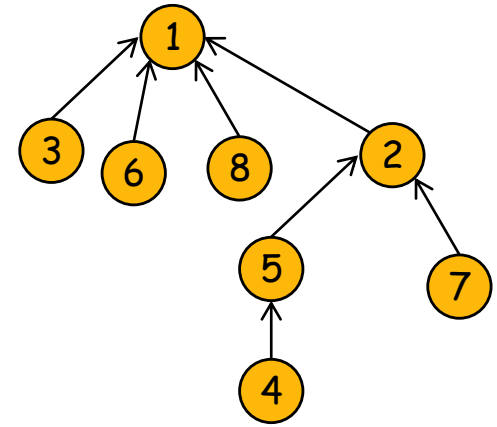
$$h \leq \lfloor \log_2 a \rfloor + 1 + 1 \leq \lfloor \log_2 m/2 \rfloor + 2 \leq \lfloor \log_2 m \rfloor + 1$$

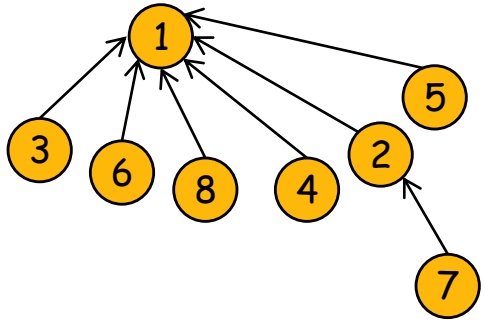
Path Compression

- If we start with singleton sets and perform an intermixed sequence of u unions and f finds, the cost of the sequence and find operations is $O(u + f \log u)$.
- Further improvements are possible by modifying the find procedure, so as to reduce the path from the element e to the root.
- This reduction in path length is obtained using a process called path compression.
- The basic technique is to change the pointers from all nodes on the path from e to the root, so that these nodes point directly to the root.

Path Compression Example

- For example, let us consider the performance of the following 4 finds in the given tree:
  - Find(4), Find(4), Find(4), Find(4)
- Each Find(4) requires going up three parent link fields, resulting in 12 moves for the 4 finds.
- During the first Find(4) nodes 5, 2 are determined to be in the 4 to the root 1.
- Now if we change the parent fields of 4 and 5 to 1 (2 already has 1 as parent), we get the resulting tree:
- Each of the remaining finds now requires going up only 1 parent link field.
- The total cost is now only 7 moves.





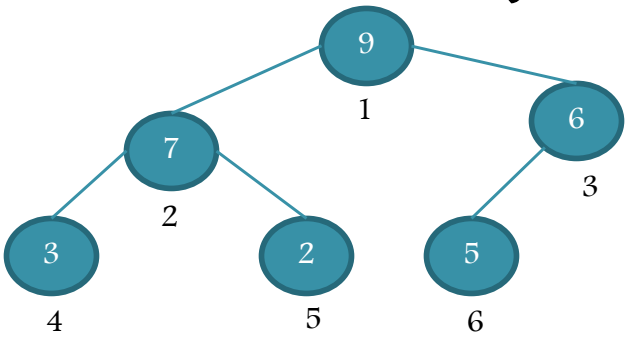
Path Compression Algorithm

```
compressedFind(e) {
        r = e;
        while (p[r] > 0) { r = p[r]; } /* find root */
        while (e ≠ r) {
                s = p[e];
                p[e] = r;
                e = s;
        }
        return r;
}
```

- The compressedFind algorithm roughly doubles the time for an individual find, but reduces the worst case time over a sequence of finds.

Heap Sort

- A max (min) Heap is a complete binary tree, such that each element is greater (less) than its children.
- We can represent a heap by an array, where if i is the index of a node, its left child is in index 2 * i and right child is in index 2 * i + 1
- The following diagram displays a heap as a binary tree and an array.



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

Heap Sort - heapify

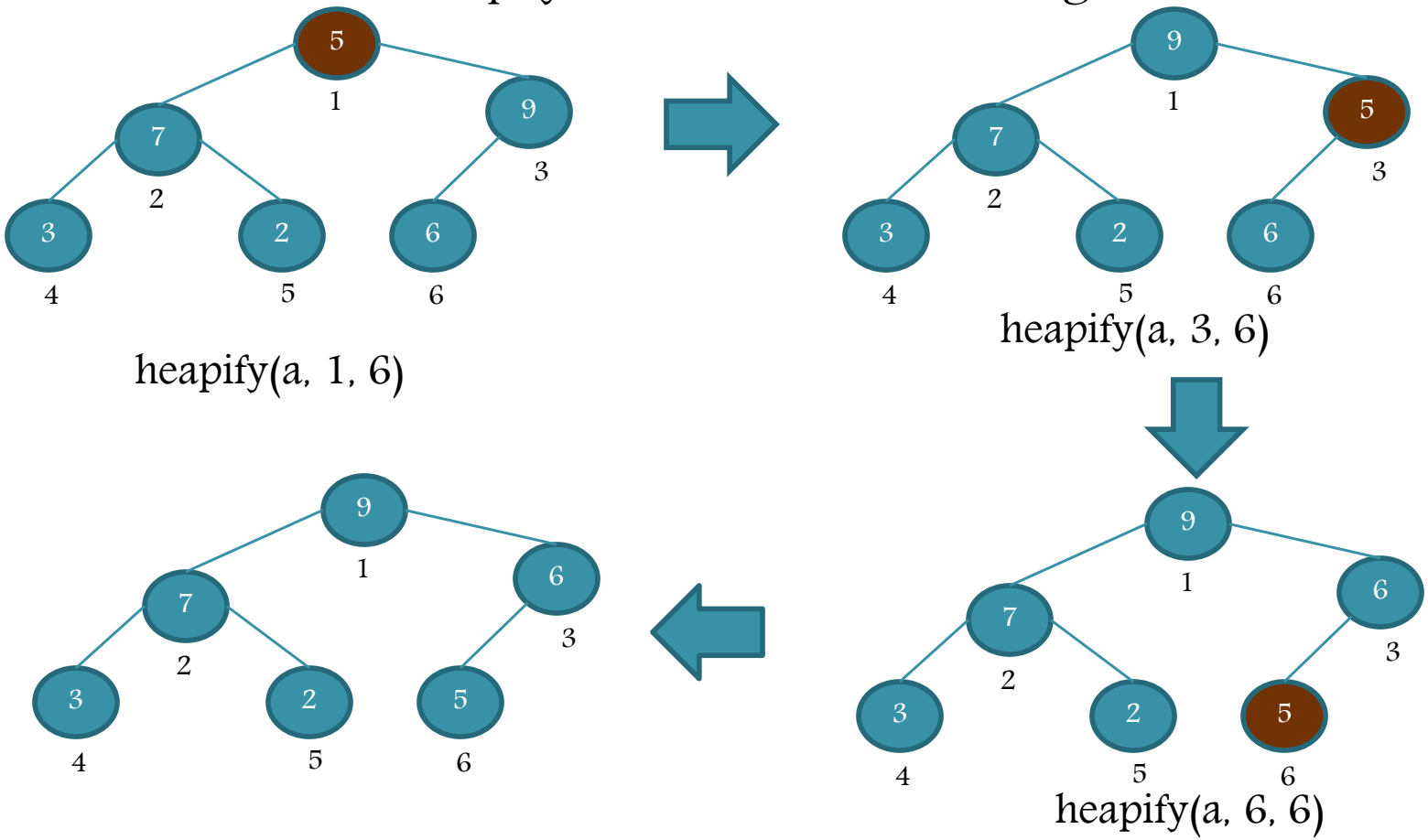
- Let us look at a procedure for maintaining heaps, *heapify*.
- Heapify takes an array a, the number of elements n and an index i of the array as input.
- It assumes that the binary trees rooted at left and right child of i are (max) heaps.
- But a[i] may be smaller than its children, thus violating the heap property.
- The function heapify re-arranges the elements of a, so that the sub-tree rooted at index i, becomes a (max) heap.

Heap Sort - heapify

```
void heapify(int a[], int i, int n)
{
    int l = 2 * i;                                  // left child of a[i]
    int r = 2 * i + 1;                              // right child of a[i]
    int max = i;
    if (l <= n && a[l] > a[i]) max = l;             // find out if a[l] or
    if (r <= n && a[r] > a[max]) max = r;           // a[r] is greater
    if (max != i) {
            swap(a[i], a[max]);
            heapify(a, max, n);                     // heapify the
    }                                               // sub-tree rooted
}                                                   // at a[max]
```

Heap Sort - heapify

- Let us look at how heapify works with the following tree.



Heapify Analysis

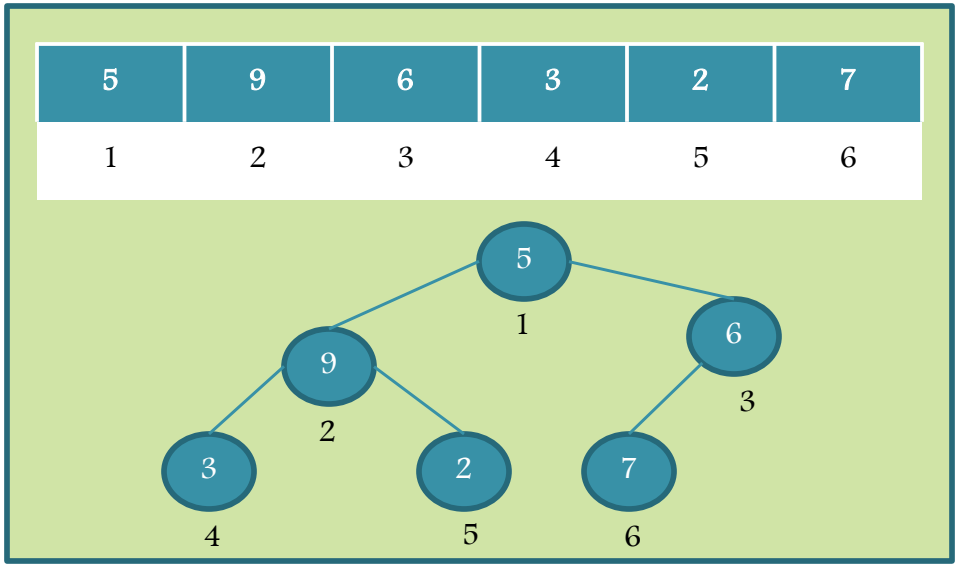
- Let T(n) be the running time for input size n
- T(n) is the time to find max of a[i], a[l] and a[r] (which is $\Theta(1)$) and the time to run heapify on a sub-tree rooted at one of the children of node i.
- The children's sub-trees have size at most 2n/3
- Thus, at the worst case
  - $T(n) <= T(2n/3) + \Theta(1)$
- Solving this recurrence by case 2 of the master method, we get
  - $T(n) = O(\lg(n))$

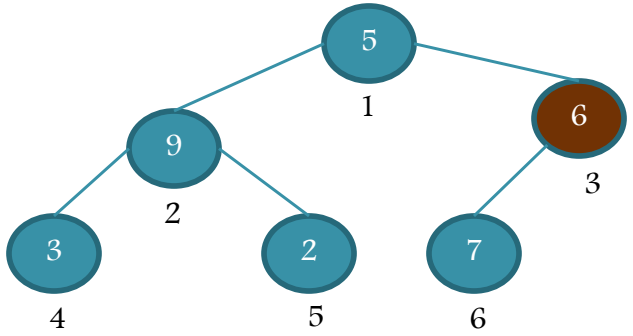
Building a Heap

- We can use the function heapify in a bottom-up manner to convert an array of n elements to a max heap.

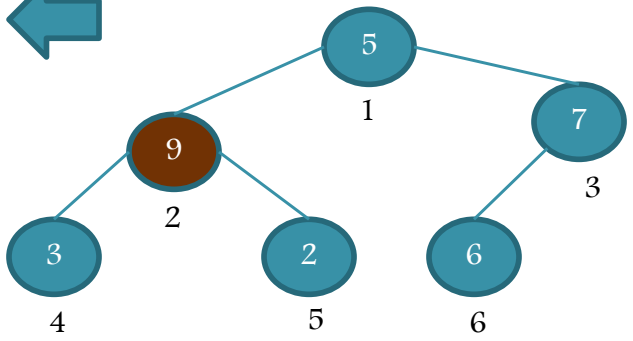
```
void buildHeap(int a[], int n)
{
        for (int i = n / 2; i >= 1; i--)
        {
                heapify(a, i, n);
        }
}
```

Build Heap example

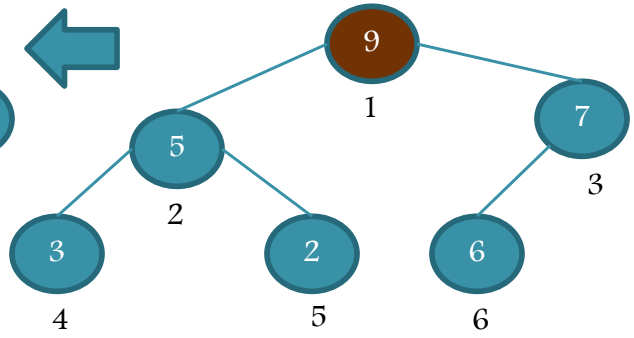




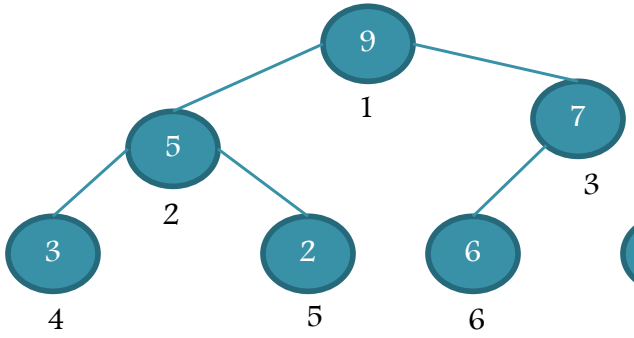
heapify(a, 3, 6)



heapify(a, 2, 6)



heapify(a, 1, 6)



Heap Sort

- We observe that in a max heap the root element is the greatest. This knowledge can be used to sort an array of n elements

1. Given an array a[1 … n] of n elements build a max heap of n elements
2. Since a[1] is the greatest element swap a[1] and a[n]
3. Now we have an array of n – 1 elements a[1 … n – 1], such that sub-trees rooted at index 2 and index 3 are max heaps but element a[1] may be less than its children.
4. We apply heapify on a[1 … n – 1] to build a heap
5. Repeat steps 2 to 4 (n – 1) times to sort the entire array

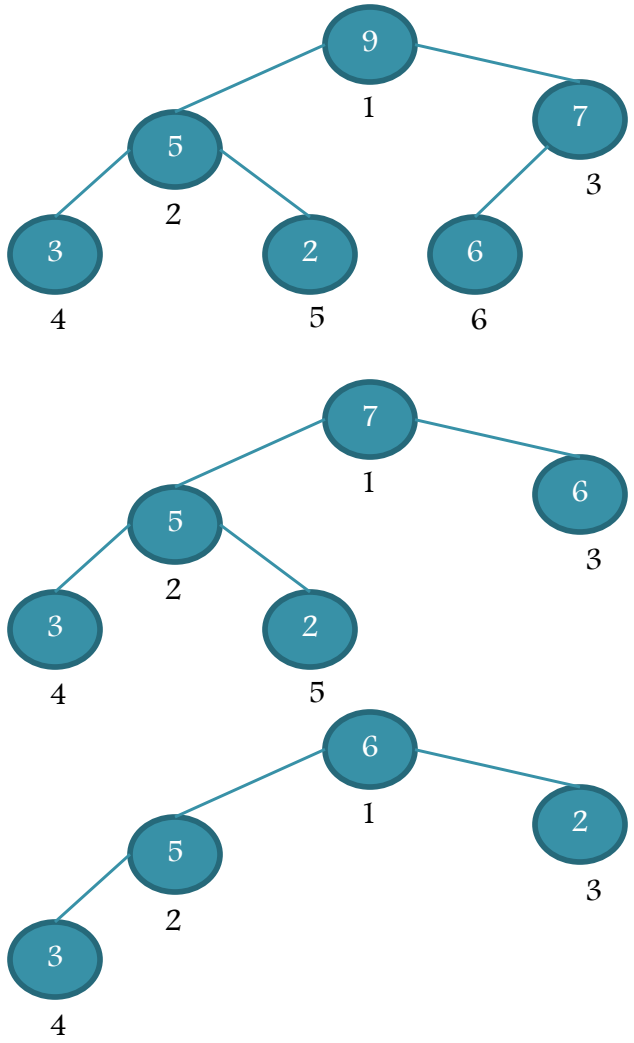
Heap Sort

```
void heapsort(int a[], int n)
{
   buildHeap(a, n);
   for (int i = n; i >= 2; i--)
   {
        swap(a[n], a[1]);
        n = n - 1;
        heapify(a, 1, n);
   }
}
```

- Each call to heapify takes O(lg(n)) time
- For an input size n, there are n such calls
- So heap sort algorithm runs in O(n.lg(n)) time

Heap Sort Example

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



Application of Heap: Priority Queues

- A queue is a data structure which supports two operations, *enqueue* and *dequeue.* An element is inserted in the back of the queue using enqueue operation. An element is deleted from the front of the queue using dequeue operation. Thus a queue is a LIFO data structure.
- A priority queue is a queue such that each element has an associated priority value.
- The dequeue operation in a priority queue removes and returns the element with the highest priority.
- We can use a heap to implement a priority queue.
- The enqueue operation is to insert an element in the heap

Application of Heap: Priority Queues

```
int dequeue(int a[], int n)
{ // Array a[] represents a heap
  // of n elements
        max = a[1];
        a[1] = a[n];
        n = n - 1;
        heapify(a, 1, n);
        return max;
}
```

```
void enqueue(int a[], int n, int priority)
{  // insert an element in the heap
        n = n + 1;
        a[i] = n;
        while(i > 1 &&
        a[parent(i)] < priotity) {
                // parent(i) = i/2
                a[i] = a[parent(i)];
                i = parent(i);
        }
        a[i] = priority;
}
```

Huffman Codes

- Suppose we have a 100,000 character data file that we wish to store compactly.
- We observe that the characters of the file occur with the frequencies as given below:

Characters	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

- We have to store the file in binary mode, wherein each character is represented by a unique binary string.
- If we use a fixed length code, we need 3 bits to represent 6 characters, given by:

Characters	a	b	c	d	e	f
Code (Fixed Length)	000	001	010	011	100	101

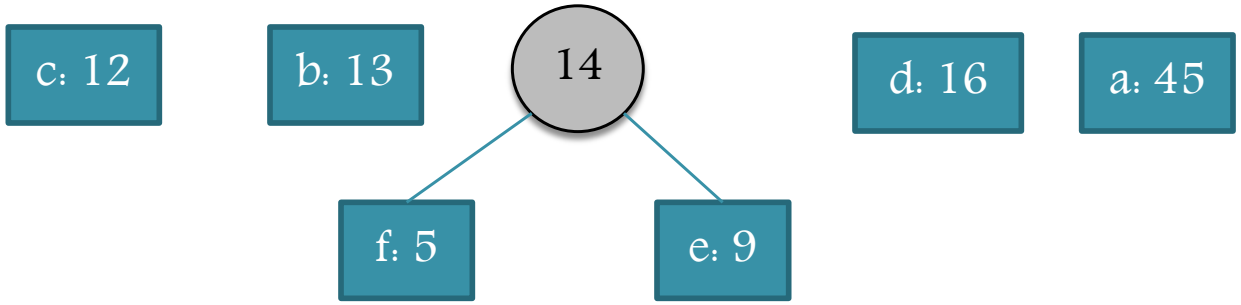
- That requires 300,000 bits to code the entire file.

Huffman Codes

- Huffman invented a greedy algorithm that constructs optimal variable length codes for the characters, and results in savings in the file size.
- The construction of Huffman codes is explained in the following example:
- Step 1: Arrange the characters in ascending order of frequencies.

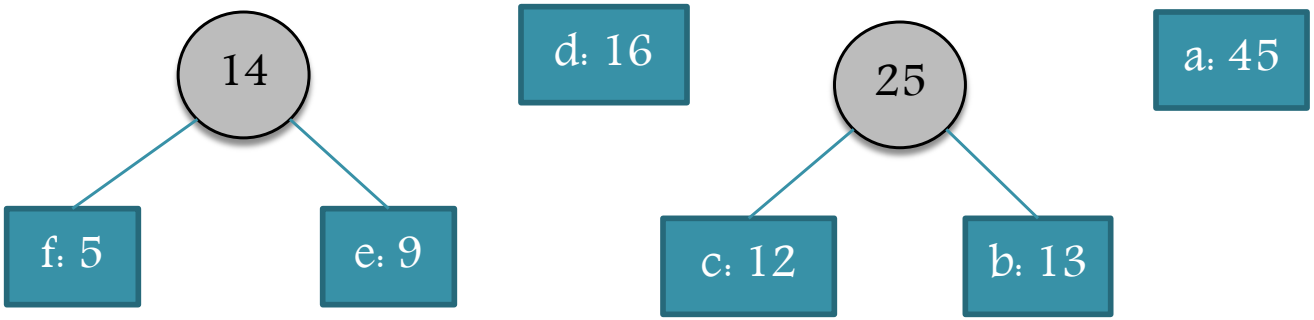
Huffman Codes

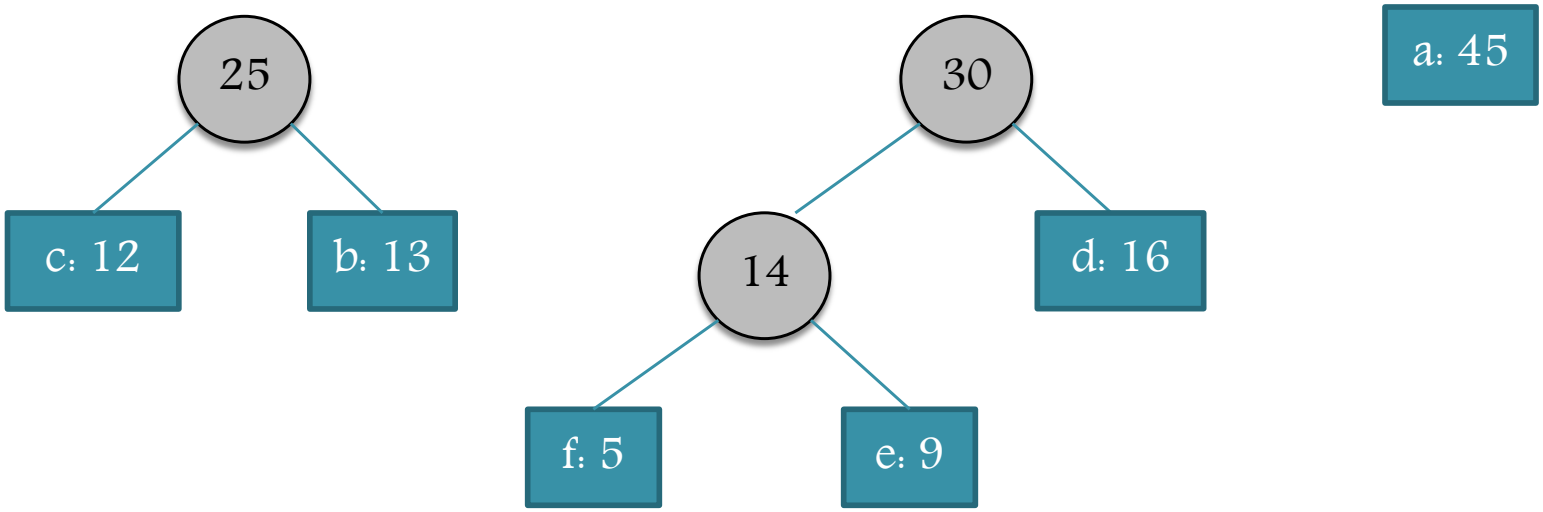
- Step2: Identify two least frequency characters and merge them together. The result of the merge is a new element whose frequency is the sum of frequencies of the two characters that were merged.



Huffman Codes

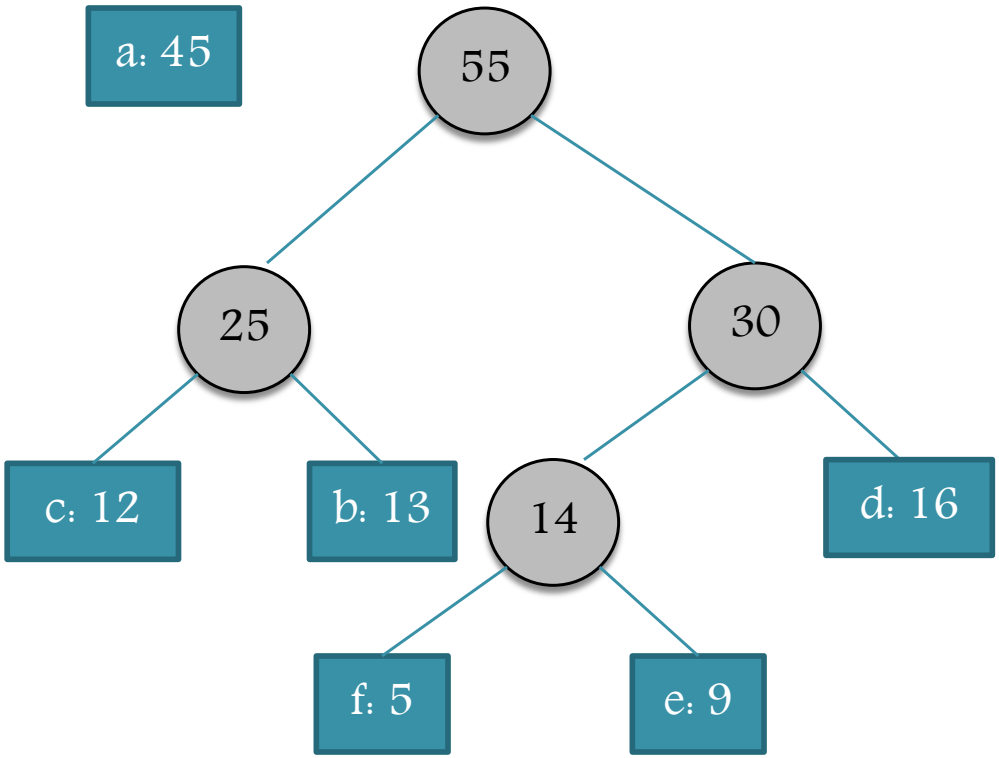
- Step3: Continue Step 2, until all the characters have been covered.



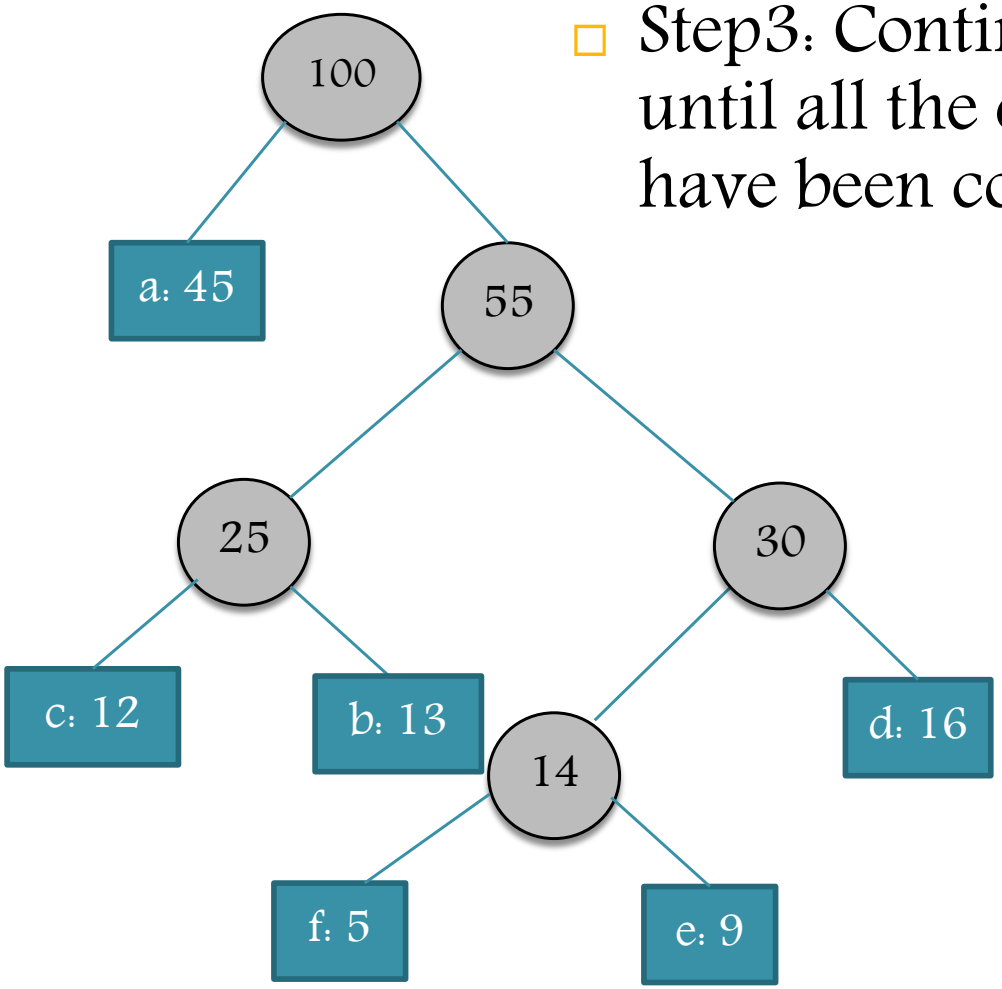


Huffman Codes

- Step3: Continue Step 2, until all the characters have been covered.

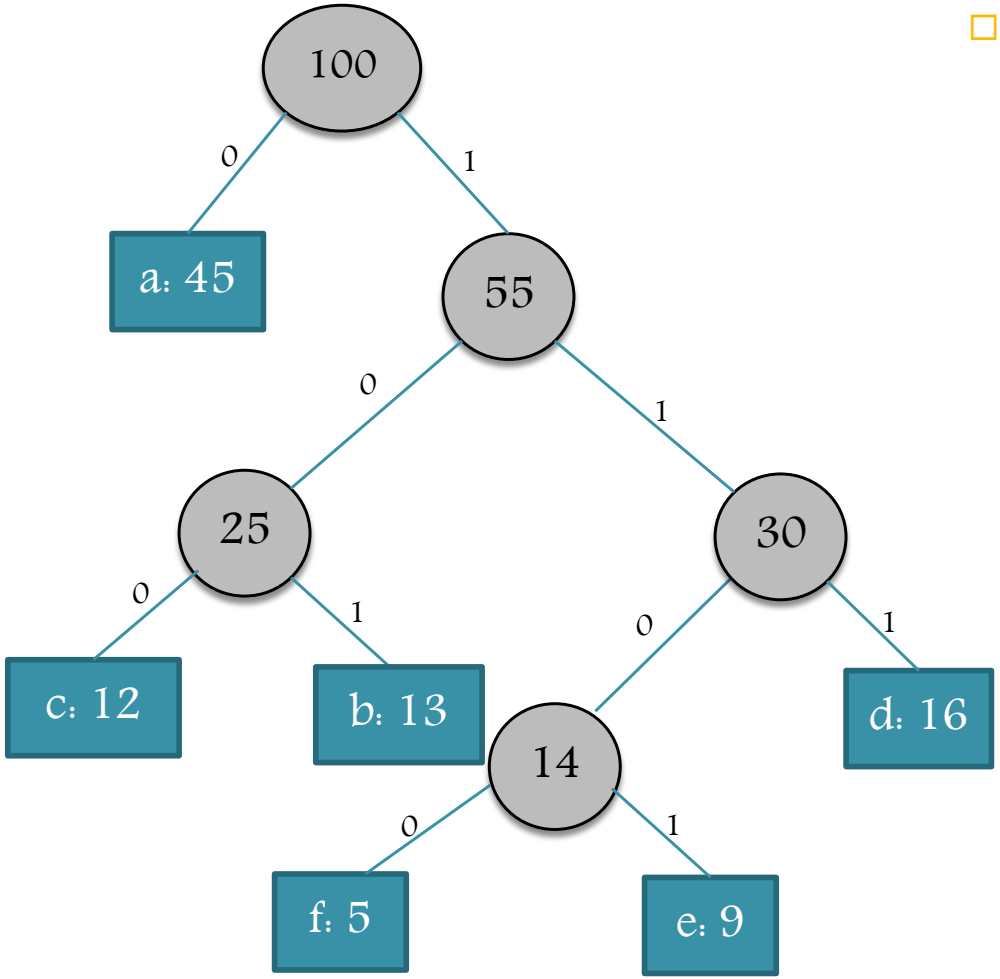


Huffman Codes



- Step3: Continue Step 2, until all the characters have been covered.

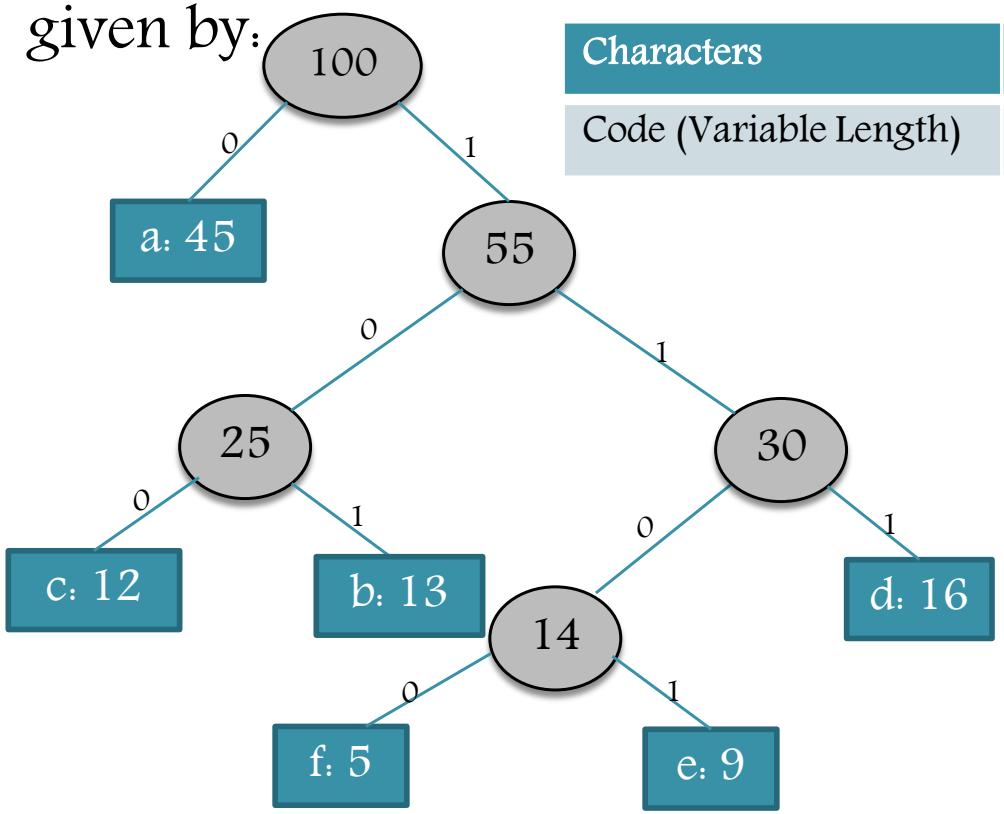
Huffman Codes



- An edge connecting a node with its left child is labeled 0, and with its right child is labeled 1.

Huffman Codes

- Thus the variable length codes for the different characters are given by:



Characters	a	b	c	d	e	f
Code (Variable Length)	0	101	100	111	1101	1100

- This code requires (45.1+13.3+12.3+16.3+9.4+12.4).1000 = 224,000 characters.

THANK YOU!