Lecture 14 - Trees

April 23, 2019

Trees

- Tree – A connected graph that contains no simple circuits.

Trees

- Tree – A connected graph that contains no simple circuits.
- Applications – chemistry (first use of trees as far back as 1857),

Trees

- Tree – A connected graph that contains no simple circuits.
- Applications – chemistry (first use of trees as far back as 1857), computer science – for efficient algorithms for example to locate items in a list,

Trees

- Tree – A connected graph that contains no simple circuits.
- Applications – chemistry (first use of trees as far back as 1857), computer science – for efficient algorithms for example to locate items in a list, coding theory like Huffman coding, gives efficients codes that save costs of data transmission and storage

Trees

- Tree – A connected graph that contains no simple circuits.
- Applications – chemistry (first use of trees as far back as 1857), computer science – for efficient algorithms for example to locate items in a list, coding theory like Huffman coding, gives efficients codes that save costs of data transmission and storage understand outcomes of games.

Trees

- Tree – A connected graph that contains no simple circuits.
- Applications – chemistry (first use of trees as far back as 1857), computer science – for efficient algorithms for example to locate items in a list, coding theory like Huffman coding, gives efficients codes that save costs of data transmission and storage understand outcomes of games.
- Very important in modeling procedures carrying out sequence of decisions.

Trees

- Tree – A connected graph that contains no simple circuits.
- Applications – chemistry (first use of trees as far back as 1857), computer science – for efficient algorithms for example to locate items in a list, coding theory like Huffman coding, gives efficients codes that save costs of data transmission and storage understand outcomes of games.
- Very important in modeling procedures carrying out sequence of decisions.
- Examples and non-examples –

Trees

- Tree – A connected graph that contains no simple circuits.
- Applications – chemistry (first use of trees as far back as 1857), computer science – for efficient algorithms for example to locate items in a list, coding theory like Huffman coding, gives efficients codes that save costs of data transmission and storage understand outcomes of games.
- Very important in modeling procedures carrying out sequence of decisions.
- Examples and non-examples –
- What about graphs containing no simple circuits that are not necessarily connected?

Trees

- Tree – A connected graph that contains no simple circuits.
- Applications – chemistry (first use of trees as far back as 1857), computer science – for efficient algorithms for example to locate items in a list, coding theory like Huffman coding, gives efficients codes that save costs of data transmission and storage understand outcomes of games.
- Very important in modeling procedures carrying out sequence of decisions.
- Examples and non-examples –
- What about graphs containing no simple circuits that are not necessarily connected? forests

Trees

- Tree – A connected graph that contains no simple circuits.
- Applications – chemistry (first use of trees as far back as 1857), computer science – for efficient algorithms for example to locate items in a list, coding theory like Huffman coding, gives efficients codes that save costs of data transmission and storage understand outcomes of games.
- Very important in modeling procedures carrying out sequence of decisions.
- Examples and non-examples –
- What about graphs containing no simple circuits that are not necessarily connected? forests
- Each of forest's connected components is a tree.

Trees

An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

Trees

An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

- Assume that T is a tree. By definition T is a connected graph with no simple circuits.

Trees

An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

- Assume that T is a tree. By definition T is a connected graph with no simple circuits.
- Let x, y be two vertices of T, then there is a simple path between x and y since T is connected.

Trees

An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

- Assume that T is a tree. By definition T is a connected graph with no simple circuits.
- Let x, y be two vertices of T, then there is a simple path between x and y since T is connected.
- If the path is not unique then by combining first path from x to y followed by the path from y to x (reverse of the path).

Trees

An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

- Assume that T is a tree. By definition T is a connected graph with no simple circuits.
- Let x, y be two vertices of T, then there is a simple path between x and y since T is connected.
- If the path is not unique then by combining first path from x to y followed by the path from y to x (reverse of the path).
- Thats a circuit - not possible, so only a unique simple path is possible.

Trees

An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

- Assume that T is a tree. By definition T is a connected graph with no simple circuits.
- Let x, y be two vertices of T, then there is a simple path between x and y since T is connected.
- If the path is not unique then by combining first path from x to y followed by the path from y to x (reverse of the path).
- Thats a circuit - not possible, so only a unique simple path is possible.
- Assume there is a unique simple path between any two vertices of a graph T.

Trees

An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

- Assume that T is a tree. By definition T is a connected graph with no simple circuits.
- Let x, y be two vertices of T, then there is a simple path between x and y since T is connected.
- If the path is not unique then by combining first path from x to y followed by the path from y to x (reverse of the path).
- Thats a circuit - not possible, so only a unique simple path is possible.
- Assume there is a unique simple path between any two vertices of a graph T.
- This implies T is connected.

Trees

- All that remains to show that T can have no simple circuits.

Trees

- All that remains to show that T can have no simple circuits.
- Suppose T had a simple circuit containing x and y.

Trees

- All that remains to show that T can have no simple circuits.
- Suppose T had a simple circuit containing x and y.
- Then there would be two simple paths between x and y – that would violate the unique simple path between any two vertices.

Rooted Trees

- A particular vertex is designated as root.

Rooted Trees

- A particular vertex is designated as root.
- Every edge is given a direction from root.

Rooted Trees

- A particular vertex is designated as root.
- Every edge is given a direction from root.
- There is after all a unique path from the root to each vertex of the graph.

Rooted Trees

- A particular vertex is designated as root.
- Every edge is given a direction from root.
- There is after all a unique path from the root to each vertex of the graph.

A rooted tree is a tree in which one vertex has been designated as the root and every edge is directed away from the root.

Rooted Trees

- A particular vertex is designated as root.
- Every edge is given a direction from root.
- There is after all a unique path from the root to each vertex of the graph.

A rooted tree is a tree in which one vertex has been designated as the root and every edge is directed away from the root.
A different choice of root leads to a different rooted tree.

Terminology of Trees

- Botanical and genealogical origins.

Terminology of Trees

- Botanical and genealogical origins.
- T is a rooted tree. v is a vertex in T other than the root, the parent of v is the unique vertex u s.t. there is a directed edge from u to v.

Terminology of Trees

- Botanical and genealogical origins.
- T is a rooted tree. v is a vertex in T other than the root, the parent of v is the unique vertex u s.t. there is a directed edge from u to v. v is called a child of u.

Terminology of Trees

- Botanical and genealogical origins.
- T is a rooted tree. v is a vertex in T other than the root, the parent of v is the unique vertex u s.t. there is a directed edge from u to v. v is called a child of u.
- Vertices with same parent - siblings.

Terminology of Trees

- Botanical and genealogical origins.
- T is a rooted tree. v is a vertex in T other than the root, the parent of v is the unique vertex u s.t. there is a directed edge from u to v. v is called a child of u.
- Vertices with same parent - siblings.
- The ancestors of a vertex (other than root) are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root.

Terminology of Trees

- Botanical and genealogical origins.
- T is a rooted tree. v is a vertex in T other than the root, the parent of v is the unique vertex u s.t. there is a directed edge from u to v. v is called a child of u.
- Vertices with same parent - siblings.
- The ancestors of a vertex (other than root) are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root.
- The descendants of a vertex v are those vertices that have v as an ancestor.

Terminology of Trees

- Botanical and genealogical origins.
- T is a rooted tree. v is a vertex in T other than the root, the parent of v is the unique vertex u s.t. there is a directed edge from u to v. v is called a child of u.
- Vertices with same parent - siblings.
- The ancestors of a vertex (other than root) are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root.
- The descendants of a vertex v are those vertices that have v as an ancestor.
- A vertex of a rooted tree is called a leaf if it has no children.

Terminology of Trees

- Botanical and genealogical origins.
- T is a rooted tree. v is a vertex in T other than the root, the parent of v is the unique vertex u s.t. there is a directed edge from u to v. v is called a child of u.
- Vertices with same parent - siblings.
- The ancestors of a vertex (other than root) are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root.
- The descendants of a vertex v are those vertices that have v as an ancestor.
- A vertex of a rooted tree is called a leaf if it has no children.
- Vertices that have children are called internal vertices. The root is an internal vertex unless it is the only vertex in the graph, in which case it is a leaf.

m-ary Trees and Ordered Trees

- A rooted tree is called an m-ary tree if every internal vertex has no more than m children.
- The tree is called a full m-ary tree if every internal vertex has exactly m children.
- An m-ary tree with $m = 2$ is called a binary tree.

m-ary Trees and Ordered Trees

- A rooted tree is called an m-ary tree if every internal vertex has no more than m children.
- The tree is called a full m-ary tree if every internal vertex has exactly m children.
- An m-ary tree with $m = 2$ is called a binary tree.
- Ordered Rooted Tree : is a rooted tree where the children of each internal vertex are ordered.

m-ary Trees and Ordered Trees

- A rooted tree is called an m-ary tree if every internal vertex has no more than m children.
- The tree is called a full m-ary tree if every internal vertex has exactly m children.
- An m-ary tree with $m = 2$ is called a binary tree.
- Ordered Rooted Tree : is a rooted tree where the children of each internal vertex are ordered.
- Ordered rooted trees are drawn so that the children of each internal vertex are shown in order from left to right.

m-ary Trees and Ordered Trees

- A rooted tree is called an m-ary tree if every internal vertex has no more than m children.
- The tree is called a full m-ary tree if every internal vertex has exactly m children.
- An m-ary tree with $m = 2$ is called a binary tree.
- Ordered Rooted Tree : is a rooted tree where the children of each internal vertex are ordered.
- Ordered rooted trees are drawn so that the children of each internal vertex are shown in order from left to right.
- Ordered binary tree or just binary tree : if an internal vertex has two children, the first child is called the left child and the second child is called the right child .

m-ary Trees and Ordered Trees

- A rooted tree is called an m-ary tree if every internal vertex has no more than m children.
- The tree is called a full m-ary tree if every internal vertex has exactly m children.
- An m-ary tree with $m = 2$ is called a binary tree.
- Ordered Rooted Tree : is a rooted tree where the children of each internal vertex are ordered.
- Ordered rooted trees are drawn so that the children of each internal vertex are shown in order from left to right.
- Ordered binary tree or just binary tree : if an internal vertex has two children, the first child is called the left child and the second child is called the right child .
- Similarly, left subtree and right subtree.

Properties of Trees

A tree with n vertices has $n - 1$ edges.

- Proof by induction. Base Step : When $n = 1$ a tree with $n = 1$ vertex has no edges.

Properties of Trees

A tree with n vertices has $n - 1$ edges.

- Proof by induction. Base Step : When $n = 1$ a tree with $n = 1$ vertex has no edges.
- Inductive step : The inductive hypothesis states that every tree with k vertices has $k - 1$ edges, where $k \in \mathbb{Z}_{\geq 0}$.

Properties of Trees

A tree with n vertices has $n-1$ edges.

- Proof by induction. Base Step : When $n=1$ a tree with $n=1$ vertex has no edges.
- Inductive step : The inductive hypothesis states that every tree with k vertices has $k-1$ edges, where $k \in \mathbb{Z}_{\geq 0}$.
- Suppose that a tree T has $k+1$ vertices and that v is a leaf of T (exists because the tree is finite).

Properties of Trees

A tree with n vertices has $n-1$ edges.

- Proof by induction. Base Step : When $n=1$ a tree with $n=1$ vertex has no edges.
- Inductive step : The inductive hypothesis states that every tree with k vertices has $k-1$ edges, where $k \in \mathbb{Z}_{\geq 0}$.
- Suppose that a tree T has $k+1$ vertices and that v is a leaf of T (exists because the tree is finite).
- Let w be the parent of v.
- Removing from T the vertex v and $\{w, v\}$ produces a tree $T^{'}$ with k vertices.

Properties of Trees

A tree with n vertices has $n-1$ edges.

- Proof by induction. Base Step : When $n=1$ a tree with $n=1$ vertex has no edges.
- Inductive step : The inductive hypothesis states that every tree with k vertices has $k-1$ edges, where $k \in \mathbb{Z}_{\geq 0}$.
- Suppose that a tree T has $k+1$ vertices and that v is a leaf of T (exists because the tree is finite).
- Let w be the parent of v.
- Removing from T the vertex v and $\{w, v\}$ produces a tree $T^{'}$ with k vertices.
- $T^{'}$ has $k-1$ edges by hypothesis.

Properties of Trees

A tree with n vertices has $n-1$ edges.

- Proof by induction. Base Step : When $n=1$ a tree with $n=1$ vertex has no edges.
- Inductive step : The inductive hypothesis states that every tree with k vertices has $k-1$ edges, where $k \in \mathbb{Z}_{\geq 0}$.
- Suppose that a tree T has $k+1$ vertices and that v is a leaf of T (exists because the tree is finite).
- Let w be the parent of v.
- Removing from T the vertex v and $\{w, v\}$ produces a tree $T^{'}$ with k vertices.
- $T^{'}$ has $k-1$ edges by hypothesis.
- T will have k edges since it includes the edge connecting v and w.

Properties of Trees

Tree is a connected undirected graph with no simple circuits. This means, consider when G is an undirected graph with n vertices,

1. G is connected,
2. G has no simple circuits,
3. G has $n-1$ edges.

From previous theorem we have *i* and *ii* implies *iii*.

Properties of Trees

Tree is a connected undirected graph with no simple circuits. This means, consider when G is an undirected graph with n vertices,

1. G is connected,
2. G has no simple circuits,
3. G has $n - 1$ edges.

From previous theorem we have i and ii implies iii.
Exercise:

1. When i and iii hold, this implies ii holds.
2. When ii and iii hold, i must hold.

Counting vertices in m-ary trees

- A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices.

Counting vertices in m-ary trees

- A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices. Proof: Root is counted in $+1$.

Counting vertices in m-ary trees

- A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices. Proof: Root is counted in $+1$.
- A full m-ary tree with
  1. n vertices has $i = (n-1)/m$ internal vertices and $l = [(m-1)n + 1]/m$ leaves,

Counting vertices in m-ary trees

- A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices. Proof: Root is counted in $+1$.
- A full m-ary tree with
  1. n vertices has $i = (n-1)/m$ internal vertices and $l = [(m-1)n + 1]/m$ leaves,
  2. i internal vertices $n = mi + 1$ vertices and $l = (m-1)i + 1$ leaves,

Counting vertices in m-ary trees

- A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices. Proof: Root is counted in $+1$.
- A full m-ary tree with
  1. n vertices has $i = (n-1)/m$ internal vertices and $l = [(m-1)n+1]/m$ leaves,
  2. i internal vertices $n = mi + 1$ vertices and $l = (m-1)i + 1$ leaves,
  3. l leaves has $n = (ml-1)/(m-1)$ vertices and $i = (l-1)/(m-1)$ internal vertices.

Counting vertices in m-ary trees

- A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices. Proof: Root is counted in $+1$.
- A full m-ary tree with
  1. n vertices has $i = (n-1)/m$ internal vertices and $l = [(m-1)n+1]/m$ leaves,
  2. i internal vertices $n = mi + 1$ vertices and $l = (m-1)i + 1$ leaves,
  3. l leaves has $n = (ml-1)/(m-1)$ vertices and $i = (l-1)/(m-1)$ internal vertices.
- All of it can be solved by $n = mi + 1$ and $n = l + i$.

Counting vertices in m-ary trees

- A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices. Proof: Root is counted in $+1$.
- A full m-ary tree with
    1. n vertices has $i = (n-1)/m$ internal vertices and $l = [(m-1)n + 1]/m$ leaves,
    2. i internal vertices $n = mi + 1$ vertices and $l = (m-1)i + 1$ leaves,
    3. l leaves has $n = (ml - 1)/(m-1)$ vertices and $i = (l-1)/(m-1)$ internal vertices.
- All of it can be solved by $n = mi + 1$ and $n = l + i$. For eg : in 1, $i = (n-1)/m$ from $n = mi + 1$, insert this in $n = l + i$ to get $l = [(m-1)n + 1]/m$.

Exercise Question

Suppose that someone starts a chain letter. Each person who receives the letter is asked to send it on to 4 other people. Some people do this, but others do not send any letters. How many people have seen the letter, including the first person, if no one receives $\gneq 1$ letter and if the chain letter ends after there have been 100 people who read it but did not send it out? How many people sent out the letter?

- We can use a 4-ary tree.

Exercise Question

Suppose that someone starts a chain letter. Each person who receives the letter is asked to send it on to 4 other people. Some people do this, but others do not send any letters. How many people have seen the letter, including the first person, if no one receives $\gneq 1$ letter and if the chain letter ends after there have been 100 people who read it but did not send it out? How many people sent out the letter?

- We can use a 4-ary tree.
- Internal vertices - people who sent out the letter, leaves-people who didn't.

Exercise Question

Suppose that someone starts a chain letter. Each person who receives the letter is asked to send it on to 4 other people. Some people do this, but others do not send any letters. How many people have seen the letter, including the first person, if no one receives $\gneq 1$ letter and if the chain letter ends after there have been 100 people who read it but did not send it out? How many people sent out the letter?

- We can use a 4-ary tree.
- Internal vertices - people who sent out the letter, leaves- people who didn't.
- No of leaves $l = 100$.

Exercise Question

Suppose that someone starts a chain letter. Each person who receives the letter is asked to send it on to 4 other people. Some people do this, but others do not send any letters. How many people have seen the letter, including the first person, if no one receives $\gneq 1$ letter and if the chain letter ends after there have been 100 people who read it but did not send it out? How many people sent out the letter?

- We can use a 4-ary tree.
- Internal vertices - people who sent out the letter, leaves- people who didn't.
- No of leaves $l = 100$.
- From previous result we have $n = (4 \cdot 100 - 1)/(4 - 1) = 133$ - these many people saw the letter

Exercise Question

Suppose that someone starts a chain letter. Each person who receives the letter is asked to send it on to 4 other people. Some people do this, but others do not send any letters. How many people have seen the letter, including the first person, if no one receives $\gneq$ 1 letter and if the chain letter ends after there have been 100 people who read it but did not send it out? How many people sent out the letter?

- We can use a 4-ary tree.
- Internal vertices - people who sent out the letter, leaves- people who didn't.
- No of leaves $l = 100$.
- From previous result we have $n = (4 \cdot 100 - 1)/(4 - 1) = 133$ - these many people saw the letter
- Number of internal vertices is $133 - 100 = 33$ - they sent out the letter.

Terminology related to m-ary trees

- Level of a vertex v in a rooted tree – length of the unique path from the root to this vertex.

Terminology related to m-ary trees

- Level of a vertex v in a rooted tree – length of the unique path from the root to this vertex.
- Level of the root – 0.

Terminology related to m-ary trees

- Level of a vertex v in a rooted tree – length of the unique path from the root to this vertex.
- Level of the root – 0.
- Height of a rooted tree – maximum of levels, i.e. length of the longest path from the root to any vertex.

Terminology related to m-ary trees

- Level of a vertex v in a rooted tree – length of the unique path from the root to this vertex.
- Level of the root – 0.
- Height of a rooted tree – maximum of levels, i.e. length of the longest path from the root to any vertex.
- Balanced - A rooted m-ary tree of height h is balanced if all leaves at levels h or $h - 1$.

Terminology related to m-ary trees

- Level of a vertex v in a rooted tree – length of the unique path from the root to this vertex.
- Level of the root – 0.
- Height of a rooted tree – maximum of levels, i.e. length of the longest path from the root to any vertex.
- Balanced - A rooted m-ary tree of height h is balanced if all leaves at levels h or $h - 1$.

Theorem

There are at most m^h leaves in an m-ary tree of height h.

If an m-ary tree of height h has l leaves, then $h \geq \lceil log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil log_m l \rceil$.

- $l \leq m^h$ from previous theorem.

If an m-ary tree of height h has l leaves, then $h \geq \lceil log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil log_m l \rceil$.

- $l \leq m^h$ from previous theorem.
- $log_m l \leq h$, h is an integer, $h \geq \lceil log_m l \rceil$.

m-ary trees

If an m-ary tree of height h has l leaves, then $h \geq \lceil log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil log_m l \rceil$.

- $l \leq m^h$ from previous theorem.
- $log_m l \leq h$, h is an integer, $h \geq \lceil log_m l \rceil$.
- Suppose that the tree is balanced, each leaf is at level h or $h - 1$.

m-ary trees

If an m-ary tree of height h has l leaves, then $h \geq \lceil log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil log_m l \rceil$.

- $l \leq m^h$ from previous theorem.
- $log_m l \leq h$, h is an integer, $h \geq \lceil log_m l \rceil$.
- Suppose that the tree is balanced, each leaf is at level h or $h - 1$.
- The height of the tree is h there is at least one leaf is at level h.

If an m-ary tree of height h has l leaves, then $h \geq \lceil log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil log_m l \rceil$.

- $l \leq m^h$ from previous theorem.
- $log_m l \leq h$, h is an integer, $h \geq \lceil log_m l \rceil$.
- Suppose that the tree is balanced, each leaf is at level h or $h - 1$.
- The height of the tree is h there is at least one leaf is at level h.
- There are therefore more than m^{h-1} leaves.

m-ary trees

If an m-ary tree of height h has l leaves, then $h \geq \lceil log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil log_m l \rceil$.

- $l \leq m^h$ from previous theorem.
- $log_m l \leq h$, h is an integer, $h \geq \lceil log_m l \rceil$.
- Suppose that the tree is balanced, each leaf is at level h or $h-1$.
- The height of the tree is h there is at least one leaf is at level h.
- There are therefore more than m^{h-1} leaves.
- We have $m^{h-1} \lneq l \leq m^h$, taking log $h-1 \lneq log_m l \leq h \Rightarrow h = \lceil log_m l \rceil$.

Applications of Trees - Binary Search Tree (BST)

- Searching for items in a list - an important task in computer science.

Applications of Trees - Binary Search Tree (BST)

- Searching for items in a list - an important task in computer science.
- Each child of a vertex is designated a right or left child.

Applications of Trees - Binary Search Tree (BST)

- Searching for items in a list - an important task in computer science.
- Each child of a vertex is designated a right or left child.
- Each vertex is assigned a key – key of a vertex is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.

Applications of Trees - Binary Search Tree (BST)

- Searching for items in a list - an important task in computer science.
- Each child of a vertex is designated a right or left child.
- Each vertex is assigned a key – key of a vertex is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.
- Construct a binary search tree –

Computational Complexity of Adding and Locating an Item in BST

- Consider a binary search tree T for a list of n items.

Computational Complexity of Adding and Locating an Item in BST

- Consider a binary search tree T for a list of n items.
- We form a full binary tree U from T by adding unlabeled vertices so that every vertex with a key has two children.

Computational Complexity of Adding and Locating an Item in BST

- Consider a binary search tree T for a list of n items.
- We form a full binary tree U from T by adding unlabeled vertices so that every vertex with a key has two children.
- The most comparisons needed to add a new item is the length of the longest path in U from the root to a leaf.

Computational Complexity of Adding and Locating an Item in BST

- Consider a binary search tree T for a list of n items.
- We form a full binary tree U from T by adding unlabeled vertices so that every vertex with a key has two children.
- The most comparisons needed to add a new item is the length of the longest path in U from the root to a leaf.
- The internal vertices of U are vertices of T, therefore U has n vertices.

Computational Complexity of Adding and Locating an Item in BST

- Consider a binary search tree T for a list of n items.
- We form a full binary tree U from T by adding unlabeled vertices so that every vertex with a key has two children.
- The most comparisons needed to add a new item is the length of the longest path in U from the root to a leaf.
- The internal vertices of U are vertices of T, therefore U has n vertices.
- From previous result (which one?) we have that U has $n+1$ leaves.

Computational Complexity of Adding and Locating an Item in BST

- From previous result we have that the height of U is $\geq h = \lceil log(n+1) \rceil$.

Computational Complexity of Adding and Locating an Item in BST

- From previous result we have that the height of U is $\geq h = \lceil log(n+1) \rceil$.
- Therefore, we need to perform at least $\lceil log(n+1) \rceil$ comparisons to add an item.

Computational Complexity of Adding and Locating an Item in BST

- From previous result we have that the height of U is $\geq h = \lceil log(n+1) \rceil$.
- Therefore, we need to perform at least $\lceil log(n+1) \rceil$ comparisons to add an item.
- If a BST is balanced then its height is $\lceil log(n+1) \rceil$ and so no more comparisons are required.

Computational Complexity of Adding and Locating an Item in BST

- From previous result we have that the height of U is $\geq h = \lceil log(n+1) \rceil$.
- Therefore, we need to perform at least $\lceil log(n+1) \rceil$ comparisons to add an item.
- If a BST is balanced then its height is $\lceil log(n+1) \rceil$ and so no more comparisons are required.
- This is why there are many algorithms that try to rebalance BSTs after items are added.

- Rooted trees for modeling problems with a series of decisions that leads to a solution.

- Rooted trees for modeling problems with a series of decisions that leads to a solution.
- Example –Suppose there are seven coins, all with the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which of the eight coins is the counterfeit one?

Applications of Trees - Decision Trees

- 3 possibilities – equal, first pan heavier, second pan heavier. Therefore 3-ary tree.

Applications of Trees - Decision Trees

- 3 possibilities – equal, first pan heavier, second pan heavier.
  Therefore 3-ary tree.
- 8 possible outcomes since each of the 8 can be fake.
  Therefore at least 8 leaves.

Applications of Trees - Decision Trees

- 3 possibilities – equal, first pan heavier, second pan heavier. Therefore 3-ary tree.
- 8 possible outcomes since each of the 8 can be fake. Therefore at least 8 leaves.
- The largest number of weighings – height of the tree is at least $\lceil log_3 8 \rceil = 2$.

Applications of Trees - Decision Trees

- 3 possibilities – equal, first pan heavier, second pan heavier. Therefore 3-ary tree.
- 8 possible outcomes since each of the 8 can be fake. Therefore at least 8 leaves.
- The largest number of weighings – height of the tree is at least $\lceil log_3 8 \rceil = 2$.
- Therefore we need at least 2 weighings.

Complexity of Comparison based sorting algorithms

- Using decisions trees we can find a lower bound for the worst-case complexity of sorting algorithms.

Complexity of Comparison based sorting algorithms

- Using decisions trees we can find a lower bound for the worst-case complexity of sorting algorithms.
- Given a list of n elements sorting algorithms are based on binary comparisons.

Complexity of Comparison based sorting algorithms

- Using decisions trees we can find a lower bound for the worst-case complexity of sorting algorithms.
- Given a list of n elements sorting algorithms are based on binary comparisons.
- A binary decision tree in which each internal vertex represents a comparison of two elements.

Complexity of Comparison based sorting algorithms

- Using decisions trees we can find a lower bound for the worst-case complexity of sorting algorithms.
- Given a list of n elements sorting algorithms are based on binary comparisons.
- A binary decision tree in which each internal vertex represents a comparison of two elements.
- Each leaf represents one of the $n!$ permutations of n elements.

Complexity of Comparison based sorting algorithms

- Using decisions trees we can find a lower bound for the worst-case complexity of sorting algorithms.
- Given a list of n elements sorting algorithms are based on binary comparisons.
- A binary decision tree in which each internal vertex represents a comparison of two elements.
- Each leaf represents one of the $n!$ permutations of n elements.
- Complexity is based on number of binary comparisons, worst case complexity is based on largest number of binary comparisons needed to sort a list with n elements.

Complexity of Comparison based sorting algorithms

- Using decisions trees we can find a lower bound for the worst-case complexity of sorting algorithms.
- Given a list of n elements sorting algorithms are based on binary comparisons.
- A binary decision tree in which each internal vertex represents a comparison of two elements.
- Each leaf represents one of the $n!$ permutations of n elements.
- Complexity is based on number of binary comparisons, worst case complexity is based on largest number of binary comparisons needed to sort a list with n elements.
- That is the height of the decision tree with $n!$ leaves - at least $\lceil log n! \rceil$

Complexity of Comparison based sorting algorithms

Theorem

A sorting algorithm based on binary comparisons requires at least $\lceil log n! \rceil$ comparisons.

Exercise : $\lceil log n! \rceil$ is $\Theta(nlogn)$.
Therefore we have,

Theorem

The number of comparisons used by a sorting algorithm to sort n elements based on binary comparisons is $\Omega(nlogn)$.

So if you have a comparison sorting algorithm that uses $\Theta(nlogn)$ comparisons in the worst case you have an optimal algorithm.

Average Case Complexity of Comparison based sorting algorithms

Theorem

The average number of comparisons used by a sorting algorithm to sort n elements based on binary comparisons is $\Omega(nlogn)$.

Proof : The average number of comparisons is average depth of a leaf in the decision tree.
Exercise: Average depth of a leaf in a binary tree with N vertices is $\Omega(logN)$.

Average Case Complexity of Comparison based sorting algorithms

Theorem

The average number of comparisons used by a sorting algorithm to sort n elements based on binary comparisons is $\Omega(nlogn)$.

Proof : The average number of comparisons is average depth of a leaf in the decision tree.
Exercise: Average depth of a leaf in a binary tree with N vertices is $\Omega(logN)$.
$\Omega(logn!)$ is $\Omega(nlogn)$ since $logn!$ is $\Theta(nlogn)$.
Interesting other applications : Huffman coding and Game Trees.

Tree Traversal

- Ordered rooted trees are used to store information and therefore we need procedures for visiting each vertex.

Tree Traversal

- Ordered rooted trees are used to store information and therefore we need procedures for visiting each vertex.
- Traversal algorithms – preorder, inorder and postorder traversal.

Tree Traversal

- Ordered rooted trees are used to store information and therefore we need procedures for visiting each vertex.
- Traversal algorithms – preorder, inorder and postorder traversal.

Definition (Preorder Traversal)

Let T be an ordered rooted tree with root r. If T contains only of r, then r is the preorder traversal of T.

- Ordered rooted trees are used to store information and therefore we need procedures for visiting each vertex.
- Traversal algorithms – preorder, inorder and postorder traversal.

Definition (Preorder Traversal)

Let T be an ordered rooted tree with root r. If T contains only of r, then r is the preorder traversal of T. Or else suppose $T_1, T_2, \ldots, T_r$ are the subtrees at r from left to right in T. The preorder traversal begins by vising r, continues by traversing T_1 in preorder, then T_2 in preorder and so on until T_n is traversed in preorder.

Tree Traversal

- Ordered rooted trees are used to store information and therefore we need procedures for visiting each vertex.
- Traversal algorithms – preorder, inorder and postorder traversal.

Definition (Preorder Traversal)

Let T be an ordered rooted tree with root r. If T contains only of r, then r is the preorder traversal of T. Or else suppose $T_1, T_2, \ldots, T_r$ are the subtrees at r from left to right in T. The preorder traversal begins by vising r, continues by traversing T_1 in preorder, then T_2 in preorder and so on until T_n is traversed in preorder.

Example –

Definition (Inorder Traversal)

Let T be an ordered rooted tree with root r. If T contains only of r, then r is the inorder traversal of T.

Tree Traversal

Definition (Inorder Traversal)

Let T be an ordered rooted tree with root r. If T contains only of r, then r is the inorder traversal of T. Or else suppose $T_1, T_2, \ldots, T_r$ are the subtrees at r from left to right in T. The inorder traversal begins by traversing T_1 in inorder, then visiting r, continues by traversing T_2 in inorder, then T_3 in inorder and so on until T_n is traversed in inorder.

Definition (Inorder Traversal)
Let T be an ordered rooted tree with root r. If T contains only of r, then r is the inorder traversal of T. Or else suppose $T_1, T_2, \ldots, T_r$ are the subtrees at r from left to right in T. The inorder traversal begins by traversing T_1 in inorder, then visiting r, continues by traversing T_2 in inorder, then T_3 in inorder and so on until T_n is traversed in inorder.

Example –

Definition (Postorder Traversal)

Let T be an ordered rooted tree with root r. If T contains only of r, then r is the postorder traversal of T.

Definition (Postorder Traversal)

Let T be an ordered rooted tree with root r. If T contains only of r, then r is the postorder traversal of T. Or else suppose $T_1, T_2, \ldots, T_r$ are the subtrees at r from left to right in T. The postorder traversal begins by traversing T_1 in postorder, continues by traversing T_2 in postorder, then T_3 in postorder and so on then T_n is traversed in postorder and finally ends by visiting r,.

Definition (Postorder Traversal)
Let T be an ordered rooted tree with root r. If T contains only of r, then r is the postorder traversal of T. Or else suppose $T_1, T_2, \ldots, T_r$ are the subtrees at r from left to right in T. The postorder traversal begins by traversing T_1 in postorder, continues by traversing T_2 in postorder, then T_3 in postorder and so on then T_n is traversed in postorder and finally ends by visiting r,.

Example –

Definition (Postorder Traversal)

Let T be an ordered rooted tree with root r. If T contains only of r, then r is the postorder traversal of T. Or else suppose $T_1, T_2, \ldots, T_r$ are the subtrees at r from left to right in T. The postorder traversal begins by traversing T_1 in postorder, continues by traversing T_2 in postorder, then T_3 in postorder and so on then T_n is traversed in postorder and finally ends by visiting r,.

Example –

Exercise – Design recursive algorithms for these traversals.

Inorder traversal of a BST gives ——-.

Infix, Prefix and Postfix Notation

- Represent complicated expressions such as compound propositions, combinations of sets and arithmetic expression using ordered rooted trees.

Infix, Prefix and Postfix Notation

- Represent complicated expressions such as compound propositions, combinations of sets and arithmetic expression using ordered rooted trees.
- Internal vertices represent operations and leaves the numbers or variables.
- Example of a tree for $((x + y)^2) + (x - 4)/3)$–

Infix, Prefix and Postfix Notation

- Inorder traversal - produces original expression except for certain cases:

Infix, Prefix and Postfix Notation

- Inorder traversal - produces original expression except for certain cases:
- Consider inorder traversals of the binary trees which represent $(x+y)/(x+3)$, $(x+(y/x))+3$, $x+(y/(x+3))$ –

Infix, Prefix and Postfix Notation

- Inorder traversal - produces original expression except for certain cases:
- Consider inorder traversals of the binary trees which represent $(x + y)/(x + 3)$, $(x + (y/x)) + 3$, $x + (y/(x + 3))$ –
- They all lead to infix expression $x + y/x + 3$.

Infix, Prefix and Postfix Notation

- Inorder traversal - produces original expression except for certain cases:
- Consider inorder traversals of the binary trees which represent $(x+y)/(x+3)$, $(x+(y/x))+3$, $x+(y/(x+3))$ –
- They all lead to infix expression $x+y/x+3$.
- You need to include paranthesis when you encounter an operation in inorder traversal - that is called infix form.

Infix, Prefix and Postfix Notation

- Prefix form : traverse the rooted tree in preorder.

Infix, Prefix and Postfix Notation

- Prefix form : traverse the rooted tree in preorder.
- Also called Polish notation after a Polish logician.

Infix, Prefix and Postfix Notation

- Prefix form : traverse the rooted tree in preorder.
- Also called Polish notation after a Polish logician.
- An expression in prefix notation is unambiguous provided each operation has specified number of operands.

Infix, Prefix and Postfix Notation

- Prefix form : traverse the rooted tree in preorder.
- Also called Polish notation after a Polish logician.
- An expression in prefix notation is unambiguous provided each operation has specified number of operands.
- Postfix form - Traverse the tree in postorder.

Infix, Prefix and Postfix Notation

- Prefix form : traverse the rooted tree in preorder.
- Also called Polish notation after a Polish logician.
- An expression in prefix notation is unambiguous provided each operation has specified number of operands.
- Postfix form - Traverse the tree in postorder.
- Called reverse Polish notation, notations are unambiguous.