Trees

Dr. Isaac Griffith

Idaho State University

Outline

The lecture is structured as follows:

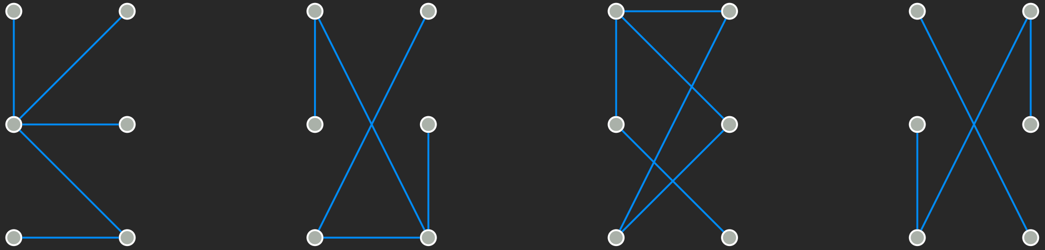
- Tree Basics
- Applications of Trees
  - Prefix Trees and Huffman Coding
  - Decision Trees
  - BSTs
  - Game Trees
- Tree Traversals
  - Pre-, In-, Post-order
- Tree Induction
- Spanning Trees
  - BFS and DFS
- Minimum Spanning Trees

Introduction to Trees

CS 1187

What's Trees?

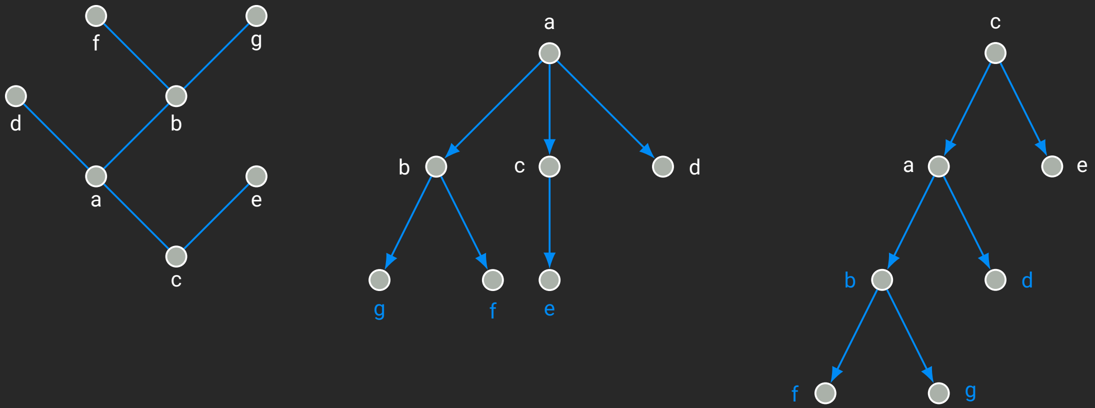
- A **tree** is a connected undirected graph with no simple circuits



- **Theorem:** An undirected graph is a tree iff there is a unique simple path between any two of its vertices

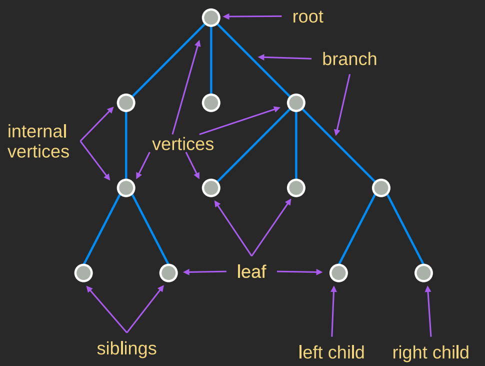
Rooted Trees

- 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.



Terminologies of Rooted Trees

- If v is a vertex in T other than the root, the **parent** of v is the unique vertex u such that there is a directed edge from u to v
- If u is the parent of v, v is called a **child** of u
- Vertices with the same parent are called **siblings**



ROAR

Terminologies of Rooted Trees

- The **ancestors** of a vertex other than the 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 tree is called a **leaf** if it has no children
- Vertices that have children are called **internal vertices**
- If *a* is a vertex is a tree, the **subtree** with *a* as its root is the subgraph of the tree consisting of *a* and its descendants and all edges incident to these descendants.

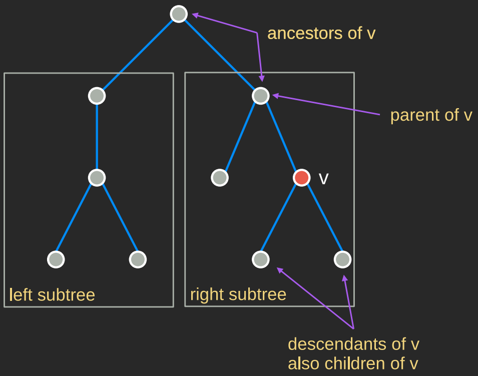
m-Ary Trees

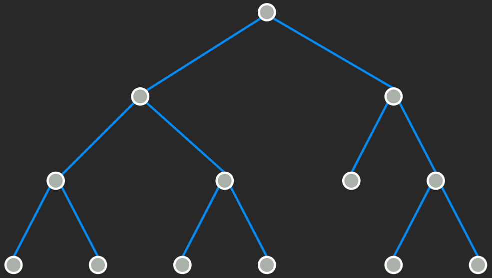
- A rooted tree is called an **m-ary tree** if every internal vertex has no more than *m* children. The tree is called a **full m-ary tree** if every internal vertex has exactly *m* children. An m-ary tree with $m = 2$ is called a **binary tree**

- An **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.

- In an ordered binary tree (usually just called a 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**. The tree rooted at the left child (or right child, respectively) of a vertex is called the **left subtree** (or **right subtree**, respectively) of this vertex.

Tree Terminologies





Example Full Binary Tree

Properties of Trees

- **Theorem:** A tree with n vertices has $n - 1$ edges
- **Theorem:** A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices

Properties of Trees

- **Theorem:** 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 has $n = mi + 1$ 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

- **Theorem:** There are at most m^h leaves in any m-ary tree of height h

ROAR

Trees in Haskell

CS 1187

ROAR

Representing Trees in Haskell

- The representation of a *binary tree* in Haskell is as follows (limited to integer data)

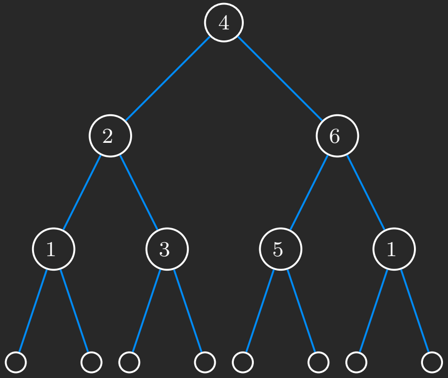
```haskell
data BinTree a
  = BinLeaf
  | BinNode a (BinTree a) (BinTree a)
  deriving Show
```

- That is a tree is either
  - A leaf without a value, or
  - A node with a value and a right and a left subtree

Haskell Example

Example tree

```haskell
tree :: BinTree Int
tree = BinNode 4
         (BinNode 2
            (BinNode 1 BinLeaf BinLeaf)
            (BinNode 3 BinLeaf BinLeaf)
         (BinNode 6
            (BinNode 5 BinLeaf BinLeaf)
            (BinNode 7 BinLeaf BinLeaf))
```



Applications of Trees

CS 1187

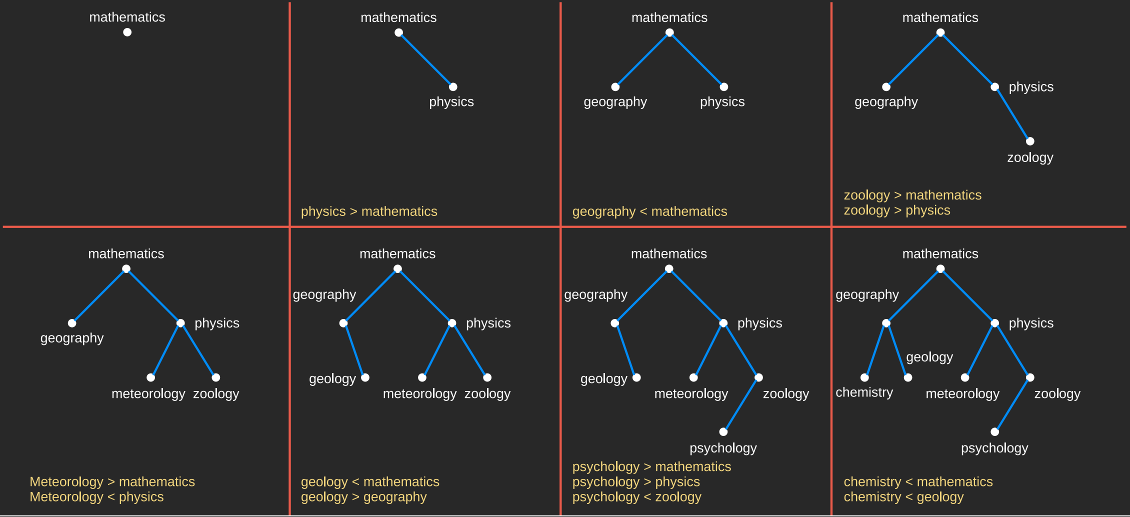
ROAR

Binary Search Trees

- **Binary Search Tree:** a binary tree in which each child of a vertex is designated as a right or left child, no vertex has more than one right or left child, and each vertex is labeled with a key, which is one of the items.

- Vertices are assigned keys so that the key of a vertex is both larger than keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.

Binary Search Trees



Searching a BST

- We can implement a BST in Haskell as follows:

```haskell
bstSearch :: Ord a => a -> BinTree (a, b) -> Maybe b
bstSearch key BinLeaf = Nothing
bstSearch key (BinNode (x, y) t1 t2) =
  if key == x
    then Just y
    else if key < x
      then bstSearch key t1
      else bstSearch key t2
```

Adding Items to a BST

Idaho State University | Computer Science

```
procedure INSERTION(T: binary search tree, x item)
    v := root of T
    {a vertex is not present in T has the value null}
    while v ≠ null and label(v) ≠ x do
        if x < label(v) then
            if left child of v ≠ null then
                v := left child of v
            else
                add new vertex as a left child of v and set v := null
        else
            if right child of v ≠ null then
                v := right child of v
            else
                add new vertex as a right child of v and set v := null
    if root of T = null then
        add a vertex v to the tree and label it with x
    else if v is null or label(v) ≠ x then
        label new vertex with x and let v be this new vertex
    return v {v = location of x}
```


ROAR

BST Insertion

- The prior algorithm for insertion can be implemented as follows in Haskell:

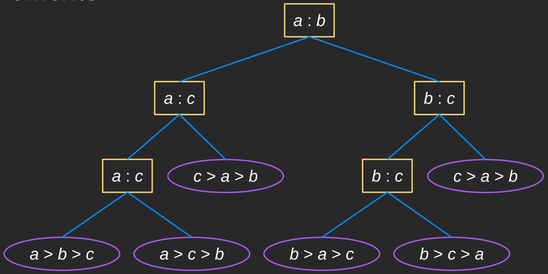
```haskell
insert :: Ord a => (a, b) -> BinTree (a, b) -> BinTree (a, b)
insert (key, d) BinLeaf = BinNode (key, d) BinLeaf BinLeaf
insert (key, d) (BinNode (x, y) t1 t2) =
  if key == x
    then BinNode (key, d) t1 t2
    else if key < x
      then BinNode (x, y) (insert (key, d) t1) t2
      else BinNode (x, y) t1 (insert (key, d) t2)
```

Decision Trees

- **Decision tree:** a rooted tree in which each internal vertex corresponds to a decision with a subtree at these vertices for each possible outcome fo the decision.

- Such trees can be used to model problems in which a series of decisions leads to a solution.

Example: Decision tree for sorting three distinct elements



Prefix Codes

- **Prefix Codes:** codes wherein a bit string represents a letter, and no bit strings corresponds to more than one sequence of letters.

- **Huffman coding:** a special case of prefix codes
  - Uses frequencies of symbols in a string and produces as output a prefix code that encodes the string using the fewest possible bits, among all possible binary prefix codes for these symbols.

Huffman Coding

Idaho State University | Computer Science

procedure HUFFMAN(C: symbols a_i with frequencies w_i, $i = 1, \ldots, n$)

$F :=$ forest of n rooted trees, each consisting of a single vertex a_i and assigned weight w_i

while F is not a tree **do**

Replace the rooted trees T and T' of least weights from F with $w(T) \geq w(T')$ with a tree having a new root that has T as it left subtree and T' as its right subtree. Label the new edge to T with 0 and the new edge to T' with 1.

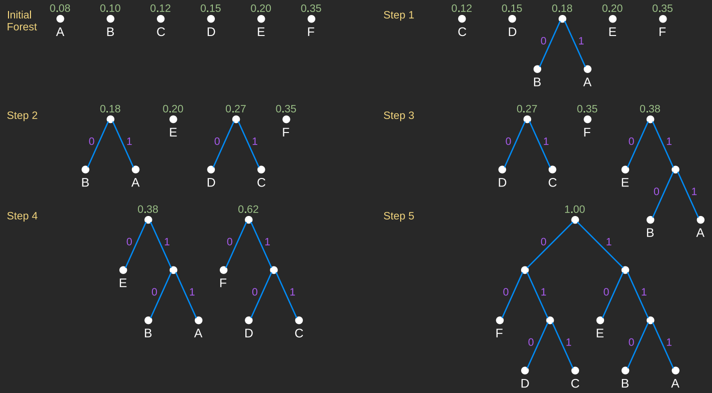
Assign $w(T) + w(T')$ as the weight of the new tree

{the Huffman coding for the symbol a_i is the concatenation of the labels of the edges in the unique path from the root to the vertex a_i}

ROAR

Huffman Coding Example

- **Example:** Use Huffman coding to encode the following symbols with the frequencies: A: 0.08, B: 0.10, C: 0.12, D: 0.15, E:0.20, F: 0.35



- **Solution:** This encodes A by 111, B by 110, C by 011, D by 010, E by 10, and F by 00.

  The average number of bits used to encode a symbol using this encoding is:

$$3 \cdot 0.08 + 3 \cdot 0.10 + 3 \cdot 0.12 + \\ 3 \cdot 0.15 + 2 \cdot 0.020 + 2 \cdot 0.35 = 2.45$$

Game Trees

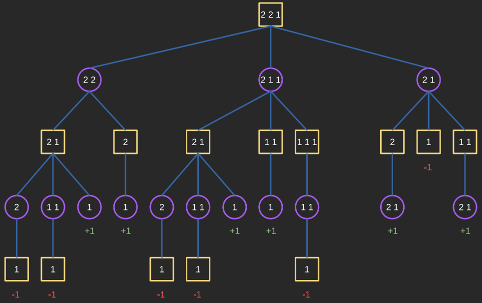
- Trees can be used to analyze certain types of games
  - Tic-tac-toe
  - Checkers
  - Chess
  - Nim
- These types of games are called perfect-information games
  - both players know the moves of the other players
  - the state of the game is known by both players
  - No chance involved

Game Trees

- **Game Trees:** where the vertices represent the positions that a game can be in as it progresses, and the edges represent legal moves.
  - Tend to be very large but can be simplified by combining all symmetric positions into the same node
  - Root node represents the starting position.
  - Even levels are represented by boxes and are the first player's move
  - Odd levels are represented by circles and are the second player's moves
  - We assign values to leaves to represent the payoff to the first player if the game terminates (terminal vertices are represented by boxes)

The Game of Nim

- At the start of the game there are a number of piles of stones.
- Players take turns making moves
  - A legal move consists of removing one or more stones from one of the piles, without removing all stones left.
  - A player without a legal move loses
- A game tree for this game is shown



ROAR

Minmax

- **Definition:** The *value of a vertex in a game tree* is defined recursively as:
  i. the value of a leaf is the payoff to the first player when the game terminates in the position represented by this leaf.
  ii. the value of an internal vertex at an even level is the maximum of the values of its children, and the value of an internal vertex at an odd level is the minimum of the values of its children.

- **Theorem:** The value of a vertex of a game tree tells us the payoff to the first player if both players follow the minmax strategy and play starts from the position represented by this vertex.

- **Minmax Strategy:** where the first player moves to a position represented by a child with maximum value and the send player moves to a position of a child with minimum value.

- Various approaches can be used to devise good strategies
  - *alpha-beta pruning* eliminates computation by pruning portions of game trees based on ancestor vertex values
  - *evaluation functions* to estimate the value of internal vertices

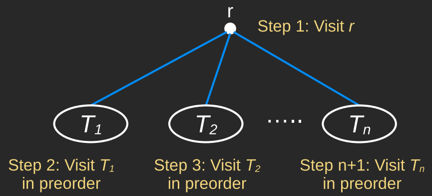
- More information can be learned by studying **combinatorial game theory**

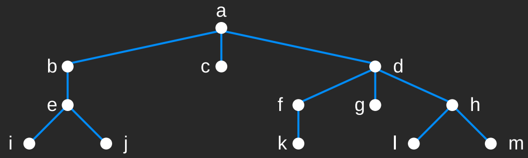
Tree Traversal

CS 1187

Preorder Traversal

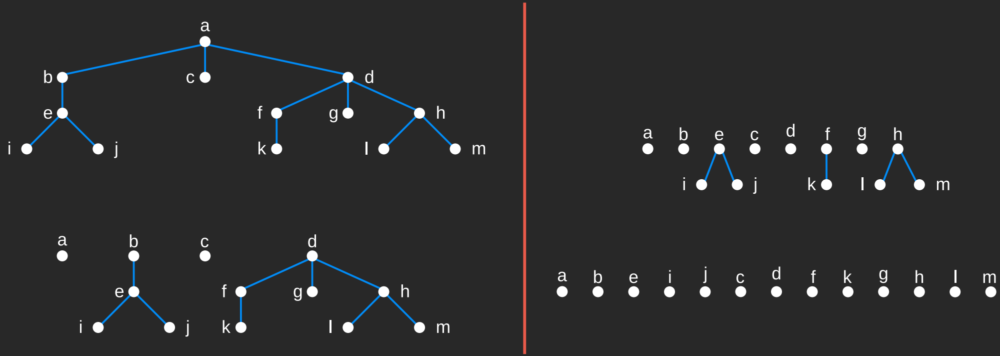
- **Definition:** Let T be an ordered rooted tree with root r. If T consists only of r, then r is the **preorder traversal** of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right in T. The **preorder traversal** begins by visiting r. It continues by traversing T_1 in preorder, then T_2 in preorder, and so on, until T_n is traversed in preorder





Examples of Preorder Traversal

Idaho State University | Computer Science



ROAR

Pseudocode of Preorder Traversal

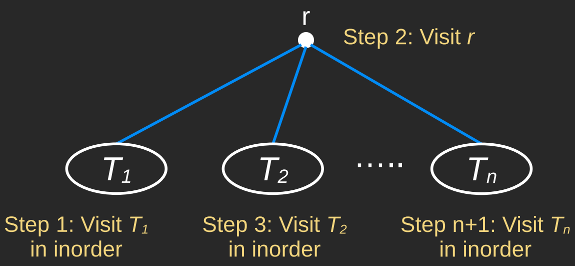
Algorithm:

procedure PREORDER(T: ordered rooted tree)
  $r :=$ root of T
  listr
  for each child c of r from left to right **do**
    $T(c) :=$ subtree with c as its root
    PREORDER($T(c)$)

```haskell
preorder :: BinTree a -> [a]
preorder BinLeaf = []
preorder (BinNode x t1 t2) =
  [x] ++ preorder t1 ++ preorder t2
```

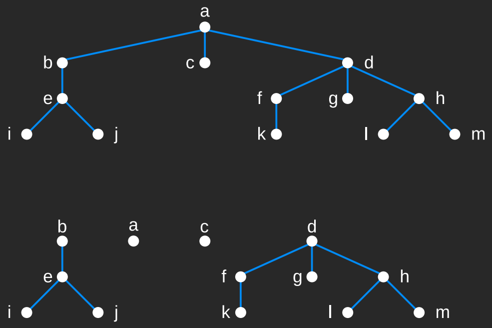
Inorder Traversal

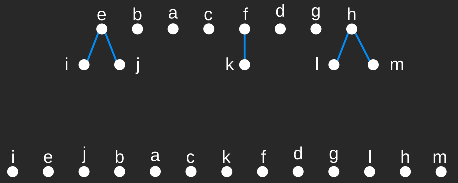
- **Defintion:** Let T be an ordered rooted tree with root r. If T consists only of r, then r is the **inorder traversal** of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right. The **inorder traversal** begins by traversing T_1 in inorder, then visiting r. It continues by traversing T_2 in inorder, then T_3 in inorder, . . ., and finally T_n in inorder.



ROAR

Examples of Inorder Traversal





Pseudocode of Inorder Traversal

Algorithm:

procedure INORDER(T: ordered rooted tree)
  r := root of T
  if r is a leaf **then**
    list r
  else
    l := first child of r from left to right
    $T(l)$:= subtree with l as its root
    INORDER($T(l)$)
    list r
    for each child c of r except for l from left to right **do**
      $T(c)$:= subtree with c as its root
      INORDER($T(c)$)

Haskell Implementation:

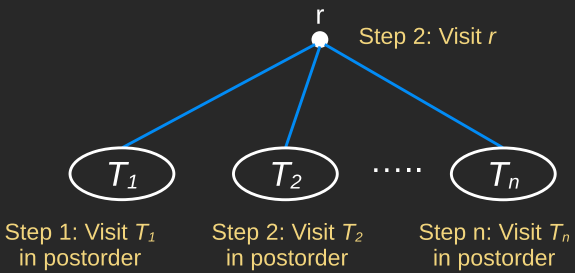
```haskell
inorder :: BinTree a -> [a]
inorder BinLeaf = []
inorder (BinNode x t1 t2) =
  inorder t1 ++ [x] ++ inorder t2
```


ROAR

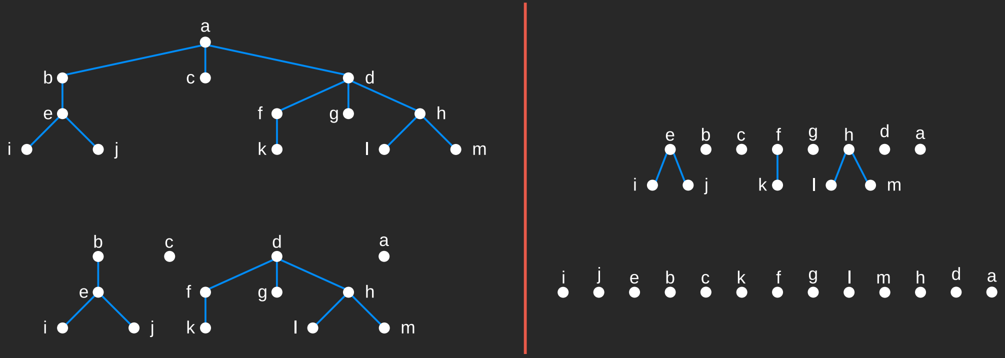
Postorder Traversal

Idaho State University | Computer Science

- **Definition:** Let T be an ordered rooted tree with root r. If T consists only of r, then r is the **postorder traversal** of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right in T. The **postorder traversal** begins by traversing T_1 in postorder, then T_2 in postorder, . . ., then T_n in postorder, and end by visiting r.



Examples of Postorder Traversal



Pseudocode of Postorder Traversal

Algorithm:

procedure POSTORDER(T: ordered rooted tree)
  $r :=$ root of T
  for each child c of r from left to right **do**
    $T(c) :=$ subtree with c as its root
    POSTORDER($T(c)$)
  list r

Haskell Implementation:

```haskell
postorder :: BinTree a -> [a]
postorder BinLeaf = []
postorder (BinNode x t1 t2) =
  postorder t1 ++ postorder t2 ++ [x]
```

Procession Tree Structure

- STDM provides several functions to process a tree, including measuring tree size or the ability to affect its shape
  - `reflect` - takes a binary tree and returns its mirror image

```haskell
reflect :: BinTree a -> BinTree a
reflect BinLeaf = BinLeaf
reflect (BinNode n l r) = BinNode n (reflect r) (reflect l)
```

  - `height` - measures the height of a tree between its root and its deepest leaf (an empty tree has height zero)

```haskell
height :: BinTree a -> Integer
height BinLeaf = 0
height (BinNode x t1 t2) = 1 + max (height t1) (height t2)
```

Procession Tree Structure

- STDM provides several functions to process a tree, including measuring tree size or the ability to affect its shape
  - `size` - calculates the size of a tree, as the number of nodes a tree has

```
size :: BinTree a -> Integer
size BinLeaf = 0
size (Node x t1 t2) = 1 + size t1 + size t2
```

  - `balanced` - determines if the binary tree is *balanced* or not

```
balanced :: BinTree a -> Bool
balanced BinLeaf = True
balanced (BinNode x t1 t2) =
  balanced t1 && balanced t2 && (height t1 == height t2)
```

- **Theorem:** Let $h = \texttt{height}\ t$. If $\texttt{balanced}\ t$, then $\texttt{size}\ t = 2^h - 1$

ROAR

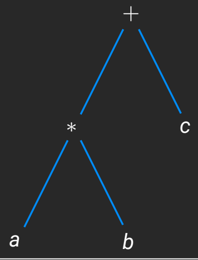
Evaluating Expression Trees

- Often when working with text we can use trees to represent documents in the language.
- Trees represent the structure of the text while omitting unimportant details
- Examples of this includes programs that manipulate language, as well as compilers and interpreters for programming languages.

Infix, Prefix, and Postfix Notation

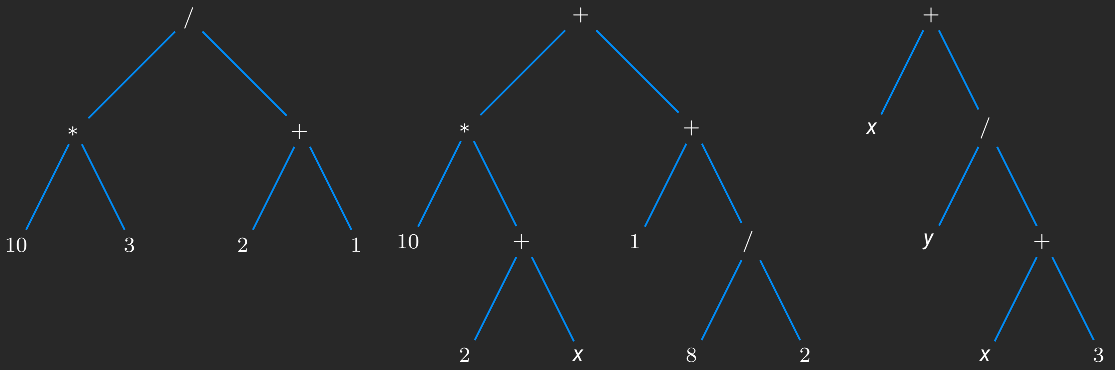
- Examples: infix, prefix, and postfix notations of $a \times b + c$
  - Infix: $a * b + c$: uses inorder traversal of an expression tree
  - Prefix: $+ * abc$ (also called Polish notation) - uses preorder traversal of an expression tree
  - Postfix: $ab * c+$ - uses postorder traversal of an expression tree

- Represented by ordered rooted trees



Examples of Expression Tree Representation

Idaho State University | Computer Science



ROAR

Evaluation Expression Trees

- Let's consider the following simple expression language

```
data Exp
  = Const Integer
  | Add Exp Exp
  | Mult Exp Exp
```

- We could then implement a simple programming language interpreter as a tree traversal, The interpretter function `eval` takes an expression tree and returns the value it represents

```
eval :: Exp -> Integer
eval (Const n) = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mult e1 e2) = eval e1 * eval e2
```

Tree Induction

CS 1187

ROAR

Tree Induction

- **Theorem:** *Principle of induction on binary trees*. Let `BinTree a` be a binary tree type as prior defined, and let $P(t)$ be a proposition on trees. Suppose the following two requirements hold:
  - *Base Case:* $P(\texttt{BinLeaf})$
  - *Induction Case:* For all `t1` and `t2` of type `BinTree a`, and all `x :: a`, suppose that the proposition holds for a tree consisting of a node, the value `a`, and the subtrees `t1` and `t2`, provided that the proposition holds for `t1` and `t2`.
    - This can be written as: $P(t_1) \land P(t_2) \rightarrow P(\texttt{BinNode x t1 t2})$
  - Then $\forall t{::}\ \texttt{BinTree a}.\ P(t)$, thus the proposition holds for all trees of finite size

ROAR

Tree Induction Example

Example: Let `t :: BinTree a` be any finite binary tree. Then `length (inorder t) = size t`

Proof:

Base Case:

```
length (inorder BinLeaf)
  = length []
  = 0
  = size BinLeaf
```

Inductive Case:

Assume the induction hypothesis:

```
length (inorder t1) = size t1
length (inorder t2) = size t2
```

Then

```
length (inorder (BinNode x t1 t2))
  = length (inorder t1 ++ [x] ++ inorder t2
  = length (inorder t1) + length [x] + leng
  = size t1 + 1 + size t2
  = size (BinNode x t1 t2)
```

Therefore the theorem holds by tree induction

Spanning Trees

CS 1187

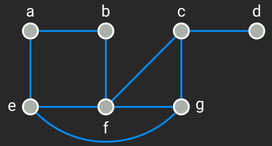
ROAR

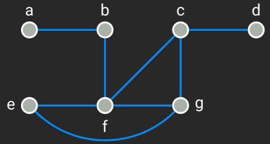
What is a Spanning Tree

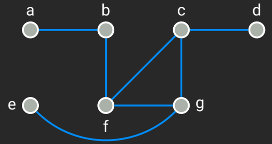
Idaho State University | Computer Science

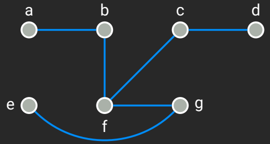
- **Definition:** Let G be a simple graph. A **spanning tree** of G is a subgraph of G that is a tree containing every vertex of G.

G









- **Theorem:** A simple graph is connected iff it has a spanning tree

How to Construct Spanning Trees?

Two approaches

- Depth-first search (DFS)
- Breadth-first search (BFS)

Algorithm: Depth-First Search

Idaho State University | Computer Science

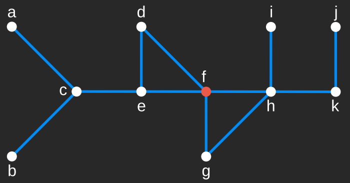
- Idea form a rooted tree by arbitrarily selecting a root
  - Form a path by successively adding vertices and edges incident with the last vertex selected
    - Only select those edges/vertices not already in the graph
  - If we can no longer go forward, we backtrack and try again.
  - This process is repeated until all nodes are visited.

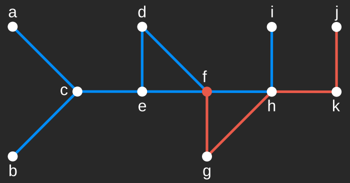
- Also called **backtracking** as the algorithm returns to vertices previously visited to add new paths.

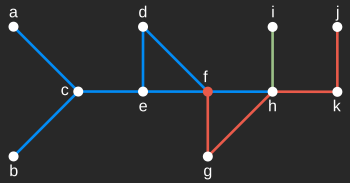
procedure DFS(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
 $T :=$ tree consisting only of the vertex v_1
 VISIT(v_1)

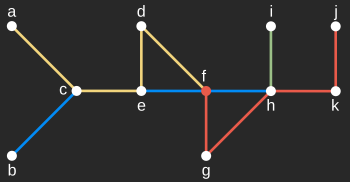
procedure VISIT(v: vertex of G)
 for each vertex w adjacent to v and not yet in T **do**
  add vertex w and edge $\{v, w\}$ to T
  VISIT(w)

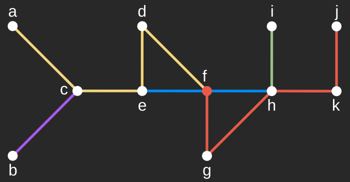
Example of DFS

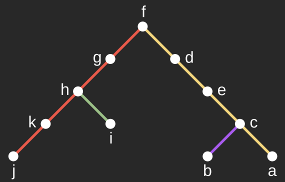












ROAR

Backtracking Applications

- Backtracking as used in DFS can also be used as a technique for the exhaustive search of all possible solutions, application of this technique include
  - Graph colorings
  - The *n*-Queens problem
  - Sums of Subsets
- Additionally, we can apply DFS (and of course BFS) in graphs and digraphs. When applied to digraphs some useful applications occur
  - Webcrawlers and Internet Search Engines
- It provides the ability to backtrack and try another path when it is known that the current path will provide no viable solutions.
  - Additionally, another *algorithmic strategy* can be applied called **branch-and-bound**
  - In **branch-and-bound** when we see a similar subtree which has already been shown to been shown that it will not render the optimal solution, rather than continuing down that path, we simply bound over it to the next subtree.

Breadth-First Search

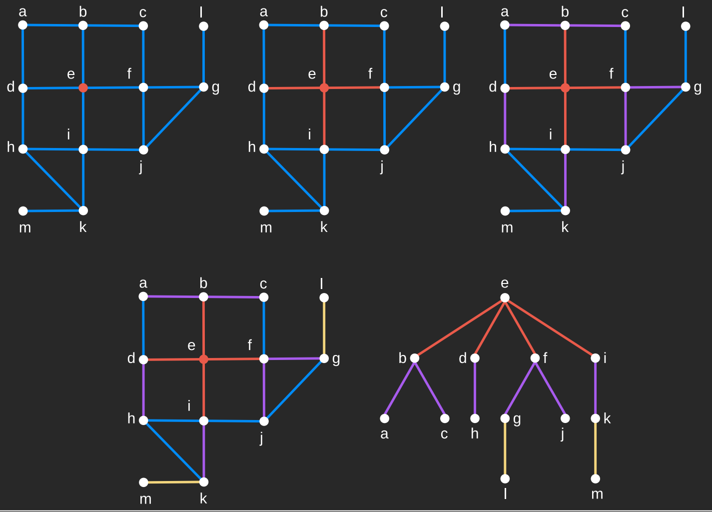
- Rather than processing all nodes and backtracking when we reach an end, here we instead process each all successive vertices of vertex.

- The idea is as follows: select an arbitrary root
  - Add all edges incident to the selected vertex
  - We then visit each of these nodes in the same way before processing their successive nodes.

Algorithm: Breadth-First Search

procedure BFS(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
    $T :=$ tree consisting only of the vertex v_1
    $L :=$ empty list
    put v_1 in the list L of unprocessed vertices
    while L is not empty **do**
        remove the first vertex, v, from L
        for each neighbor w of v and not yet in T **do**
            if w is not in L and not in T **then**
                add w to the end of the list L
                add w and edge $\{v, w\}$ to T

Example of Breadth-First Search



ROAR

DFS vs. BFS

Idaho State University | Computer Science

- Although we can specify the DFS and BFS algorithms as already stated, we can create a general iterative algorithm for both:

```
procedure GENERALSEARCH(G: connected graph with vertices v_1, v_2, ..., v_n)
    T := tree consisting only of the vertex v_1
    L := empty list
    ENQUE(L, v_1)
    while L is not empty do
        v := DEQUE(L)
        for each neighbor w of v and not yet in T do
            if w is not in L and not in T then
                ENQUE(L, w)
                add w and edge {v, w} to T
```

DFS vs. BFS

- For DFS, *L* is a Stack (LIFO)
  - `enque` is the stack `push` method (adding an item to the front of the list)
  - `deque` is the stack `pop` method (removing the first item from the list)

- For BFS, *L* is a Queue (FIFO)
  - `enque` is the queue `offer` method (adding an item to the end of the list)
  - `deque` is the queue `poll` method (removing the first item from the list)

Minimum Spanning Trees

CS 1187

Minimum Spanning Trees

- If T is a spanning tree in a weighted graph $G(V, E, w)$, the weight of T, denoted by $w(T)$, is the sum of weights of edges in T.

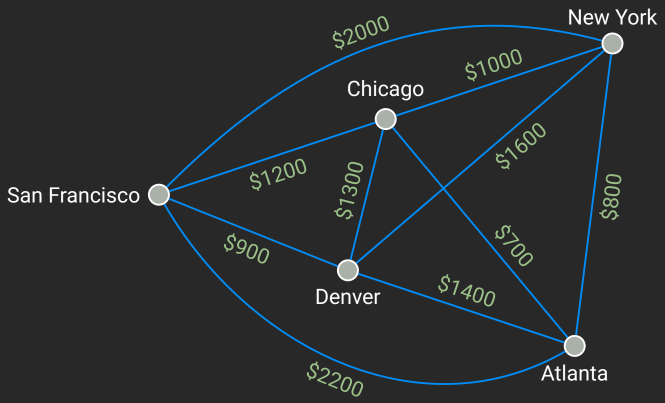
$$w(t) = \sum_{e \in T} w(e)$$

- Given a weighted graph $G(V, E, w)$, the minimum spanning tree problem is to find a spanning tree in G that has the smallest weight

Cost of a Computer Network

Idaho State University | Computer Science

- What is the smallest total cost to maintain a connected network between those five cities?

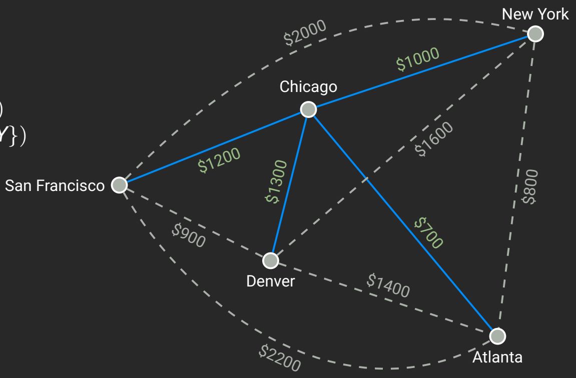


Some Spanning Trees

Idaho State University | Computer Science

- $T_1 = \left\{ \begin{array}{l} \{Chicago, SF\}, \{Chicago, Denver\}, \\ \{Chicago, Atlanta\}, \{Chicago, NY\} \end{array} \right\}$

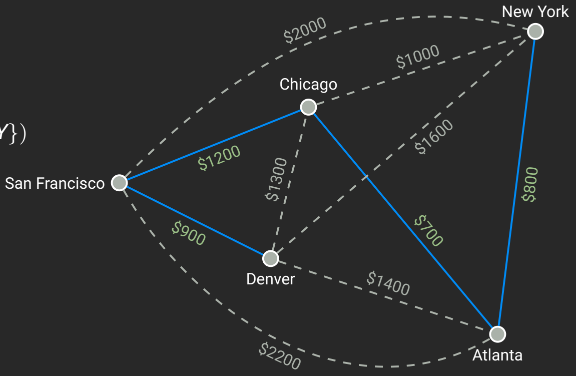
$$\begin{aligned} w(T_1) &= w(\{Chicago, SF\}) + w(\{Chicago, Denver\}) \\ &\quad + w(\{Chicago, Atlanta\}) + w(\{Chicago, NY\}) \\ &= \$1200 + \$1300 + \$700 + \$1000 = \$4200 \end{aligned}$$



Some Spanning Trees

- $T_2 = \left\{ \begin{array}{c} \{Chicago, SF\}, \{SF, Denver\}, \\ \{Chicago, Atlanta\}, \{Atlanta, NY\} \end{array} \right\}$

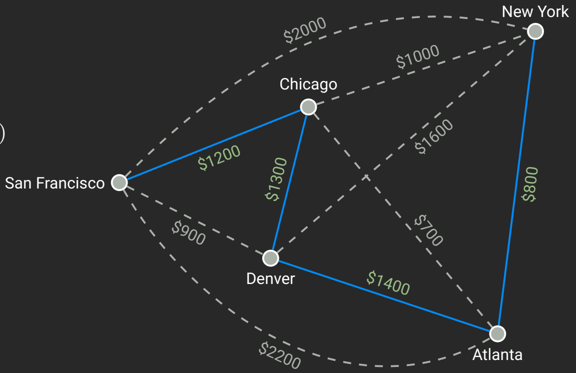
$$\begin{aligned} w(T_1) &= w(\{Chicago, SF\}) + w(\{SF, Denver\}) \\ &\quad + w(\{Chicago, Atlanta\}) + w(\{Chicago, NY\}) \\ &= \$1200 + \$900 + \$700 + \$800 = \$3600 \end{aligned}$$



Some Spanning Trees

- $T_3 = \left\{ \begin{array}{c} \{Chicago, Denver\}, \{Denver, SF\}, \\ \{Denver, Atlanta\}, \{Atlanta, NY\} \end{array} \right\}$

$$\begin{aligned} w(T_1) &= w(\{Chicago, Denver\}) + w(\{Denver, SF\}) \\ &\quad + w(\{Denver, Atlanta\}) + w(\{Atlanta, NY\}) \\ &= \$1300 + \$900 + \$1400 + \$800 = \$4400 \end{aligned}$$



- Problem: Which one is the one with the smallest weight among all possible spanning trees?

ROAR

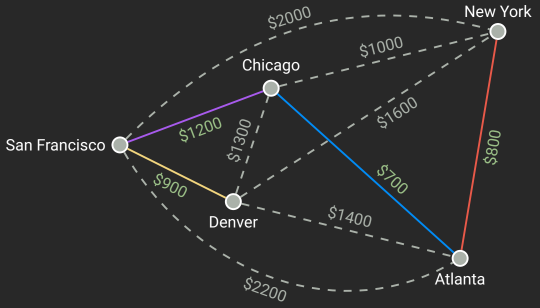
Prim's Algorithm

procedure PRIM(G: weighted connected undirected graph with n vertices)
    $T := $ a minimum-weighted edge
    for $i := 1$ **to** $n - 2$ **do**
        $e :=$ an edge of minimum weight incident to a vertex in T and not forming a simple circuit in T if added to T
        $T := T$ with e added
    {T is a minimum spanning tree of G}

An Example of Prim's Algorithm

Idaho State University | Computer Science

Choice	Edge	Cost
1	{Atlanta, Chicago}	$700
2	{Atlanta, NY}	$800
3	{Chicago, SF}	$900
4	{Denver, SF}	$1200
	Total	$3600



ROAR

Kruskal's Algorithm

procedure KRUSKAL(G: weighted connected undirected graph with n vertices)
$T := $ empty graph
for $i := 1$ **to** $n - 1$ **do**
$e :=$ an edge in G with smallest weight that does not form a simple circuit when added to T
$T := T$ with e added
$\{T$ is a minimum spanning tree of $G\}$

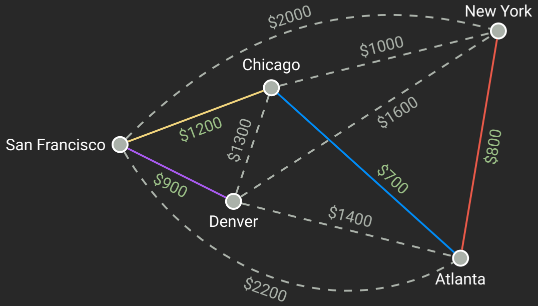
ROAR

An Example of Kruskal's Algorithm

Idaho State University | Computer Science

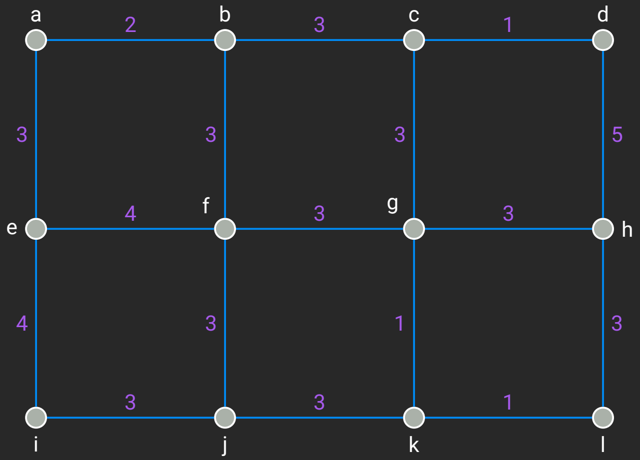
- First, sort all edges based on their weight in ascending order.
  - {Atlanta, Chicago}, {Atlanta, NY}, {Denver, SF}, {Chicago, NY}, {Chicago, SF}, {Chicago, Denver}, {Atlanta, Denver}, {Denver, NY}, {NY, SF}, {Atlanta, SF}
- Examine each edge one by one until a spanning tree is constructed

Choice	Edge	Cost
1	{Atlanta, Chicago}	$700
2	{Atlanta, NY}	$800
3	{Denver, SF}	$900
4	{Chicago, SF}	$1200
	Total	$3600

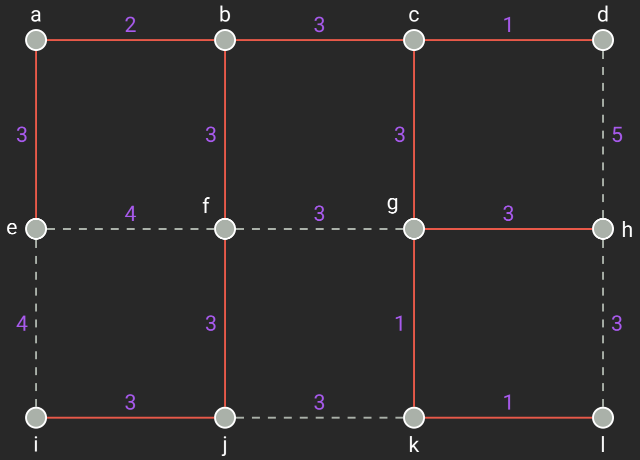


Finding a Spanning Tree with Minimum Weight

Idaho State University | Computer Science



Finding a Spanning Tree with Minimum Weight



Are there any questions?