## Graph Search and Trees

<table>
<thead>
<tr>
<th>Lecture A</th>
<th>Tiefenbruck</th>
<th>MWF 9-9:50am</th>
<th>Center 212</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lecture B</td>
<td>Jones</td>
<td>MWF 2-2:50pm</td>
<td>Center 214</td>
</tr>
<tr>
<td>Lecture C</td>
<td>Tiefenbruck</td>
<td>MWF 11-11:50am</td>
<td>Center 212</td>
</tr>
</tbody>
</table>

[http://cseweb.ucsd.edu/classes/wi16/cse21-abc/](http://cseweb.ucsd.edu/classes/wi16/cse21-abc/)

February 8, 2016
Recall: Tartaglia's Pouring Problem

Large cup: contains 8 ounces, can hold more.
Medium cup: is empty, has 5 ounce capacity.
Small cup: is empty, has 3 ounce capacity

You can pour from one cup to another until the first is empty or the second is full.
Tartaglia's Pouring Problem

Rephrasing the problem:

(1) Is there a path from (8,0,0) to (4,4,0)?
(2) If so, what's the best path?

"Best" means "shortest length"
Tartaglia's Pouring Problem

Rephrasing the problem: using configurations

(1) Is there a path from (8,0,0) to (4,4,0)?
(2) If so, what's the best path?

"Best" means "shortest length"

(l, m, s) means
l ounces in large cup
m ounces in medium cup
s ounces in small cup
Tartaglia's Pouring Problem

How many configurations are possible?

A. Infinitely many 
B. $4 \times 6 \times 9 = 216$
C. 24
D. 16
E. None of the above.

(l, m, s) means
l ounces in large cup
m ounces in medium cup
s ounces in small cup
Tartaglia's Pouring Problem

How many configurations are possible?

Small cup: 0, 1, 2, or 3
Medium cup: 0, 1, 2, 3, 4, or 5
Large cup: 0, 1, 2, 3, 4, 5, 6, 7, or 8

(l, m, s)** means

l ounces in large cup
m ounces in medium cup
s ounces in small cup

**integer values
Tartaglia's Pouring Problem

---

(l, m, s) **

means

l ounces in large cup
m ounces in medium cup
s ounces in small cup

**integer values

---

How many configurations are possible?

Small cup: 0, 1, 2, or 3
Medium cup: 0, 1, 2, 3, 4, or 5
Large cup: 0, 1, 2, 3, 4, 5, 6, 7, or 8

But can't have 3 in small AND 5 in medium AND 8 in large: Total must be 8.
Tartaglia's Pouring Problem

The three columns total 8 in each row. 24 rows.

(l, m, s) **

means

l ounces in large cup
m ounces in medium cup
s ounces in small cup

**integer values

<table>
<thead>
<tr>
<th>l</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>l</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>l</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>l</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>
Tartaglia's Pouring Problem

You can pour from one cup to another until the first is empty or the second is full: so either (or both) of the small cup and the medium cup must always be either full or empty.

Which configurations are actually possible?

(8,0,0)  
Pour into medium cup

(3,5,0)  
Pour into small cup

(5,0,3)
Pour from medium cup

Tartaglia's Pouring Problem

You can pour from one cup to another until the first is empty or the second is full: so either (or both) of the small cup and the medium cup must always be either full or empty.

Which configurations are actually possible?

- (8,0,0)
- (3,5,0)
- (5,0,3)
- (0,5,3)
- (3,2,3)
Tartaglia's Pouring Problem

You can pour from one cup to another until the first is empty or the second is full: so either (or both) of the small cup and the medium cup must always be either full or empty.

Which configurations are actually possible?

\[(8,0,0)\]  \[\rightarrow\]  \[(3,5,0)\]  \[\rightarrow\]  \[(0,5,3)\]  \[\rightarrow\]  \[(5,0,3)\]  \[\rightarrow\]  \[(5,3,0)\]  \[\rightarrow\]  \[(3,2,3)\]
Tartaglia's Pouring Problem

You can pour from one cup to another until the first is empty or the second is full: so either (or both) of the small cup and the medium cup must always be either full or empty.

Which configurations are actually possible?
Tartaglia's Pouring Problem

You can pour from one cup to another until the first is empty or the second is full: so either (or both) of the small cup and the medium cup must always be either full or empty.

<table>
<thead>
<tr>
<th>l</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>
Are all 16 configurations reachable from (8,0,0)?

A. Yes
B. All except for (8,0,0)
C. No
Given a directed graph $G$ and a start vertex $s$, produce a list of all vertices $v$ reachable from $s$ by a directed path in $G$. 

Reachable:

1, 2, 4, 5

Not reachable:

3
Given a directed graph $G$ and a start vertex $s$,

produce a list of all vertices $v$ reachable from $s$ by a directed path in $G$.

At each point in a graph search algorithm, the vertices are partitioned into

$X$: eXplored
$F$: frontier (reached but haven't yet explored)
$U$: unreachable
Graph reachability: HOW

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X= empty, F = \{s\}, U = V – F.
While F is not empty:
   Pick v in F.
   For each neighbor u of v:
      If u is not in X or F, then move u from U to F.
   Move v from F to X.

Return X.
procedure **GraphSearch** (G: directed graph, s: vertex)

Initialize $X = \text{empty}$, $F = \{s\}$, $U = V - F$.

While $F$ is not empty:
   - Pick $v$ in $F$.
   - For each neighbor $u$ of $v$:
     - If $u$ is not in $X$ or $F$, then move $u$ from $U$ to $F$.
   - Move $v$ from $F$ to $X$.

Return $X$. 

**Before any iterations of while loop…**

- $X = \text{empty}$
- $F = \{(8,0,0)\}$
- $U = \text{green nodes}$
procedure GraphSearch \((G: \text{ directed graph}, s: \text{ vertex})\)

Initialize \(X = \text{ empty}, F = \{s\}, U = V − F\).
While \(F\) is not empty:
Pick \(v\) in \(F\).
For each neighbor \(u\) of \(v\):
    If \(u\) is not in \(X\) or \(F\), then move \(u\) from \(U\) to \(F\).
Move \(v\) from \(F\) to \(X\).

Return \(X\).
procedure GraphSearch \((G: \text{directed graph}, s: \text{vertex})\)

Initialize \(X = \text{empty}, \ F = \{s\}, \ U = V - F.\)
While \(F\) is not empty:
Pick \(v\) in \(F.\)
For each neighbor \(u\) of \(v:\)
  If \(u\) is not in \(X\) or \(F,\) then move \(u\) from \(U\) to \(F.\)
Move \(v\) from \(F\) to \(X.\)
Return \(X.\)
Graph reachability: WHY

**procedure GraphSearch** (G: directed graph, s: vertex)

Initialize $X = \text{empty}$, $F = \{s\}$, $U = V - F$.
While $F$ is not empty:
   Pick $v$ in $F$.
   For each neighbor $u$ of $v$:
      If $u$ is not in $X$ or $F$, then move $u$ from $U$ to $F$.
   Move $v$ from $F$ to $X$.

Return $X$. 

*Does this algorithm output the collection of vertices $v$ reachable from $s$ by a directed path in $G$?*
procedure GraphSearch (G: directed graph, s: vertex)

Initialize X= empty, F = \{s\}, U = V – F. 
While F is not empty: 
  Pick v in F. 
  For each neighbor u of v: 
    If u is not in X or F, then move u from U to F. 
  Move v from F to X. 

Return X.

Does this algorithm output the collection of vertices v reachable from s by a directed path in G?

Goal:
1. Every element of output X is reachable from s in G.
2. Every reachable vertex is in X (by end of algorithm).
Graph reachability: WHY

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X= empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F, then move u from U to F.
  Move v from F to X.

Return X.

Claim: After $t^{th}$ iteration through while loop, every element of (current version of) X or F is reachable from s in G.

Proof by induction on t.
**Graph reachability: WHY**

**procedure GraphSearch** (G: directed graph, s: vertex)

Initialize X= empty, F = {s}, U = V – F.
While F is not empty:
    Pick v in F.
    For each neighbor u of v:
        If u is not in X or F, then move u from U to F.
    Move v from F to X.

Return X.

**Claim:** After $t^{th}$ iteration through while loop, every element of (current version of) X or F is reachable from s in G.

**Base case ($t=0$):**
Before any iterations of loop, X is initialized as empty and F is initialized as {s}. WTS s is reachable from s in G.
Graph reachability: WHY

**procedure GraphSearch** (G: directed graph, s: vertex)

Initialize X= empty, F = \{s\}, U = V – F.
While F is not empty:
   Pick v in F.
   For each neighbor u of v:
      If u is not in X or F, then move u from U to F.
   Move v from F to X.

Return X.

**Claim:** After \( t^{th} \) iteration through while loop, every element of (current version of) X or F is reachable from s in G.

**Induction step:** Suppose after \( t^{th} \) iteration, every element of X or F is reachable from s in G.

*What happens in \( t+1^{st} \) iteration?*
procedure GraphSearch (G: directed graph, s: vertex)

Initialize X= empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F, then move u from U to F.
  Move v from F to X.

Claim: After t\textsuperscript{th} iteration through while loop, every element of (current version of) X or F is reachable from s in G.

Induction step: Suppose that after t\textsuperscript{th} iteration, every element of X or F is reachable from s in G.

What happens in t+1\textsuperscript{st} iteration?
Graph reachability: WHY

**procedure GraphSearch** (G: directed graph, s: vertex)

Initialize X = empty, F = \{s\}, U = V – F.

While F is not empty:

Pick v in F.

For each neighbor u of v:

If u is not in X or F, then move u from U to F.

Move v from F to X.

Return X.

---

**Claim**: After \( t \)th iteration through while loop, every element of (current version of) X or F is reachable from s in G.

**Using Claim to prove Goal 1**: After the final iteration, output X, which by claim, only contains vertices that are reachable from s.
procedure GraphSearch (G: directed graph, s: vertex)

Initialize X= empty, F = {s}, U = V – F.
While F is not empty:
   Pick v in F.
   For each neighbor u of v:
      If u is not in X or F, then move u from U to F.
   Move v from F to X.

Return X.

Does this algorithm output the collection of vertices v reachable from s by a directed path in G?

Goal:
1. Every element of output X is reachable from s in G. 😊
2. Every reachable vertex is in X (by end of algorithm).
procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
    Pick v in F.
    For each neighbor u of v:
        If u is not in X or F, then move u from U to F.
    Move v from F to X.

Return X.

WTS Goal 2: Every reachable vertex is in X.

*Hint: assume, towards a contradiction that some vertex is reachable from s but not in X. Look for first vertex on the path between s that is not in X.*
Graph reachability: WHEN

**procedure GraphSearch** (G: directed graph, s: vertex)

Initialize $X = \text{empty}$, $F = \{s\}$, $U = V - F$.
While $F$ is not empty:
  1. Pick $v$ in $F$.
  2. For each neighbor $u$ of $v$:
     - If $u$ is not in $X$ or $F$, then move $u$ from $U$ to $F$.
  3. Move $v$ from $F$ to $X$.

Return $X$.

*How long does it take to pick $v$ in $F$?*
*How long does it take to iterate over neighbors of $v$?*

*Need to know some implementation decisions.*
procedure GraphSearch (G: directed graph, s: vertex)

Initialize X= empty, F = {s}, U = V − F.
While F is not empty:  
Pick v in F.
For each neighbor u of v:  
If u is not in X or F, then move u from U to F.
Move v from F to X.
Return X.

What's an upper bound on the time it takes to do one iteration of the body of the for loop?

A. O( n^2 )  
B. O(n)  
C. O( degree (v) )  
D. O(|F|)  
E. None of the above.
**Graph reachability: WHEN**

**procedure GraphSearch** (G: directed graph, s: vertex)

- Initialize $X = \text{empty}$, $F = \{s\}$, $U = V - F$.
- While $F$ is not empty:
  - Pick $v$ in $F$.
  - For each neighbor $u$ of $v$:
    - If $u$ is not in $X$ or $F$, then move $u$ from $U$ to $F$.
  - Move $v$ from $F$ to $X$.
- Return $X$.

Assume $G$ stored as adjacency list.
Assume have array $\text{Status[]}$:
* length $n$ array
  * each entry either $F$, $X$, $U$

What's an upper bound on the time it takes to go through the whole for loop **for a given $v$**?

A. $O(n^2)$
B. $O(n)$
C. $O(\text{degree}(v))$
D. $O(|F|)$
E. None of the above.
procedure GraphSearch \( (G: \text{directed graph, } s: \text{vertex}) \)

Initialize \( X = \text{empty}, \ F = \{s\}, \ U = V - F \).
While \( F \) is not empty:
  Pick \( v \) in \( F \).
  For each neighbor \( u \) of \( v \):
  If \( u \) is not in \( X \) or \( F \), then move \( u \) from \( U \) to \( F \).
  Move \( v \) from \( F \) to \( X \).

Return \( X \).

Assume \( G \) stored as adjacency list.
Assume have array Status\[
\]
* length \( n \) array
* each entry either \( F, X, U \)

What's an upper bound on the time spent on the for loop throughout the whole algorithm?
A. \( O( n ) \)
B. \( O( |V| ) \)
C. \( O( |E| ) \)
D. \( O( |F| ) \)
E. None of the above.
procedure GraphSearch (G: directed graph, s: vertex)

Initialize X= empty, F = {s}, U = V – F.
While F is not empty:
    Pick v in F.
    For each neighbor u of v:
        If u is not in X or F, then move u from U to F.
    Move v from F to X.

Return X.

Assume G stored as adjacency list.
Assume have array Status[]
* length n array
* each entry either F, X, U

Total time is asymptotically upper bounded by sum of degrees of all vertices
i.e. O (2 |E| )
i.e. O( |E| )
Another Special Type of Graph: Trees
1. Definitions of trees

2. Properties of trees

3. Revisiting uses of trees
A **rooted tree** is a connected directed acyclic graph in which one vertex has been designated the root, which has no incoming edges, and every other vertex has exactly one incoming edge.

(Rooted) Trees: definitions

Rosen p. 747-749
A rooted tree is a connected directed acyclic graph in which one vertex has been designated the root, which has no incoming edges, and every other vertex has exactly one incoming edge.

Special case of DAGs from last class. Note that each vertex in middle has exactly one incoming edge from layer above. Edges are directed away from the root.
Which of the following directed graphs are trees (with root indicated in green)?

- **A.** [Diagram with 2 incoming edges and a cycle, marked as incorrect.]
- **B.** [Disconnected graph, marked as incorrect.]
- **C.** [Correct tree with root indicated in green.]
- **D.** [Graph with 2 incoming edges and a cycle, marked as incorrect.]
(Rooted) Trees: definitions

- Root
- Leaf
- Internal vertices

Rosen p. 747-749
If vertex $v$ is not the root, it has exactly one incoming edge, which is from its parent, $p(v)$. Height of vertex $v$ is given by the recurrence:

$$h(v) = h(p(v)) + 1 \quad \text{if } v \text{ is not the root, and}$$

$$h(r) = 0$$
(Rooted) Trees: definitions

**Height** of vertex $v$: $h(v) = h(p(v)) + 1$ if $v$ is not the root, and $h(r) = 0$

What is the height of the red vertex?
A. 0
B. 1
C. 2
D. 3
E. None of the above.
(Rooted) Trees: definitions

**Height** of vertex v: \( h(v) = h(p(v)) + 1 \) if v is not the root, and \( h(r) = 0 \)

**Height** of tree is maximum height of a vertex in the tree.

*Rosen p. 753*

What is the height of the tree?

A. 0  
B. 1  
C. 2  
D. 3  
E. None of the above.
A binary tree is a rooted tree where every (internal) vertex has no more than 2 children.

How many leaves does a binary tree of height 3 have?

A. 2
B. 3
C. 6
D. 8
E. None of the above.
A binary tree is a rooted tree where every (internal) vertex has no more than 2 children.

How many leaves does a binary tree of height 3 have?
A. 2
B. 3
C. 6
D. 8
E. None of the above.

*See Theorem 5 for proof of upper bound*
A full binary tree is a rooted tree where every internal vertex has exactly 2 children.

Which of the following are full binary trees?

A.

C.

D.
A full binary tree is a rooted tree where every internal vertex has exactly 2 children.

At most how many vertices are there in a full binary tree of height $h$?

A. $\Theta(h)$  
B. $\Theta(2^h)$  
C. $\Theta(h^2)$  
D. $\Theta(\log h)$
A **full** binary tree is a rooted tree where every internal vertex has exactly 2 children.

**Key insight:** number of vertices doubles on each level.

\[ 1 + 2 + 4 + 8 + \ldots + 2^h = \Theta(2^h) \]

If \( n \) is number of vertices:

\[ n = 2^{h+1} - 1 \]

so

\[ h = \log(n+1) - 1 \quad \text{i.e.} \quad \Theta(\log n) \]
Relating height and number of vertices:

\[ \log(n+1) - 1 \leq h \leq ___ \]

This is what we just proved.

How do we prove?

What tree with \( n \) vertices has the greatest possible height?
Relating height and number of vertices:

\[ \log(n+1) - 1 \leq h \leq n-1 \]

This is what we just proved.

How do we prove?

What tree with \( n \) vertices has the greatest possible height?
Trees

1. Definitions of trees
2. Properties of trees
3. Revisiting uses of trees

In data structures:
Binary search trees
Binary Search Trees

• Facilitate binary search (must maintain sorted order of data)
• Dynamic

Implementation

Each vertex is an object with the fields

\[ p = \text{parent} \]
\[ lc = \text{left child} \]
\[ rc = \text{right child} \]
\[ \text{value} \]

When is \( p \) null?

A. If we have an error in our implementation.
B. When the value is 0.
C. When the vertex is a leaf node.
D. When the vertex is the root node.
E. None of the above.
Binary Search Trees

- Facilitate binary search (must maintain sorted order of data)
- Dynamic

Implementation

Each vertex is an object with the fields

\[\begin{align*}
p &= \text{parent} \\
lc &= \text{left child} \\
r_c &= \text{right child} \\
value &= \text{value}
\end{align*}\]

When is \textit{lc} null?

- A. If we have an error in our implementation.
- B. When the value is 0.
- C. When the vertex is a leaf node.
- D. When the vertex is the root node.
- E. None of the above.
Binary Search Trees

• Facilitate binary search (must maintain sorted order of data)
• Dynamic

For each vertex $v$
  • If $x$ is in subtree rooted at $lc(v)$, $value(x) \leq value(v)$.
  • If $x$ is in the subtree rooted at $rc(v)$, $value(x) \geq value(v)$. 
Binary Search Trees

- Facilitate binary search (must **maintain sorted order** of data)
- Dynamic

How would you search for "orange?"
Binary Search Trees

- Facilitate binary search (must maintain sorted order of data)
- Dynamic

To search for target T in a binary search tree.

1. Compare T to value(r) where r is the root.
2. If T = value(r), done 😊.
3. If T < value(r), search recursively starting at lc(r).
4. If T > value(r), search recursively starting at rc(r).
Binary Search Trees

- Facilitate binary search (must maintain sorted order of data)
- Dynamic

To search for target T in a binary search tree.

1. Compare T to value(r) where r is the root.
2. If T = value(r), done 😊.
3. If T < value(r), search recursively starting at lc(r).
4. If T > value(r), search recursively starting at rc(r).

*How long does this take?*
Binary Search Trees

- Facilitate binary search (must maintain sorted order of data)
- Dynamic

To search for target $T$ in a binary search tree.

1. Compare $T$ to $\text{value}(r)$ where $r$ is the root.
2. If $T = \text{value}(r)$, done $\smile$.
3. If $T < \text{value}(r)$, search recursively starting at $\text{lc}(r)$.
4. If $T > \text{value}(r)$, search recursively starting at $\text{rc}(r)$.

**How long does this take?**

Constant time at each level, number of levels is height+1.
Binary Search Trees

- Facilitate binary search (must maintain sorted order of data)
- Dynamic

To search for target $T$ in a binary search tree.

1. Compare $T$ to $value(r)$ where $r$ is the root.
2. If $T = value(r)$, done 😊.
3. If $T < value(r)$, search recursively starting at $lc(r)$.
4. If $T > value(r)$, search recursively starting at $rc(r)$.

*How long does this take?* Time proportional to height!
An unrooted tree is a connected undirected graph with no cycles.
Theorem: An undirected graph is an unrooted tree if and only if it contains all the edges of some rooted tree.

What does this mean?

(1) If we replace all directed edges in a rooted tree with undirected edges, the result will be an unrooted tree.

(2) There is always some way to put directions on the edges of an unrooted tree to make it a rooted tree.
Equivalence between rooted and unrooted trees

**Goal (1):** If we replace all directed edges in a rooted tree with undirected edges, the result will be an unrooted tree.

**What do we need to prove?**

A. The resulting undirected graph will be connected.
B. The resulting undirected graph will be undirected.
C. The resulting undirected graph will not have cycles.
D. All of the above.
Equivalence between rooted and unrooted trees

**Goal (1):** If we replace all directed edges in a rooted tree with undirected edges, the result will be an **unrooted tree**.

**SubGoal (1a):** this resulting graph is connected, i.e. between any two vertices $u$ and $v$ there is a path in the graph.

*Idea:* To find path between purple and orange, follow parents of purple all the way to root, then follow its children down to orange.
**Equivalence between rooted and unrooted trees**

**Goal (1):** If we replace all directed edges in a rooted tree with undirected edges, the result will be an **unrooted tree**.

**SubGoal (1b):** this resulting graph has no cycles.

**Idea:** Towards a contradiction, assume there is a cycle and consider the simplest cycle (with no repeated vertices).

Start at vertex at highest level in the cycle. Next step must go to a child node, etc. Can never go up to higher level again because vertices in rooted tree only have one incoming edge.
Goal (2): There is always some way to put directions on the edges of an unrooted tree to make it a rooted tree.

Idea: finding right directions for edges will be similar to finding topological sort last class.
Equivalence between rooted and unrooted trees

**Goal (2):** There is always some way to put directions on the edges of an unrooted tree to make it a rooted tree.

**SubGoal (2a):** Any unrooted tree with at least two vertices has a vertex of degree exactly 1.

**Proof:** Towards a contradiction, assume that all vertices have degree 0 or >=2. Since a tree is connected, eliminate the case of degree-0 vertices.

**Goal:** construct a cycle to arrive at a contradiction.

Start at any vertex $u_0$.
Pick $u_{i+1}$ so that it is adjacent to $u_i$ but is **not** $u_{i-1}$. **Why?**

Get $u_0, u_1, \ldots, u_n$. By Pigeonhole Principle, must repeat. **Cycle!**
Equivalence between rooted and unrooted trees

**Goal (2):** There is always some way to put directions on the edges of an unrooted tree to make it a rooted tree.

**SubGoal (2b):** If $T$ is unrooted tree and $v$ has degree 1 in $T$, then $T\{-v\}$ is unrooted tree.

**Proof:** To check that $T\{-v\}$ is unrooted tree,

* confirm $T\{-v\}$ is connected and

* $T\{-v\}$ does not have a cycle.
Equivalence between rooted and unrooted trees

**Goal (2):** There is always some way to put directions on the edges of an unrooted tree to make it a rooted tree.

**SubGoal (2b):** If $T$ is an unrooted tree and $v$ has degree 1 in $T$, then $T\{-v\}$ is an unrooted tree.

**Proof:** To check that $T\{-v\}$ is an unrooted tree,

* confirm $T\{-v\}$ is connected and

* $T\{-v\}$ does not have a cycle.
**Goal (2):** There is always some way to put directions on the edges of an unrooted tree to make it a rooted tree.

Using the subgoals to achieve the goal:

Root(\(T\): unrooted tree with \(n\) nodes)
1. If \(n=1\), let the only vertex \(v\) be the root, set \(h(v):=0\), and return.
2. Find a vertex \(v\) of degree 1 in \(T\), and let \(u\) be its only neighbor.
3. Root(\(T\-{v}\)).
4. Set \(p(v):=u\) and \(h(v):=h(u)+1\).
Reminders

HW 5 due **Wednesday 11:59pm** via Gradescope.