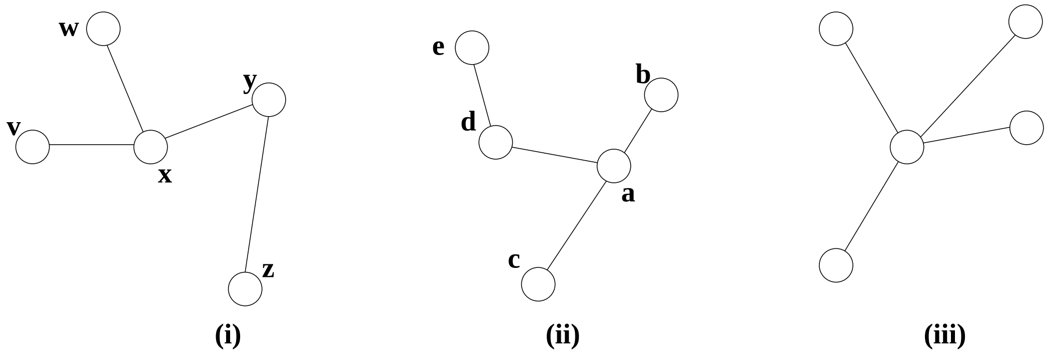
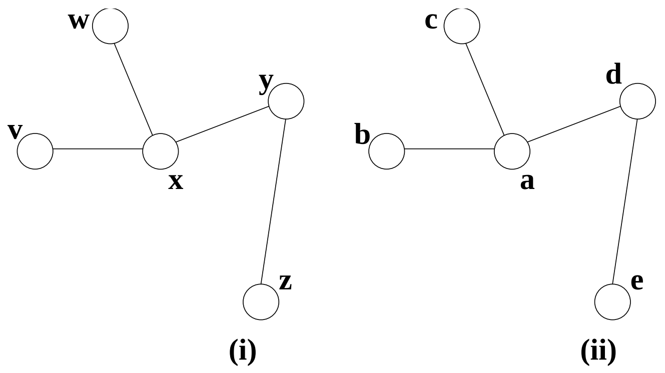
Assignment 5: Tree Isomorphism

Important Note: You do not need to completely understand all of the discussion written here to successfully complete the assignment. The discussion is meant to give you an idea of what you are doing and hopefully clear up any ambiguities in the algorithm descriptions. Send comments or questions to: msuder@cs.mcgill.ca.

Informally, we say that two graphs are *isomorphic* if one can be transformed into the other simply by renaming nodes. For example, in the illustration below, the first two trees are isomorphic but the third is not isomorphic to either of the other two trees.



In the illustration below, we draw trees (i) and (ii) so that they look exactly the same.



Formally, we say that two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there is a 1-1 mapping $f : V_1 \to V_2$ such that $\{v, w\}$ is an edge in E_1 if and only if $\{f(v), f(w)\}$ is an edge in E_2. We call such a mapping an *isomorphism*.

For trees (i) and (ii) in the example above, there exists a 1-1 mapping f, with $f(v) = b$, $f(w) = c$, $f(x) = a$, $f(y) = d$ and $f(z) = e$, such that:

- $\{v, x\}$ is an edge in the first tree and $\{f(v), f(x)\} = \{b, a\}$ is an edge in the second tree;
- $\{w, x\}$ is an edge in the first tree and $\{f(w), f(x)\} = \{c, a\}$ is an edge in the second tree;
- $\{x, y\}$ is an edge in the first tree and $\{f(x), f(y)\} = \{a, d\}$ is an edge in the second tree; and
- $\{y, z\}$ is an edge in the first tree and $\{f(y), f(z)\} = \{d, e\}$ is an edge in the second tree.

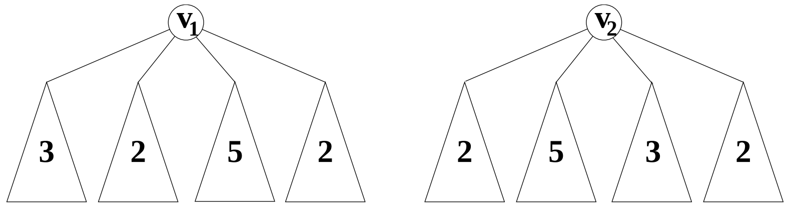
Another possible 1-1 mapping is g with $g(v) = c$, $g(w) = b$, $g(x) = a$, $g(y) = d$ and $g(z) = e$.

For this assignment, you will implement an algorithm that determines whether or not two trees are isomorphic and, if so, returns an isomorphism between the two trees. The algorithm runs in $\Theta(n \log(n))$ time where n is the number of vertices in each tree.

1 Rooted Tree Isomorphism

Before tackling the general problem for trees, we first consider a slightly simpler problem: determining whether or two *rooted* trees are isomorphic. In other words, given two trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ whose roots are v_1 and v_2, respectively, we want to know if there is an isomorphism $f : V_1 \to V_2$ between T_1 and T_2 that maps the roots onto each other, that is, with $f(v_1) = v_2$.

If such an isomorphism exists then for each child v of v_1 there is a child w of v_2 such that $f(v) = w$ and the subtree rooted at v is isomorphic to w. In fact, this implies a recursive algorithm for solving the problem: the algorithm when applied to two rooted trees recursively determines which subtrees rooted at children of the tree roots are isomorphic. When the recursive calls terminate it is then easy to determine whether or not the trees are isomorphic. For example, in the illustration below isomorphic subtrees are assigned the same integer labels so it is easy to see that the two rooted trees are isomorphic— both trees have two subtrees labelled 2, one labelled 3 and one more labelled 5.



Unfortunately, this algorithm is very inefficient because it must determine whether or not *each pair* of subtrees is isomorphic.

However, generalizing the observations above, we see that for each vertex v at depth i in T_1 there is a vertex w at the same depth i in T_2 such that:

1. $f(v) = w$ and
2. the subtree rooted at v is isomorphic to w.

From this generalization we can now obtain an efficient algorithm. Rather than starting with the tree root, this algorithm begins with vertices at depth h where h is the height of each tree. The algorithm then works its way up the tree labelling the subtrees rooted at each level so that if two subtrees rooted at the same level are given the same label then they are isomorphic. We store the label of each subtree at its root.

The subtrees rooted at depth h each consists of exactly one vertex, a leaf; therefore, these subtrees are all isomorphic to one another. Thus, we assign each of these subtrees the same integer label 0.

Moving up to subtrees rooted at depth $h - 1$, we see that these subtrees consist of a root and zero or more children at depth h. Two of these subtrees are isomorphic iff their roots each have the same number of children. Therefore, we label each subtree with the number of children the subtree root has. Thus, subtrees whose roots are leaves are labelled 0, subtrees whose roots have one child are labelled 1, subtrees whose roots have two children are labelled 2, and so on.

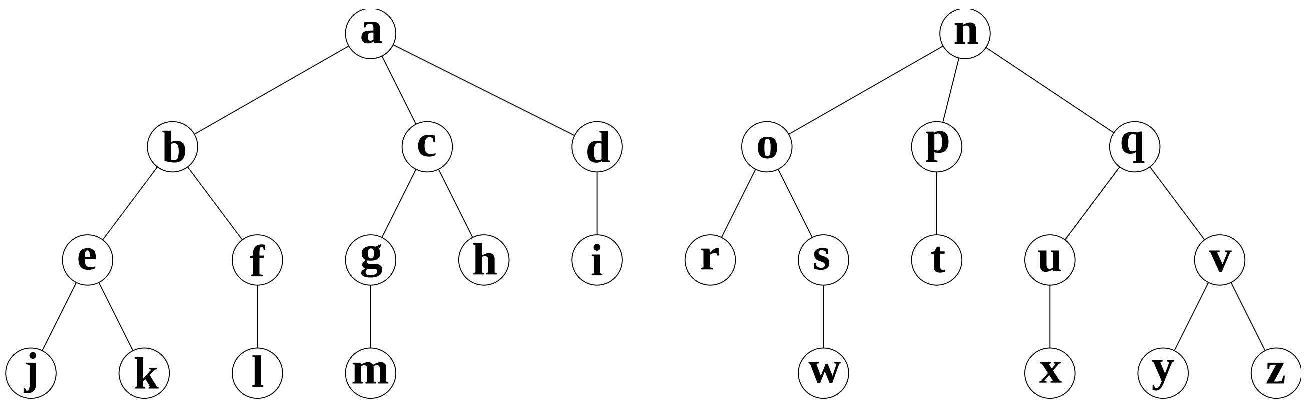
Once we reach depth $0 \leq i \leq h - 1$, all subtrees rooted at depths $i + 1,\ i + 2,\ \ldots,\ h$ have been labelled so that isomorphic subtrees at the same depth have the same label. Each subtree rooted at depth i consists of a root and zero or more labelled subtrees at depth $i + 1$; therefore, two subtrees are isomorphic iff their depth $i + 1$ subtrees have the same labels. For example, if subtree A contains subtrees labelled $3,\ 2,\ 5,\ 2$ and another subtree B contains subtrees labelled

2, 5, 3, 2 then they are isomorphic. On the other hand, if a subtree C contains subtrees labelled 2, 5, 3 then A is not isomorphic to C because C has only one subtree labelled 2. To make comparing sets of labels efficient, we store the labels as a sorted sequence in the subtree root. For example, the roots of A and B would store the sequence (2, 2, 3, 5) and the root of C would store (2, 3, 5).

The algorithm terminates after labelling the trees at depth 0. The two trees are isomorphic if and only if the roots are assigned the same labels.

If the two trees are isomorphic, then the algorithm constructs an isomorphism first by pairing up the tree roots and then by recursively constructing an isomorphism for corresponding isomorphic subtrees rooted at their children. We can make this part of the algorithm efficient by having each vertex store a sequence of its children sorted by their subtree vertex labels. For example, if the root v of subtree A stores the sorted sequence (2, 2, 3, 5) then v has four children, one corresponding to each of the four labels. If w and x are the children corresponding to the label 2, y the child corresponding to 3 and z the child corresponding to 5 then v would contain the additional sequence $(w,\ x,\ y,\ z)$. Then, if the root v' of subtree B stores the same sequence (2, 2, 3, 5) and the additional sequence of ordered children $(w',\ x',\ y',\ z')$ than one isomorphism between A and B contains the pairs (v,v'), (w,w'), (x,x'), (y,y') and (z,z').

Before giving the algorithm in detail, we apply it to the following trees:



(i) (ii)

We let $L[0]$, $L[1]$, $L[2]$ and $L[3]$ be sequences storing the vertices of each tree at depths 0, 1, 2 and 3, respectively. Thus:

$$\begin{aligned} L[0] &= (a,\ n)\\ L[1] &= (b,\ c,\ d,\ o,\ p,\ q)\\ L[2] &= (e,\ f,\ g,\ h,\ i,\ r,\ s,\ t,\ u,\ v)\\ L[3] &= (j,\ k,\ l,\ m,\ w,\ x,\ y,\ z)\end{aligned}$$

We begin by assigning each vertex in $L[3]$ an empty sequence of subtree labels and sorted children and an integer label 0:

$L[3]$	j	k	l	m	w	x	y	z
subtree labels	()	()	()	()	()	()	()	()
label	0	0	0	0	0	0	0	0
sorted children	()	()	()	()	()	()	()	()

We then move to the next depth 2:

$L[2]$	e	f	g	h	i	r	s	t	u	v
subtree labels	$(0,0)$	(0)	(0)	$()$	$()$	$()$	(0)	$()$	(0)	$(0,0)$
label	2	1	1	0	0	0	1	0	1	2
sorted children	(j,k)	(l)	(m)	$()$	$()$	$()$	(w)	$()$	(x)	(y,z)

Then to the next depth 1:

$L[1]$	b	c	d	o	p	q
subtree labels	$(1,2)$	$(0,1)$	(0)	$(0,1)$	(0)	$(1,2)$
label	2	1	0	1	0	2
sorted children	(f,e)	(h,g)	(i)	(r,s)	(t)	(u,v)

Finally, we finish by labelling the roots:

$L[0]$	a	n
subtree labels	$(0,1,2)$	$(0,1,2)$
label	0	0
sorted children	(d,c,b)	(p,o,q)

The labels assigned to each root is 0 so the two trees are isomorphic. The resulting isomorphism is:

$$\{(a,n),\ (b,q),\ (e,v),\ (j,y),\ (k,z),\ (f,u),\ (l,x),\ (c,o),\ (g,s),\ (m,w),\ (h,r),\ (d,p),\ (i,t)\}.$$

The algorithm below efficiently assigns integer labels to each vertex at depth i by first sorting $L[i]$ according to the sorted subtree labels sequence stored at each vertex in $L[i]$. After $L[i]$ is sorted, roots of isomorphic subtrees in $L[i]$ appear next to one another. For example, sorting $L[1]$ in the example above would change it from $(b,\ c,\ d,\ o,\ p,\ q)$ to $(d,\ p,\ c,\ o,\ b, q)$ because

$$(0) \leq (0) \leq (0,1) \leq (0,1) \leq (1,2) \leq (1,2).$$

Notice that we're sorting lexicographically just like we sort words in a dictionary.

Algorithm RootedIsomorphic(T_1, T_2)

Input: T_1 and T_2 are rooted trees.
Output: If T_1 and T_2 are rooted isomorphic trees then return an isomorphism from T_1 to T_2; otherwise, return an empty mapping.

```
$h \leftarrow$ max($T_1$.height(), $T_2$.height())
$L \leftarrow$ an array length $h+1$
Use BFS to add all vertices of depth $i$ in $T_1$ and $T_2$ to $L[i]$, $0 \leq i \leq h$
**for each** vertex $v$ in $T_1$ or $T_2$ **do**
    $v$.label $\leftarrow$ 0
    $v$.orderedlabels $\leftarrow$ empty sequence
    $v$.orderedchildren $\leftarrow$ empty sequence
**for** $i \leftarrow h-1,\ h-2,\ \ldots,\ 0$ **do**
    **for** $j \leftarrow 0,\ 1,\ \ldots,\ L[i+1]$.size() $- 1$ **do**
        $v \leftarrow$ vertex $j$ in $L[i+1]$
        $v$.parent().orderedlabels.insertLast($v$.label)
```

```
            $v$.parent().orderedchildren.insertLast($v$)
        Sort the vertices of $L[i]$ in ascending order of their orderedlabels sequences.
        **for each** vertex $v$ in $L[i]$ **do**
            $v$.label ← $k-1$ where $v$.orderedlabels is the $k^{\text{th}}$ largest in $L[i]$
    $M$ ← empty sequence
    **if** $T_1$.root().label = $T_2$.root().label **then**
        GenerateMapping($T_1$.root(), $T_2$.root(), $M$)
    **return** M
```

Algorithm GenerateMapping(v, w, M)

Input: Vertices v and w, roots of isomorphic subtrees, and sequence M.
Output: Adds an isomorphism to M between the subtree rooted at v and the subtree rooted at w.

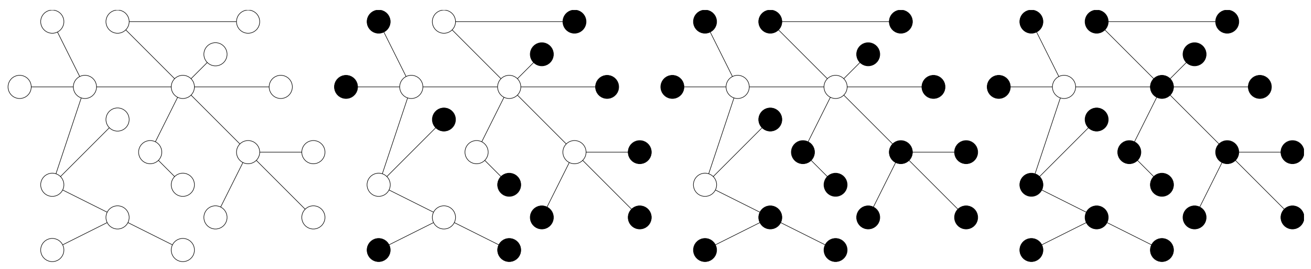
```
    $M$.insert(pair $v$,$w$)
    **for** $i$ ← 0, 1, ..., $v$.orderedchildren.size()−1 **do**
        $x$ ← vertex $i$ in $v$.orderedchildren
        $y$ ← vertex $i$ in $w$.orderedchildren
        GenerateMapping($x$, $y$, $M$)
```

2 General Tree Isomorphism

Surprisingly, most of the difficult work has already been done for solving the general tree isomorphism problem. All we need to do is to root each tree so that if the two trees are isomorphic then there is an isomorphism that maps the roots onto each other. After we have rooted the trees, we simply use the algorithm for rooted trees to find an isomorphism if one exists. One way to do this is to choose the roots from the centers of each tree.

2.1 Center of a Tree

The **center** of a tree is the set of vertices furthest from a leaf. One way to find the center is to repeatedly remove all leaves until either a single vertex or two vertices connected by an edge are left. The sequence of trees drawn below illustrates this algorithm. The black circles represent vertices that have been removed from the tree. After removing all leaves three times, we finally discover the center of the tree, represented by the only remaining white circle.



The algorithm below implements this algorithm but without actually destroying the tree. It keeps track of the degree of each vertex as it “removes” vertices from the tree in a variable called `centerdegree`. A vertex becomes a leaf when its `centerdegree` variable equals 1. At that point, the algorithm “removes” the vertex from the tree by decrementing the `centerdegree` variable of each neighbor.

ALGORITHM FindCenter(T)

Input: Rooted tree T.
Output: Returns the set of vertices at the center of T. The set has size either one or two.

```
$r \leftarrow$ the number of vertices in $T$
$S \leftarrow$ empty set
**for each** vertex $v$ in $T$ **do**
    $v$.centerdegree $\leftarrow$ $v$.degree()
    **if** $v$.degree() $\leq 1$ **then**
        $S$.insert($v$)
        $r \leftarrow r - 1$
**while** $r > 0$ **do**
    $T \leftarrow$ empty set
    **for each** $v$ in $S$ **do**
        **for each** neighbor $w$ of $v$ **do**
            $w$.centerdegree $\leftarrow$ $w$.centerdegree $-1$
            **if** $w$.centerdegree $= 1$ **then**
                $T$.insert($w$)
                $r \leftarrow r - 1$
    $S \leftarrow T$
**return** $S$
```

2.2 Rooting a Tree

The center of a tree consists of either one or two vertices. If the center of each tree contains only one vertex then we simply root each tree at its center. Otherwise, if each center contains two vertices, say $\{u, v\}$ for the first tree and $\{x, y\}$ for the second and the two trees are isomorphic then any isomorphism will map u to either x or y. Thus, we root the first tree at u and the second at x. If the rooted isomorphism algorithm is successful then we return the resulting isomorphism; otherwise, we root the second tree at y and run the algorithm again.

Once we have chosen a root we must root the tree. Here’s how:

ALGORITHM ReRoot(T, v)

Input: T is a rooted tree and v a vertex in T.
Output: T rooted at v.

```
$r \leftarrow v$
$p \leftarrow v$.parent()
**while** $p \neq$ null **do**
    remove $v$ from $p$'s list of children
    add $p$ to $v$'s list of children
    $g \leftarrow p$.parent()
    make $p$'s parent $v$
    $v \leftarrow p$
    $p \leftarrow g$
  $T$.setRoot($r$)
```

2.3 Final Algorithm

ALGORITHM Isomorphic(T_1, T_2)

Input: Rooted trees T_1 and T_2.
Output: If T_1 is isomorphic to T_2 then returns an isomorphism from T_1 to T_2; otherwise an empty mapping.

```
$C_1 \leftarrow$ FindCenter($T_1$)
ReRoot($T_1$, first vertex in $C_1$)
$C_2 \leftarrow$ FindCenter($T_2$)
ReRoot($T_2$, first vertex in $C_2$)
$M \leftarrow$ RootedIsomorphic($T_1$, $T_2$)
if $M$ is empty and $C_2$.size() $> 1$ then
  ReRoot($T_2$, second vertex in $C_2$)
  M $\leftarrow$ RootedIsomorphic($T_1$, $T_2$)
return $M$
```