Chapter 7

Trees

Eng. Eman R. Habib

November, 2013
**Tree**

A tree $T$ is a set of nodes storing elements such that the nodes have a parent-child relationship, that satisfies the following properties:

- If $T$ is nonempty, it has a special node, called the root of $T$, that has no parent.
- Each node $v$ of $T$ different from the root has a unique parent node $w$; every node. With parent $w$ is a child of $w$.

Ordered Trees:

A tree is ordered if there is a linear ordering defined for the children of each node; that is, we can identify the children of a node as being the first, second, third, and so on. Such an ordering is usually visualized by arranging siblings left to right, according to their ordering. Ordered trees typically indicate the linear order among siblings by listing them in the correct order.

**Methods:**

- `element()`: Return the object stored at this position.
- `root()`: Return the tree's root; an error occurs if the tree is empty.
- `parent(v)`: Return the parent of $v$; an error occurs if $v$ is the root.
- `children(v)`: Return an iterable collection containing the children of node $v$.
- `isInternal(v)`: Test whether node $v$ is internal.
- **isExternal(v)**: Test whether node $v$ is external.
- **isRoot(v)**: Test whether node $v$ is the root.
- **Size()**: Return the number of nodes in the tree.
- **isLeaf()**: Test whether the tree has any nodes or not.
- **iterator()**: Return an iterator of all the elements stored at nodes of the tree.
- **positions()**: Return an iterable collection of all the nodes of the tree.
- **replace(v, e)**: Replace with $e$ and return the element stored at node $v$.

### Linked Structure for General Trees:

- **Depth**: Let $v$ be a node of a tree $T$. The depth of $v$ is the number of ancestors of $v$. For example, in the tree in Figure 1, the node storing ‘K’ has depth 3. The depth of the root of $T$ is $0$. The depth of a node $v$ can also be recursively defined as follows:
  - If $v$ is the root, then the depth of $v$ is 0
  - Otherwise, the depth of $v$ is one plus the depth of the parent of $v$.

```java
public static <E> int depth (Tree<E> T, Position<E> v) {
    if (T.isRoot(v))
        return 0;
    else
        return 1 + depth(T, T.parent(v));
}
```

The running time of algorithm depth $(T, v)$ is $O(dv)$, where $dv$ denotes the depth of the node $v$ in the tree $T$.
Height:
The height of a node v in a tree T is also defined recursively:

- If v is an external node, then the height of v is 0
- Otherwise, the height of v is one plus the maximum height of a child of v.

The height of a nonempty tree T is the height of the root of T.
For example, the tree in Figure 1 has height 3.
The height of a nonempty tree T is equal to the maximum depth of an external node of T.

```java
public static <E> int height1 (Tree<E> T) {
    int h = 0;
    for (Position<E> v : T.positions()) {
        if (T.isExternal(v))
            h = Math.max(h, depth(T, v));
    }
    return h;
}
```

The running time of height1 is given by \(O(n + \sum_{v \in Ex} (1 + dv))\), where n is the number of nodes of T, dv is the depth of node v, and Ex is the set of external nodes of T. In the worst case, the sum \(\sum_{v \in Ex} (1 + dv)\) is proportional to \(n^2\). Thus, algorithm height1 runs in \(O(n^2)\) time.

```java
public static <E> int height2(Tree<E> T, Position<E> v) {
    if (T.isExternal(v)) return 0;
    int h = 0;
    for (Position<E> w : T.children(v))
        h = Math.max(h, height2(T, w));
    return 1 + h;
}
```

The running time is \(\sum_v C_v = n - 1\), is \(O(n)\).

Preorder Traversal:

Figure 3: Preorder Traversal

```
Preorder Traversal: print(StringPreorder)
```

_root_ child 1 then child 2 then child 3 then root

Figure 3: Preorder Traversal
public static <E> String toStringPreorder(Tree<E> T, Position<E> v) {
    String s = v.element().toString(); // the main "visit" action
    for (Position<E> w : T.children(v))
        s += ', ' + toStringPreorder(T, w);
    return s;
}

public static <E> String parentheticRepresentation(Tree<E> T, Position<E> v) {
    String s = v.element().toString(); // main visit action
    if (T.isInternal(v)) {
        Boolean firstTime = true;
        for (Position<E> w : T.children(v))
            if (firstTime) {
                s += "(" + parentheticRepresentation(T, w); // the first child
                firstTime = false;
            }
            else s += ', ' + parentheticRepresentation(T, w); // subsequent child
        s += ")"; // close parenthesis
    }
    return s;
}

Postorder Traversal:

Postorder Traversal
public static <E> String toStringPostorder(Tree<E> T, Position<E> v) {
    String s = "";
    for (Position<E> w : T.children(v))
        s += toStringPostorder(T, w) + ",";
    s += v.element(); // main visit action
    return s;
}

Postorder Traversal

public static <E> int diskSpace(Tree<E> T, Position<E> v) {
    int sum = 0;
    for (Position<E> w : T.children(v))
        // add the recursively computed space used by the children of v
        sum += diskSpace(T, w);
    sum += size(v);
    return sum;
}

Binary Trees:
A binary tree is an ordered tree with the following properties:
1. Every node has at most two children.
2. Each child node is labeled as being either a left child or a right child.
3. A left child precedes a right child in the ordering of children of a node.
Methods:
- \texttt{left(v)}: Return the left child of \( v \); an error condition occurs if \( v \) has no left child.
- \texttt{right(v)}: Return the right child of \( v \); an error condition occurs if \( v \) has no right child.
- \texttt{hasLeft(v)}: Test whether \( v \) has a left child.
- \texttt{hasRight(v)}: Test whether \( v \) has a right child.
- \texttt{addRoot(e)}: Create and return a new node \( r \) storing element \( e \) and make \( r \) the root of the tree; an error occurs if the tree is not empty.
- \texttt{insertLeft(v,e)}: Create and return a new node \( w \) storing element \( e \), add \( w \) as the the left child of \( v \) and return \( w \); an error occurs if \( v \) already has a left child.
- \texttt{insertRight(v,e)}: Create and return a new node \( z \) storing element \( e \), add \( z \) as the the right child of \( v \) and return \( z \); an error occurs if \( v \) already has a right child.
- \texttt{remove(v)}: Remove node \( v \), replace it with its child, if any, and return the element stored at \( v \); an error occurs if \( v \) has two children.
- \texttt{attach(v, T1, T2)}: Attach \( T1 \) and \( T2 \), respectively, as the left and right sub-trees of the external node \( v \); an error condition occurs if \( v \) is not external.

A Linked Structure for Binary Trees:
An Array-List Representation of a Binary Tree:
For every node v of T, let p(v) be the integer defined as follows:
- If v is the root of T, then p(v) = 1.
- If v is the left child of node u, then p(v) = 2p(u).
- If v is the right child of node u, then p(v) = 2p(u) + 1.

Inorder Traversal:

Algorithm inorder(T, v):
  if v has a left child u in T then
    inorder(T, u)  \{recursively traverse left subtree\}
  perform the "visit" action for node v
  if v has a right child w in r then
    inorder(T, w)  \{recursively traverse right subtree\}

Euler Tour Traversal:
- "On the left" (before the Euler tour of v's left subtree)
- "From below" (between the Euler tours of v's two subtrees)
- "On the right" (after the Euler tour of v's right subtree).
Algorithm printExpression, accomplishes this task by performing the following actions in an Euler tour:
"On the left" action: if the node is internal, print "(
"From below" action: print the value or operator stored at the node
"On the right" action: if the node is internal, print ")".

Algorithm printExpression(T, v):
    if T.isInternal(v) then
        print "("
    if T.hasLeft(v) then
        printExpression(T, T.left(v))
    if T.isInternal(v) then
        print the operator stored at v
    else
        print the value stored at v
    if T.hasRight(v) then
        printExpression(T, T.right(v))
    if T.isInternal(v) then
        print ")"
Exercises:

C-7.3:

Design algorithms for the following operations for a binary tree $T$:

- $\text{preorderNext}(v)$: return the node visited after node $v$ in a preorder traversal of $T$.
- $\text{inorderNext}(v)$: return the node visited after node $v$ in an inorder traversal of $T$.
- $\text{postorderNext}(v)$: return the node visited after node $v$ in a postorder traversal of $T$.

What are the worst-case running times of your algorithms?

$\text{preorderNext}(v)$:

Algorithm $\text{preorderNext}(\text{Node } v)$:

```
if $v$ isInternal() then
    return $v$’s left child
else
    Node $p = \text{parent of } v$
    if $v$ is left child of $p$ then
        return right child of $p$
    else
        while $v$ is not left child of $p$
            $v = p$
            $p = p$.parent
        return right child of $p$
```

$\text{inorderNext}(v)$:

Algorithm $\text{inorderNext}(\text{Node } v)$:

```
if $v$ isInternal() then
    $v = \text{right child of } v$
    while $v$ is not external
        $v = \text{left child of } v$
    return $v$
else
    Node $p = \text{parent of } v$
    if $v$ is left child of $p$ then
        return $p$
    else
        while $v$ is not left child of $p$
            $v = p$
            $p = p$.parent
        return $p$
```
postorderNext(v):

Algorithm postorderNext(Node v):
    if v isInternal() then
        p = parent of v
        if v = right child of p then
            return p
        else
            v = right child of p
            while v is not external do
                v = left child of v
            return v
    else
        p = parent of v
        if v is left child of p then
            return right child of p
        else
            return p

The worst case running times for these algorithms are all \( O(\log n) \) where \( n \) is the height of the tree \( T \).

C-7.8:

Describe how to clone a proper binary tree using the attach method instead of methods insertLeft and insertRight.

Clone(BinaryTree T, E e1, E e2){
    BinaryTree T1 = new BinaryTree();
    T1.addRoot(e1);
    BinaryTree T2 = new BinaryTree();
    T1.addRoot(e2);
    T.attach(T.root(), T1, T2);
}
C-7.21:

Let T be a tree with n nodes; Define the lowest common ancestor (LCA) between two nodes v and w as the lowest node in T that has both v and w as descendents (where we allow a node to be a descendent of itself). Given two nodes v and w, describe an efficient algorithm for finding the LCA of v and w. What is the running time of your algorithm?

Algorithm LCA(Node v, Node w):
    int vdpth ← v.depth
    int wdpth ← w.depth
    while vdpth > wdpth do
        v ← v.parent
    while wdpth > vdpth do
        w ← w.parent
    while v ≠ w do
        v ← v.parent
        w ← w.parent
    return v

C-7.29:

Describe, in pseudo-code, a nonrecursive method for performing an inorder traversal of a binary tree in linear time.

Algorithm inorder(Tree T):
    Stack S ← new Stack()
    Node v ← T.root()
    push v
    while S is not empty do
        while v is internal do
            v ← v.left
            push v
        while S is not empty do
            pop v
            visit v
            if v is internal then
                v ← v.right
                push v
        while v is internal do
            v ← v.left
            push v
C-7.30

Let T be a binary tree with n nodes (T may be realized with an array list or a linked structure). Give a linear-time algorithm that uses the methods of the BinaryTree interface to traverse the nodes of T by increasing values of the level numbering function p given in Section 7.3.5. This traversal is known as the level order traversal.

Algorithm levelOrderTraversal(BinaryTree T):
   Queue Q = new Queue()
   Q.enqueue(T.root())
   while Q is not empty do
      Node v ← Q.dequeue()
      if T.isInternal(v) then
         Q.enqueue(v.leftchild)
         Q.enqueue(v.rightchild)