Binary Trees (1)

Outline and Required Reading:

- Binary Trees (§ 6.3)
- Data Structures for Representing Trees (§ 6.4)
Binary Tree

- tree with the following properties
  - each internal node has two children
  - children of internal nodes form ordered pairs: left node – 1st, right node – 2nd

Application – representation of arithmetic expression, decision process, …
Binary Tree (cont.)

Example 1  [ binary tree for decision process ]

Are you a mammal?

Yes

Are you bigger than cat?

Yes

Kangaroo

No

Mouse

No

Do you live under water?

Yes

Trout

No

Can you fly?

Yes

Robin

No

Snake
Properties of Binary Trees

Binary Tree Notation

- \( n \) – number of nodes
- \( e \) – number of external nodes
- \( i \) – number of internal nodes
- \( h \) – height of the tree
- \( \text{level} \) – set of nodes with the same depth

Property 1.1  Level \( d \) has at most \( 2^d \) nodes.

Proof  Let us annotate max number of nodes at level \( d \) with \( \text{mn}(d) \).

Clearly, \( \text{mn}(0) = 1 \), and
\[
\text{mn}(d) = 2 \times \text{mn}(d-1) \quad \text{for} \quad \forall d \geq 0 .
\]

Hence,
\[
\text{mn}(d) = 2 \times \text{mn}(d-1) = 2 \times 2 \times \text{mn}(d-2) = 2 \times [2 \times \ldots \times 2 \times \text{mn}(0)] = 2^d
\]
Property 1.2  A full binary tree of height $h$ has $(2^{h+1} - 1)$ nodes.

**Proof**

$$n = mn(0) + mn(1) + .. + mn(d) =$$

$$= 2^0 + 2^1 + 2^2 + \ldots + 2^h =$$

$$= \frac{1 - 2^{h+1}}{1 - 2} = 2^{h+1} - 1$$
Properties of Binary Trees (cont.)

Induction as a Proof Technique

Assume we want to verify the correctness of a statement \( P(n) \).

1. First, prove that \( P(n) \) holds for \( n=1 \) (2, 3);
2. Assume it holds for an arbitrary \( n \) and try to prove it holds for \( (n+1) \)

Property 2  In a binary tree, the number of external nodes is 1 more than the number of internal nodes, i.e. \( e = i + 1 \).

Proof

Clearly true for one node.

Clearly true for tree nodes.

Assume true for trees with up to \( n \) nodes.
Let T be a tree with $n+1$ nodes (top diagram).

1. Choose a leaf and its parent (which, of course, is internal). For example, the leaf a and parent p.

2. Remove the leaf and its parent (middle diagram).

3. Splice the tree back without the two nodes (bottom diagram).

4. Since S has $n-1$ nodes, S satisfies initial assumption.

5. T is just S + one leaf + one internal so it also satisfies the assumption.
Approach (2):

Assume true for a tree with \( n \) nodes \( (e = i + 1) \). Now, we want to add new external nodes:

1. Cannot add only one external – that would violate the property of proper binary tree. Hence, it cannot be: \( e = i + 2 \)

2. Add two externals. In this case, one old external becomes internal, so we have:

\[
e_{\text{new}} = (e-1)+2 = e+1 = i+1+1 = i+2
\]

\[
i_{\text{new}} = i+1
\]

Hence, \( e_{\text{new}} = i_{\text{new}} + 1 \)
**Properties of Binary Trees (cont.)**

**Property 3**  
The number of external nodes \( (e) \) satisfies:  
\[(h+1) \leq e \leq 2^h.\]

**Proof**

- At every level (except last) there is only one internal node.
- Full binary tree.

**Property 4**  
The number of internal nodes \( (i) \) satisfies:  
\[h \leq i \leq 2^{h-1}.\]

**Proof**

- Based on Property 3:  
  \[h+1 \leq e \leq 2^h\]
- Based on Property 2:  
  \[h+1 \leq i+1 \leq 2^h\]
  \[h \leq i \leq 2^{h-1}\]
Properties of Binary Trees (cont.)

**Property 5** The total number of nodes \( n \) satisfies: \( 2h+1 \leq n \leq 2^{h+1}-1 \).

**Proof**

Based on Property 3:

\[
(h+1) \leq e \leq 2^h
\]

Based on Property 2 and \( n = i + e \):

\[
(h+1) \leq (n+1)/2 \leq 2^h \quad \ldots
\]

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

**Property 6** The height \( h \) satisfies: \( \log_2(n+1)-1 \leq h \leq (n-1)/2 \).

**Proof**

Based on Property 5, the following two inequalities hold:

\[
2h+1 \leq n \\
h \leq (n-1)2 \\
\]

\[
n \leq 2^{h+1}-1 \\
n+1 \leq 2^{h+1} \\
\log_2(n+1) \leq h + 1 \\
\log_2(n+1) - 1 \leq h
\]
Properties of Binary Trees  (cont.)

**Property 7**  The height ($h$) satisfies: $\log_2(e) \leq h \leq e-1$.

**Proof**

Based on Property 6: $\log_2(n+1)-1 \leq h \leq (n-1)/2$

Based on Property 2: $\log_2(2e-1+1)-1 \leq h \leq (2e-1-1)/2$

$\log_2(2e)-1 \leq h \leq e-1$

$\log_2(2)+\log_2(e)-1 \leq h \leq e-1$

$\log_2(e) \leq h \leq e-1$
Properties of Binary Trees (cont.)

Summary of Properties

\[ n = e + i \]
\[ e = i + 1 \]

Number of external, internal, and overall nodes as a function of tree’s height

\[ (h+1) \leq e \leq 2^h \]
\[ h \leq i \leq 2^{h-1} \]
\[ 2h+1 \leq n \leq 2^{h+1}-1 \]

Tree’s height as a function of number of external, internal, of overall nodes

\[ \log_2(n+1)-1 \leq h \leq (n-1)/2 \]
\[ \log_2(e) \leq h \leq e-1 \]
\[ \log_2(i+1) \leq h \leq i \]

All other expressions can be obtained from these three.
Binary Tree ADT: Interface

**Binary Tree ADT** – extends Tree ADT, i.e. inherits all its methods

**Additional Methods**

```java
class BinaryTreeNode {
    public Position leftChild(Position v) {
        /* return the left child of a node */
        /* error occurs if v is an external node */
    }

class BinaryTreeNode {
    public Position rightChild(Position v) {
        /* return the right child of a node */
        /* error occurs if v is an external node */
    }

class BinaryTreeNode {
    public Position sibling(Position v) {
        /* return the sibling of a node */
        /* error occurs if v is the root */
    }
}
```
**Indexing Scheme**— for every node $v$ of $T$, let its index/rank $p(v)$ be defined as follows

- if $v$ is the root: $p(v) = 1$
- if $v$ is the left child of node $u$: $p(v) = 2p(u)$
- if $v$ is the right child of node $u$: $p(v) = 2p(u) + 1$

**Advantages**— simple implementation, easy access
**Space Complexity** – let use the following notation

- **n** – number of nodes in T
- **p_M** – maximum value of p(v)
- **N** – array size (N=p_M+1), i.e. space usage

1) **Best Case**: full, balanced tree ⇒ all array slots occupied

\[ N = p_M + 1 = n + 1 = \Theta(n) \]

2) **Worst Case**: highly unbalanced tree ⇒ many slots empty

- **height**: \( h = \frac{(n-1)}{2} \)
- **max p(v)**: \( p_M = 2^{h+1} - 1 = 2^{(n+1)/2} - 1 \)
- **required array size**: \( N = p_M + 1 = 2^{(n+1)/2} = O(2^n) \)
Array-Based Binary Tree: Performance

**Run Times** – Good! all methods, except positions and elements run in constant $O(1)$ time

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>position, elements</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>swapElements, replaceElement</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>root, parent, children</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>leftChild, rightChild, sibling</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>isInternal, isExternal, isRoot</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>expandExternal, removeAboveExternal</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

**Space Usage** – Poor! (in general)

best case (full balanced tree): $O(n)$
worst case (highly unbalanced tree): $O(2^n)$
Link Structure - Based Implementation of Binary Tree

Node in Linked Structure for Binary Trees

- object containing
  1) element
  2) reference to parent
  3) reference to the right child
  4) reference to the left child

- if node is the root: reference to parent = null
- if node is external: references to children = null
Example 4  [ binary tree and its linked list implementation ]
**BTNode Class** – generalization of Position ADT, i.e. implements Position interface

```java
public class BTNode implements Position {
    private Object element;
    private BTNode left, right, parent;

    public BTNode(Object o, BTNode u, BTNode v, BTNode w) {
        setElement(o);
        setParent(u);
        setLeft(v);
        setRight(w);
    }

    public Object element() { return element; }
    public void setElement(Object o) { element = o; }
}
```
public BTNode getLeft() { return left; }
public void setLeft(BTNode v) { left = v; }

public BTNode getRight() { return right; }
public void setRight(BTNode v) { right = v; }

public BTNode getParent() { return parent; }
public void setParent(BTNode v) { parent = v; }

Root Node:  
BTNode root = BTNode(o, null, v, w)

External Node:  
BTNode root = BTNode(o, u, null, null)
**Null_Node** – contains no elements, no children, has only reference to the parent
  - implements Position interface!

**Extended BT** – every external node reference becomes reference to **Null_Node**
  - with this approach, there is never a need to check whether a reference to a child is null
  - implementation presented here, and in the textbook, does not employ Null_Node

```java
(A.getLeft()).element();
```

```java
(A.getLeft()).element();
```
LinkedBinaryTree ADT: Implementation

LinkedBinaryTree Class – implements BinaryTree interface, and also provides 2 additional methods
1) expandExternal
2) removeAboveExternal

public class LinkedBinaryTree implements BinaryTree {
    private Position root; /* reference to the root */
    private int size; /* number of nodes */
    public LinkedBinaryTree() {
        root = new BTNode(null, null, null, null);
        size = 1;
    }
    public void expandExternal (Position v) { ... }
    public void removeAboveExternal (Position v) { ... }
    ...
}

For other methods, and their implementation details, see pp. 268-269 of the textbook.
**LinkedBinaryTree ADT: Implementation (cont.)**

**expandExternal() Method** – transforms v from external into internal node, by creating 2 new external nodes and making them the children of v
- error occurs if v is internal

```java
public void expandExternal(Position v) {
    if (isExternal(v)) {
        ((BTNode) v).setLeft(new BTNode(null, (BTNode) v, null));
        ((BTNode) v).setRight(new BTNode(null, (BTNode) v, null));
        size += 2;
    }
}
```

**Application of expandExternal()** – used for building a tree – see pp. 27
**LinkedBinaryTree ADT: Implementation (cont.)**

**removeAboveExternal() Method** — removes external node w together with its parent v, replacing v with the sibling of w
- error occurs if w is internal

---

**Application of removeAboveExternal()** — used to dismantle a tree
public void removeAboveExternal (Position v) {
    if (isExternal(v)) {
        BTNode p = (BTNode) parent(v);
        BTNode s = (BTNode) sibling(v);
        if (isRoot(p)) {
            s.setParent(null);
            root = s; }
        else {
            BTNode g = (BTNode) parent(p);
            if (p == lefChild(g)) g.setLeft(s);
            else g.setRight(s);
            s.setParent(g);
        }
    size-=2;
    }
    ...
}
LinkedBinaryTree ADT: Implementation (cont.)

Run Times – Good! all methods, except positions and elements run in constant O(1) time

Space Complexity – Good! only O(n), since there is one BTNnode object per every node of the tree

- no empty slots as in array-based implementation

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>position, elements</td>
<td>O(n)</td>
</tr>
<tr>
<td>swapElements, replaceElement</td>
<td>O(1)</td>
</tr>
<tr>
<td>root, parent, children</td>
<td>O(1)</td>
</tr>
<tr>
<td>leftChild, rightChild, sibling</td>
<td>O(1)</td>
</tr>
<tr>
<td>isInternal, isExternal, isRoot</td>
<td>O(1)</td>
</tr>
<tr>
<td>expandExternal, removeAboveExternal</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

Only immediate children!
Example 5  [ creating a tree ]

```java
LinkedBinaryTree t = new LinkedBinaryTree();
t.root().setElement("Albert");
t.expandExternal(tree.root());
t.root().leftChild().setElement("Betty");
t.root().rightChild().setElement("Chris");
t.expandExternal(tree.root().leftChild());
t.root().leftChild().leftChild().setElement("David");
t.root().leftChild().rightChild().setElement("Elvis");
```

![Diagram of a tree with nodes labeled "Albert", "Betty", "Chris", "David", and "Elvis".]