Hierarchical Collections: Trees

2501ICT/7421ICT

Nathan

René Hexel

School of Information and Communication Technology
Griffith University

Semester 1, 2012
Outline

1. Trees
2. Expressions and Grammar Parsing
3. Search Trees
Hierarchical Collections

- Tree definition
- Types of Trees
- Binary Expressions
  - expression trees
  - tree traversals: pre-, in-, postorder
- Examples
  - generating Postfix
  - parsing
Tree Definition

- Each node has at most one predecessor
  - Parent
- Many Successors
  - Children
- Siblings
  - nodes sharing the same parent (e.g., $D_2$ and $D_3$)
Tree Definition (2)

- Topmost Node
  - root
- Children, children of children, …
  - Descendants
  - Successors

⇒ All nodes are successors of root
Tree Definition (3)

- **Leaf Nodes**
  - nodes without successors
  - \( D_3 \) and \( D_4 \)

- **Frontier**
  - set of all leaf nodes
**Tree Definition (4)**

- **Interior Nodes**
  - nodes with at least one successor
  - $D_1$ and $D_2$

- **Ancestors**
  - immediate or indirect predecessors
  - $D_1$ is an ancestor of $D_2$, $D_3$, and $D_4$
Levels are numbered from 0
→ level 0 is always the root

This tree has 3 Levels
- Level 0: $D_1$
- Level 1: $D_2$ and $D_3$
- Level 2: $D_4$
Binary Trees

- Binary Trees
  → allow at most *two children* per node

- Generic Trees
  → allow any number of children per node
Generic Trees

- **Order of the Tree**
  - maximum *number of children* allowed for any given node
  - e.g. Order 3
Tree Applications

- Parsing Languages
  - Computer Languages, Mathematical Formulae
  - Natural Languages
- Searchable Data Structures
  - Databases (e.g., B-Trees)
  - Heaps and Balanced Trees
- Sorting and organising Data
Read in Expressions
   → (2 + 3) * 5

Check Syntactical Correctness
   is everything where it should be?

Create Parse Tree
   evaluator checks semantic meaning and processes the
data in the Tree to produce meaningful output
Binary Expressions

- Stored in Binary Trees
  - $3 + 5$
- Numbers
  - leaf nodes
- Operators
  - interior nodes
- Operands
  - contained in a subtree of the expression
Example Expression

$3 \times 4 + 5$
Operator Precedence

3 \times (4 + 5)

The *higher* the precedence, the *lower* in the tree

\rightarrow \text{ overridden by parentheses}
Operator Precedence (2)

3 + 4 + 5

If operators have equal precedence, the ones on the left appear lower in the tree when parsed from left to right!
Evaluating an Expression Tree

- Begin at the root Node
- If a number, return it, otherwise
- Run the operator with the results of
  - evaluating its left and right subtrees, and
  - return this value
Evaluating Example

3 \times (4 + 5)

- Evaluation starts at the top
- $\times$ is an operator
  - $\Rightarrow$ evaluate left and right subtrees first!
- 3 is a number
  - $\Rightarrow$ return 3
- + is an operator
  - $\Rightarrow$ evaluate left and right subtrees first!
- 4 is a number
  - $\Rightarrow$ return 4
- 5 is a number
  - $\Rightarrow$ return 5

$27$
Pseudo code for tree evaluation

evaluate(node)
{
    if node is a number
        return number;
    else
    {
        left = evaluate(node.left);
        right = evaluate(node.right);
        return compute(node, left, right);
    }
}
Binary Tree Traversals

- **Preorder**
  → visit node, then go left, then go right

- **Inorder**
  → go left, then visit node then go right

- **Postorder: Depth First**
  → go left, then go right, then visit node

- **Breadth First**
  → level 0, then level 1, then level 2, etc.
Preorder, Inorder, and Postorder

→ correspond with Prefix, Infix, and Postfix notations of an expression

- Infix:
- Prefix = Polish notation (PN):
- Postfix = reverse Polish notation (RPN):

⇒ use the same generic recursive algorithm!
Prefix Pseudocode

Prefix Evaluation

```java
String prefix(node) {
    if (node == NULL)
        return " ";
    else
        return node + prefix(node.left) + prefix(node.right);
}
```
Infix Pseudocode

Infix Evaluation

```java
String infix(node) {
    if (node == NULL) return "";
    else return infix(node.left) + node + infix(node.right);
}
```
Postfix Pseudocode

```java
String postfix(node) {
    if (node == NULL) return "";
    else return postfix(node.left) + postfix(node.right) + node;
}
```
Infix Expressions

Expression = Term { + | - Term }
Term = Factor { * | / Factor }
Factor = number | ( Expression )

- Represents standard maths formulas
  - e.g.: \( 3 + 4 \times (5 - (6/7)) \)
- can be used to create a parse tree!
  → recursive descent parsing
Recursive Descent Parsing

Expression = Term { + | - Term }

Expression()
{
    Term();
    while (token == '+') ||
            token == '-')
    {
        get_token();
        Term();
    }
}
Recursive Descent Parsing

Term = Factor { * | / Factor }

Term()
{
    Factor();
    
    while (token == '*' ||
            token == '/')
    {
        get_token();
        Factor();
    }
}
Recursive Descent Parsing

Factor = number | ( Expression )

Factor() {
    switch (token) {
        case number: get_token(); break;
        case '(': get_token(); Expression();
           if (token != ')')
              error("No closing ")
           get_token();
           break;
        
        default: error("Error '%s'\n", token);
    }
}
Binary Search Tree

- “Sorted Array” stored in a tree
  - left to right order
  - e.g. A B C
Binary Tree Search

1. Start at the root node $n$
   - searching for an object $s$
2. if $s = n$ then we are finished
3. if $s < n$ then $n :=$ left child
4. if $s > n$ then $n :=$ right child
5. repeat from step 2 until finished
   - ... either $s$ has been found
   - ... a leaf node has been reached, but $s$ has not been found
Recursive Pseudocode

```c
recursive_pseudocode

search(s, node) {
    if node == nil
        return nil;  // not in tree
    else if s == node->content
        return node;  // found
    else if s < node->content
        return search(s, node->left);
    else  // s > node
        return search(s, node->right);
}
```
Search Tree Complexity

- Depends on the Balance of the Tree
- Unbalanced Tree:
  - $O(n)$
- Balanced Tree
  - $O(\log n)$
  - equivalent to Binary Search in Sorted Array
Balanced Trees

- **Balanced Tree**
  - Difference in height of both subtrees of any node in the tree is either 0 or 1

- **Unbalanced Tree:**
  - Difference of subtree heights > 1

- **Perfectly Balanced Tree**
  - Balanced Tree with leaves only on one or two levels
Creating a Search Tree

1. Incrementally
   - Sort in a new Node $n$

2. Search if $n$ already exists
   - Finished if $n$ exists (do nothing)
   - Otherwise add $n$ as the left or right child of the last node searched (depending on whether $n$ was smaller or bigger than the last node)

3. Produces an ad-hoc Search Tree
   - Not guaranteed to be balanced!
Balancing a Complete Tree

1. Write out the Search Tree in sorted order
   - e.g. in alphabetical order
   - write to sorted array/list
   - write to file

2. Read back the sorted data, creating a Balanced Tree
   - Recursively create Left Children, Root, then Right Children for each subtree
   - Creates a *perfectly balanced tree*!
Balancing ReadTree Algorithm

```c
BTNode *readTree(BufferedReader *file, int n) {
    if (n <= 0) return nil;

    BTNode *node = [BTNode new];
    [node setLeft: readTree(file, n/2)];
    [node setValue: [file readLine]];  
    [node setRight: readTree(file, (n-1)/2)];

    return node;
}
```
Self-Balancing Trees

- Problem: writing out and reading back
  - takes time
  - requires space
  - Read back the sorted data, creating a Balanced Tree
    - Sorted data are available in 3 places (original tree, file/array, and final, balanced tree)

- Alternative: keep the tree balanced
  - insertion operation needs to check if tree is still balanced
  - re-balance if adding a node breaks balance
Red-Black Tree

1. Every node is either red or black
2. The root node is black
3. All leaves are black
   - Leaves are dummy empty nodes at the end of the tree
4. Both children of red nodes are black
5. All paths from any given node to its descendant leaves contain the same number of black nodes
Red-Black Tree Definitions

- **Grandparent**
  - the parent of the parent node

- **Uncle**
  - the “other child” of the grandparent, i.e.
    - if (parent == grandparent.left)
      - uncle = grandparent.right)
    - else // if (parent != grandparent.left)
      - uncle = grandparent.left)

- Both children of red nodes are **black**

- All paths from any given node to its descendant leaves contain the *same number of black* nodes
Red-Black Tree Insertion

- Add node as in a binary search tree
  → default colour is red
- Case 1: new node \( n \) is root
  → repaint as \textit{black}
- Case 2: parent \( p \) of \( n \) is \textit{black}
  ⇒ everything is fine!
case 3: both parent and uncle are red

→ repaint parent and uncle as black
→ repaint grandparent as red (property 5)

• may now violate property 2 (root is black) or property 4 (both children of red nodes are black)

⇒ therefore recursively restart with case 1 on the grandparent
Case 4: parent $p$ of new node $n$ is red, uncle $u$ is black

- grandparent $g$
  - if $n == p.\text{right} \&\& p == g.\text{left}$
    - perform left rotation to switch roles of $n$ and $p$
  - if $n == p.\text{left} \&\& p == g.\text{right}$
    - perform right rotation to switch roles of $n$ and $p$
  - continue with Case 5!
Red-Black Tree Insertion

Case 5: parent $p$ of new node $n$ is red, uncle $u$ is black
- switch the colours of $p$ and grandparent $g$
- if $n == p.left$ && $p == g.left$
  → perform right rotation on $g$
- if $n == p.right$ && $p == g.right$
  → perform left rotation on $g$
⇒ Terminal manoeuvre, no further repaint needed!
Strings in Search Trees

- Storing long strings in binary search trees can be inefficient
  - Requires full string (key) comparisons for every node
    \[ O(n \log n) \] search complexity if average string length approximates the number of nodes \( n \)

- Trie
  - Retrieval of keys while traversing a search tree
trees that store the individual characters of the key strings

common prefixes share the same path through the search tree, e.g.

- on
- off
- often
Trie Efficiency

- Time and Space efficiency
  → large number of long words

- Efficient for spell checking
  → common prefixes determine tree height
    - English words do not share long common prefixes
      - 5-7 node visits, regardless of whether 10,000 or 100,000 words are stored!
      - compare with $13 = \log_2 10000$ or $17 = \log_2 100000$ node visits for optimal binary search trees!
Trie Challenges

- **Prefix detection**
  - how to distinguish words such as “are” and “area”
  - requires a separate *end of word* mark

- **Efficient search requires** $O(1)$ character search in nodes
  - requires (array) space for each node, indexed by char
    - 26+1 pointers for A-Z (plus end of word mark)
    - 127+1 pointers for ASCII
    - 65536 pointers for UTF-16
    - 4294967296 pointers for UTF-32 (full Unicode)

- **Suffixes are different node types**
  - makes trie handling code more complex