About this lecture

• We introduce some popular algorithms to traverse a rooted ordered binary tree
  1. Level Order (similar to BFS)
  2. Pre-order, Post-order, In-order (similar to DFS)

• Then, we will discuss a related topic called expression tree
Level Order Traversal
Level Order

• Imagine we have a rooted binary tree, and we apply the BFS algorithm on the root (as the source)
• What will happen?
Level Order

- The nodes of the tree will be visited in the following order:

- This is called the level order traversal
Implementation

- To implement level order traversal, we just run BFS on the root.
- Since each node (except root) in a rooted tree has exactly one parent, it can only be discovered once during BFS.
- No need to have an extra array to remember if a node is marked or not, and we need only a queue.
- Running time: $O(|V|)$
Preorder/Postorder/Inorder Traversal
DFS Traversal on a Tree

• We now describe 3 popular algorithms to traverse a tree
  • Preorder, Postorder, Inorder
  • They are all based on DFS
• The only difference is:
  “During the traversal, what time they will output the content of a node”
DFS on a Tree

• When we apply DFS on a tree, when it visits a node:
  • it calls DFS recursively on left child
  • then DFS recursively on right child
DFS on a Tree

• A node is actually visited a few times
  • Exactly 3 times for binary tree
• They include: the time before the first DFS, and the times after each DFS
**Preorder Traversal**

- The *preorder* traversal prints the content of a node when it is first visited.
- In our example, we print: FBDEAC
Postorder Traversal

- The **postorder** traversal prints the content of a node when it is last visited.
- In our example, we print: DEBCAF
Inorder Traversal

• The **inorder** traversal prints the content of a node just before we visit right child
• In our example, we print: DBEFAC
Implementation

To implement the above traversal algorithms, we first see that DFS on a binary tree can be done as follows:

\[
\text{DFS}(u) \{
    \begin{array}{l}
    1. \text{Call DFS}(u.\text{left}) \\
    2. \text{Call DFS}(u.\text{right}) \\
    \end{array}
\}
\]

At the main program, we call DFS(root)
Implementation

• Then the preorder traversal is implemented as follows:

```plaintext
Preorder (u) {
    1. Print content of u;
    2. Call Preorder (u.left);
    3. Call Preorder (u.right);
}
```

At the main program, we call `Preorder (root)`
Implementation

• Similarly, the postorder traversal is implemented as follows:

\[
\text{Postorder} \ (u) \ \{ \\
\quad 1. \ \text{Call Postorder} \ (u.\text{left}) ; \\
\quad 2. \ \text{Call Postorder} \ (u.\text{right}) ; \\
\quad 3. \ \text{Print content of } u ; \\
\} 
\]

At the main program, we call Postorder (root)
Implementation

• And the inorder traversal is implemented as follows:

    Inorder (u) {
        1. Call Inorder (u.left);
        2. Print content of u;
        3. Call Inorder (u.right);
    }

    At the main program, we call Inorder (root)
Remarks

• Running time: \( O( |V| ) \) time

• The preorder and postorder traversals are well-defined for non-binary trees

• For inorder, to visit a node with degree more than 2, there are 2 common ways: One prints the content after the first DFS, and one prints after every DFS except the last

Two versions of Inorder: EBCF vs EBCBF
Expression Tree
Expression Tree

- We can use rooted binary trees to represent mathematical expressions that involve only binary operators.
- Each internal node stores an operator.
- Each leaf stores an operand.
- Ex: $\times$ $\frac{3}{4} + \frac{2}{2}$
Expression Tree

• Each internal node $u$ corresponds to a value computed recursively as follows:
  1. Compute the value $x$ corresponding to left child of $u$
  2. Compute the value $y$ corresponding to right child of $u$
  3. The value of $u = x \Delta y$ where $\Delta$ is the operator stored in $u$

• value of expression = value of the root
Expression Tree

- Ex:

Value: $3 \times 4$  
Value: $3+4$  
Value: $(3 \times 4) + 2$  
Value: $(3+4) \times 2$
Expression Tree

• Each mathematical expression has a corresponding expression tree
• To find such a tree, we can:
  1. First determine which operator is last applied, then put it inside the root;
  2. After that, recursively construct the left and right subtrees of the root based on the contents on the left and right sides of the operator.
Expression Tree

- Ex: $5 + ((1 + 2) \times 4) - 3$
Expression Tree

• Ex: $5 + ((1 + 2) \times 4) - 3$
Expression Tree

- If we now perform preorder traversal on the expression tree, we get the **prefix notation** of the expression.

Prefix Notation: 
$- + 5 \times + 1 2 4 3$
Expression Tree

- If we perform postorder traversal instead, we get the **postfix notation of the expression**

Postfix Notation:

```
5 1 2 + 4 × + 3 –
```
Evaluation

• In prefix or postfix notations, we do not need any parentheses
  • Both notations can allow us to compute the value of the original expression
  • Idea: Using a stack

• Remark: the original expression is stored in the infix notation
Evaluating Prefix Notation

• In prefix notation, when there are two consecutive "values", we can apply the operator before the two values.

• So the evaluation can be done as follows:
  • Push operator or value on a stack, but ..
  • Whenever there are two values $x$ and $y$ on top of the stack, pop $x$ and $y$, and also the next operator $\Delta$. Then push a new value $x \Delta y$ back to stack.
Evaluating Prefix Notation

• Ex: \(- \div + 5 \times + 1 2 4 3\)
  (Prefix notation of \(5 + ((1 + 2) \times 4) - 3\))

<table>
<thead>
<tr>
<th>contents of stack after key operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(- \div + 5 \times + 1 2)</td>
</tr>
<tr>
<td>(- \div + 5 \times 3)</td>
</tr>
<tr>
<td>(- \div + 5 \times 3 4)</td>
</tr>
<tr>
<td>(- \div + 5 12)</td>
</tr>
<tr>
<td>(- 17)</td>
</tr>
<tr>
<td>(- 17 3)</td>
</tr>
<tr>
<td>14</td>
</tr>
</tbody>
</table>
Evaluating Postfix Notation

• In postfix notation, when we see an operator, we can apply the operator to the two values before the operator.
• So the evaluation can be done as follows:
  • Push operator or value on a stack, but ..
  • Whenever we see an operator $\Delta$, we pop $\Delta$, and the next two values $x$ and $y$ on top of the stack. Then push a new value $x \Delta y$ back to stack.
Evaluating Postfix Notation

- **Ex**: $5\ 1\ 2\ +\ 4\ \times\ +\ 3\ -$  
  
  (Postfix notation of $5 + ((1 + 2) \times 4) - 3$)

<table>
<thead>
<tr>
<th>contents of stack after key operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5\ 1\ 2\ +$</td>
</tr>
<tr>
<td>$5\ 3$</td>
</tr>
<tr>
<td>$5\ 3\ 4\ \times$</td>
</tr>
<tr>
<td>$5\ 12$</td>
</tr>
<tr>
<td>$5\ 12\ +$</td>
</tr>
<tr>
<td>$17$</td>
</tr>
<tr>
<td>$17\ 3\ -$</td>
</tr>
<tr>
<td>$14$</td>
</tr>
</tbody>
</table>
Remarks

• Prefix or postfix notations are very useful because they can evaluate an expression easily (in one pass)

• In the next assignment, we will examine how to convert an expression from infix to postfix

• This can also be done with a stack!!