Lecture 27

Today:

- Binary Trees
- Recursive Definitions of Trees
- Binary Tree Implementation
- Expression Trees
- Traversals
- Grammars
Rooted Trees (Review)

A *rooted tree* is a tree where all but one vertex has exactly one inbound edge (from its *parent*).

- usually drawn by level, top down
- *root* vertex has no inbound edge
- *leaf* vertex has no outbound edge
- parents point to *children*
- *ancestors* point to *descendants* via a downward path

A *binary tree* is a rooted tree in which no vertex has more than 2 children.
Subtrees and Recursive Definitions

There are many *subtrees* within a binary tree:

- the two most important are the left and right subtrees
- rooted at the left and right children of the root
- to visualize, remove the root!

Leads to a recursive definition:

*\( T \) is a binary tree when:

- \( T \) is an empty tree (i.e., no vertices)
  OR . . .
- \( T \) has a root vertex whose left and right subtrees are binary trees
Subtrees and Recursive Definitions

There are many *subtrees* within a binary tree:

- the two most important are the left and right subtrees
- rooted at the left and right children of the root
- to visualize, remove the root!

Leads to a recursive definition:

$T$ is a binary tree when:

- $T$ is an empty tree (i.e., no vertices) OR . . .
- $T$ has a root vertex whose left and right subtrees are binary trees
**Trees and Recursion**

Recursive definitions benefit you in two ways:

- allow you to write recursive algorithms
- allow you to reason about the structure by recursion (or induction)

**How to build a binary tree in C++?**

- use the recursive definition
- adopt similar strategy to a linked list

```c
struct LLnode {
    int data;
    struct LLnode * next;
};
```

A binary tree $T$ is when either:

- $T$ is an empty tree
- $T$ has a root vertex whose left and right subtrees are binary trees
Trees and Recursion

Recursive definitions benefit you in two ways:

- allow you to write recursive algorithms
- allow you to reason about the structure by recursion (or induction)

How to build a binary tree in C++?

- use the recursive definition
- adopt similar strategy to a linked list

```c
struct BTnode {
    int data;
    struct BTnode * next;
};
```
Trees and Recursion

Recursive definitions benefit you in two ways:

- allow you to write recursive algorithms
- allow you to reason about the structure by recursion (or induction)

How to build a binary tree in C++?

- use the recursive definition
- adopt similar strategy to a linked list

```c
struct BTnode {
    int data;
    struct BTnode * left;
    struct BTnode * right;
};
```

$T$ is a binary tree when either:

- $T$ is an empty tree
- $T$ has a root vertex whose left and right subtrees are binary trees
Trees and Recursion

Recursive definitions benefit you in two ways:

- allow you to write recursive algorithms
- allow you to reason about the structure by recursion (or induction)

How to build a binary tree in C++?

- use the recursive definition
- adopt similar strategy to a linked list

```c
struct BTnode {
    int data;
    struct BTnode * left;
    struct BTnode * right;
    struct BTnode * parent;
};
```
A full binary tree is a non-empty binary tree, where each vertex has exactly 0 or 2 children.

**Theorem:** A full binary tree always has an odd number of vertices.

**Proof by induction:**

- If the root has 0 children, then the tree has only one vertex, which is odd.
- If the root has 2 children, then their subtrees must also be full, and by induction, odd. The total number of vertices is the sum of 3 odd numbers, which is odd.

Expression trees are full.
Expression Trees

An expression tree is a full binary tree that represents an arithmetic calculation:

- internal nodes are binary operators
- leaves are numbers

```
  +
 /|
/ | /
4 7
```

left exp’n tree  +  right exp’n tree
Expression Trees and Postfix

An expression tree is a full binary tree that represents an arithmetic calculation:

- internal nodes are binary operators
- leaves are numbers

Thus, postfix expressions can be defined recursively, too.

$E$ is a postfix expression when:

- $E$ is a number, OR . . .
- $E$ is two postfix expressions followed by a binary operator ($+, -, \times, \div$)

Algorithm to evaluate expression tree:

- bottom up
Tree Evaluation: Traversals

Algorithm to evaluate a tree rooted at vertex $x$:

- If $x$ has a number, then return that number
- If $x$ has an operator, then:
  - evaluate the left subtree
  - evaluate the right subtree
  - return $(\text{left } op \text{ right})$

Known as a post-order traversal

- evaluate the children first, then yourself
- follows the order: left $\rightarrow$ right $\rightarrow$ self

Other common traversals:

- self $\rightarrow$ left $\rightarrow$ right: pre-order
- left $\rightarrow$ self $\rightarrow$ right: in-order

Q. What’s the in-order traversal?

Q. What is this sequence?
Stack-Based Postfix Calculator

Use a Stack ADT to evaluate postfix.

Algorithm:

Create an empty stack $S$
while there is still input {
    if next input token is a number
        push the number to $S$
    if next input token is an operator {
        pop from $S$ → $b$
        pop from $S$ → $a$
        push ($a \text{ op } b$) to $S$
    }
}

pop from $S$ → result
Stack-Based Postfix Calculator

Use a Stack ADT to evaluate postfix.

Algorithm:

Create an empty stack $S$
while there is still input {
    if next input token is a number
        push the number to $S$
    if next input token is an operator {
        pop from $S \rightarrow b$
        pop from $S \rightarrow a$
        push $(a \text{ op } b)$ to $S$
    }
}
pop from $S \rightarrow \text{result}$

If any pop fails, then it's invalid postfix.
If $S$ ends nonempty then it's invalid postfix.
Building Expression Trees from Postfix

Adapt postfix calculator algorithm to build trees from postfix.

Algorithm:

Create an empty stack S

while there is still input {
    if next input token is a number
        push the number to S
    if next input token is an operator {
        pop from S → b
        pop from S → a
        push (a op b) to S
    }
}

pop from S → result

Example:

9 6 5 + 6 9 - * -

S:
Use a Stack ADT to evaluate postfix.

Algorithm:

Create an empty stack S
while there is still input {
    if next input token is a number
        push to S
    if next input token is an operator {
        pop from S → b
        pop from S → a
        push (a op b) to S
    }
}
pop from S → result

Building Expression Trees from Postfix

Adapt postfix calculator algorithm to build trees from postfix.

Algorithm:
Create an empty stack S
while there is still input {
    if next input token is a number
        push to S
    if next input token is an operator {
        pop from S → b
        pop from S → a
        push (a op b) to S
    }
}
pop from S → result

Example:
9 6 5 + 6 9 − * −

S:
Building Expression Trees from Postfix

Adapt postfix calculator algorithm to build trees from postfix.

**Algorithm:**

Create an empty stack S

while there is still input {

    if next input token is a number
        push \# to S

    if next input token is an operator {
        pop from S → b
        pop from S → a
        push (a op b) to S
    }

    }
Building Expression Trees from Postfix

Adapt postfix calculator algorithm to build trees from postfix.

Algorithm:

Create an empty stack $S$
while there is still input {
    if next input token is a number
        push to $S$
    if next input token is an operator {
        pop from $S \rightarrow b$
        pop from $S \rightarrow a$
        push $(a \ op \ b)$ to $S$
    }
}
pop from $S \rightarrow$ result

Example:

9 6 5 + 6 9 − * −

S:

```
         −
        /   \
      6     9
       / \
     +   5
       \
        6
```

```
          op
         /   \
        a     b
       / \
     +   5
       \
        6
```
Building Expression Trees from Postfix

Adapt postfix calculator algorithm to build trees from postfix.

Algorithm:
Create an empty stack S
while there is still input {
    if next input token is a number
        push to S
    if next input token is an operator {
        pop from S → b
        pop from S → a
        push (a op b) to S
    }
}
pop from S → result

Example:

```
9 6 5 + 6 9 - * -
```

S:
Building Expression Trees from Postfix

Adapt postfix calculator algorithm to build trees from postfix.

Algorithm:

Create an empty stack \( S \)

while there is still input {

if next input token is a number

push \( \# \) to \( S \)

if next input token is an operator {

pop from \( S \) → \( b \)

pop from \( S \) → \( a \)

push \( a \ op b \) to \( S \)

}

}

pop from \( S \) → result

Example:

\[
\begin{array}{ccccccc}
9 & 6 & 5 & + & 6 & 9 & - & \cdot & - \\
\end{array}
\]
Expression Grammars

You can express the recursion using a grammar

Grammar for $E$

- $E \rightarrow \text{number}$
- $E \rightarrow EE\text{operator}$

Production rules should be able to derive any valid expression, by stepwise substitution until no $E$'s remain.

Q. What’s the grammar for infix?

E.g. Derive: $7 4 - 8 * 2 9 * +$

$E \Rightarrow EE+$

$\Rightarrow EE* E+$
$\Rightarrow EE- E* E+$
$\Rightarrow 7 4 - E* E+$
$\Rightarrow 7 4 - 8 * E+$
$\Rightarrow 7 4 - 8 * EE* +$
$\Rightarrow 7 4 - 8 * 2 E* +$
$\Rightarrow 7 4 - 8 * 2 9 * +$

$E$ is a *postfix expression* when:

- $E$ is a number, OR . . .
- $E$ is two postfix expressions followed by a binary operator