is a minimally connected graph; equivalently, a connected graph with no circuits.

In the course, we cover only rooted trees. In graph theory, a (non-rooted) tree

Trees

Stacks

CSI 310: Lecture 19
Label 3 is to do calculations in the stack-based post-fix syntax language Postscript.

Label 4 is to observe the stack of activation records during the run of a

C/C++ function calls and returns, both recursive and non-recursive.

Implementing and organizing local variables and other data relevant to all

runs C/C++ programs. References and pointers provide access into non-top stack frames.

1. The "run-time stack" of activation records, internal to the system when it

exits.

Expressions, such as in the "two-stacks" algorithm of Project 4.

2. Storing and organizing intermediate results when evaluating or parsing

parenthesized expressions.

3. Figure which pairs of parentheses MATCH in a correctly nested

uses for stacks:

ONLY ONE END (called the top).

are permitted at

(push) (()pop) (())pop

What is a stack? A stack is a sequence that is restricted so that access
2. How can we organize computer memory so the value of each subexpression is stored after it is computed and can be retrieved when it will be used?

2. How can we organize computer memory so the value of each subexpression is stored after it is computed and can be retrieved when it will be used?

\[
\left( \sum_{i=1}^{n} (z_i - 2z_i) + \sum_{i=1}^{n} (y_i - 2y_i) + \sum_{i=1}^{n} (x_i - 2x) \right)
\]

3. How can we program the computer to do the arithmetic or other operations in the order expressed by the expression?

Examples of sequences of expressions:

1. How can we program the computer to do the arithmetic or other operations in the order expressed by the expression?

Examples of sequences of expressions:

1. How can we program the computer to do the arithmetic or other operations in the order expressed by the expression?

Problems with expressions:

Problems with expressions:
The particular expressions input to the computer for an expression evaluator program to evaluate are NOT KNOWN AHEAD OF TIME.
which are strings, not trees.

They don't solve the problems (1) and (2) because the users input expressions

But what about expression trees?

Complicated algorithms are needed to find them.

anywhere in the expression.

2. The first operation as well as the top level operation can be located

1. They require precedence rules and/or ( ) to express the order of operations.

Facts about Infix Expressions:

University at Albany Computer Science Dept.
4. Demonstrate tree printing and evaluation on that tree.

3. Repeat it so it builds the expression tree.

\[
\begin{align*}
&\text{(3)} \quad 5 \\
&\text{(2)} \quad 15/3 \\
&\text{(1)} \quad [6+9]/3
\end{align*}
\]

The value of the expression is 7.

Print the subexpression equaling that number.

Enhance it so each intermediate “number” also carries information.

Implement the two-stacks algorithm to evaluate infix expressions.

Project 4.
Let's look at the 2-stack algorithm from DSO chapter 7.
computes the INTEGER whose value is signaled by that digit character.

\texttt{integer \texttt{\textbackslash n value = mychar} - \texttt{0} \texttt{\textbackslash n...} \texttt{. 1, \ldots, 9, \ldots, 9, \ldots, 9}}$

\texttt{\textbackslash nstatus (mychar) == true \texttt{\textbackslash nso mychar == \texttt{0}, \texttt{\textbackslash n1, \ldots, 9, \ldots, 9}}} and

\texttt{\textbackslash ncin >> mychar; \texttt{or mychar = butt}[0]; \texttt{\textbackslash ncin >> mychar; \texttt{char butt}[size]}}

\texttt{\textbackslash nchar mychar; \texttt{\textbackslash nchar butt}[size]; \texttt{\textbackslash n}}

A practical tip: How to evaluate a decimal digit: Suppose...
// The value of the expression has been evaluated as an arithmetic expression. The value of the expression has been read from the stream and this is a

// precondition: A line has been read from the stream this

// the four operators - * + /

// parentheses are arithmetic expression formed from non-negative numbers.

// precondition: The next line of characters in the stream is a 

double read-and-evaluate(stream_t &s);

// prototypes for functions used by this demonstration program:

using namespace std;

#include <stack>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cassert>

FILE *file; // #include <cstdio>

The purpose is to illustrate a fundamental use of stacks.

The arithmetic expression input and evaluates the arithmetic expression.

Basic calculator program that takes a fully parenthesized arithmetic input.
```cpp
void evaluate(char* stack, stack<tops> stack<Top>, stack<numbers> stack<Double>> numbers, char operation, Double answer)
{
    if (operation == '+' || operation == '-' || operation == '/' || operation == '*')
    {
        double answer = (tops(stack, stack<Top>, stack<Double>> numbers, operation, answer));
        cout << "That evaluates to " << answer << endl;
    }
}
```
if (isdigit(instream.peek()) || (instream.peek() == DECIMAL))
{
    while (isdigit(instream.peek()) || (instream.peek() != ',') && next_number)
    {
        // Loop continues while next number is not bad (tested by instream)
    }
}

char symbol;
double number;
stack<char> operations;
stack<double> numbers;

const char LEFT_PARENTHESIS = ',';
const char RIGHT_PARENTHESIS = ')
const char DECIMAL = ';

library math facilities used: string, istream, stack
double read-and-evaluate(istream instream)
{
    return EXIT_SUCCESS;

return numbers.top

{
  this.ignore
  else
    { 
      evaluate-stack-tops(numbers, operations)
      this.ignore
    }

    else if (this.peek() == RIGHTPARENTHESIS)
      { 
        operations.push(symbol)
        this >> symbol
      }

    else if (is(CHAR) "\+\-\*/"")
      { 
        numbers.push(number)
        this >> number
      }
break;
    case "+": numbers.push(operand1 + operand2);
    break;
}

switch (operations.top()) {
    case "(":
        operand1 = numbers.pop();
        operand2 = numbers.pop();
        double operand1, operand2;
        // evaluate operation
        break;
}
{ operations.pop();
    {
        break;
    }
    case numbers.push(operand1 / operand2):
ValadposNode *next; //addr on next node or NULL
Valadpos data;
} struct ValadposNode
{

ValadposNode *MyExpR

//

//

//

//

double value; //value of some subexpression.
} struct Valadpos

ValadposNode *head = NULL;

//this is the last.

//

ValadposNode *hasVal;

// ValadposNode

//

ValadposNode *delNode;

//

ValadposNode *delNext;

//

//

//

//

ValadposNode *delNext;

//

ValadposNode *delNext;

//

ValadposNode *delNext;

//

ValadposNode *delNext;

//

ValadposNode *delNext;

//

ValadposNode *delNext;

//

ValadposNode *delNext;

//

ValadposNode *delNext;

//

ValadposNode *delNext;

//

ValadposNode *delNext;

//

ValadposNode *delNext;
{ return ret;
dele te pt;
VpStackHea d = pt->next;
VAlu ndpos ret = pt->data;
VAlu ndpos pt = VpStackHEAD;
assert(VpStackHEAD);
}
}
VAlu ndpos Vpop()
{
VpStackHEAD = pt;
pt->next = VpStackHEAD;
pt->data = Vp;
VAlu ndposNode *pt = new VAlu ndposNode;
}

void Vpush(VAlu ndpos Vp)
VAlu ndposNode *VpStackHEAD; /*GLOBAL
char MYEXPR[EXPRSIZE]; /*GLOBAL
#define EXPRSIZE 200

University at Albany Computer Science Dept.
Operator ValAndPos

in position 10.

6 in position 6.

* in position 4.

MyExpr

( ( 3 + 4 ) * ( 2 − 6 ) )

stacks after processing the stacks after processing the stacks after processing the
Data Structure diagram of Linked List implementation of this ValAndPos stack, after processing the stack after the 6
in position 10.

VPStackHead

ValAndPos Operator

MyExpr

( ( 3 + 4 ) * ( 2 - 6 ) ) / 0 (garbage)
void VPPush(ValAndPosNode *pt) {
    new ValAndPosNode{
        pt = ValAndPosNode;
    }
}

void VPPush(ValAndPosNode *vp) {
    vp = ValAndPosNode;
}

VPPush(ValAndPosNode);

//call VPPush with a Value Argument!

va

3.0

2

VPPush(TMP);

3.0

2
void VPPush(ValAndPos vp)
{
    ValAndPosNode *pt = new ValAndPosNode;
    pt->data = vp;
    pt->next = VPStackHEAD;
    VPStackHEAD = pt;
}
```c
void VPPush(ValAndPos vp)
{
    ValAndPosNode *pt = VPStackHEAD;
    VPStackHEAD = pt;
    pt->val = vp->val;
    pt->pos = vp->pos;
    vp = VPPush(vp);
}
```
{ }
// you wrote!
stack<char> operators;
stack<ValAndPos> ValAndPosStack;

tack<ValAndPos> Poststack;

} void doOneExpression (char * MYEXPR )
	//

#include "ValAndPos.h"
#include "stack2.h"

Main and Switch style:
# How to push a ValAndPos record on this stack:

```c
{ ValAndPos stack, push(TMPE)

    // Your program would generate the position
    TMP.position = 2;
    // Your program would compute the value
    TMP.value = 3.0;

    // stack<char> operations:
    stack<char> stack, stackValAndPos; ValAndPos
}
```

```c
void DoOneExpression(char *MYEXPR)

    #include "ValAndPos.h"
    #include "stack2.h"
```

University at Albany Computer Science Dept.
{  
    // Print a report for debugging.
    from the VALANPSE stack" >> end;
    " position=" >> TMP.position  
    count >> we popped (valute=" >> TMP.value  
    VALANPS.stack.pop();
    TMP = VALANPS.stack.top();  
    VALANPS.TMP;
    ... // stack<char> operations;
    stack<VALANPS> VALANPSstack;
}

void DoKnapsack(char *MYEXPR) {  
#include "VALANPS.h"
#include "stack2.h"
#include "stack1.h"

How to pop and use a VALANPS record on this stack:
description of the process on p. 345-346:

Compare the operation (p. 346-348) of the 2-stacks algorithm (p. 348) WITH:

Why the 2-stacks algorithm works:
Each subexpression corresponds to a number stack entry. When two numbers are popped, combined and pushed, the new stack entry corresponds to the combination of the two old stack entries.
$((3+4)\times2)$
Parentheses!

Prefix and Postfix notation can express every expression without any parentheses.

Fact: Suppose each operator has its own fixed number of operands, the

- Prefix: Each operator is written after its operand(s).
- Postfix: Each operator is written after its operand(s).
- Infix: Each operator is written between its operand(s).

Expressions (as strings) are very smart.

Computer Scientists know two alternatives to infix notation for writing
expression in postfix form.

are used by recursive evaluation of an expression tree, you will write the

3. If you write the constants and variable, and the operations, in the order they

suffices to store and retrieve subexpression values (solves problem (2)).

(2). A stack of subexpression values (store by push, access and removal by pop)

expression. (Solves Problem (1)).

exactly matches the order of data access when the computer evaluates the

The left-to-right order of atoms (constants and variables) and operators

What’s cool about Postfix:

University at Albany Computer Science Dept.
Suppose we write down the constants and operations in the order that we use them when we evaluate this expression. First we take 6, and then 9, and then add them.

Next, we write 3, then divide / 6 + 9 / 3 + 6 9 - 4.

So, we first write: 6 9 + 3 / 6 4 - *

Here’s the result: The expression tree:

Next, we show the stack used to evaluate this postfix notation:

expression after each step of evaluation:

expression tree:

expression for this postfix notation:

( (4 - 6) * (3 / (6 + 9)) )
We can verify the result is 10 by calculating directly from the expression tree.

We can also verify this from the expression by using elementary school methods:

\[( ( 4 - 6 ) \times ( 3 / ( 6 + 6 ) ) ) \]

* Postfix: \[6 \ 9 \ + \ 3 \ / \ 6 \ 4 \ - \]

\[6+9=15 \ 15/3=5 \ 6-4=2 \ 5\times2=10 \]

10 by calculating directly from the expression tree.
Each arc expresses the structural relation between the root node and the subtrees.

(1) An expression as operands under or overlaps (and)
(2) Any operator and operands under
(3) If it has an operator, it has one operator
(4) Either is an identifier or constant

A (non-empty) tree has:

Zero or more rooted trees called its subtrees, with no nodes or arcs in common with each other or the root.

One root node (and)
The root of each of the trees specified
under (q).

†‡
3. Solution to expression problem (2): The stack of activation records stores the subexpression values until they are used by the recursive evaluator.

2. The expression tree directly reveals the order of operations.

I. The expression tree easily finds and executes all the operations in the right order.

Remember about expression trees:
1. There is one special node, called the root.

2. Each node may be [is] associated with [zero] or more different nodes, called children. If a node c is the child of another node d, then we say that d is c’s parent.

3. Each node except for the root has exactly one parent; the root has no parent.

4. If you start at any node and move to the node’s parent (provided there is one), then move again to that node’s parent (provided there is one), and keep moving upward to each node’s parent, you will eventually reach the root.

Main/Savitch’s definition: A tree is a finite set of nodes. The set might be empty (no nodes), which is called the empty tree. But logically, “null” means “and” [if the set is not empty, then it is the set of nodes] must follow these rules:

University at Albany Computer Science Dept.

34
\[
\{ \\
\text{\texttt{halt()}\;}; \\
\text{\texttt{count \gg \texttt{temp} \gg \texttt{is\;the\;root}\;\gg \texttt{end};}} \\
\text{\{ \text{\texttt{temp = parent\;temp}} \};} \\
\text{\texttt{while\;\texttt{temp\;is\;NOT\;the\;root}} \;\} \\
\text{\texttt{temp = u;}} \\
\text{\texttt{node\;temp;}} \\
\} \\
\text{\texttt{find\;root\;node\;u}} \\
\]

The following algorithm, started at any node, always halts, eventually:

A more formal description of property :
halts when it is started on $n$.

The no-cycle property of trees: For every node $u$ the $\text{find-root}(n)$ algorithm

$$\begin{align*}
\text{return } & \text{find-root( parent( } n \text{ ) } \text{ ) } \text{; } \text{ } \\
\text{else} & \\
\text{ } & \{ \text{ } \\
\text{return } n; \text{ } \text{ } \\
\text{count } & \text{ } \geq n \text{ } \geq \text{ "is the root." } \text{ } \geq \text{ end}; \text{ } \\
( \text{ if } n \text{ is the root } ) \text{ } & \\
( \text{ node find-root( node } u \text{ ) } ) \text{ } \\
\text{Recursive root-finder} \end{align*}$$
The purpose of clause 4 of the definition and not clause 4 requires situations which satisfies clauses 1-3 relationship which satisfies clauses 1-3 parent of 1, and 3 is the parent of 4. This is an example of a parent-child relationship which satisfies clauses 1-3, 1 is the parent of 2, 4 is the parent of 3, 2 is the parent of 1, 4, 3, etc. where 2 is the unique root is 5.

The unique root is 5.
is c's parent.

Let's child and its right child. If a node c is the child of node p, we say: "c is p's child." Each node may be associated with up to two other different nodes, called its

Replace M/S tree definition clause (2) by:

(Left and Right)

Binary Tree (in brief, each node's children are distinguishable from one another as
Complete Binary Tree: Every non-leaf has exactly 2 children.

Full Binary Tree: Every leaf has the same depth.

William, Cormen, Leiserson, Rivest, and others call this the height (illumination, corresponding, maximum, etc.). Better authors write for more sophisticated readers (like Knuth, Aho, Hopcroft, Etc). But all of these leaves.

Depth of a whole tree: Maximum depth of any of its leaves.

Depth (root) = zero.

Depth of a node: Number of parent-to-child steps from the root to this node.

Left and right subtrees of a node.

Consisting of this new root’s descendants (i.e., subtree (we can view any non-root node as the root of a new, smaller tree).

Parent, Sibling, Ancestor, Decendant.

No children.

Tell a node with computer sci. examinations and job placement interviews: Leave this one.
Recursive algorithm to compute the depth of a node:

```java
int depth(node n) {
    if (n is the root) return 0;
    else return 1 + depth(parent(n));
}
```
You can simplify this by omitting the if test:

```java
if (n is a leaf) {
    return max;
} else {
    int max_left = max_height(node(n) .
    for each of the children nodes c of n
    int max_right = max_height(c);
    max = max(max_left, max_right);
    return max;
}
```

Recursive algorithm to compute the height of the subtree with the given root:
Example of an expression and its Parse Tree

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```

```plaintext
Example of an expression and its Parse Tree
```
The top level operator is multiplication ($\ast$).

Diagram:

```
   G
 /    \
(\ast)  F
```
(C-(D*(E++)))

The top level operator is subtraction.

(D*(E++))

The top level operator is multiplication.

(E++)

The top level operator is increment.
The following C/C++ expression has an expression tree that is binary but not complete binary:

\[(9 * (++)(XX*)))\]
Binary Search Trees: A decision tree for answering whether or not a given question (which is in a finite set) by using questions of the form: "Is it in the set?" or "Is it equal to?" and number is in each node, not the question.

"Is it in the set?" or "Is it equal to?" and number is in each node, not the question.

Binary Search Trees: A decision tree for answering whether or not a given question (which is in a finite set) by using questions of the form: "Is it in the set?" or "Is it equal to?" and number is in each node, not the question.

(Binary) Decision Trees: Each leaf is an answer, each non-leaf is a yes-no question, (M/S counts trees with taxonomy trees).

(46) Other Name Space Trees: EC, the Domain Name System of the Internet

File Name Trees: Express a system to identity files using a sequence of directory names, plus a file name.

Expression Trees: Express the structure of the computation expressed by an expression (string, web document, program, etc.).


Tree Examples/Applications

University at Albany Computer Science Dept.
For the tree and each subtree $T$, the root contains the largest of the numbers in $T$. A heap-ordered tree ("heap") is a tree of numbers with the heap property:

- The value of each node is greater than or equal to the values of its children (or equal to the value of its only child).
The worst case use an amount of computer work proportional to the height of the tree, in the insertion into a heap, and search and insertion in a binary search tree.

Main conclusion:

number into a heap ordered tree.

We then viewed Main/Savitch's page 520 and described how to insert a new search tree.

example of how a dictionary of states (of the USA) is implemented by a binary search and inserted a new number into it.

We viewed Main's slide set 10b on the dictionary abstract data type and the search tree.

We viewed Main/Savitch's page 498 on a binary search tree and described how to search and inserted a new number into it.

We viewed Main/Savitch's page 470 on a decision tree.