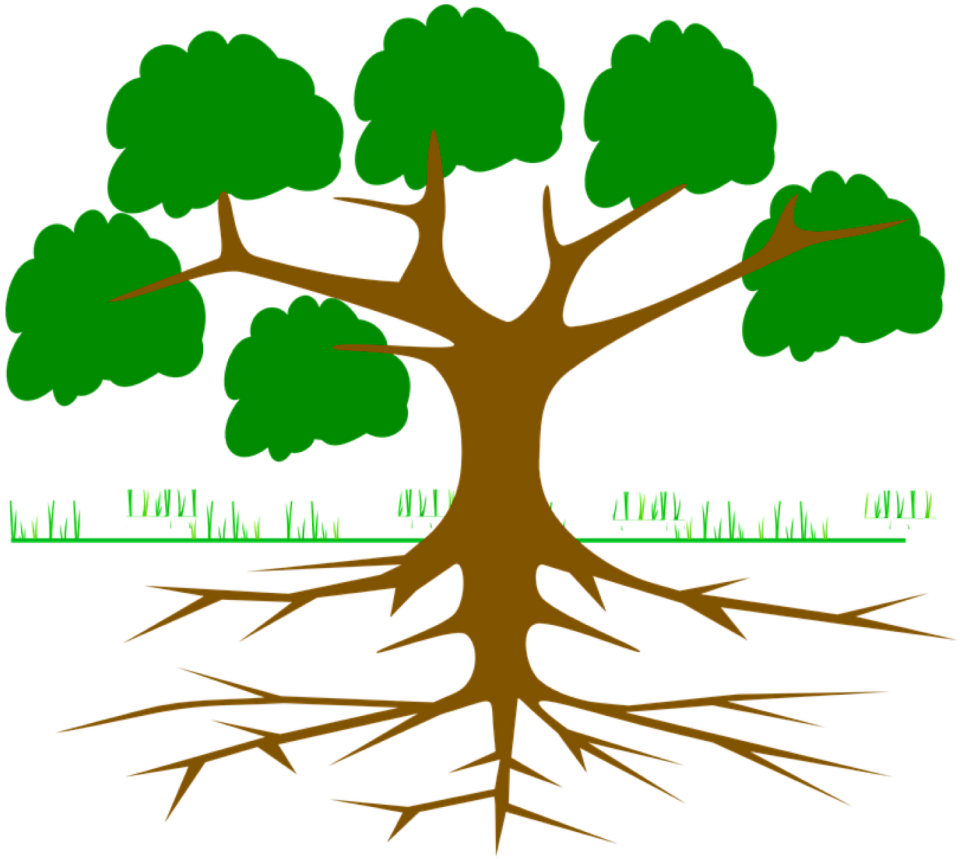
# Binary tree algorithms

Simon Powers

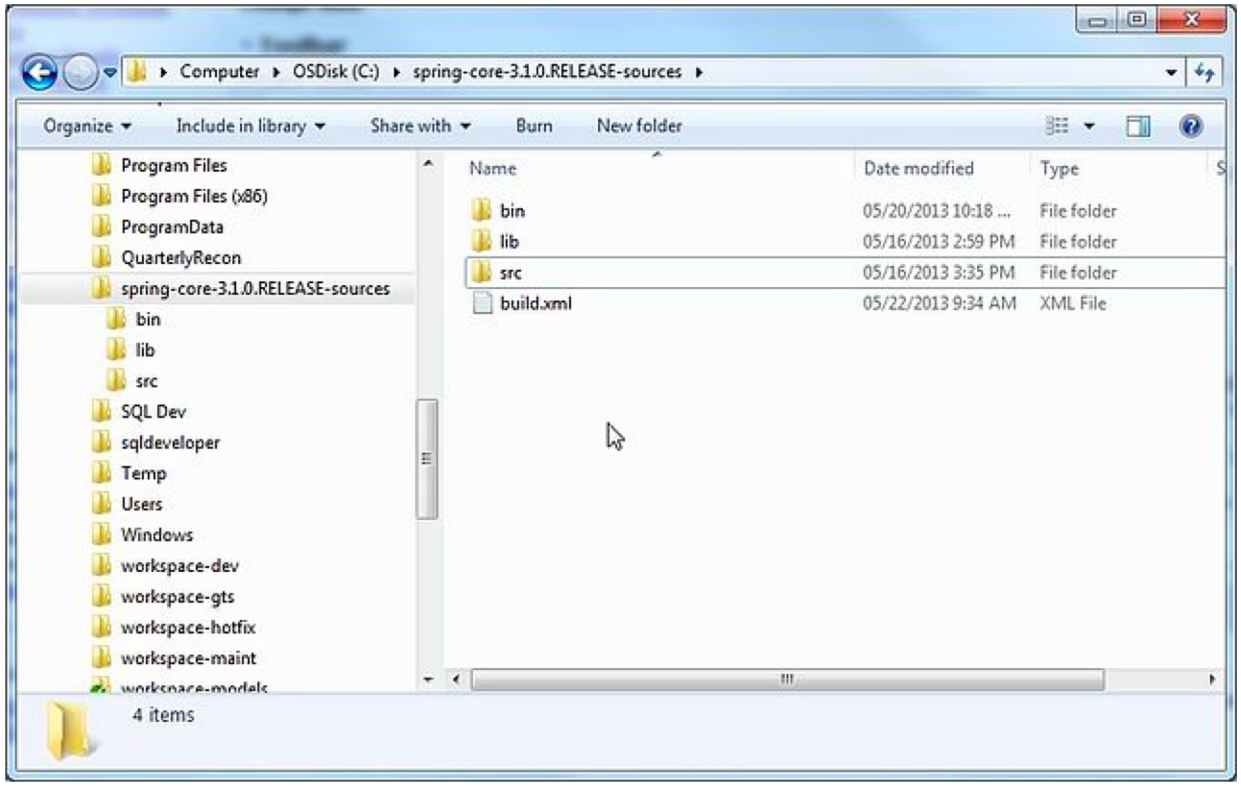
SET08122 – Algorithms and Data Structures

# Trees: hierarchical data structures



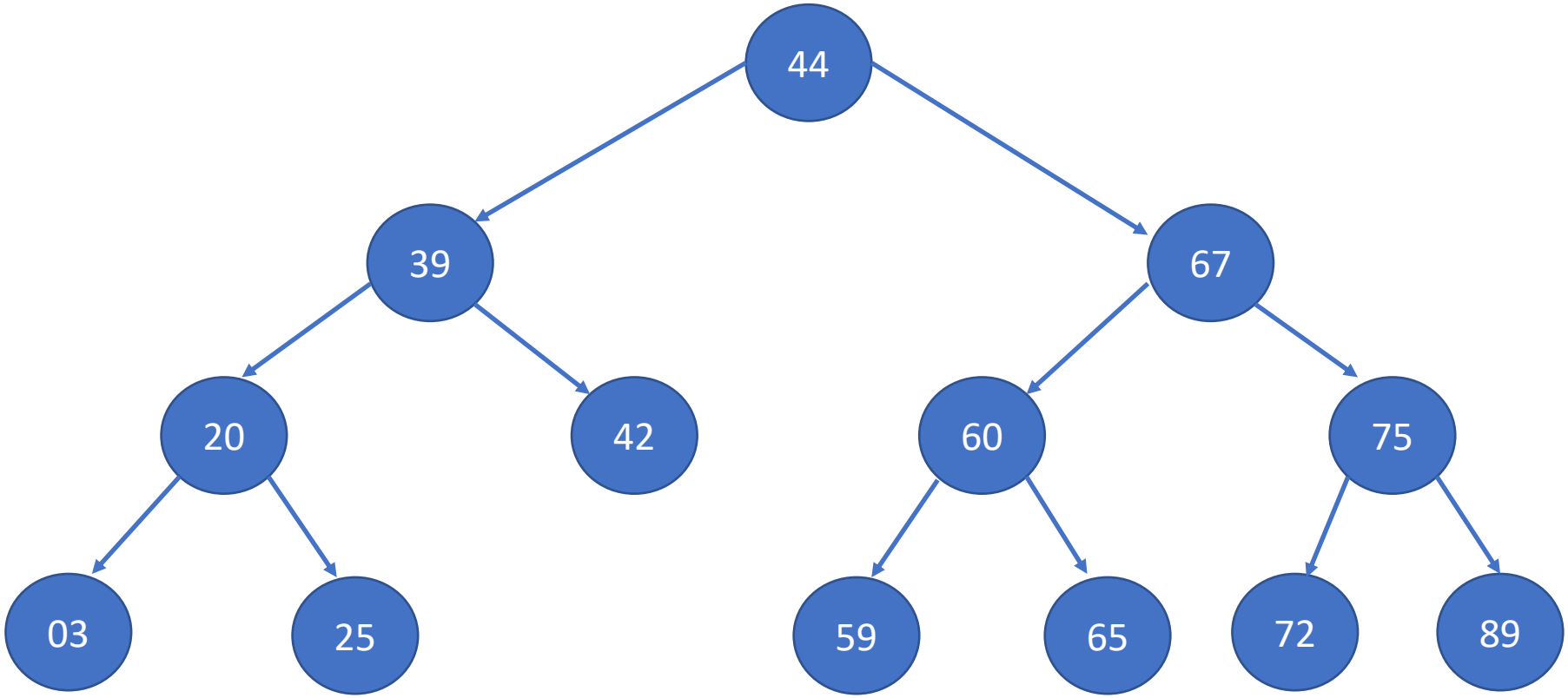
# Trees: hierarchical data structures

# Trees: hierarchical data structures

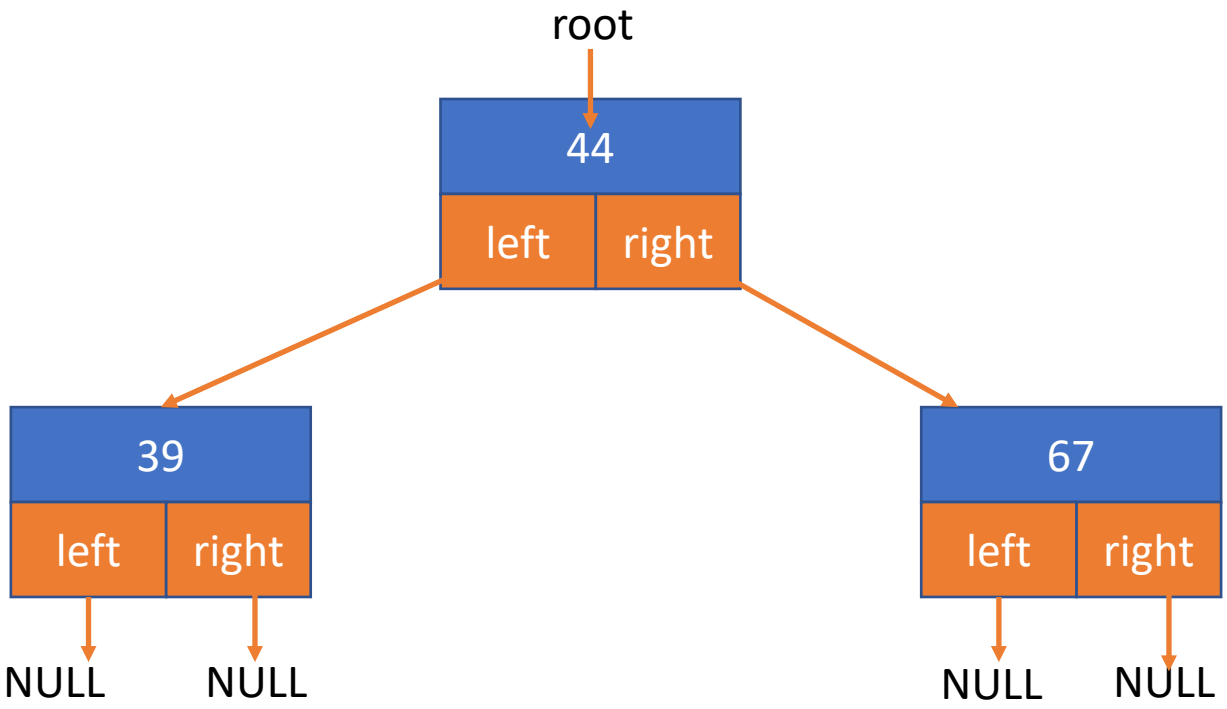


# Trees: more formally

- A tree consists of *nodes*.
- The node at the top of the hierarchy is the *root*.
- The nodes below the root are its *children*, which in turn have children of their own.
- Each node has one parent.
- But a node can have many children – the maximum number is the tree's branching factor.


44
39
67
20
42
60
75
03
25
59
65
72
89

# A tree is implemented with nodes




```
struct Node {
        int      data;
        Node    *left;
        Node    *right;
};
```

# Visiting the nodes of a tree

- There is no “natural” order to visit nodes.
- A tree is recursively composed of many smaller subtrees.

# Traversing trees recursively

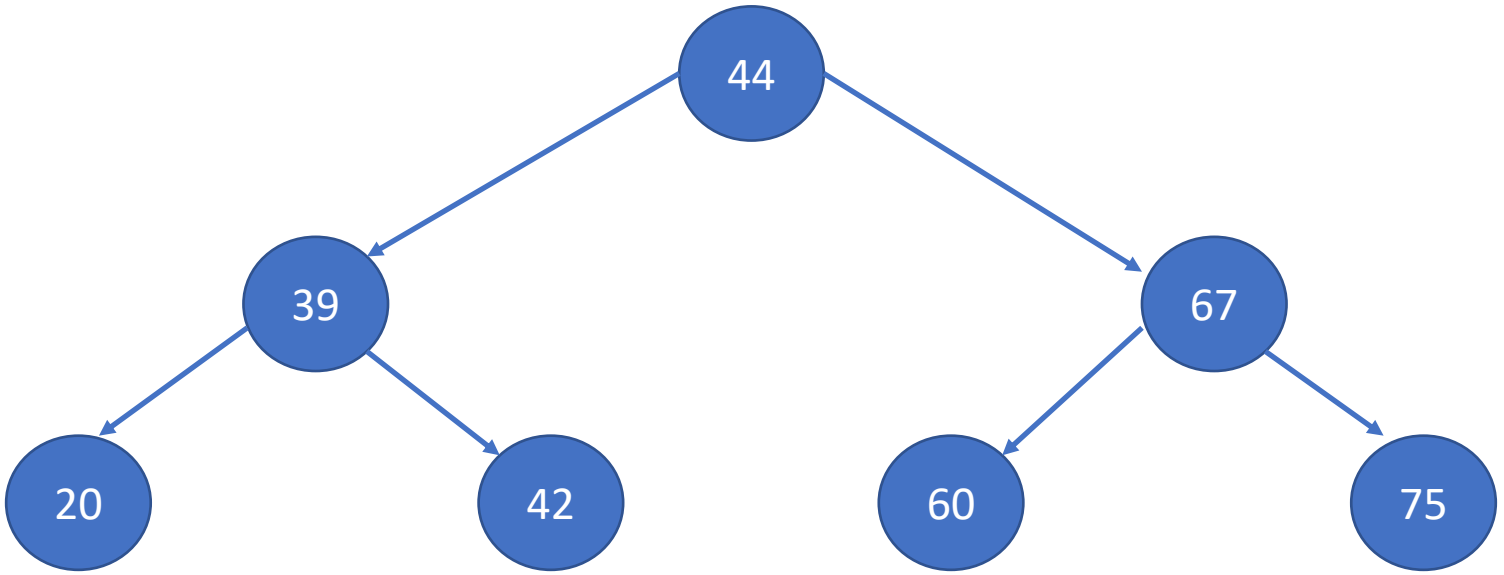
# Preorder traversal

Preorder: Visit the root

Visit the left subtree

Visit the right subtree

```
void pre_order(struct Node *root){
    if(root == NULL)
        return;
    printf("%d ", root->data);
    pre_order(root->left);
    pre_order(root->right);
}
```



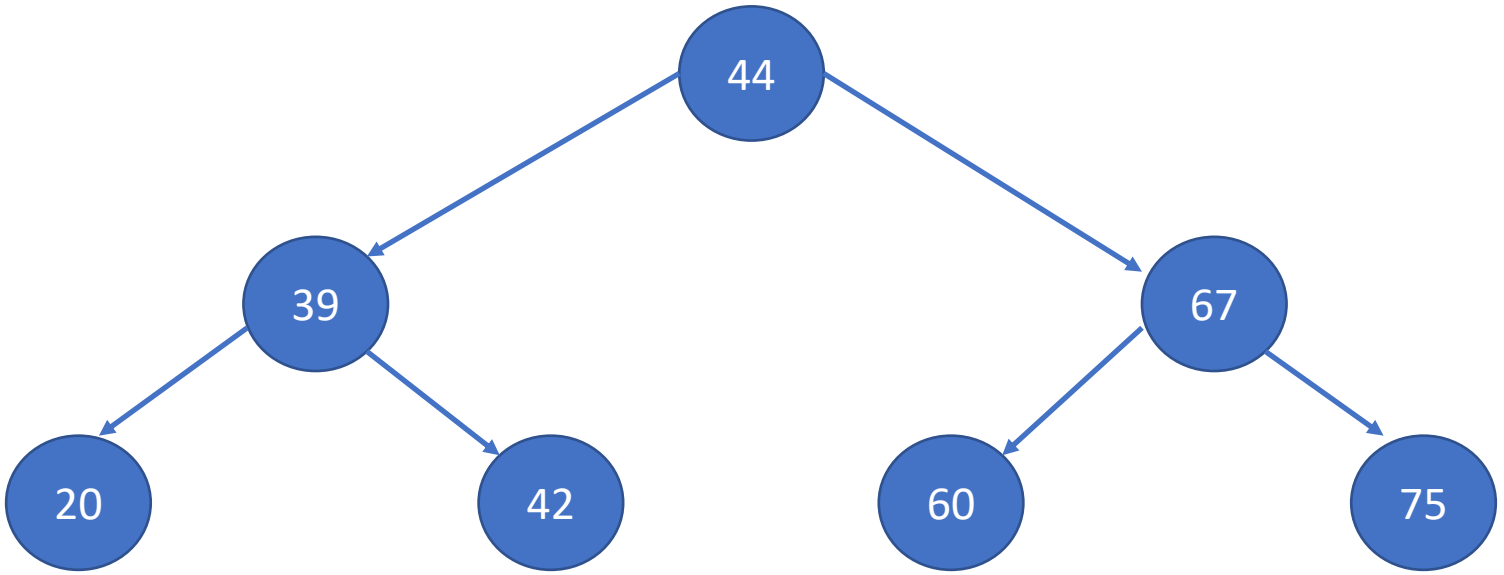
# Preorder traversal

Preorder: Visit the root

Visit the left subtree

Visit the right subtree

```
void pre_order(struct Node *root){
    if(root == NULL)
        return;
    printf("%d ", root->data);
    pre_order(root->left);
    pre_order(root->right);
}
```




44, 39, 20, 42, 67, 60, 75

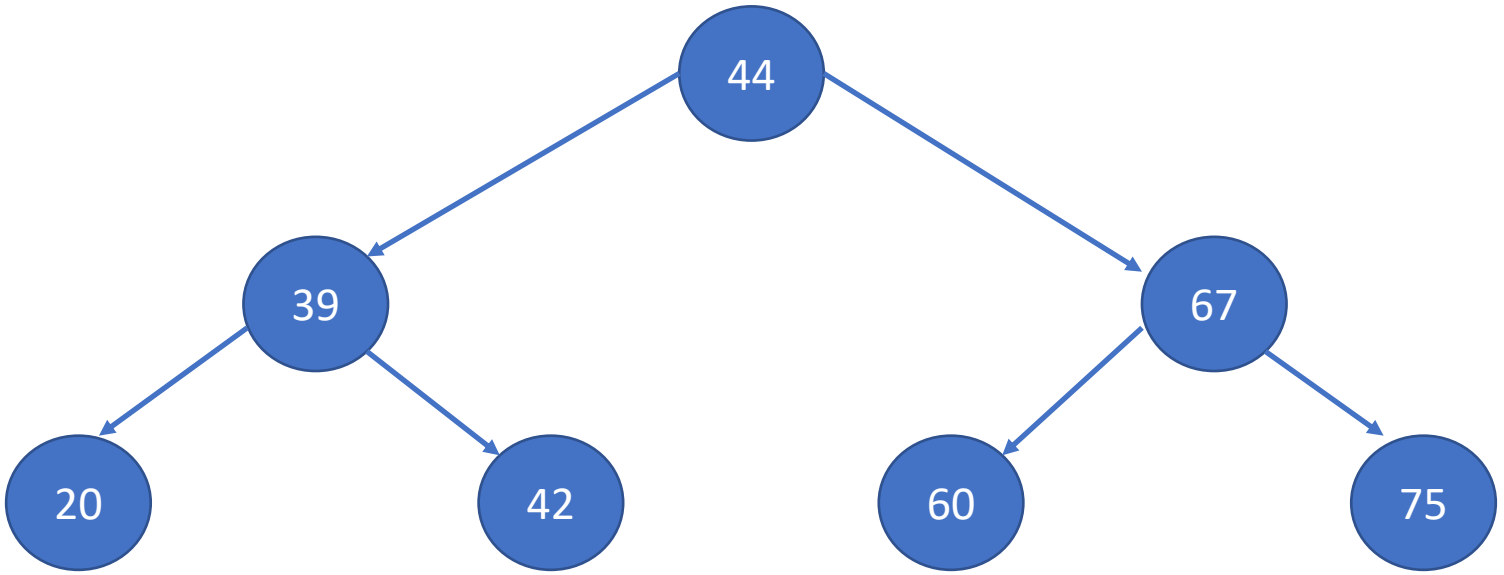
# Inorder traversal

Preorder: Visit the left subtree

Visit the root

Visit the right subtree

```
void in_order(struct Node *root){
    if(root == NULL)
        return;
    in_order(root->left);
    printf("%d ", root->data);
    in_order(root->right);
}
```



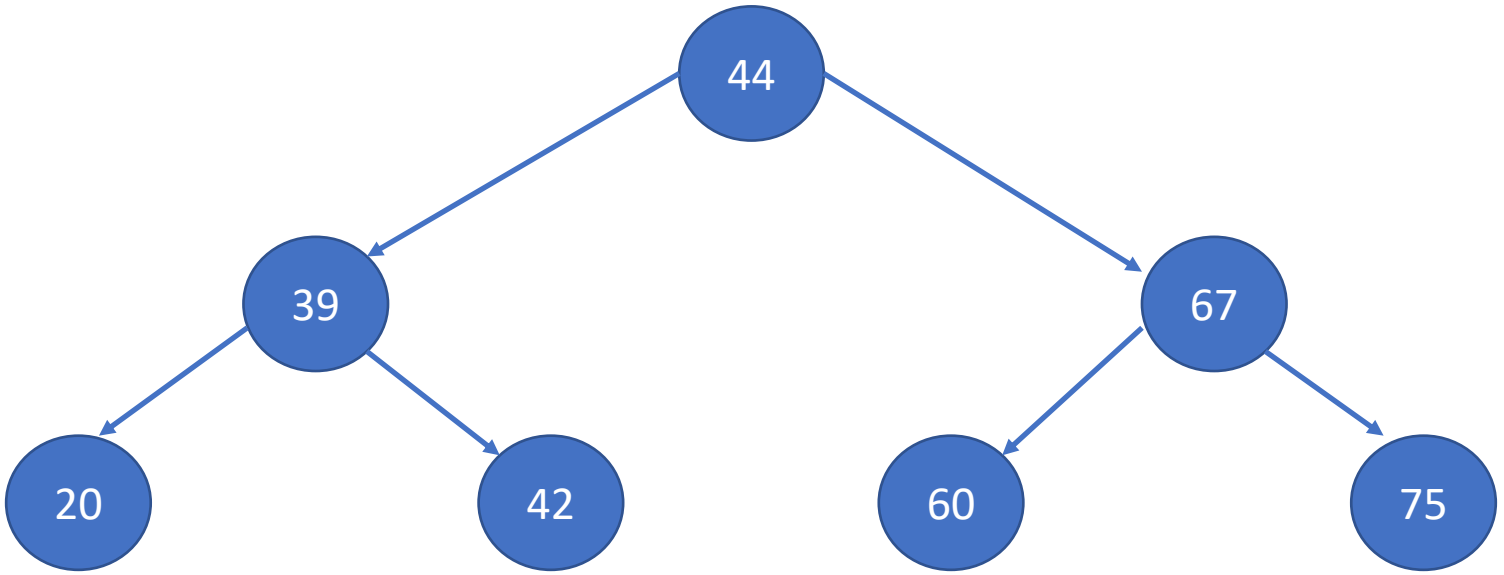
# Inorder traversal

Preorder: Visit the left subtree

Visit the root

Visit the right subtree

```
void in_order(struct Node *root){
    if(root == NULL)
        return;
    in_order(root->left);
    printf("%d ", root->data);
    in_order(root->right);
}
```




20, 39, 42, 44, 60, 67, 75

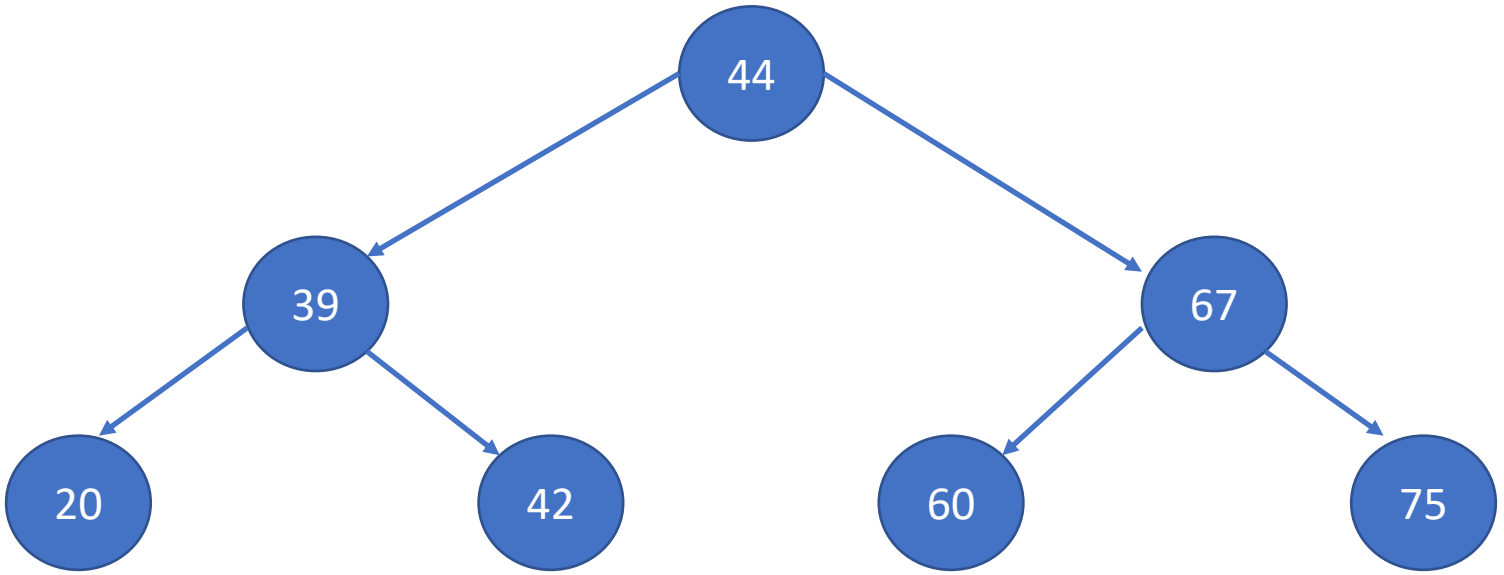
# Postorder traversal

Preorder: Visit the left subtree

Visit the right subtree

Visit the root

```
void post_order(struct Node *root){
    if(root == NULL)
        return;
    post_order(root->left);
    post_order(root->right);
    printf("%d ", root->data);
}
```



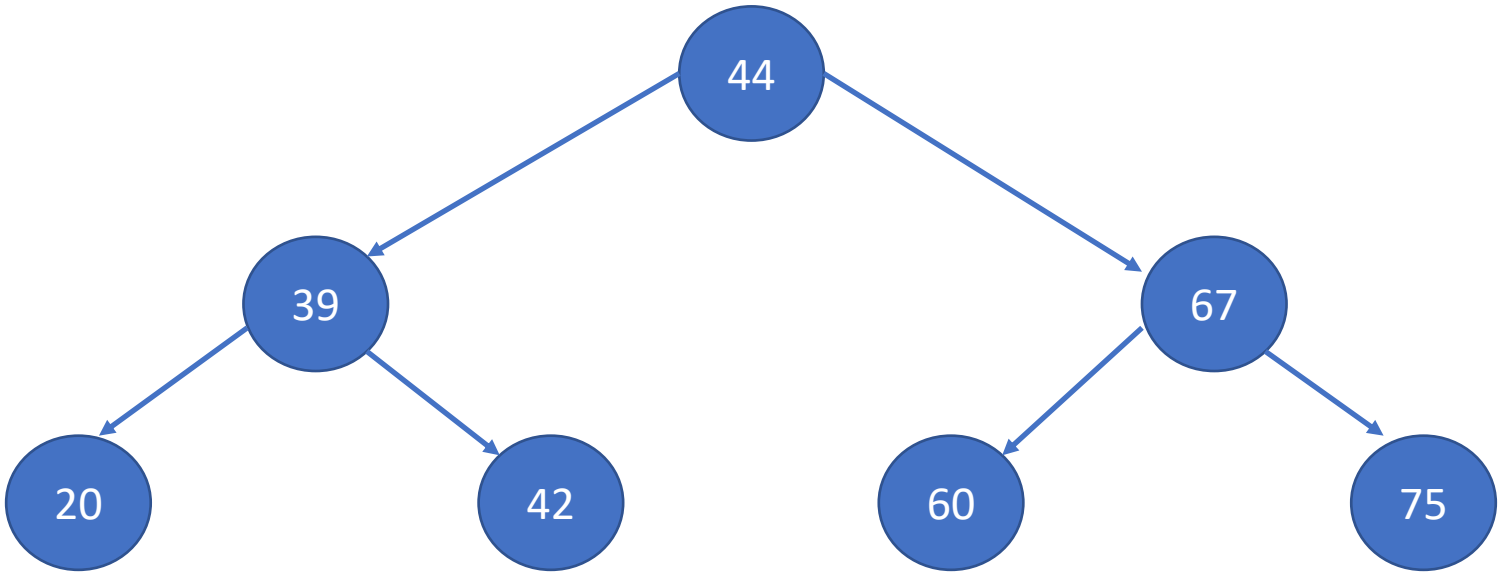
# Postorder traversal

Preorder: Visit the left subtree

Visit the right subtree

Visit the root

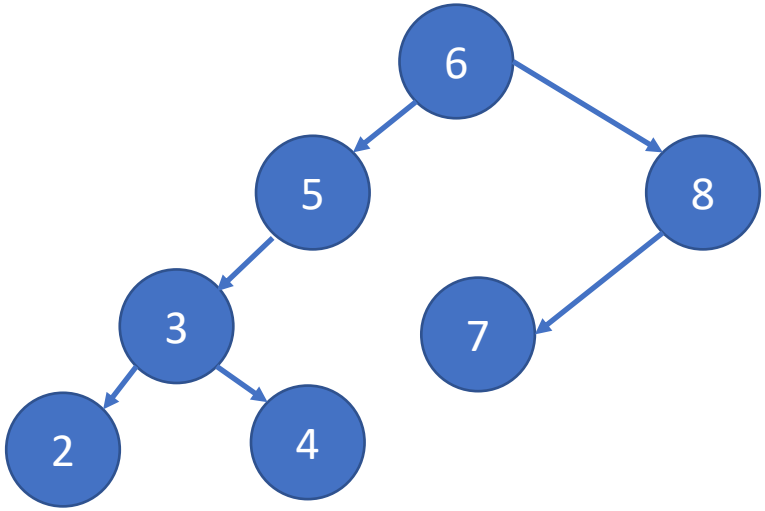
```
void post_order(struct Node *root){
    if(root == null)
        return;
    post_order(root->left);
    post_order(root->right);
    printf("%d ", root->data);
}
```




20, 42, 39, 60, 75, 67, 44

# The height of a tree

- For a tree with just one node, the height is 0.
- The height of a null tree is -1.
- The height of a tree with two levels is 1.
- Write a recursive algorithm to calculate the height of a tree.




Height = 3

# The height of a tree

```
int height (struct Node *root) {
      if(root == NULL)
            return -1;
      int hl = height(root->left);
      int hr = height(root->right);
      return 1 + max(hl, hr);
}
```

# Expression trees

# Expressions

- Consider A * (B + C) / D.
- This is written in infix notation. Problem: This needs rules about precedence in the language, and brackets to override these.

# Expressions

- Consider A * (B + C) / D.
- This is written in infix notation. Problem: This needs rules about precedence in the language, and brackets to override these.
- Prefix (Polish) notation: / * A + B C D
  - (/ (* A (+ B C)) D). Operators written before their operands, evaluated left to right. No brackets needed.

# Expressions

- Consider A * (B + C) / D.
- This is written in infix notation. Problem: This needs rules about precedence in the language, and brackets to override these.
- Prefix (Polish) notation: / * A + B C D
  - (/ (* A (+ B C)) D). Operators written before their operands, evaluated left to right. No brackets needed.
- Postfix (Reverse Polish) notation: A B C + * D /
  - ((A (B C +) *) D /). Operators written after their operands, evaluated left to right. No brackets needed.

# Converting between the notations

- Insert all of the implicit brackets showing the order of evaluation:

(A * (B + C)) /D)

# Converting between the notations

- Insert all of the implicit brackets showing the order of evaluation:

((A * (B + C)) / D)

- To get a prefix expression, simply move each operator to beside its left bracket:

(/(* A (+ B C))D)

# Converting between the notations

- Insert all of the implicit brackets showing the order of evaluation:

(A * (B + C)) / D)

- To get a prefix expression, simply move each operator to beside its left bracket:

(/(* A (+ B C))D)

- To get a postfix expression, simply move each operator to beside its right bracket:
- (A (B C +) *) D /)

# Evaluating a postfix expression

- To evaluate a postfix expression, we apply each operator to the operands immediately preceding it.
- They are easy to evaluate using an abstract stack machine.
  - Used by compilers and hand held calculators.

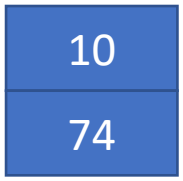

# Algorithm to evaluate a postfix expression

**While** not at end of string **do**
Move from left to right through the string
**If** value is an operand **Then**
    push it on the stack
**Else**
    pop the number of operands required by the operator
    apply the operator to them
    push the result onto the stack
**End while**
pop result from stack and return it

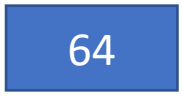
# Algorithm to evaluate a postfix expression

**While** not at end of string **do**
Move from left to right through the string
**If** value is an operand **Then**
    push it on the stack
**Else**
    pop the number of operands required by the operator
    apply the operator to them
    push the result onto the stack
**End while**
pop result from stack and return it

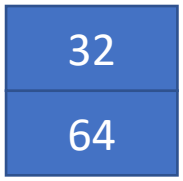
Exercise: Evaluate the following postfix expression, showing the content of the stack at each stage
74 10 – 32 / 23 17 + x




Push 74
Push 10



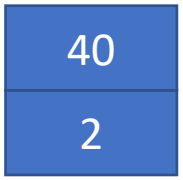

Pop 10
Pop 74
Apply –
Push 64




Push 32


Pop 32
Pop 64
Apply /
Push 2


Push 23
Push 17




Pop 17
Pop 23
Apply +
Push 40


Pop 40
Pop 2
Apply x
Push 80

# Stack machines

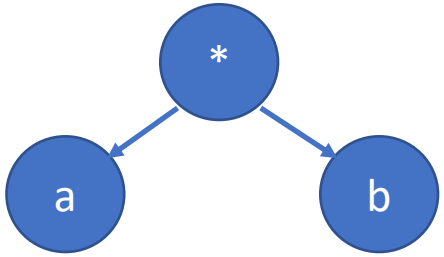
- A type of (virtual) computer where most instructions operate on a stack rather than registers.
- Instructions take their operands from the top most values on the stack.
- Very fast to access operands.
- Example: The Java Virtual Machine.
  - Java Optimized Processor

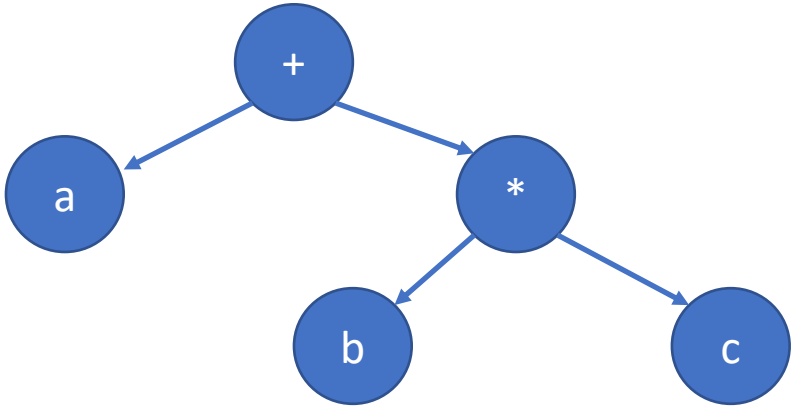
# Evaluating a prefix expression

- Easier to evaluate the string from right to left.
- You can then push operators onto the stack as before, pop them when you encounter an operator, apply the operator to them, and push the result back onto the stack.
- More fiddly to process from left to right.
- Any ideas?

# Expression trees

- A compiler uses a binary tree to represent an arithmetic expression.
- Expression tress contain operators and terminal values.



# Building an expression tree

Input: expression string in postfix form
**While** not at end of string **do**
**If** token is an operand, **Then**
    create a new leaf node storing its value.
    set left and right child pointers to null
    push the new node onto stack
**Else**
    create a new node storing the operator
    pop a node from the stack, attach to right child pointer
    pop a node from the stack, attach to left child pointer
    push the new node onto the stack
**End while**
Return single item on stack -- this is the root of the tree

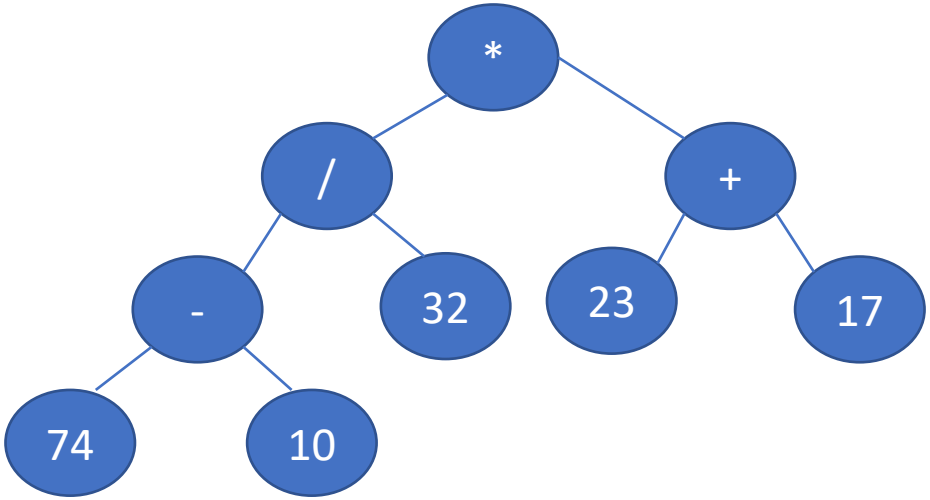
# Building an expression tree

Input: expression string in postfix form
**While** not at end of string **do**
**If** token is an operand, **Then**
    create a new leaf node storing its value.
    set left and right child pointers to null
    push the new node onto stack
**Else**
    create a new node storing the operator
    pop a node from the stack, attach to right child pointer
    pop a node from the stack, attach to left child pointer
    push the new node onto the stack
**End while**
Return single item on stack -- this is the root of the tree

Exercise: Produce the expression tree for the following expression string

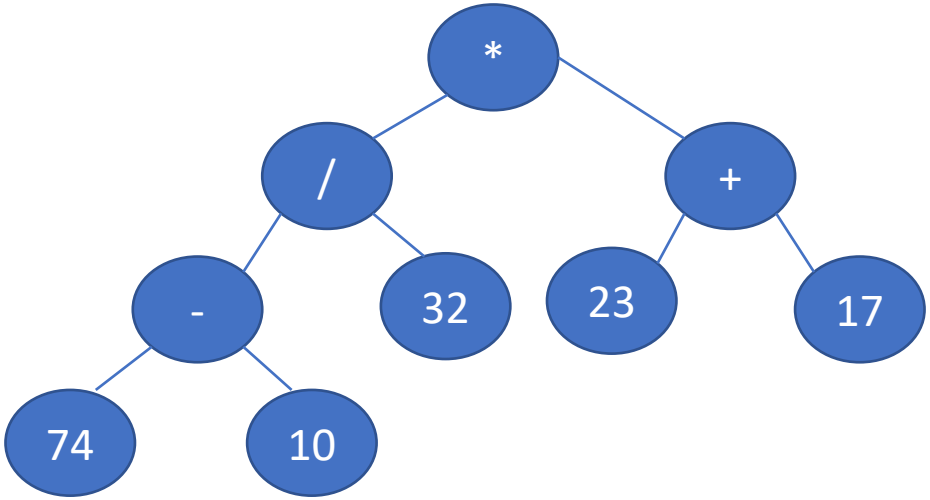
74 10 - 32 / 23 17 + *

# Traversing an expression tree




- Preorder produces a prefix expression.
- Inorder produces an infix expression.
- Postorder produces a postfix expression.

# Traversing an expression tree




- Preorder: * / - 74 10 32 + 23 17
- Inorder: 74 – 10 / 32 * 23 + 17
- Postorder: 74 10 – 32 / 23 17 + *

# Modified inorder algorithm

- Need to modify our inorder traversal to preserve the correct order of evaluation.

```
void in_order_expression(struct Node *root){
   if(root == NULL)
      return;

   if(isOperator(root->data))
       printf(“(“);

   in_order(root->left);
   printf(“%d ”, root->data);
   in_order(root->right);

   if(isOperator(root->data))
       printf(“)“);
}
```

# Code generation

- Traverse the tree in preorder.
- Consider our favourite expression

* / - 74 10 32 + 23 17

# Code generation

- Traverse the tree in preorder.
- Consider our favourite expression

*/ - 74 10 32 + 23 17

MUL(DIV(SUB(74, 10),32),ADD(23,17))

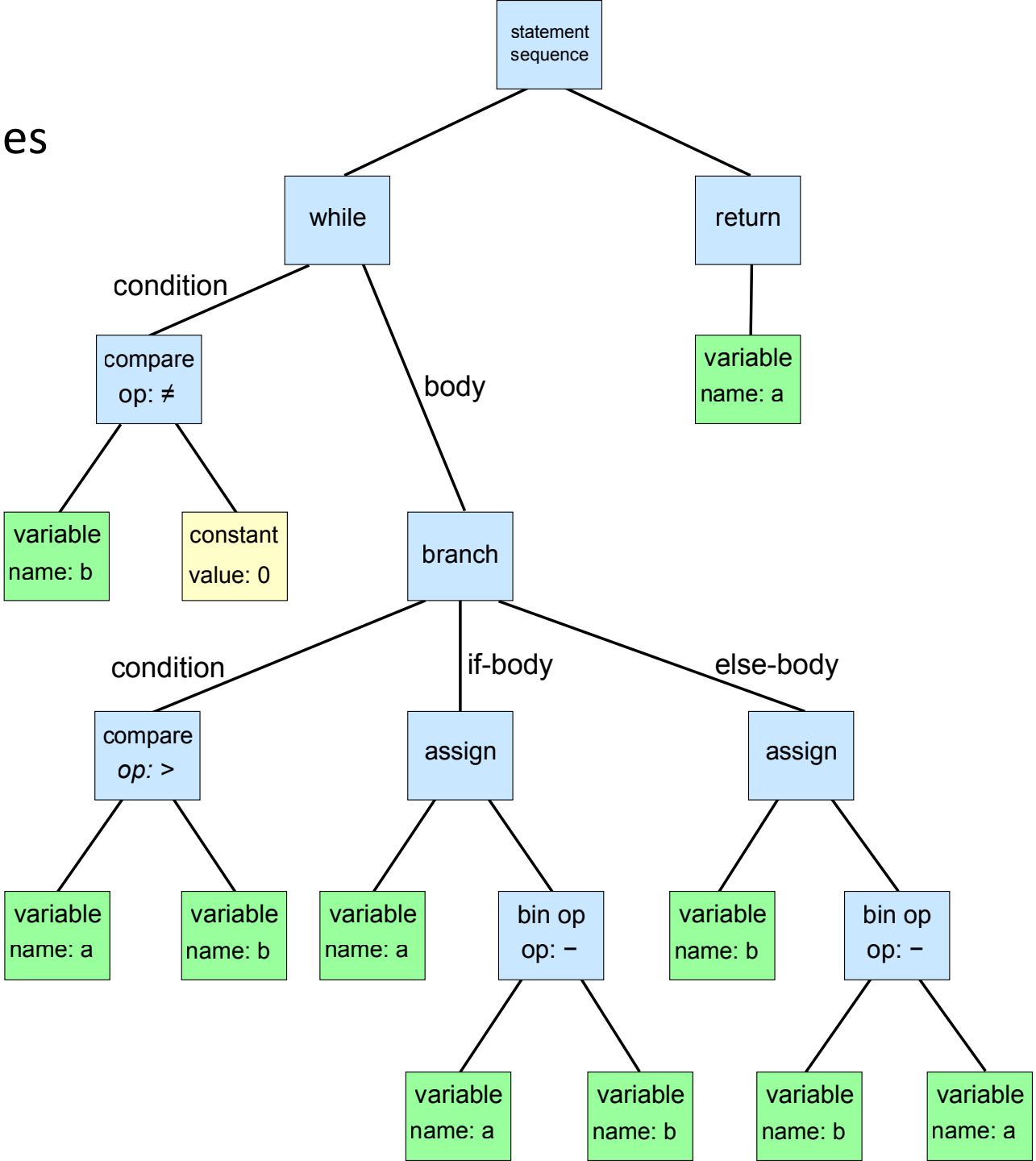
- This why the compiler has to build a tree – the processor doesn’t want your expressions in infix notation.

# Code generation

- Need to allocate registers during the preorder traversal (a stack machine doesn't have to do this).

```
void generate_code(Node *root, int resultRegNum) {
  if (is_operator(root->data)) {
        generate_code(root->left, resultRegNum);
        generate_code(root->right, resultRegNum + 1);
        generate_op_instruction(root->data, resultRegNum,
resultRegNum + 1);
   }
   else {
      generate_load_instruction(root->data, resultRegNum);
   }
}
```

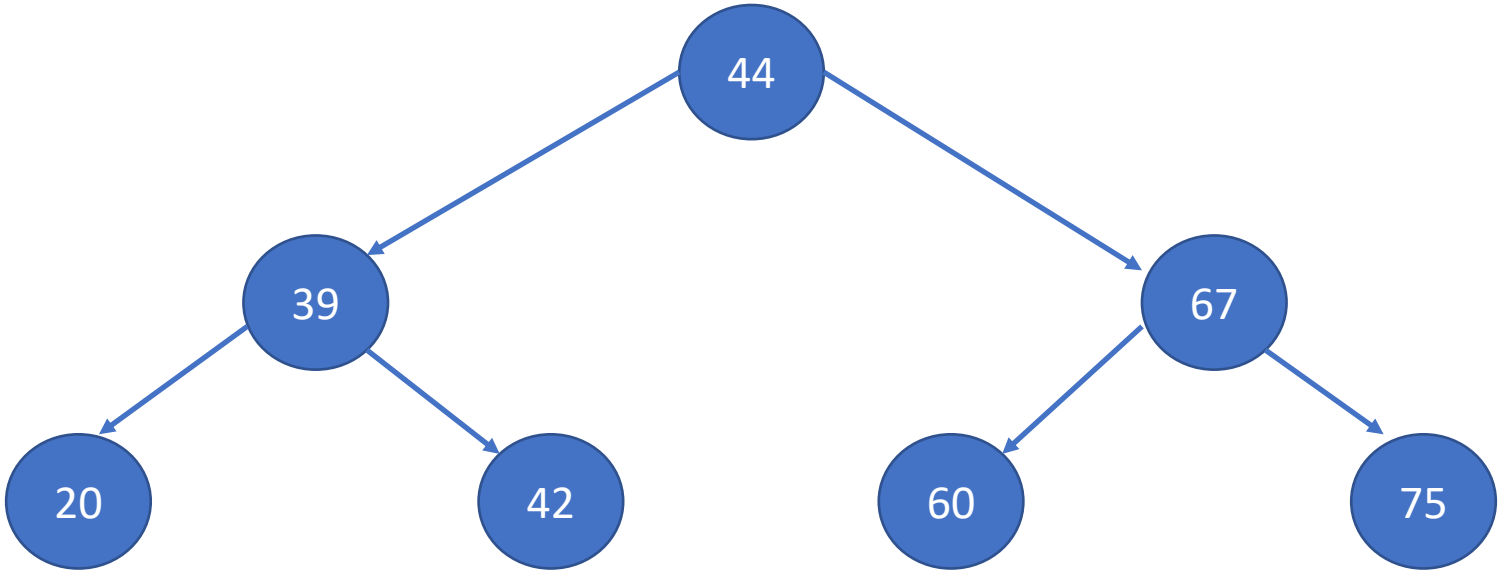
# Abstract Syntax Trees



# Binary Search Trees

# Binary Search Trees

- Organise data for efficient searching.
- Everything to the left of the root is smaller than it.
- Everything to the right of the root is greater than it.
- Apply this rule recursively.

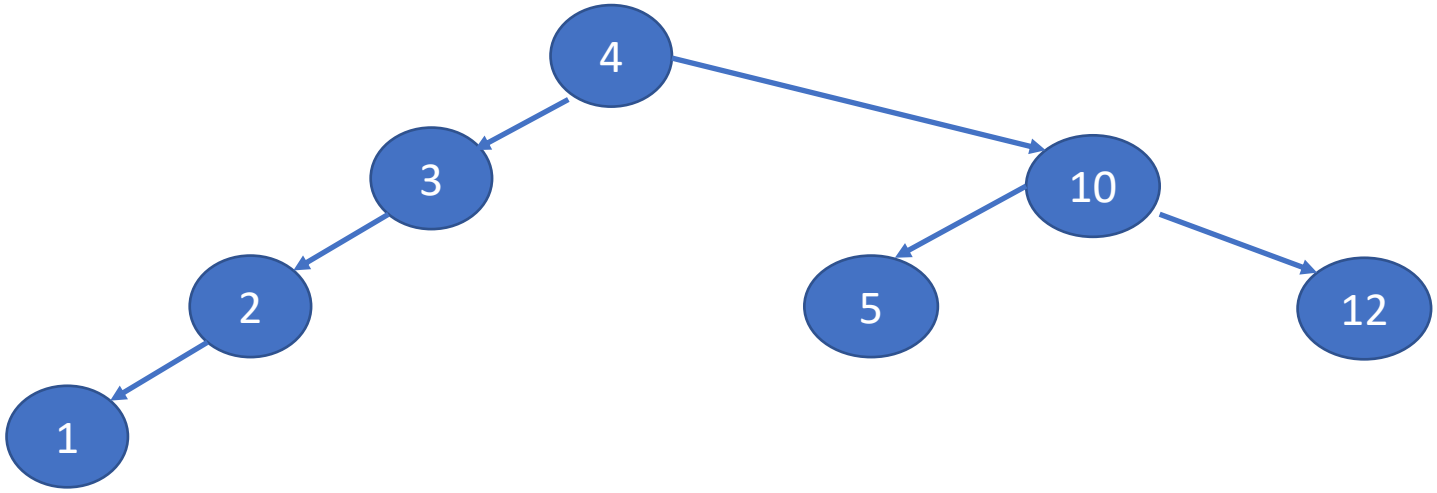


# Checking if a tree contains an item

```
boolean contains(struct Node *root, int target) {

  if (root == NULL) {
    return false;
  }

  else {

    if (target == root->data)

        return true;
    else {
      if (target < root->data)

         return contains(root->left, target);
      else

         return contains(root->right, target);
    }
  }
}
```

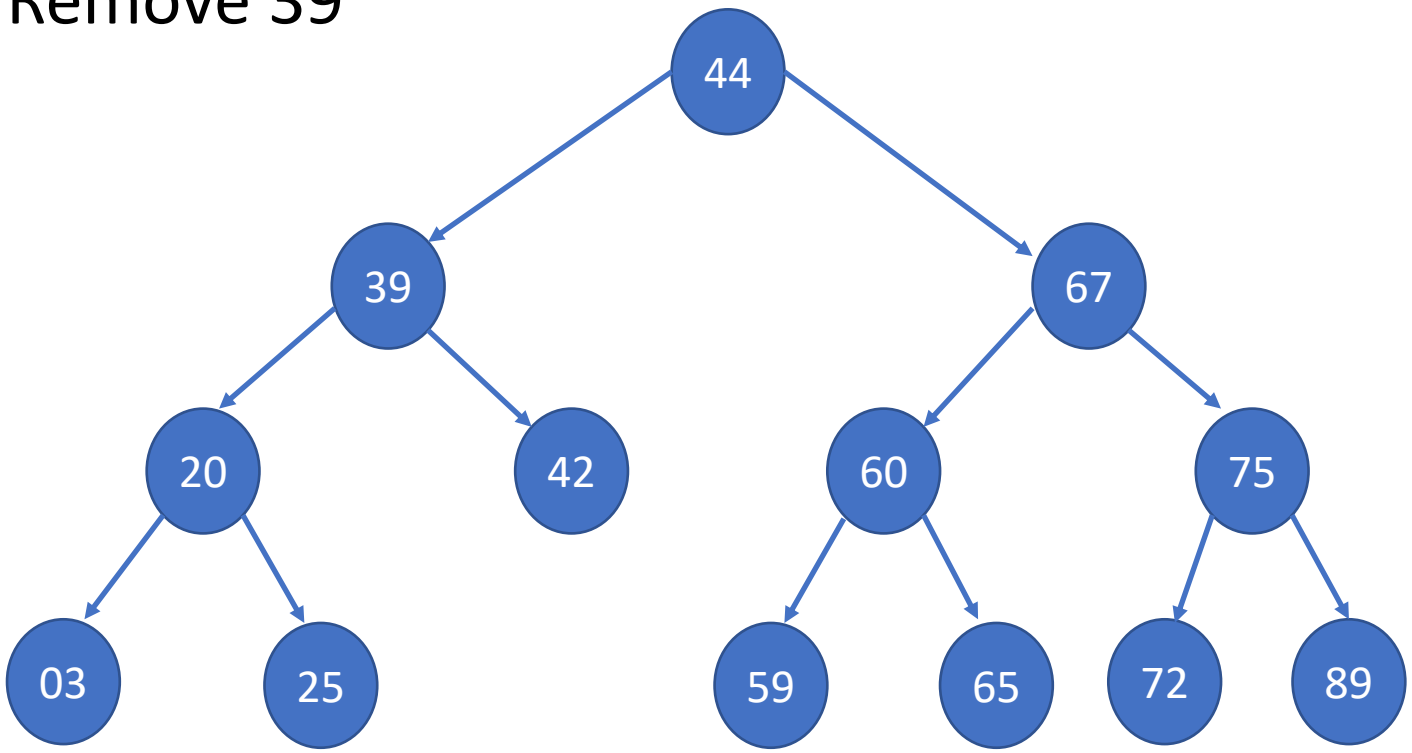
# Inserting an item

```
private void insert(struct Node **root, int data) {
    if (*root==NULL) {
      *root = (struct Node*)malloc(sizeof(struct Node));
      (*root)->data = data;
      (*root)->left = NULL;
      (*root)->right = NULL;
    }
    else {
      if (data < root->data) {
        insert(&(*root)->left, data);
     }
      else {
        insert(&(*root)->right, data);
      }
    }
}
```


4
3
10
2
5
12
1

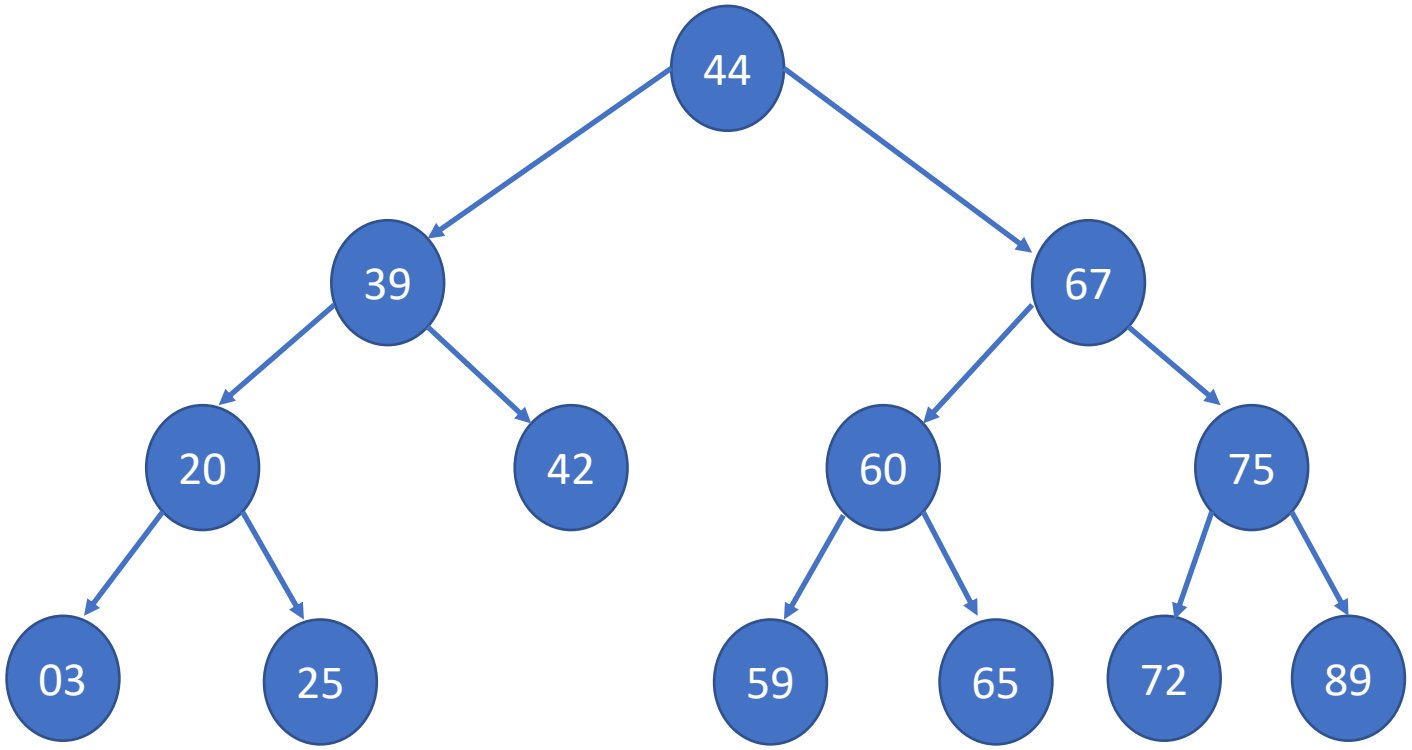
# Removing an item

- Remove 39



# Removing an item

- Could replace 39 with either 25 or 42 and still have a valid binary search tree.

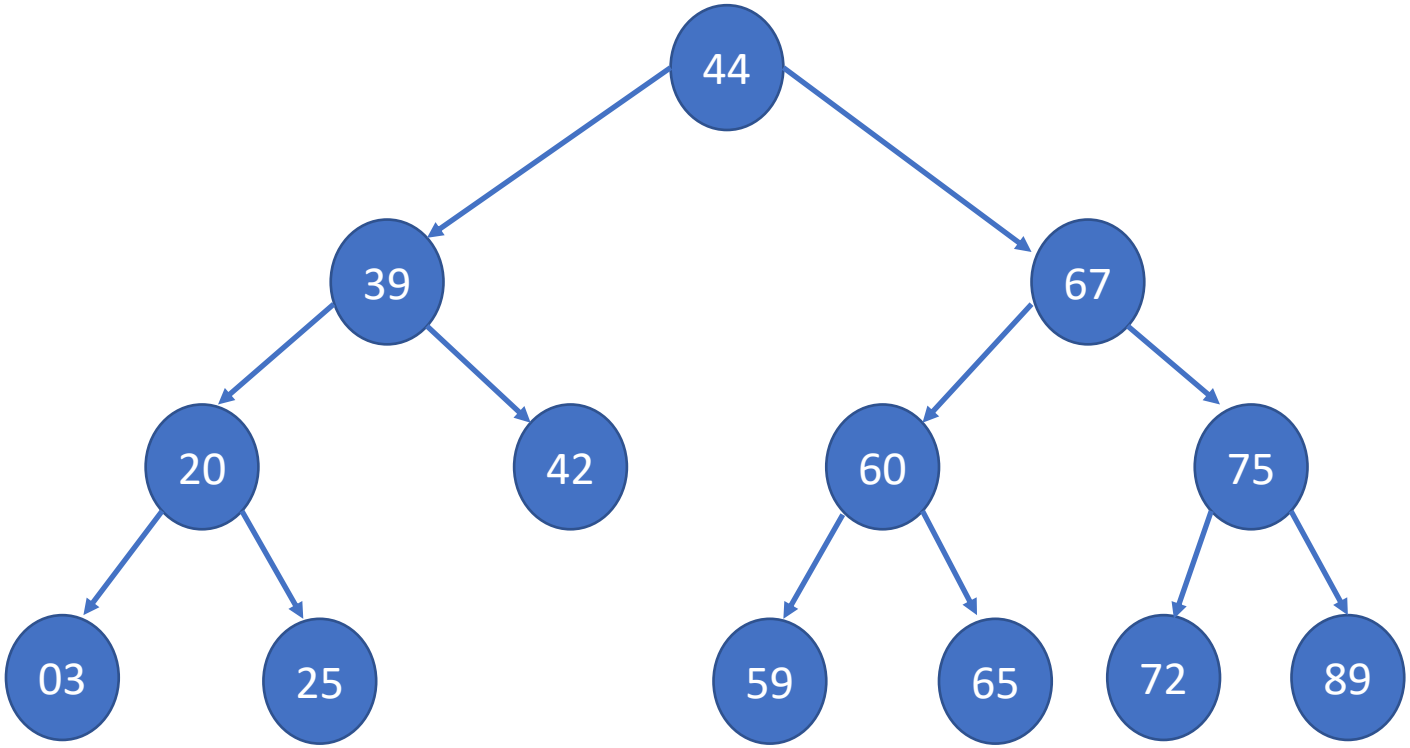


# Removing an item

- Case 1: The node is a leaf, just delete it.
- Case 2: The node has one child, replace the node with that child.
- Case 3: The node has two children, replace it with either the largest node in its left subtree, or the smallest node in its right subtree.

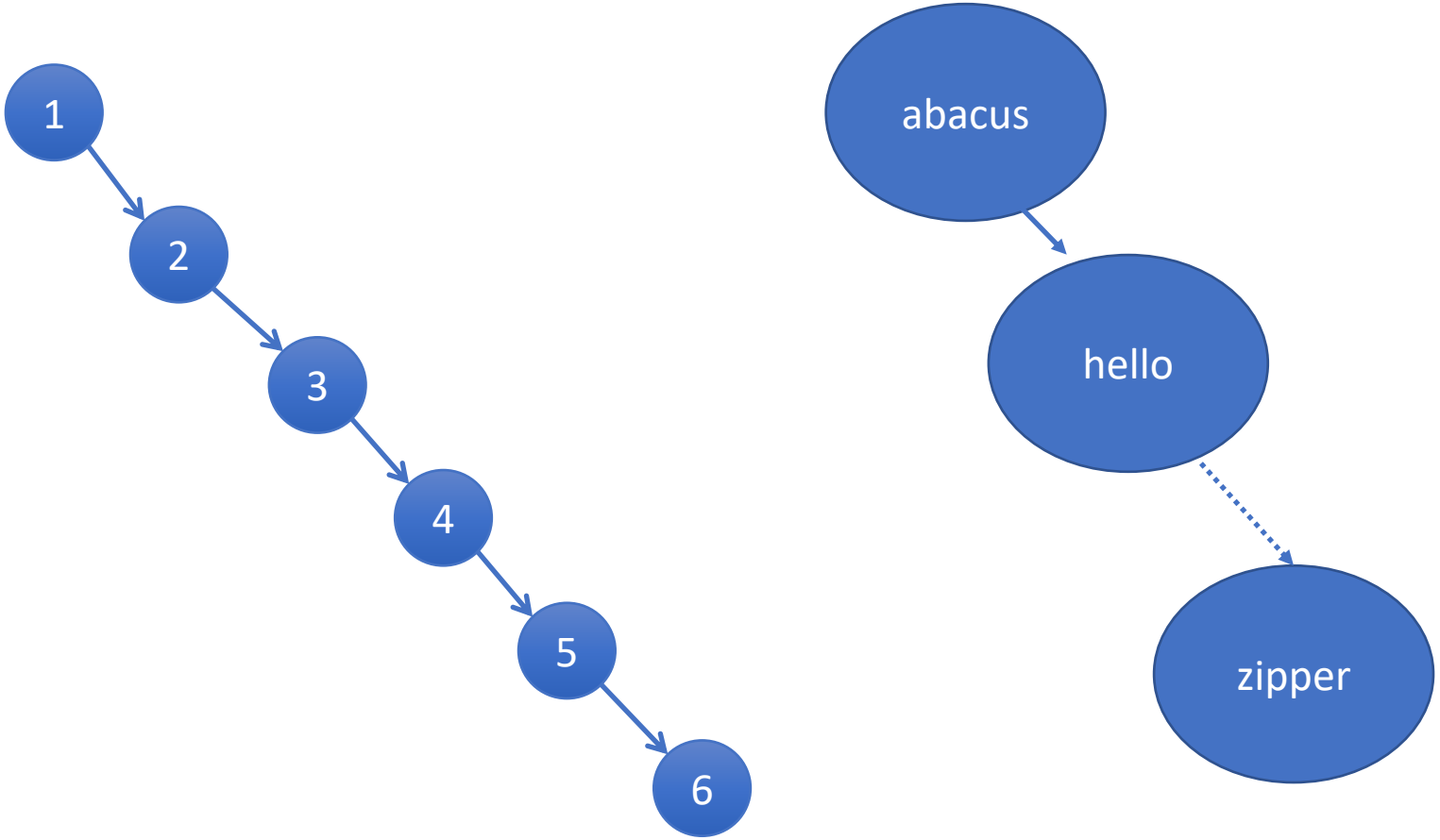
# Efficiency

- In the worst case we only end up searching one branch.
- If the tree is balanced, will be $O(\log_2 n)$.



# Efficiency

- If unbalanced, can drop to O(*n*).



# Balancing a tree

- More difficult than it seems.
- Can’t rely on insertion order.
- Sophisticated approaches include AVL trees and Red Black trees.
- These modify the tree on insertion and removal (if necessary) to keep it balanced.

# We've covered…

- Trees are very widely used in computer science.
- A tree consists of nodes where each node has a single parent.
- Binary trees are useful for representing expressions.
  - And evaluating them using an Abstract Stack Machine.
- Binary search trees can provide $O(\log_2 n)$ searching if correctly balanced.
  - Otherwise can drop to $O(n)$.

# Data Compression

# Data compression

- Reduce the number of bits used to represent data.
- All data can be characterised by its entropy.
- Compression is possible because most data is represented by more bits than its entropy suggests is optimal.
- Compression ratio:
  - 1 – ratio of size of compressed data to size of original data
- Can be lossy or lossless.

# Huffman coding

- A type of minimum redundancy coding
  - encode symbols that occur with greater frequency with fewer bits than for those that occur less often
- The entropy *S* of a symbol *z* is defined as

$S_z = -\log_2 P_z$

where $P_z$ is the probability of *z* being found in the data.

e.g. if *z* occurs 8 times in 32 symbols:

$S_z = -\log_2(8/32) = 2$ bits

- So using any more than 2 bits to represent *z* is (quite literally) a waste of space.

# Example

- Suppose a text file contains the characters A, B, C and D.
- If all four characters are equally likely, we need 2 bits to encode each data, e.g. A=00, B=01, C=10, D=11.

# Example

- Suppose a text file contains the characters A, B, C and D.
- If all four characters are equally likely, we need 2 bits to encode each data, e.g. A=00, B=01, C=10, D=11.
- But suppose A occurs 70% of time, B 26%, C 2%, and D 2%
- We can compress by assigning shorter codes to A (1 bit), and longer to C and D (3 bits).

# Example

- Suppose a text file contains the characters A, B, C and D.
- If all four characters are equally likely, we need 2 bits to encode each data, e.g. A=00, B=01, C=10, D=11.
- But suppose A occurs 70% of time, B 26%, C 2%, and D 2%
- We can compress by assigning shorter codes to A (1 bit), and longer to C and D (3 bits).
- e.g. A=0, B=10, C=110, D=111

# Image compression

- Suppose we want to compress an image with a uniform distribution of grey-level intensity.
  - And there are 256 shades of grey.
- $P_z = \frac{1}{256}$, for all *z*.
  - Because each grey shade is equally likely and there are 256 of them.
- $S_z = -\log_2(1/256) = 8$ bits

# In reality images are rarely uniform in their shades of grey

By Šarūnas Burdulis from USA - Dartmouth IndustriesUploaded by GiW, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=30714779

## Entropy of a set of data containing 72 instances of 5 different symbols

| Symbol | Probability (frequency) | Entropy of each instance | Total entropy |
|---|---|---|---|
| U | 12/72 | 2.584963 | 31.01955 |
| V | 18/72 | 2.000000 | 36.00000 |
| W | 7/72 | 3.362570 | 23.53799 |
| X | 15/72 | 2.263034 | 33.94552 |
| Y | 20/72 | 1.847997 | 36.95994 |

Overall entropy of the data is the sum of the total entropies contributed by each symbol = 161.463 bits.

Using 8 bits to represent each symbol yields a data size of 72*8 = 576 bits.

We should be able to compress this data (losslessly) by up to 1 – (161.463/576) = 72% compared to ASCII coding.

# Huffman coding

- Approximate optimal encoding of data based on its entropy.
- It uses a binary tree called a Huffman Tree to generate Huffman Codes
  - these are assigned to symbols in the data to achieve compression

# Building a Huffman Tree

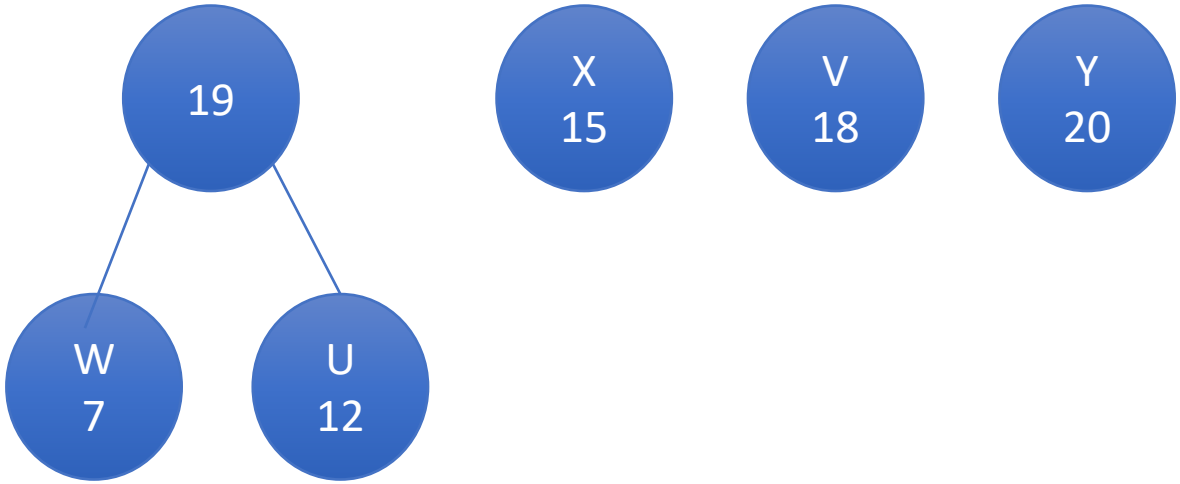
1. Place each symbol and its frequency in its own tree.
2. Merge the two trees whose root nodes have the smallest frequencies, and store the sum of the frequencies in the new tree's root.
3. Repeat step 2 until there is a single tree.

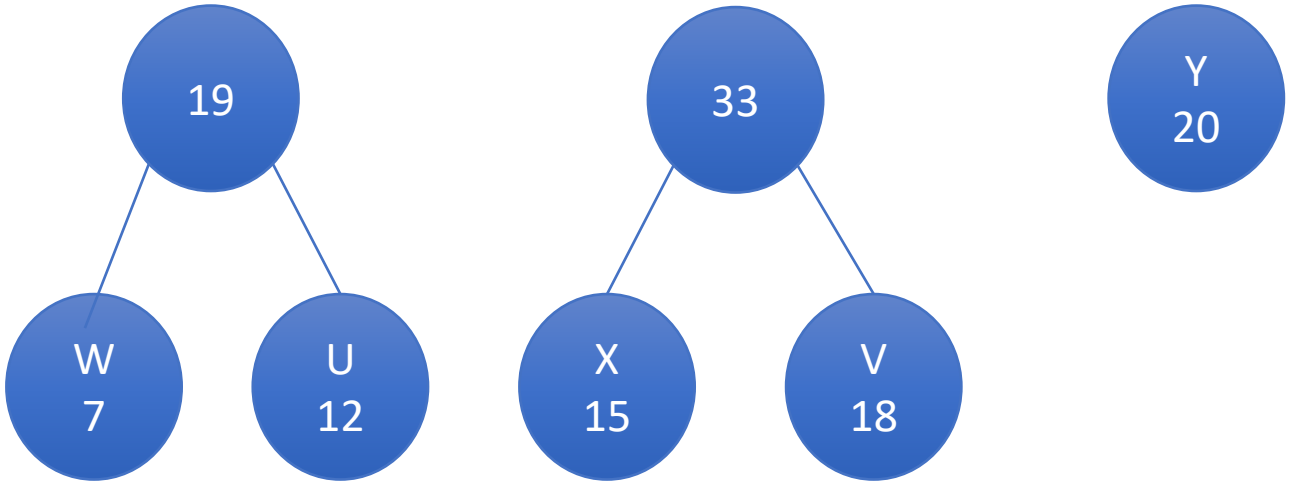
Root node contains the total number of symbols in the data.

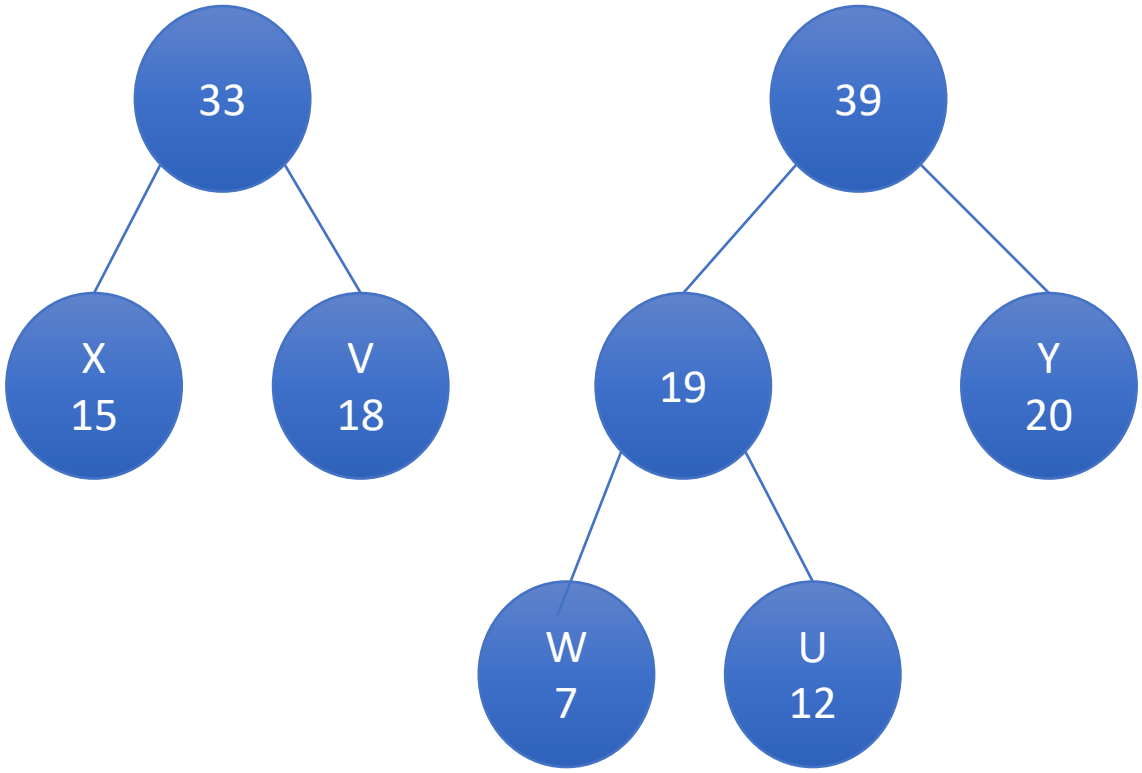
Leaf nodes contain original symbols and their frequencies.

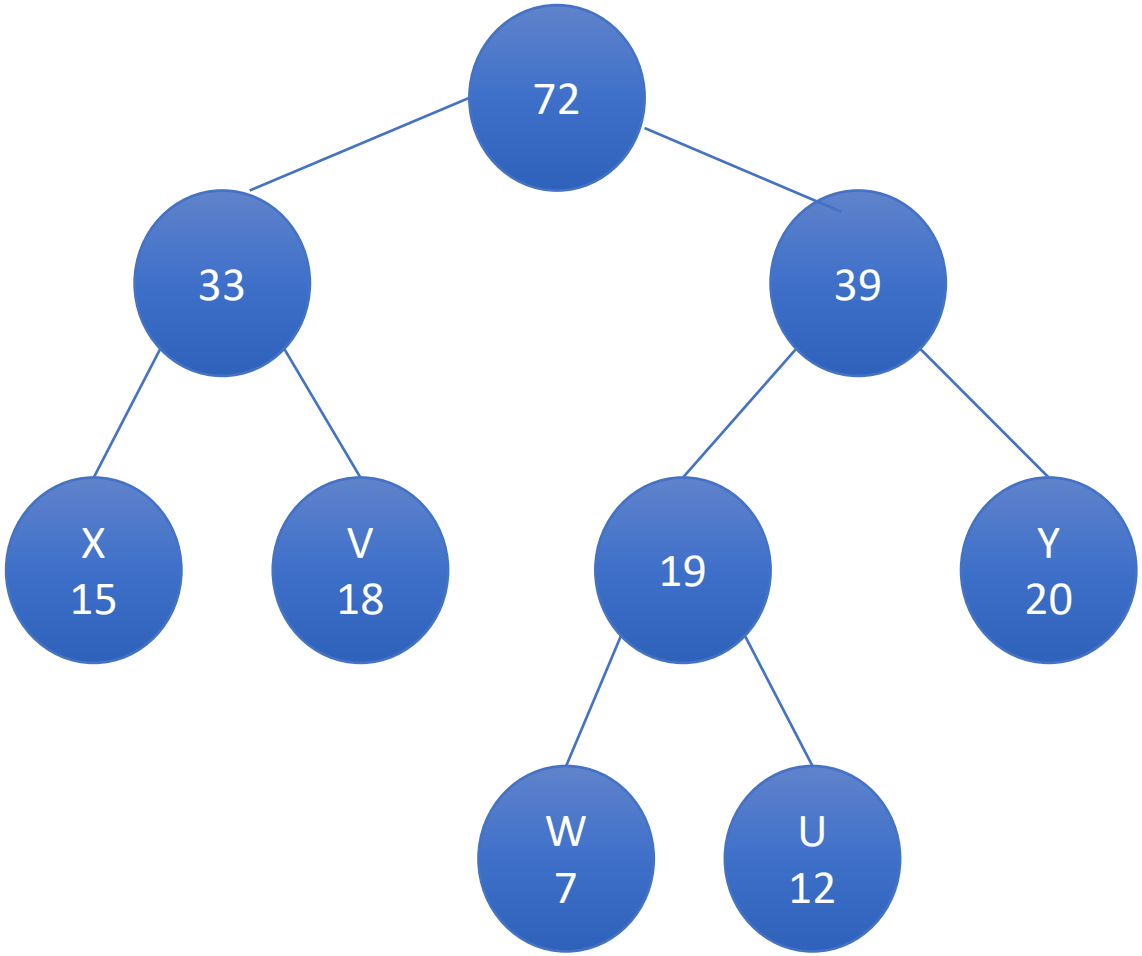
This is a greedy algorithm.


W
7
U
12
X
15
V
18
Y
20


19
W
7
U
12
X
15
V
18
Y
20

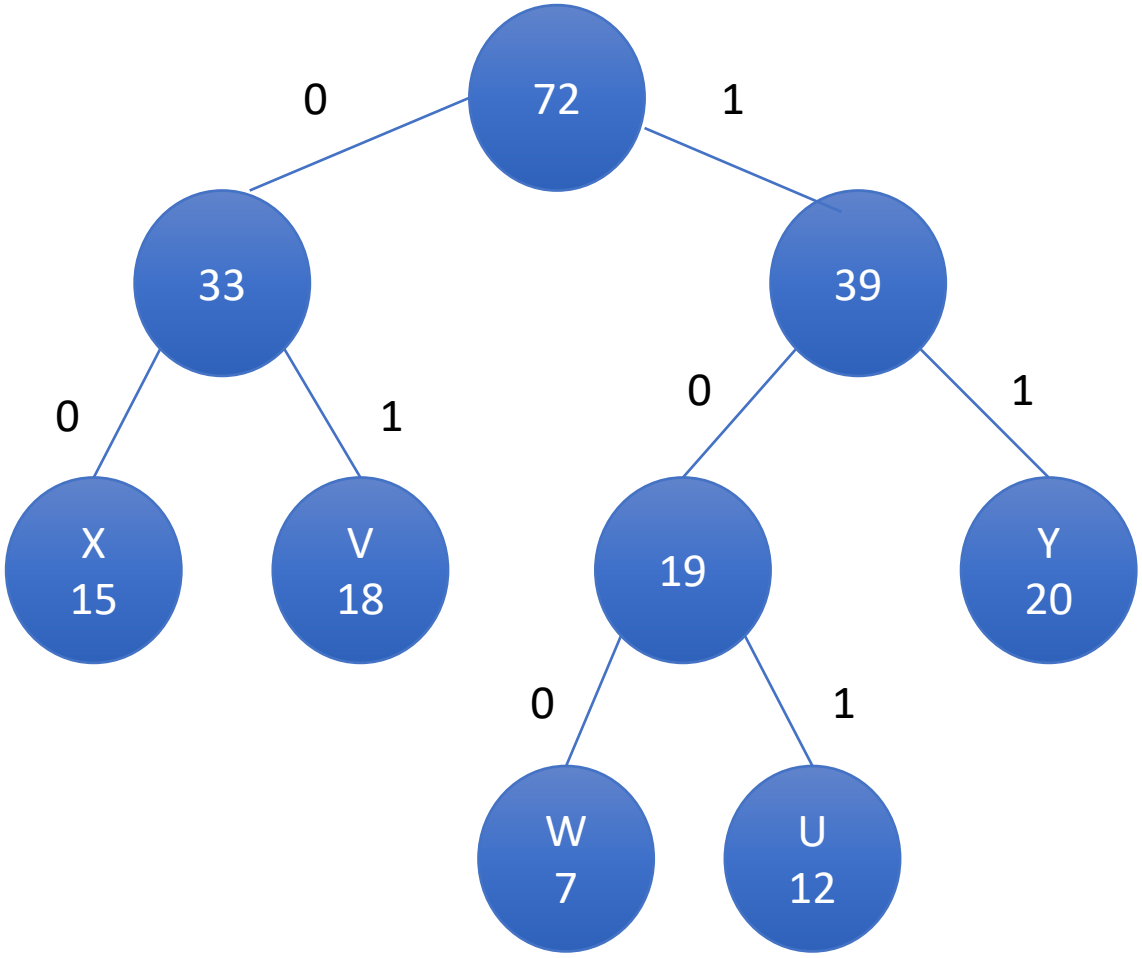

19
W
7
U
12
33
X
15
V
18
Y
20


33
X
15
V
18
39
19
Y
20
W
7
U
12


72
33
39
X
15
V
18
19
Y
20
W
7
U
12

# Compressing data

- Given a symbol, start at the root of the tree and trace a path to the symbol's leaf.
- As we descend along the path, whenever we move to the left, we append a 0 to the current code.
- Whenever we move to the right, we append a 1.


72
0
1
33
39
0
1
0
1
X
15
V
18
19
Y
20
0
1
W
7
U
12

# Uncompressing data

- Read the compressed data bit by bit.
- Starting at the root whenever we encounter a 0 we move left, whenever we encounter a 1 we move right.
- Once we reach a leaf node, we generate the symbol it contains and move back to the root.
- Possible because Huffman codes are prefix free
  - no code is a prefix of any other

# Effectiveness of Huffman coding

- Size of compressed data is sum of each symbol's frequency * number of bits in its Huffman code.
- So size of data in our example is:

12*3 + 18*2 + 7*3 + 15*2 + 20*2 = 163 bits

- If we used 8-bit ASCII codes uncompressed the size would be 72*8 = 576 bits
- So the compression ratio is 1 – (163/576) = 71.7%
- Theoretical maximum was 72%
- Difference is because we cannot take account of fractional bits in Huffman coding.
- Not the most effective form of compression but runs fast.

# Performance

# Performance

- Need to scan data twice to compress
  - once to gather frequencies
  - second time to compress data
  - compression is O(*n*).
- Uncompressing is fast because decoding each symbol requires only following the bits left and right through the tree.
  - uncompression is O(*n*).

# Automatic Code Generation

Genetic Programming

# Data Structures..

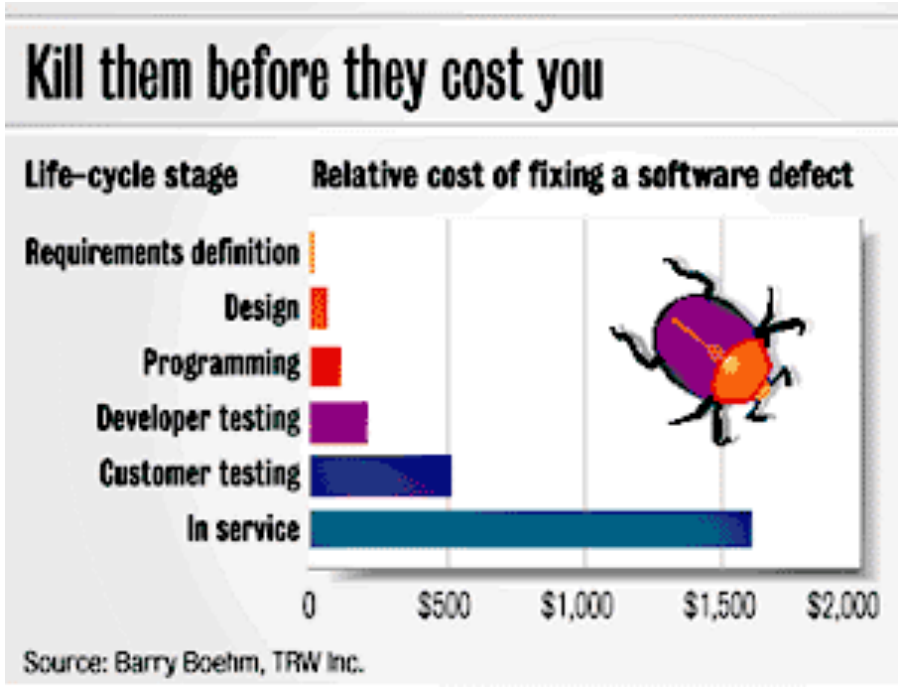
- This course has been about methods for representing data in different ways and algorithms for calculating things using the data-structures
- We have discussed that different structures/algorithms have different complexity and running time
  - Very important for large, difficult problems
- A new field called Genetic Improvement is trying to find ways to automate program design
  - auto-generating code

Automatic Program Repair with
Evolutionary Computation By Westley Weimer, Stephanie
Forrest, Claire Le Goues, and ThanhVu Nguyen

Evolutionary Improvement of Programs
David R. White, Andrea Arcuri, and John A. Clark

# Software Bug Fixing

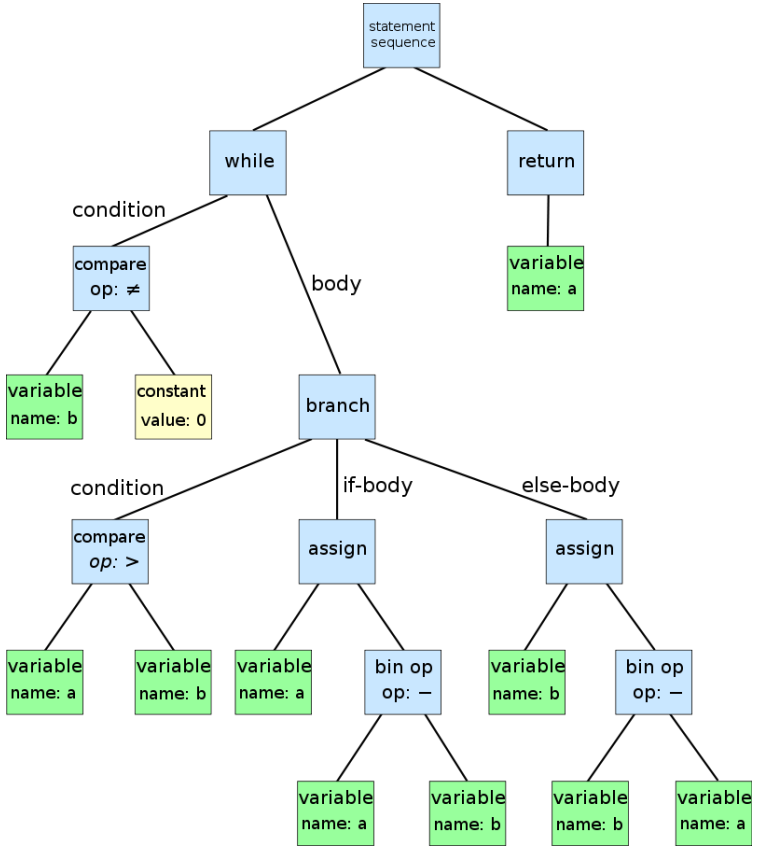
- Bug fixing is difficult, time-consuming, manual process
- Maintenance can be 90% of a software project
- The number of outstanding software defects typically exceeds the resources available to address them.
- Mature software projects are forced to ship with both known and unknown bugs because they lack the development resources to deal with every defect.




Ramamoothy, C.V., Tsai, W.-T. Advances in software engineering. *IEEE Comput. 29*, 10 (1996), 47–58.

# Programs as trees

- A C program is represented as an abstract syntax tree (AST), in which each node corresponds to an executable statement or control-flow structure in the program.
- A “negative” test case that exercises the fault
- A set of “positive” test cases

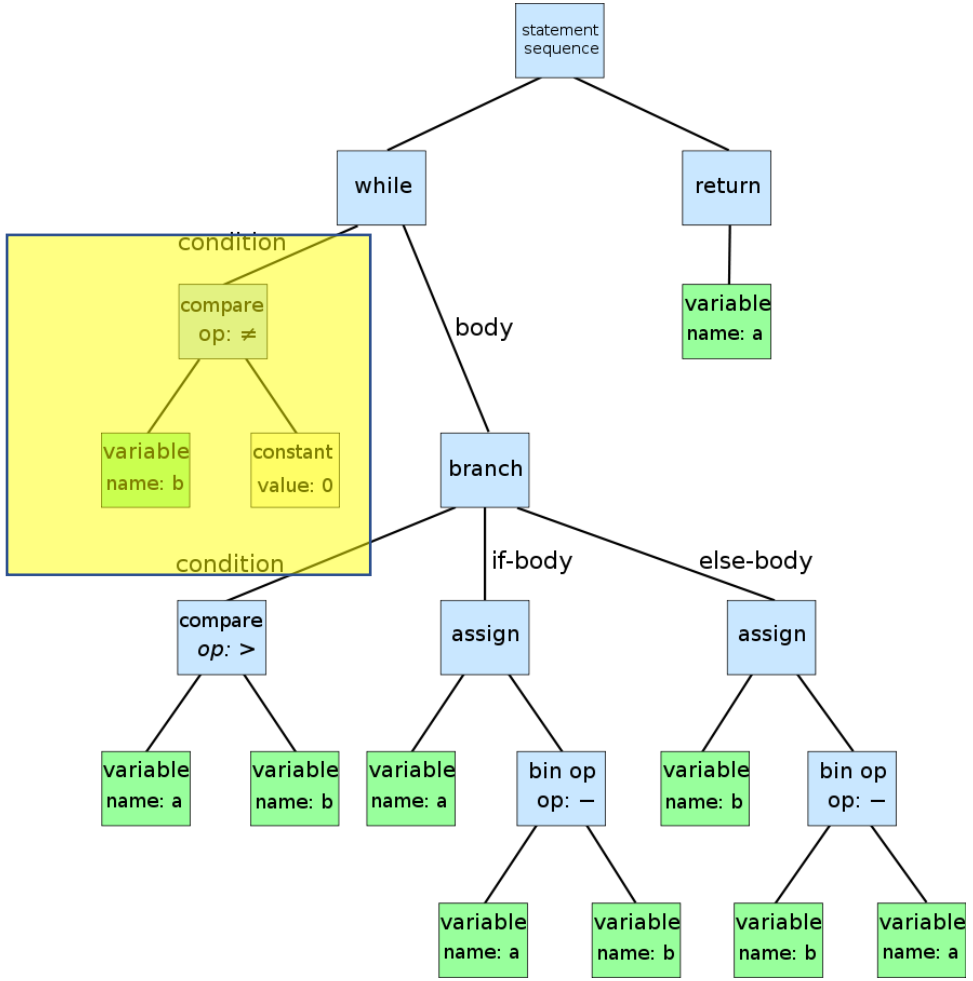



Euclidean algorithm:

**while** b ≠ 0
  **if** a > b
    a := a − b
  **else**
    b := b −a
**return** a

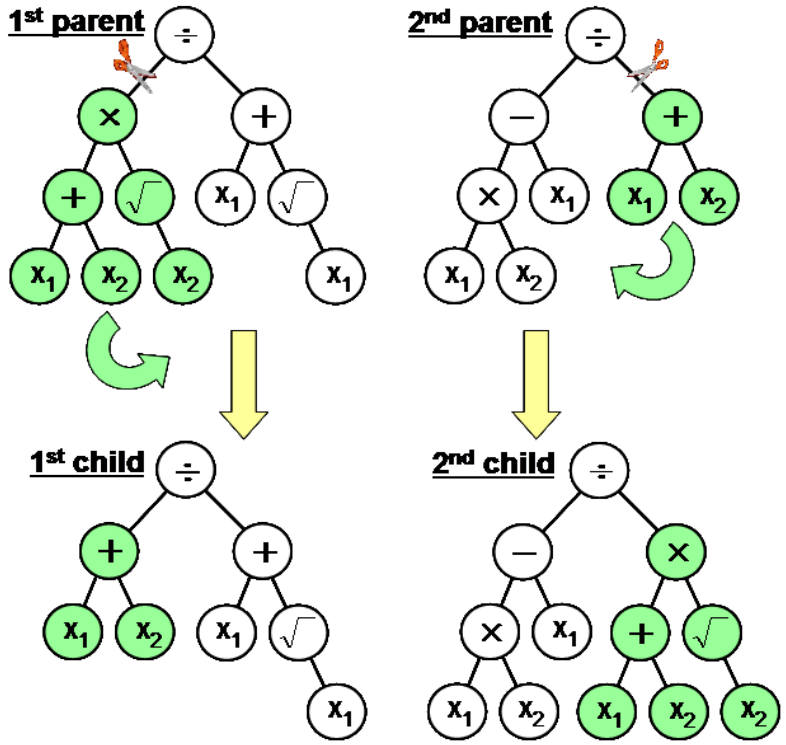
# Weighted Paths through Programs

- Bias changes towards nodes that were visited during the negative test case execution
  - Increases scalability
  - E.g. program with 8068 nodes reduced to 34 possibilities
- Create a weighted path through program:
  - List of pairs (s,w) containing nodes visited in negative case with associated weight:
    - 1.0 if in negative but not positive
    - 0.1 if in negative and positive case
    - 0 otherwise
- Summed weighted path length gives some idea of complexity



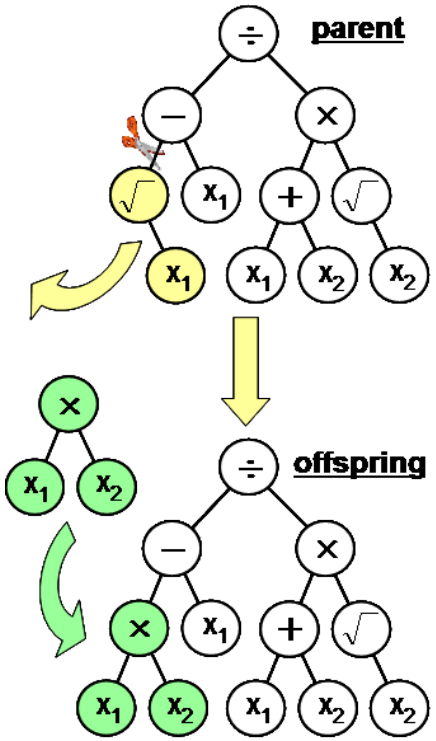
# Genetic Programming

- Techniques that uses a large population containing **random variants** of a program to search for a ”good variant”
- Variants undergo modification:
  - **Crossover** (exchanges subtrees between two programs)
  - Mutation (makes a random change to a node on the weighted path)
  - Each variant has a fitness (weighted sum of the number of positive and negative cases passed)
- A selection process deletes the bottom ranked 50% and creates a new population via the operators above
  - This is repeated many times

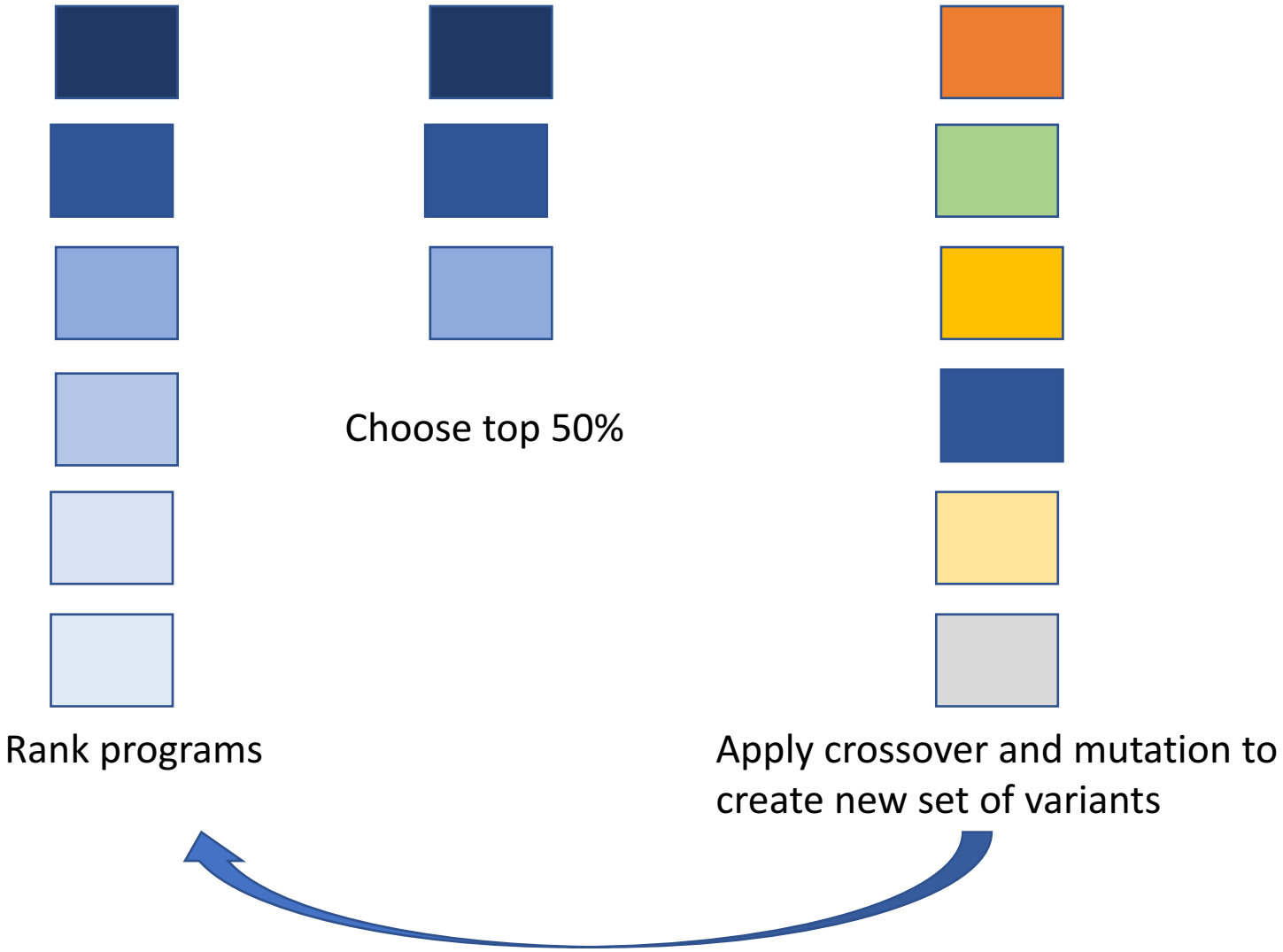


# Genetic Programming

- Techniques that uses a large population containing variants of a program to search for a ”good variant”
- Variants undergo modification:
  - Crossover (exchanges subtrees between two programs)
  - **Mutation** (makes a random change to a node on the weighted path)
  - Each variant has a fitness (weighted sum of the number of positive and negative cases passed)
- A selection process deletes the bottom ranked 50% and creates a new population via the operators above
  - This is repeated many times

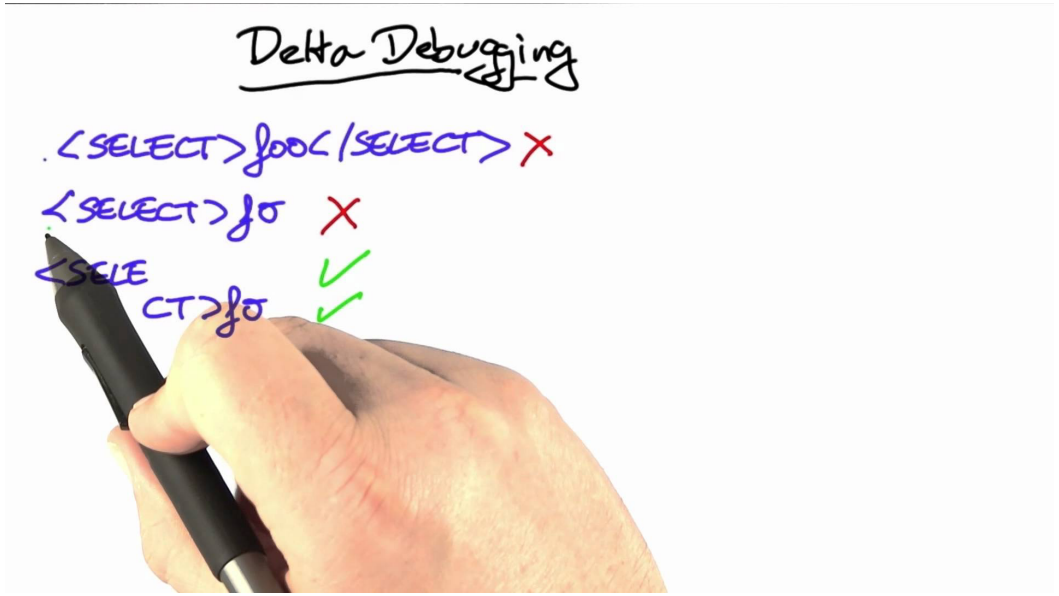


# Evolution…


Choose top 50%
Rank programs
Apply crossover and mutation to create new set of variants

# Minimising the Repair

- GP might introduce **dead code**
  - *x=3; x=5;*
  - *calls to irrelevant functions*
- Find a small subset of changes that still passes all the test cases
- Potentially time-consuming
  - But in practice, this turns out to be linear



# Microsoft Zune Bug

- Bug reported that caused media players to freeze up
- When the value of the input days is the last day of a leap year (such as 10,593, which corresponds to 31 December 2008), the program enters an infinite loop on lines 3–16

```
1  void zunebug(int days) {
2    int year = 1980;
3    while (days > 365) {
4      if (isLeapYear (year)){
5        if (days > 366) {
6          days -= 366;
7          year += 1;
8        }
9        else {
10       }
11     }
12     else {
13       days -= 365;
14       year += 1;
15     }
16   }
17   printf("the year is %d\n", year);
18 }
```

See *Microsoft Zune affected by "Bug,"* BBC News, December 2008, http://news.bbc.co.uk/2/hi/technology/7806683.stm

# Walk through example

- Negative Test Cases:
  - Inputs 366, 10,593
- Positive Test Cases:
  - Inputs 1000,2000,3000,4000,5000
- The positive test case zunebug (1,000) visits lines 1–8, 11–18.
- The negative test case zunebug (10,593) visits lines 1–16, and then repeats lines 3, 4, 8, and 11 infinitely

```
 1  void zunebug(int days) {
 2    int year = 1980;
 3    while (days > 365) {
 4      if (isLeapYear (year)){
 5        if (days > 366) {
 6          days -= 366;
 7          year += 1;
 8        }
 9        else {
10        }
11      }
12      else {
13        days -= 365;
14        year += 1;
15      }
16    }
17    printf("the year is %d\n", year);
18  }
```

# One variant

- A conditional statement inserted between lines 6-7 of original
- A new statement added here at line 15
- This variant:
  - Passes one negative case (366)
  - Passes one positive variant (1000)

```
 5  if (days > 366)   {
 6    days -= 366;
 7    if (days > 366) {   // insert #1
 8      days -= 366;      // insert #1
 9      year += 1;        // insert #1
10    }                   // insert #1
11    year += 1;
12  }
13  else {
14  }
15  days -= 366;          // insert #2
```

# Next good variant

- 6 generations later:
  - Lines 6-10 deleted
  - A new insert at line 14
- At this point it passes all the test cases
- Program terminates
  - Minimisation procedure invoked to delete unecessary code

```
 5  if (days > 366)   {
 6    // days -= 366;          // delete
 7    // if (days > 366) {     // delete
 8    //   days -= 366;        // delete
 9    //   year += 1;          // delete
10    // }                     // delete
11    year += 1;
12  }
13  else {
14    days -= 366;         // insert
15  }
16  days -= 366;
```

# Final Repair

- 3 key changes to original program (line numbers from original):
  1) Delete at line 6.  days -=366
  2) Insert between 9-10: days -= 366
  3) Insert between 10-11: days-=366
- Only 1) and 3) necessary so 2) deleted to give final repair

```
 1  void zunebug_repair (int days) {
 2    int year = 1980;
 3    while (days > 365) {
 4      if (isLeapYear (year)) {
 5        if (days > 366) {
 6          // days -= 366;  // deleted
 7          year += 1;
 8        }
 9        else {
10        }
11        days -= 366;        // inserted
12      } else {
13        days -= 365;
14        year += 1;
15      }
16    }
17    printf ("the year is %dn", year);
18  }
```

# Eleven defects repaired

| Program | Lines of Code | Weighted Path | Description | Fault | Time (s) | Fitness Evals | Repair Size |
|---|---|---|---|---|---|---|---|
| gcd | 22 | 1.3 | Euclid's algorithm | Infinite loop | 153 | 45.0 | 2 |
| zune | 28 | 2.9 | MS Zune excerpt | Infinite loop | 42 | 203.5 | 4 |
| uniq utx 4.3 | 1146 | 81.5 | Duplicate filtering | Segmentation fault | 34 | 15.5 | 4 |
| look utx 4.3 | 1169 | 213.0 | Dictionary lookup | Segmentation fault | 45 | 20.1 | 11 |
| look svr 4.0 | 1363 | 32.4 | Dictionary lookup | Infinite loop | 55 | 13.5 | 3 |
| units svr 4.0 | 1504 | 2159.7 | Metric conversion | Segmentation fault | 109 | 61.7 | 4 |
| deroff utx 4.3 | 2236 | 251.4 | Document processing | Segmentation fault | 131 | 28.6 | 3 |
| nullhttpd 0.5.0 | 5575 | 768.5 | Webserver | Heap buffer overrun | 578 | 95.1 | 5 |
| indent 1.9.1 | 9906 | 1435.9 | Source code formatting | Infinite loop | 546 | 108.6 | 2 |
| flex 2.5.4a | 18775 | 3836.6 | Lexical analyzer generator | Segmentation fault | 230 | 39.4 | 3 |
| atris 1.0.6 | 21553 | 34.0 | Graphical tetris game | Stack buffer overrun | 80 | 20.2 | 3 |
| Total | 63277 | 8817.2 | | | 2003 | 651.2 | 39 |

# Acknowledgements


- Chapters 3 and 9 of Loudon, K. (1999). *Mastering Algorithms with C*. O'Reilly & Associates, Inc.