Trees

Binary trees
Traversals – prefix / infix / postfix

Read chapter 19 of the textbook!

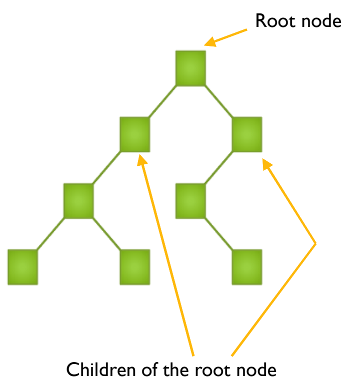
These slides will cover some of the contents, but the textbook is much more detailed.

Trees

- The *tree* is an abstract data structure (ADT) that provides a hierarchy of the elements stored in it
- Each element has exactly one *parent*
  - except for the *root* element
- Each element has one or more *children* (successors of the element)
  - except for the *leaf* elements

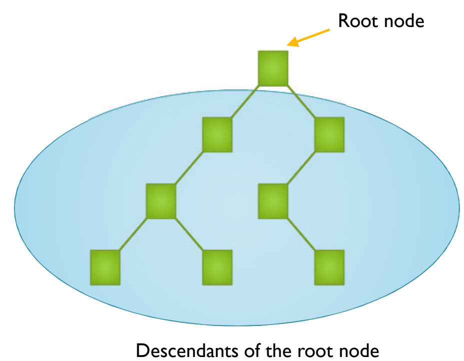
Trees

- Positions in the tree are termed *nodes*, for example.



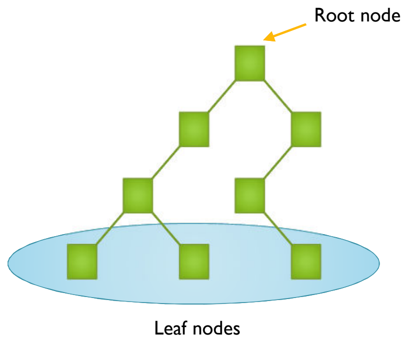
Trees

- Successors are also called descendants.



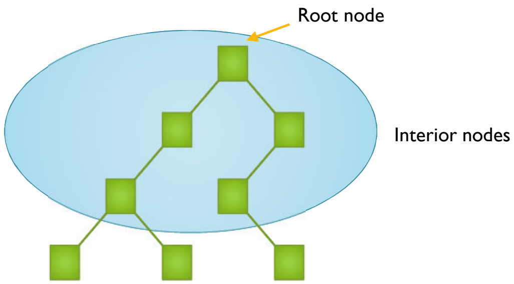
Trees

- Nodes without descendants are called leaf nodes.



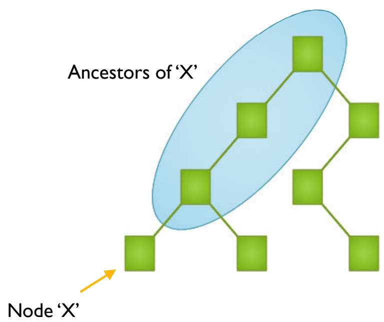
Trees

- Nodes with at least one descendant are called interior nodes.



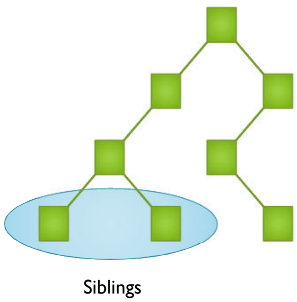
Trees

- Predecessors are also called ancestors. The immediate predecessor is called the parent.



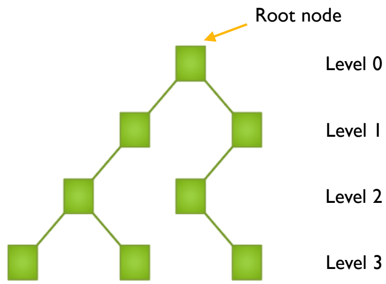
Trees

- Nodes with the same parent are called siblings.



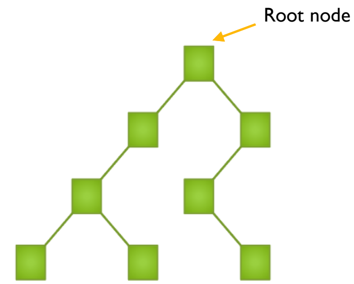
Trees

- Levels in a tree are numbered from top to bottom, starting at zero.



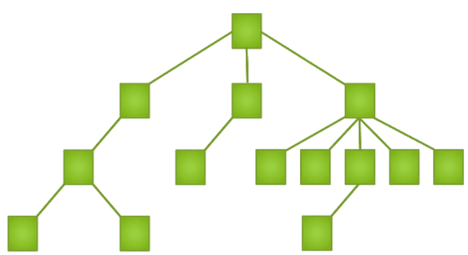
Trees

- Binary trees have at most two successors per node.



Trees

- *General* trees allow an arbitrary number of children per node.

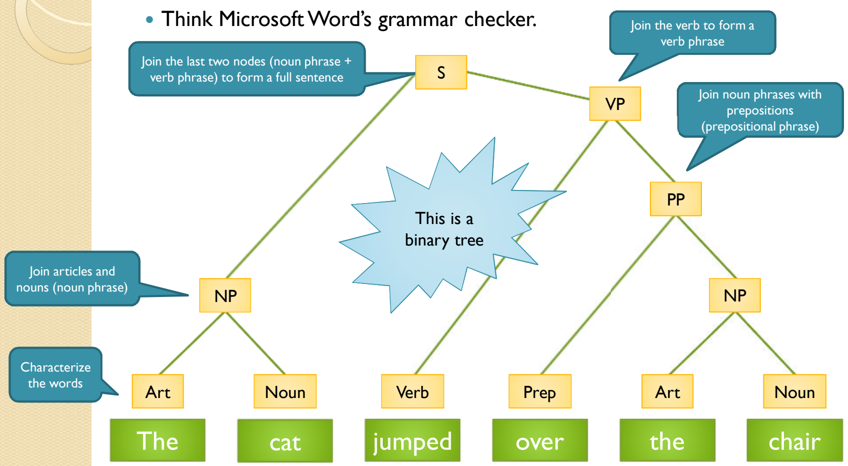


Where do you use trees?

- Processing sentences (computer programs or natural languages)
- Searchable data structures
- Heaps (implement heap sort, priority queues)

Processing sentences

- Think Microsoft Word's grammar checker.



Where do you use trees?

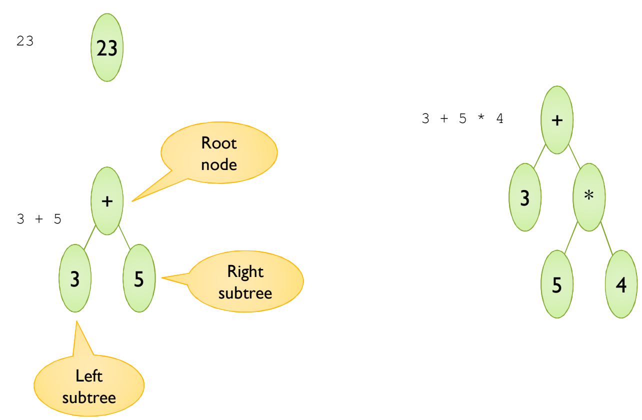
- Processing sentences (computer programs or natural languages)
- Searchable data structures
- Heaps (to implement heap sort, priority queues)

These two you will learn in this course

Expression trees

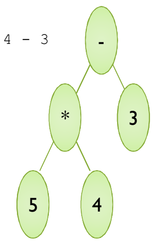
- Used for evaluating mathematical expressions. Think "=3*24/(4+8)" in Microsoft Excel.
- Simple rules:
  - An expression tree for a single number is a node containing the number
  - Otherwise, the tree is a node containing an operator and links to left and right subtrees
  - The subtrees contain the operands of the expression

Expression trees

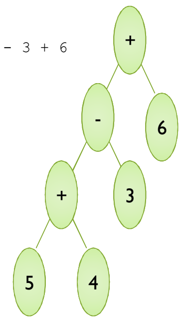


Expression trees

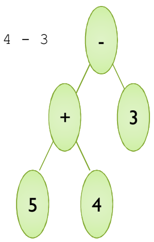
5 * 4 - 3



5 + 4 - 3 + 6



5 + 4 - 3



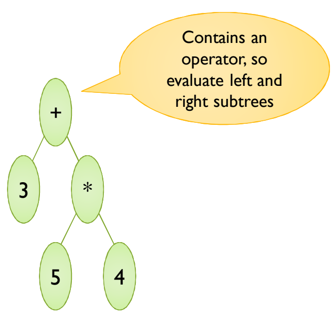
- When operators have equal precedence, the ones to the left appear lower in the tree.

Expression trees

- Expression trees have the following extra properties:
  - Numbers are stored at leaf nodes
  - Operators are stored at interior nodes
  - All interior nodes have two children
- To evaluate an expression tree:
  - Begin at the root node
  - If the node contains a number, return it
  - Otherwise, run the operator in the node with the results of evaluating its left and right subtrees, and return this value

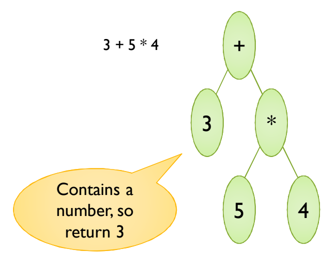
Example of an expression tree evaluation

3 + 5 * 4



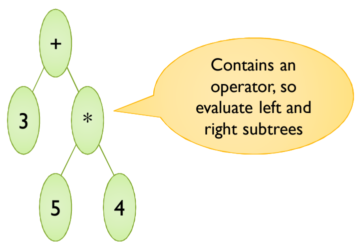
Example of an expression tree evaluation

3 + 5 * 4



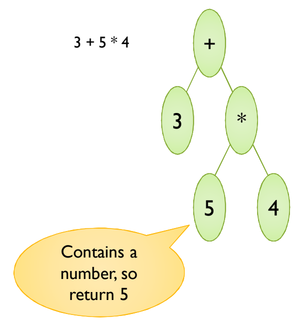
Example of an expression tree evaluation

3 + 5 * 4



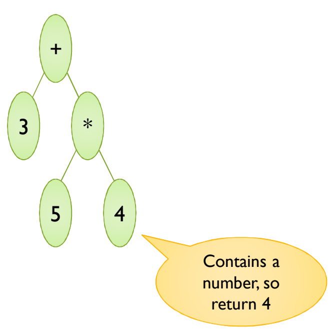
Example of an expression tree evaluation

3 + 5 * 4



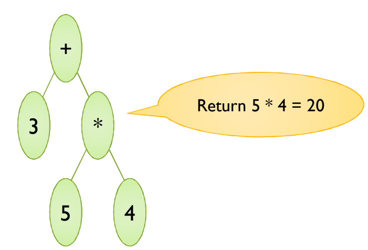
Example of an expression tree evaluation

3 + 5 * 4

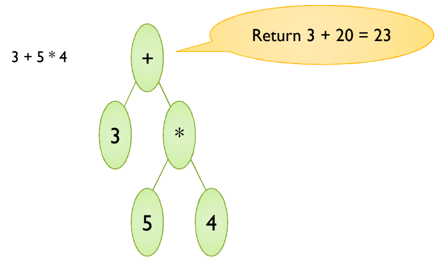


Example of an expression tree evaluation

3 + 5 * 4



Example of an expression tree evaluation



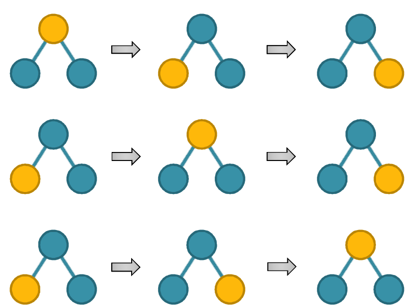
Pseudocode for `evaluate()`

- This is the method for evaluating the expression tree and returning the resulting value.

```
// this is a method to evaluate an expression tree. It receives a pointer
// to a node and recursively traverses the tree to calculate the result.
int evaluate(node)

if (node stores a number)
    return the number
else
    set leftOperand to evaluate(node.left)
    set rightOperand to evaluate(node.right)
    return computeValue(node, leftOperand, rightOperand)
```

Traversing a tree

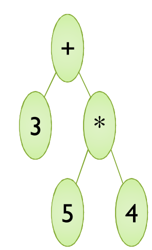
- An expression tree supports three types of traversals
  - Preorder
    - visit node,
    - then go left,
    - then go right.
  - Inorder
    - go left,
    - then visit node,
    - then go right.
  - Postorder
    - go left,
    - then go right,
    - then visit node.



- These traversals can generate the prefix, infix, and postfix notations of an expression

Traversing a tree

- For the tree on the right, these are the three traversals:
  - Preorder traversal
    + 3 * 5 4
  - Inorder traversal
    3 + 5 * 4
  - Postorder traversal
    3 5 4 * +



Pseucocode for `prefix()`

```
// In this pseudocode, get_data() returns the value stored at a node.
// get_left() returns the node at the top of the left subtree.
// Similarly, get_right() returns the node at the top of the right subtree.


string prefix(node)

if (get_data() is null)
    return ""
else
    return get_data() + prefix(get_left()) + prefix(get_right())
```

- `get_data()` returns a string with the content of the node.

Pseucocode for `infix()`

```
// In this pseudocode, get_data() returns the value stored at a node.
// get_left() returns the node at the top of the left subtree.
// Similarly, get_right() returns the node at the top of the right subtree.


string infix(node)

if (get_data() is null)
    return ""
else
    return infix(get_left()) + get_data() + infix(get_right())
```

- `get_data()` returns a string with the content of the node.

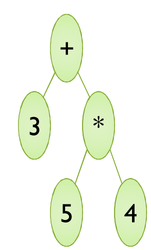
Pseucocode for `postfix()`

```
// In this pseudocode, get_data() returns the value stored at a node.
// get_left() returns the node at the top of the left subtree.
// Similarly, get_right() returns the node at the top of the right subtree.


string postfix(node)

if (get_data() is null)
    return ""
else
    return postfix(get_left()) + postfix(get_right()) + get_data()
```

- `get_data()` returns a string with the content of the node.

Traversing a tree

- For the tree on the right, these are the three traversals:
  - Preorder traversal
    `+ 3 * 5 4`
  - Inorder traversal
    `3 + 5 * 4`
  - Postorder traversal
    `3 5 4 * +`



Class BTNode

```
template <typename value_type>
class BTNode
{
public:
// Constructors
    BTNode();
    BTNode(value_type, BTNode*, BTNode*, BTNode*);
// Destructor
    ~BTNode();
// Mutators
    void set_data(const value_type);
    void set_parent(BTNode*);
    void set_left(BTNode*);
    void set_right(BTNode*);
// Query
    value_type get_data() const;
    BTNode* get_parent();
    BTNode* get_right();
    BTNode* get_left();
private:
    value_type data;
    BTNode* left;
    BTNode* right;
    BTNode* parent;
};
```

Class ETree

```
// More functionality may be added later
#include "BTNode.h"
template <typename Item>
class ETree
{
public:
// Constructors
    ETree();
    ETree(BTNode<Item>*);
    ETree(Item, ETree*, ETree*);
// Destructor
    ~ETree();
// Mutators
// Query
   BTNode<Item>* get_root();
   BTNode<Item>* get_left();
   BTNode<Item>* get_right();
   Item current();
private:
   BTNode<Item>* root;
};

#include "ETree.template"
```

Implementing ETree

```
// No parameter -> root = NULL
template <typename Item>
ETree<Item>::ETree(){root = NULL;}

// parameter is a node -> becomes the root node
template <typename Item>
ETree<Item>::ETree(BTNode<Item>* root_node){root = root_node;}

// parameters are item + 2 branches
template <typename Item>
ETree<Item>::ETree(Item data, ETree* left_sub, ETree* right_sub)
{
    root = new BTNode<Item>();
    root->set_data(data);
    if (left_sub != NULL) {
        root->set_left(left_sub->get_root());
        left_sub->get_root()->set_parent(root);
    }
    else {root->set_left(NULL);}

    if (right_sub != NULL) {
        root->set_right(right_sub->get_root());
        right_sub->get_root()->set_parent(root);
    }
    else {root->set_right(NULL);}
}
// etc
```

Implementing ETree

```
// Note that it checks for leaf nodes
template <typename Item>
Item ETree::current()
{
    if (root != NULL)
        return root->get_data();
    else
        return NULL;
}

// Returns pointer to root BTNode
template <typename Item>
BTNode<Item>* ETree::get_root()
{
    return root;
}

// Returns pointer to left node
template <typename Item>
BTNode<Item>* ETree::get_left()
{
    return root->get_left();
}

.
.
.
```

Implementing `prefix()`

```
#include <cstdlib>
#include <string>
#include "BTNode.h"
using namespace std;

string prefix(BTNode<char>* node)
{
    if (node == NULL)
        return "";
    else
        {return node->current() + prefix(node->get_left()) + prefix(node->get_right());}
}
```

Implementing `infix()`

```
#include <cstdlib>
#include <string>
#include "BTNode.h"
using namespace std;

string infix(BTNode<char>* node)
{
    if (node == NULL)
        return "";
    else
        {return infix(node->get_left()) + node->current() + infix(node->get_right());}
}
```

Implementing `postfix()`

```
#include <cstdlib>
#include <string>
#include "BTNode.h"
using namespace std;

string postfix(BTNode<char>* node)
{
    if (node == NULL)
        return "";
    else
        {return postfix(node->get_left()) + postfix(node->get_right()) + node->current();}
}
```

Sample code

```
#include <cstdlib>
#include <iostream>
#include "ETree.h"
#include "BTNode.h"
#include "functions.h" // contains in, pre and post fix
using namespace std;
int main()
{
  char curr = '3';
  ETree<char>* t1 = new ETree<char>(curr, NULL, NULL);
  curr = '4';
  ETree<char>* t2 = new ETree<char> (curr, NULL, NULL);
  curr = '+';
  ETree<char>* t3 = new ETree<char> (curr, t1, t2);
  curr = '6';
  ETree<char>* t4 = new ETree<char> (curr, NULL, NULL);
  curr = '*';
  ETree<char>* t5 = new ETree<char> (curr, t4, t3);

  cout << infix(t5->get_root()) << endl;
  cout << prefix(t5->get_root()) << endl;
  cout << postfix(t5->get_root()) << endl;
  return 0;
}

Output is:
6*3+4
*6+34
634+*
```

DEMO

Second sample code

```
#include <cstdlib>
#include "ETree.h"
#include <iostream>
#include "functions.h" // contains in, pre and post fix
using namespace std;
int main()
{
  char curr = '*';
  ETree<char>* t1 = new ETree<char>(curr);
  t1->reset(); curr = '6';
  t1->add_left(curr);
  curr = '+';
  t1->add_right(curr);
  t1->go_right(); curr = '3';
  t1->add_left(curr);
  curr = '4';
  t1->add_right(curr);

  t1->reset();
  cout << infix(t1->get_current()) << endl;
  cout << prefix(t1->get_current()) << endl;
  cout << postfix(t1->get_current()) << endl;

  return 0;
}

Output is:
6*3+4
*6+34
634+*
```

DEMO

Inversion of trees

```
#include <cstdlib>
#include <string>
#include "BTNode.h"
using namespace std;

string invert(BTNode<char>* node)
{
    if ((node->get_left() != NULL) || (node->get_right() != NULL))
    {
        // at least one of the two branches has information, so a swap is executed
        BTNode<char>* temp;
        temp = node->get_right();
        node->set_right(node->get_left());
        node->set_left(temp);
    }
    // recursively calls invert on the left branch
    if (node->get_left() != NULL) {invert(node->get_left());}
    // recursively calls invert on the right branch
    if (node->get_right() != NULL) {invert(node->get_right());}
}
```

See you next week!