Data Structures and Algorithms

XMUT-COMP 103 - 2024 T1

Traversing a binary tree

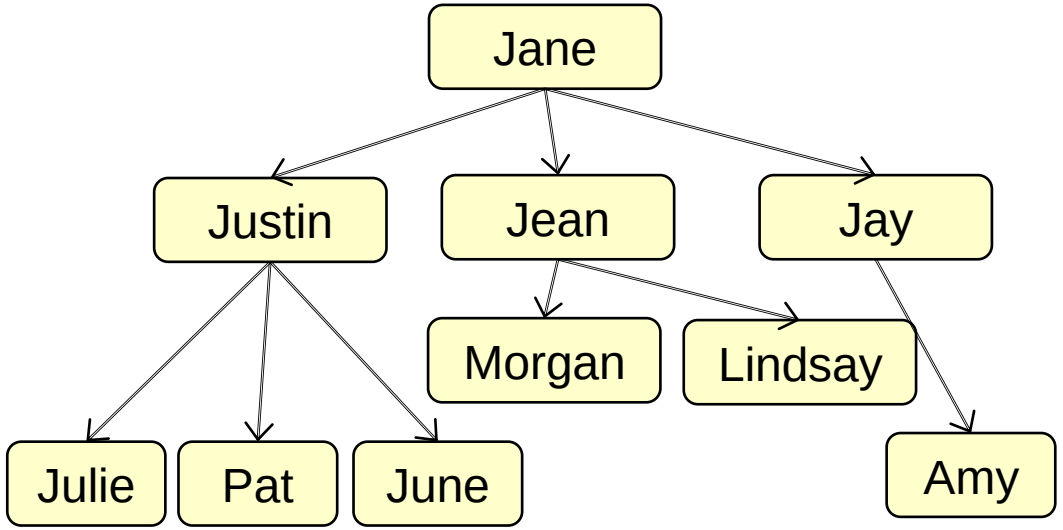
A/Prof. Pawel Dmochowski

School of Engineering and Computer Science

Victoria University of Wellington

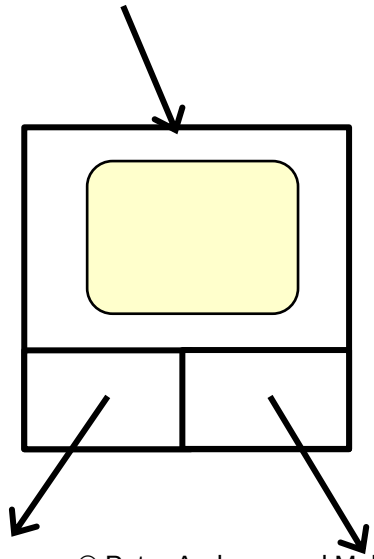
Trees of Data *vs* Trees containing Data

- Person, Position,…
  - The data object contains the links that make the tree.
  - Person: father, mother, children,…
  - Employee: manager, team



- Typical Collection (Set, Map, Queue…)
  - The collection has the structure,
  - The data objects sit inside the structure

- Tree data structures:
  - Node object that has fields for the data item, and for the links.



Tree Node types

- There is no single way of defining tree structures
  - Need to make your own.

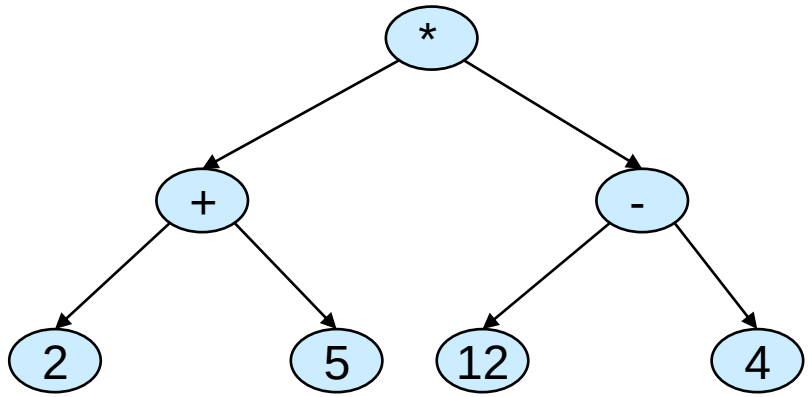
eg: Binary tree node:

Type variable: parameter of the ***type*** specify when you make a new node:

```
public class BTNode <E> {
    private E item;
    private BTNode<E>  left;
    private BTNode<E>  right;

    public BTNode(E item){  this.item = item;  }

    public E getItem()              {  return item; }
    public void setItem(E item)   { this.item = item; }

    public BTNode<E> getLeft()              { return left;}
    public void setLeft(BTNode<E> left)     { this.left = left;}

    public BTNode<E> getRight()             { return right;}
    public void setRight(BTNode<E> right)  { this.right = right;}
}
```

Using BTNode: Expressions

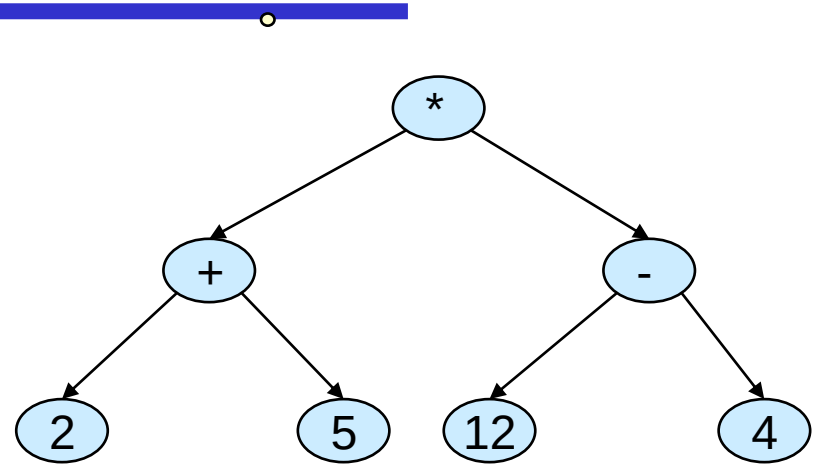
- (2 + 5) * (12 - 4)



- Printing:
  - standard format with (…)
  - pre-order: * + 2 5 – 12 4 "Polish notation" - never need brackets!
  - post-order: 2 5 + 12 4 - * "Reverse Polish notation"
- Evaluating
  - recursive, post-order traversal of tree
- Reading
  - pre-order and post-order: easy
  - in order: hard! (COMP 261, simple parsing algorithms)

Same tree, different print format.
- Polish and Reverse Polish are easier to parse
- Require fewer keystrokes on a calculator.
- Reverse Polish can be evaluated while reading

Using BTNode: Expressions



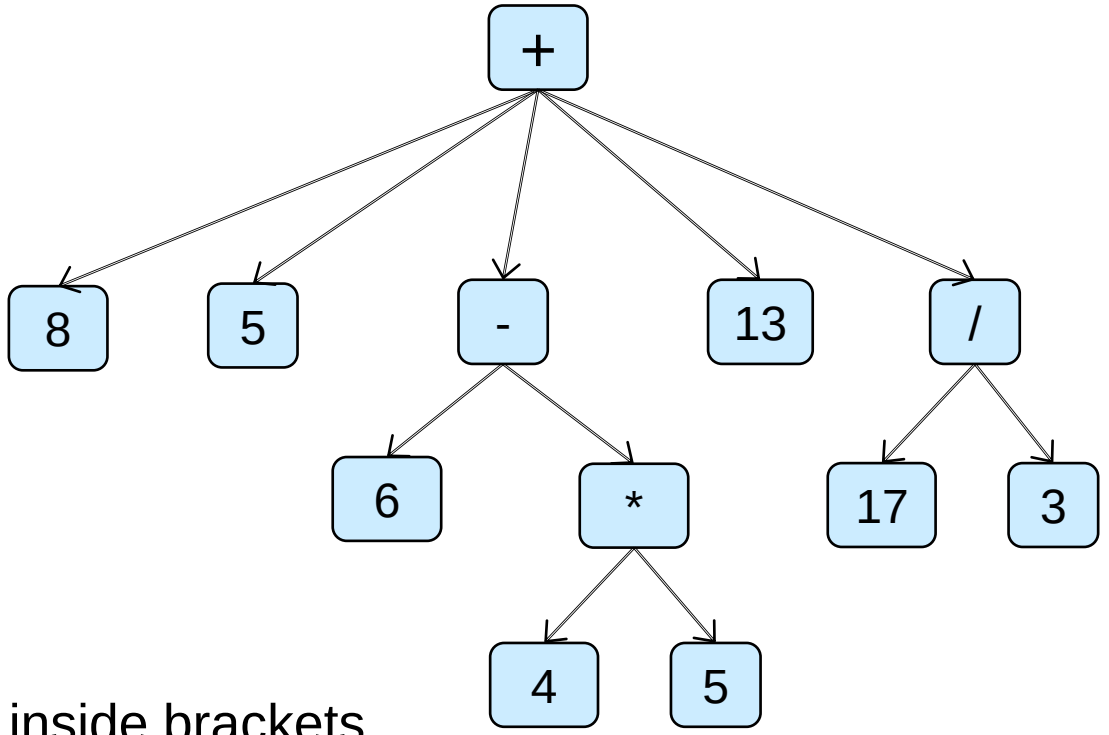
- pre-order: (Polish notation)
  * + 2 5 – 12 4
- post-order: (Reverse Polish)
  2 5 + 12 4 - *
- in-order: (infix)
  (2 + 5) * (12 – 4)

Same tree, different print format.

- Polish and Reverse Polish are easier to parse
- Require fewer keystrokes on a calculator.
- Reverse Polish can be evaluated while reading without storing the operators

Cambridge Polish notation

- Expressions with operators with variable number of arguments
- => general tree.



- Writing such expressions:
  - Normal functional notation:
    Pre-order, with brackets and commas
    operator before brackets
    eg: +(8, 5, -(6, *(4, 5)), 13. /(17 3))
  - Cambridge Polish Notation (Lisp)
    Pre-order, in brackets, no commas, operators inside brackets
    eg: (+ 8 5 (- 6 (* 4 5)) 13 (/ 17 3))

Same tree, different print format.
CPN is easier to parse

- Assignment: Make a calculator for this:
  - read (given), evaluate, print (REPL)

General Tree Nodes.

```
public class GTNode <E>  {
    private E item;
    private List<GTNode<E>> children;                // List, therefore children kept in order.

    /**Constructor for objects of class GTNode    */
    public GTNode(E item){
        this.item = item;
        this.children = new ArrayList<GTNode<E>>();
    }
    /** Getters and Setters     */
    public E getItem()              { return item; }
    public void setItem(E item) { this.item = item; }
```

Good design:
Protect the inner structure of the objects from modification by the rest of the program!

- What about the children?

```
public List<GTNode<E>> getChildren() { return Collections.unmodifiableList(children); }
```

General Tree Node: Alternative design

- Keep the List of children internal to the class.
- Provide methods to add and remove children

```
/** Getters and Setters     */
public GTNode<E> getChild(int indx)                { return children.get(indx); }
public void addChild(GTNode<E> child)              { children.add(child); }
public void addChild(int indx, GTNode<E> child) { children.add(indx, child); }
public GTNode<E> removeChild(int indx)             {return children.remove(indx); }
public void setChild(int indx, GTNode<E> child)    { children.set(indx, child); }
public int numChildren()                           {return children.size(); }
```

- What about iterating through the children?
  - Could use a counted for loop:

    `for (int i=0; i<node.numChildren(); i++){ GTNode<String> child =node.getChild(i); ... }`

  - Could enable foreach loop:

    `for (GTNode<String> child : node) { …. }`

    HOW?

Iterable and Iterators

- To be able to iterate along an object using foreach loop, the object must be Iterable:

- The object's class must implement Iterable<??> and have a

  public Iterator<??> iterator(){…}

  method which returns an Iterator object

- An Iterator object must have a

  public boolean hasNext() {…}

  method, and a

  public ??? next() {…}

  method

General Tree Nodes.

```
public class GTNode <E> implements Iterable <GTNode<E>>  {
    private E item;
    private List<GTNode<E>> children;                    // List, therefore children kept in order.

    /**Constructor for objects of class GTNode    */
    public GTNode(E item){
        this.item = item;
        this.children = new ArrayList<GTNode<E>>();
    }
    /** Getters and Setters     */
    public E getItem()              { return item; }
    public void setItem(E item) { this.item = item; }
      :
      :
    public Iterator<GTNode<E>> iterator() { return children.iterator(); }
}
```

Using GTNode with iterator

- look for a node in a tree with a particular item (recursive depth-first traversal)

```
public GTNode<String> findNode(GTNode<String> root,  String label){
    if (root == null){ return null;}
    if (root.getItem().equals(label) ) {
        return root;
    }
    for (GTNode<String> child : root) {
        GTNode<String> ans = findNode(child, label);
        if (ans != null) {
            return ans;
        }
    }
    return null;
}
```

Using GTNode with iterator

- Evaluate an expression tree with + opertator (recursive depth-first traversal)

```
public double evaluate(GTNode<ExpElem> root){
    if( root == null) { return Double.NaN; }
    ExpElem elem = root.getItem();
    if (elem.getItem().equals("#") ) { return elem.value; }
    double sum = 0;
    for (GTNode< ExpElem > child : root) {
        sum = sum + evaluate(child);
    }
    return sum;
}
```