TREES

WE SAW TREES AS A DATA STRUCTURE (WAY TO STRUCTURE DATA) FOR EFFICIENT STORAGE AND RETRIEVAL FOR THE SET ADT

BUT TREES ARE A NATURAL REPRESENTATION FOR DATA THAT IS EITHER HIERARCHICAL OR STRUCTURALLY SELF-SIMILAR

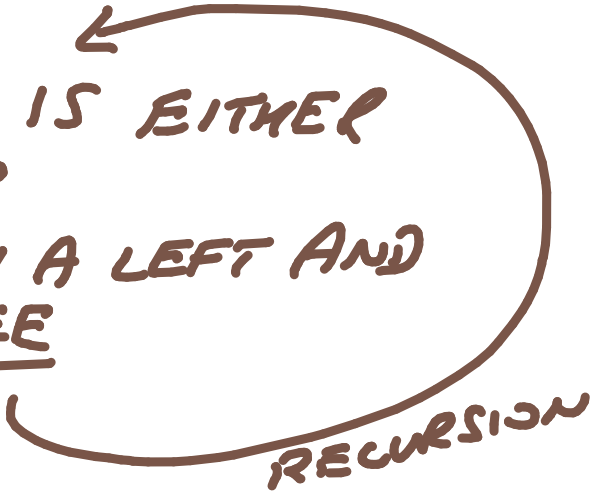
- HTML DOCUMENTS
- PARSE TREES
- GAME TREES

TODAY: TREES AND RECURSION
GENERAL TREES
TREE TRAVERSALS

A (BINARY) TREE IS FUNDAMENTALLY A RECURSIVE STRUCTURE

↳ STRUCTURE DEFINED IN TERMS OF ITSELF

DEF: A BINARY TREE IS EITHER
- EMPTY, OR
- A NODE WITH A LEFT AND RIGHT TREE



MANY OPERATIONS ON RECURSIVE STRUCTURES (ESPECIALLY TREES) ARE MOST EASILY WRITTEN USING RECURSIVE FUNCTIONS

↳ FUNCTIONS THAT CALL THEMSELVES

HERE ARE SOME EXAMPLES WITH THE FOLLOWING TYPE FOR TREES:

```
TYPE NODE STRUCT {
    VALUE INT
    LEFT *NODE
    RIGHT *NODE
}
```

```
FUNC SIZE (T *NODE) INT {
    IF T==NIL {
        RETURN 0
    } ELSE {
        S1 := SIZE(T.LEFT)
        S2 := SIZE(T.RIGHT)
        RETURN 1+S1+S2
    }
}
```

RETURN # NODES IN A TREE

```
FUNC HEIGHT(T *NODE) INT {
    IF T==NIL {
        RETURN 0
    } ELSE {
        H1 := HEIGHT(T.LEFT)
        H2 := HEIGHT(T.RIGHT)
        RETURN 1+MAX(H1, H2)
    }
}
```

RETURN # NODES ON LONGEST PATH FROM THE ROOT OF A TREE

```
FUNC REFLECT(T *NODE) {
    IF T==NIL {
        RETURN
    } ELSE {
        REFLECT(T.LEFT)
        REFLECT(T.RIGHT)
        TEMP := T.LEFT
        T.LEFT = T.RIGHT
        T.RIGHT = TEMP
    }
}
```

SWAP THE LEFT/RIGHT SUBTREE OF EVERY NODE OF A TREE, MODIFYING THE TREE

```
FUNC MAP(T *NODE, F FUNC(INT)INT) {
    IF T==NIL {
        RETURN
    } ELSE {
        MAP(T.LEFT, F)
        MAP(T.RIGHT, F)
        T.VALUE = F(T.VALUE)
    }
}
```

TRANSFORM THE VALUE AT EVERY NODE OF A TREE USING FUNCTION F, MODIFYING THE TREE

```
FUNC MAPI (T *NODE, F FUNC(INT)INT) *NODE {
    IF T==NIL {
        RETURN NIL
    } ELSE {
        NEWL := MAPI(T.LEFT, F)
        NEWR := MADI(T.RIGHT, F)
        NEWV := F(T.VALUE)
        NEWT := &NODE{NEWV, NEWL, NEWR}
        RETURN NEWT
    }
}
```

VERSION OF MAP THAT CREATES A NEW TREE AND DOES NOT MODIFY THE ORIGINAL

ALL OF THESE FUNCTIONS HAVE THE SAME SHAPE, AND THAT IS NOT AN ACCIDENT

↳ THEY FOLLOW THE STRUCTURE OF THE DATA — A TREE IS EITHER EMPTY, OR A NODE WITH TWO TREES

⟶ SO THE FUNCTIONS HAVE A CASE FOR AN EMPTY TREE, AND A CASE FOR A NODE IN WHICH THE FUNCTION IS CALLED RECURSIVELY ON THE SUBTREES

RECIPE FOR WRITING RECURSIVE FUNCTIONS OVER BINARY TREES:

- TAKE TREE T AS INPUT
- CONSIDER TWO CASES
  - IF T IS EMPTY, RETURN A VALUE (NO RECURSION)
  - IF T IS NOT EMPTY, CALL THE FUNCTION RECURSIVELY ON THE LEFT AND THE RIGHT SUBTREE OF T, AND COMBINE THE RESULTS WITH THE VALUE AT THE ROOT OF T TO GET THE RESULT FOR T

FOR A RECURSIVE FUNCTION TO TERMINATE, WE NEED TO MAKE SURE EVERY RECURSIVE CALL IS ON A "SMALLER" INPUT
→ MAKES PROGRESS TOWARDS THE BASE CASE

WE HAVE THAT IN THE RECIPE ABOVE SINCE RECURSIVE CALLS ARE OVER SUBTREES OF T, WHICH ARE OF COURSE SMALLER THAN T!

CORRECTNESS OF RECURSIVE FUNCTIONS (OR WHY DOES RECURSION WORK?) :

- MAKE SURE FUNCTION RETURNS THE CORRECT RESULT FOR THE BASE CASE (E.G., EMPTY TREE)

- UNDER THE ASSUMPTION THAT THE RECURSIVE CALLS RETURN THE CORRECT RESULT, MAKE SURE THAT THE FUNCTION RETURNS THE CORRECT RESULT IN THE RECURSIVE CASE

↳ THIS IS JUST INDUCTION.

(AND INDEED, THE NATURAL NUMBERS FORM A RECURSIVE STRUCTURE:

A NATURAL NUMBER IS EITHER:
- 0, OR
- THE SUCCESSOR OF A NATURAL NUMBER)

YOU CAN CERTAINLY USE MORE COMPLICATED RECURSION SCHEMES, BUT THEY ARE HARDER TO GET RIGHT.

NON-RECURSIVE IMPLEMENTATIONS OF RECURSIVE FUNCTIONS:

SOMETIMES YOU DO NOT WANT A RECURSIVE FUNCTION — BECAUSE OF A LIMITED CALL STACK, SAY.

YOU CAN USUALLY REPLACE A RECURSIVE FUNCTION BY A NON-RECURSIVE FUNCTION THAT USES AN EXPLICIT STACK TO KEEP TRACK OF SUBTREES THAT YOU HAVE NOT WORKED ON YET AS YOU WORK ON A SUBTREE

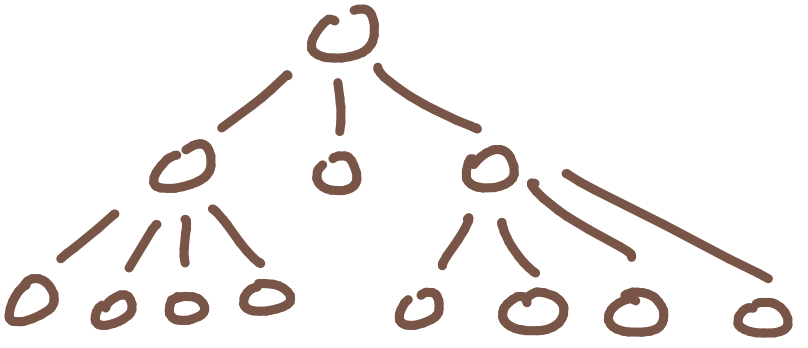
```
FUNC MAP (T*NODE, F FUNC(INT)INT) {
    S := NEWSTACK()
    PUSH(S, T)
    FOR !ISEMPTY(S) {
        TT := POP(S)
        IF TT != NIL {
            TT.VALUE = F(TT.VALUE)
            PUSH(S, TT.RIGHT)
            PUSH(S, TT.LEFT)
        }
    }
}
```

IT GETS TRICKIER TO DO WITH RECURSIVE FUNCTIONS THAT RETURN A RESULT, LIKE SIZE, HEIGHT, MAP!

GENERAL TREES

WE HAVE FOCUSED OUR ATTENTION ON BINARY TREES, BUT MORE GENERAL TREES ARE COMMON

- TREES IN WHICH NODES HAVE AN ARBITRARY NUMBER OF CHILDREN NODES



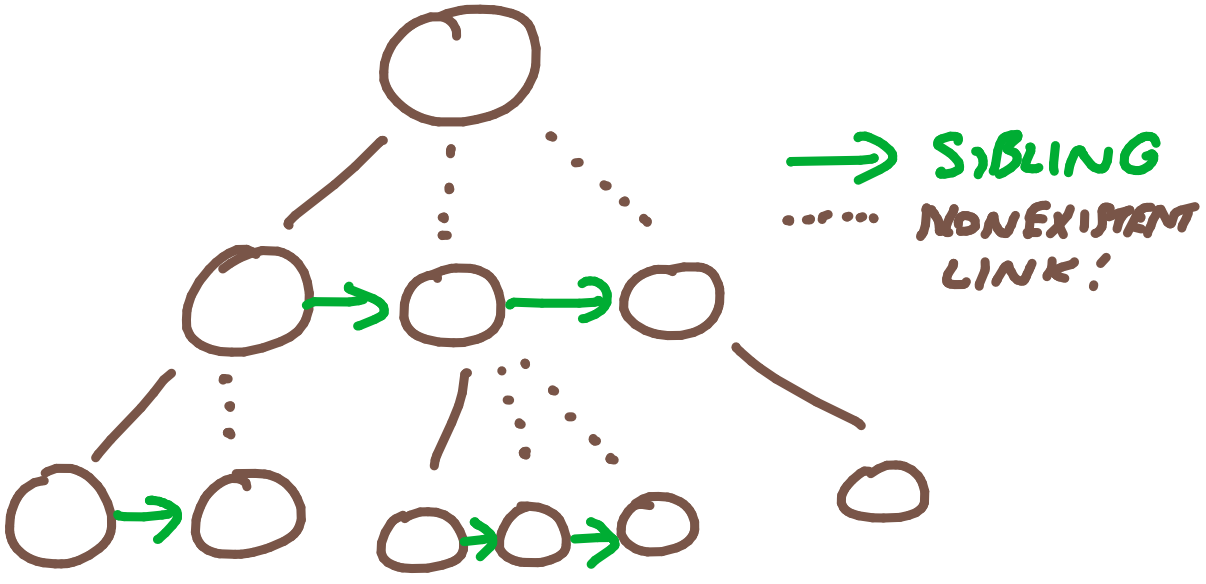
A COMMON REPRESENTATION WHERE THE CHILDREN OF A NODE ARE KEPT AS A LINKED LIST:

```
TYPE NODE STRUCT {
    VALUE INT
    CHILDREN *NODELIST
}

TYPE NODELIST STRUCT {
    NODE *NODE
    NEXT *NODELIST
}
```

AN ALTERNATIVE MORE DIRECT REPRESENTATION

```
TYPE NODE STRUCT {
    VALUE INT
    CHILD *NODE
    SIBLING *NODE
}
```



SO TO FIND ALL THE CHILDREN OF A NODE N, YOU NEED TO NAVIGATE TO THE CHILD OF N, N.CHILD, AND FOLLOW THE SIBLING POINTERS.

THIS IS THE REPRESENTATON OF DOCUMENTS IN WEB BROWSERS — THE SO-CALLED DOM — DOCUMENT OBJECT MODEL

TREE TRAVERSALS

VISIT ALL NODES IN A GENERAL TREE IN SOME ORDER

- TO SEARCH
- TO PERFORM UPDATES
- TO COMPUTE A VALUE

TWO MAIN KIND OF TRAVERSALS DEFINING A SPECIFIC VISIT ORDER

DEPTH-FIRST TRAVERSAL:

> VISIT CHILDREN OF A NODE BEFORE VISITING SIBLINGS

BREADTH-FIRST TRAVERSAL: (AKA LEVEL TRAVERSAL)

> VISIT SIBLINGS OF A NODE BEFORE VISITING CHILDREN

DEPTH-FIRST TRAVERSAL IS NATURALLY RECURSIVE, BUT HERE IS A NON-RECURSIVE STACK-BASED IMPLEMENTATION — LET'S DO SEARCH

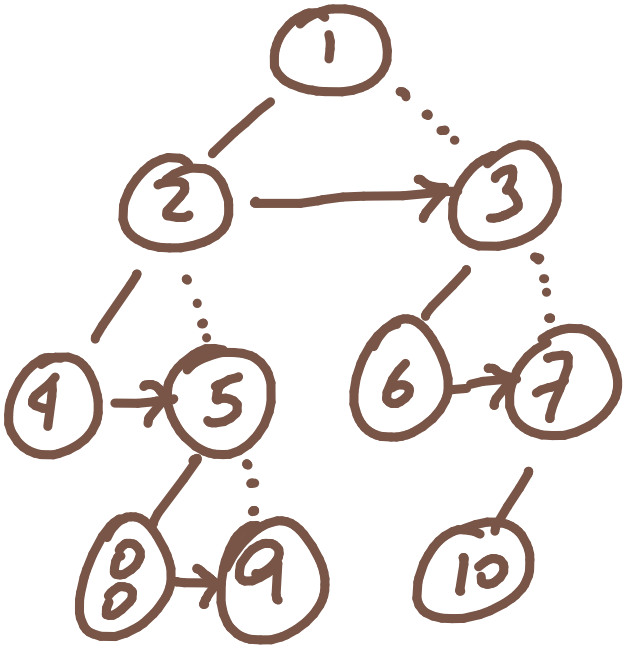
I'M USING THE SECOND REPRESENTATION OF TREES

```
FUNC SEARCHDF (T *NODE, V INT) *NODE {
    S := NEWSTACK()
    PUSH(S, T)
    FOR !ISEMPTY(S) {
        TT := POP(S)
        IF TT != NIL {
            IF TT.VALUE == V {
                RETURN TT
            }
            PUSH(S, TT.SIBLING)
            PUSH(S, TT.CHILD)
        }
    }
    RETURN NIL
}
```

NODE THE ORDER IN WHICH WE PUSH — WE FIRST PUSH THE SIBLING, THEN THE CHILD. THIS ENSURES THE CHILD WILL BE POPPED FIRST, ENSURING DEPTH-FIRST TRAVERSAL.

WHAT ABOUT BREADTH-FIRST TRAVERSAL?

YOU MIGHT THINK IT SUFFICES TO SWAP THE ORDER OF THE PUSHES IN THE ABOVE CODE, BUT THAT WON'T WORK



YOU'LL END UP VISITING 10 BEFORE VISITING 4, WHICH IS NOT BREADTH-FIRST TRAVERSAL

THE SOLUTION IS TO USE A QUEUE INSTEAD OF A STACK!

(WE ALSO NEED TO ENQUEUE ALL SIBLINGS AT ONCE)

```
FUNC SEARCHBF(T *NODE, V INT) *NODE {
    S := NEW QUEUE()
    ENQUEUE(S, T)
    FOR !ISEMPTY(S) {
        TT := DEQUEUE(S)
        IF TT != NIL {
            IF TT.VALUE == V {
                RETURN TT
            }

            CURR := TT.CHILD
            FOR CURR != NIL {
                ENQUEUE(S, CURR)
                CURR = CURR.SIBLING
            }
        }
    }
    RETURN NIL
}
```