Introduction to Trees and Traversals

Florestan Brunck

April 18, 2022

This introduction to trees builds from a transcription of a lecture given by Luc Devroye on the 28th of February 2019 for the undergraduate class on Data Structures and Algorithms at McGill University (COMP 251). This lecture introduces trees and their traversals.

1 *Introduction and Definitions*

Trees are fundamental objects which can be endowed with various structures according to more specific purposes. In our particular context, the main characteristic of trees we will exploit is their recursive character. To hint at the underlying data structures at work, we refer to the vertices of a graph by *nodes* in the present chapter.

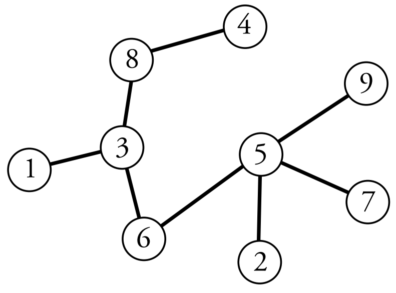


Figure 1: An example of a free tree.

Definition 1. A **free tree** is a connected acyclic graph.

Definition 2. A **rooted tree** is a free tree with a single marked node called the *root*.

Equivalently, we have the following recursive definition for rooted trees.

Definition 3 (Recursive definition)**.** A **rooted tree** T is a non-empty finite set of nodes such that:

(i) A single designated node is marked as the *root* of T ($root(T)$)

(ii) The remaining nodes (with the exception of the root) are partitioned into disjoint sets $T_1, \ldots, T_n$, each of which is a tree. The trees $T_1, \ldots, T_n$ are called the *subtrees* of the root.

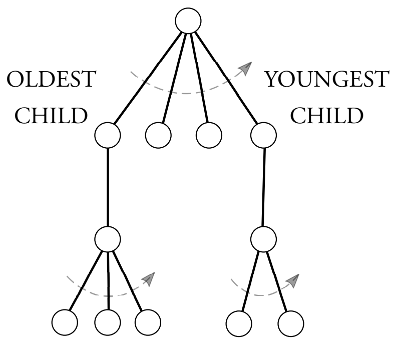


Figure 2: An example of an ordered rooted tree (or plane tree).

Definition 4. An **ordered rooted tree** (or **plane tree**) is a rooted tree for which the recursive criterion (ii) in definition 3 associates an n-tuple of disjoint Trees $(T_1, \ldots, T_n)$ to the root (and not just n disjoint trees). That is, each subtree is given an *order*.

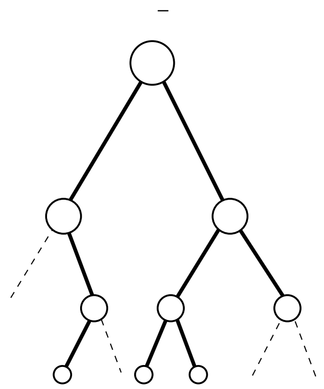


Figure 3: An example of a binary tree.

Definition 5. A ***k*-ary tree** is a tree in which every node has exactly k (possibly empty) subtrees. In the particular instances where $k = 2$ and $k = 3$ we call such trees *binary* and *ternary* trees respectively. If there is an order on subtrees and each child has a *position* $P \in \{1, \ldots, k\}$, we call such a tree a **position tree**.

Remark 6. The definition of a k-ary tree in computer science is different from the regular definition of a k-ary, in which all nodes have degree k. Indeed, with our definition, the root of the tree has degree $k-1$ (see Figure 4 and 5).

Pictorial notation: When it comes to representing trees in diagrams, the *top-down* presentation is preferred (we refer the reader to the sidenote[1] quotation of Knuth's *The Art of Programming* to motivate this choice).

Rather than formally define the remaining tree terminology we point the reader to figure below.

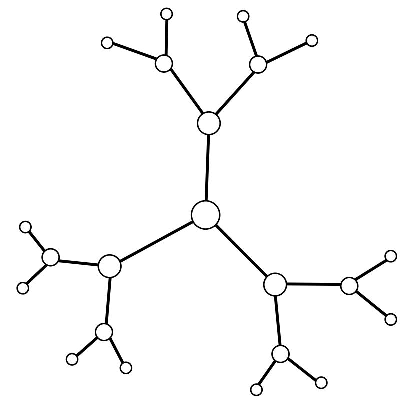


Figure 4: a ternary tree, in the common sense of the term (where every vertex is of degree 3).

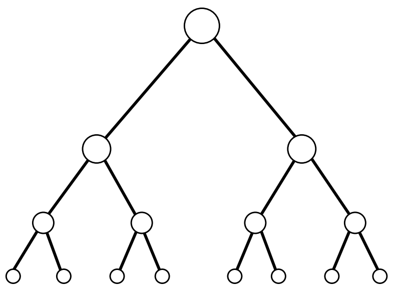
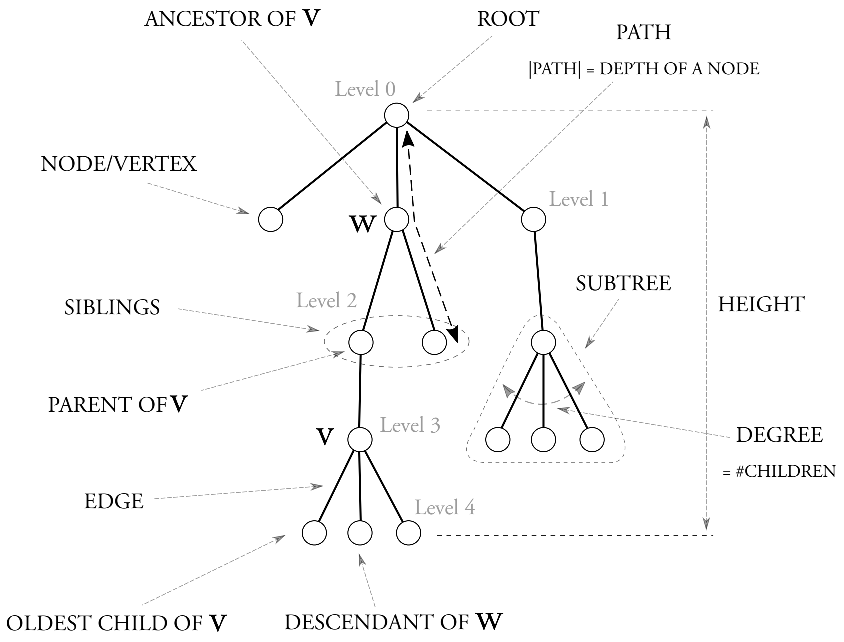


Figure 5: A binary tree, in the computer science of the term (where every vertex *except for the root* is of degree 3).



2 Complete Binary Trees

Definition 7. A **complete binary tree** is a binary tree for which all the levels are filled, with the possible exception of the last one, which is filled from left to right.

Let h denote the height of a complete binary tree and n its total number of node. The previous definition yields the following inequality:

[1] *"It may seem that the [bottom-up representation] would be preferable simply because that is how trees grow in nature; in the absence of any compelling reason to adopt any of the other three forms, we might as well adopt nature's time-honored tradition. With real trees in mind, the author consistently followed a root-at-the-bottom convention as the present set of books was first being prepared, but after two years of trial it was found to be a mistake: Observations of the computer literature and numerous informal discussions with computer scientists about a wide variety of algorithms showed that trees were drawn with the root at the top in more than 80 percent of the cases examined. There is an overwhelming tendency to make hand-drawn charts grow downwards instead of upwards (and this is easy to understand in view of the way we write); even the word "subtree", as opposed to "supertree", tends to connote a downward relationship. [...] Henceforth we will almost always draw trees [...] with the root at the top and leaves at the bottom. Corresponding to this orientation, we should perhaps call the root node the apex of the tree, and speak of nodes at shallow and deep levels." [1]*

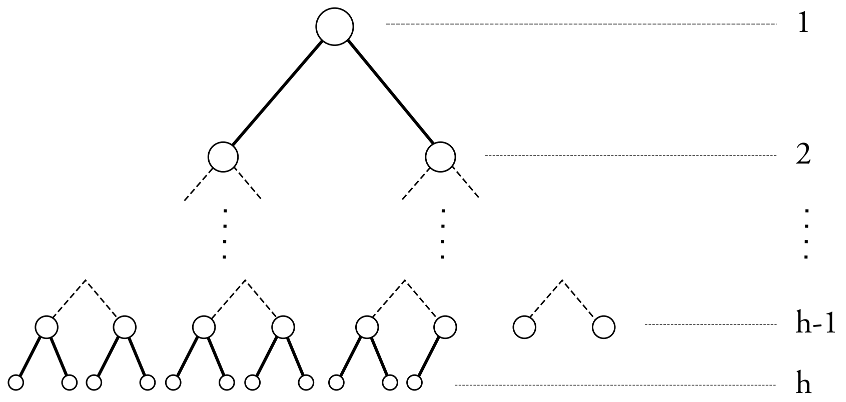


Figure 6: Computing the height of a complete binary tree on n nodes.

$$1+2+\ldots+2^{h-1} < n \leq 1+2+\ldots+2^{h}$$
$$2^{h}-1 < n \leq 2^{h+1}-1$$
$$2^{h} \leq n < 2^{h+1}$$
$$h \leq \log_2 n < h+1$$

From which we infer that $h = \lfloor \log_2 n \rfloor$.

Exercise 8. *Derive a similar formula for a complete k-ary tree.*

3 *Binary Trees and Ordered Trees*

There is a very useful trick which allows one to reduce *every tree* to a *binary tree*. Namely, there is a *bijection* between the set of binary trees on $n-1$ nodes and the set of ordered trees on n nodes. The bijection is the described as follows (see Fig. 7): start with the root of an ordered tree on n nodes and create a first corresponding node in the binary tree we want to construct and associate uniquely to this ordered tree. The oldest child of the root is to be placed as a left child of the newly created node, while the other siblings will be placed as successive right children of this left child. We then proceed inductively on the rest of the tree. This process may be seen as a

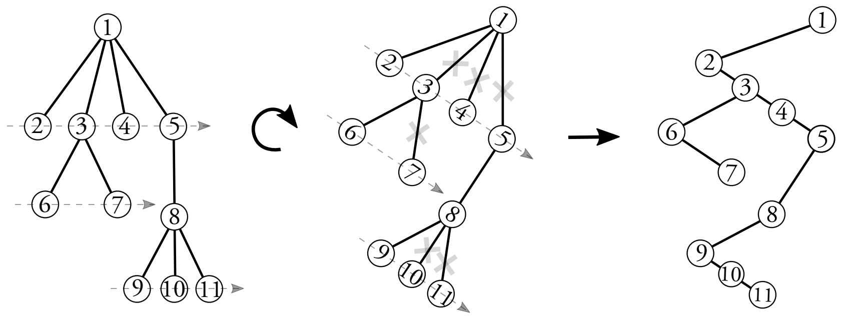


Figure 7: There is a bijection between ordered trees on n nodes and binary trees on $n-1$ nodes.

"rotation" of the original ordered tree. So far, it would seem that this process defines a bijection between ordered trees on n nodes and binary trees on n nodes. Notice, however, that the root of the binary we obtained contains no information on the tree and is redundant. Indeed, the root cannot have a right child (by construction) and we can therefore trim the root and always tacitly assume we have to start building the root first when going back from a binary tree to an ordered tree.

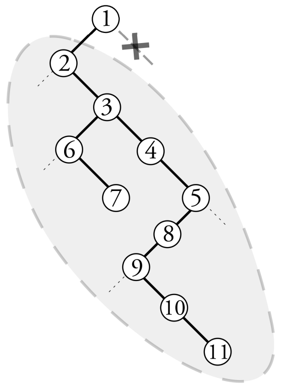


Figure 8: The root is redundant in the binary tree we obtain, thus defining a bijection on binary trees with $n - 1$ nodes.

4 Implementations

A first straight-forward way to implement a tree data structure is to associate linked *cells* to each node which each consist of a *data element* and *two pointers* to the left child and the right child of the associated node in the tree. In this construction, the tree *is* the pointer to the cell associated with the root node.

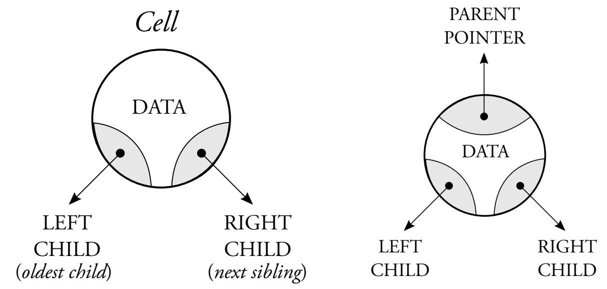


Figure 9: The standard implementation of a tree, which sometimes include a pointer to the parent node.

Very concretely, this can be implemented with an array quite easily, the following example corresponds to the tree shown in Figure 10.

Node #	1	2	3	4	5	6	7	8	9	10
Data										
Left Child	5	∅	∅	∅	6	2	∅	∅	∅	9
Right Child	3	∅	7	∅	10	11	∅	∅	∅	8

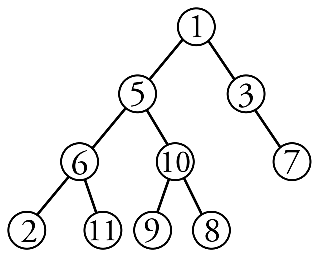


Figure 10: Example of an array implementation of a tree.

4.1 Implicit Storage of a Complete Binary Tree

In the particular case of complete binary trees there is an elegant way to store a tree in a linear array. Numbering nodes from top to bottom and from left to right within layers, we proceed as follows:

1. We store the i-th node in position $[i]$ in our array
2. The left child of the i-th node is stored in position $[2i]$
3. The right child of the i-th node is stored in position $[2i + 1]$

4. By construction we then get that the parent node of the i-th node is stored in position $\lfloor \frac{i}{2} \rfloor$

With this construction, we see that a necessary and sufficient condition for the i-th node to be a leaf is to satisty the inequality $2i > n$, if n denotes the number of nodes in the tree.

Another elegant property of this construction is the fact that the position of the node in the array (written in binary) directly reads a path from the root to the node (if we omit the first bit). This is of course due to the fact that the k-th digit in the binary expansion is either 0 or 1 depending on whether the node was obtained as a left or right child from its parent. As an example, the 10-th node reads $(1010)_2$ in binary, corresponding to the path 010, i.e., starting from the root: left, then right, then left (see Figure 11, in which node numbers refer to the position of the node in the array).

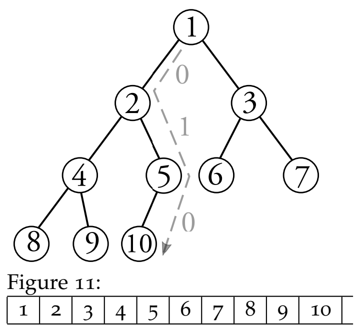


Figure 11: Example of implicit storage of a complete binary tree via an array.

5 *Properties of Trees*

In this section we list various properties on trees which are useful to keep in mind.

1. Every tree on n nodes has $n - 1$ edges.
2. In a binary tree, let n_i denote the number of nodes with i children. Then:

$$n = n_0 + n_1 + n_2 \implies n - 1 = n_1 + 2n_2$$

3. The number of binary trees on n nodes is given by the n-th Catalan number: $\frac{1}{n+1}\binom{2n}{n}$.[2]

[2] We prove this identity later on in this chapter (c.f. section 8).

6 *Traversals*

6.1 *Traversals in Binary Trees*

Unlike lists, trees are not inherently 1-dimensional objects. For representation purposes at the computer level and many other purposes one needs ways to linearise trees. As we will see in this section, trees can be linearised in multiple ways.

Definition 9. By a **traversal** of an ordered tree we mean any means of exhaustively listing the nodes of the tree as they are visited *exactly once*.

Traversals can become quite elaborate, but we concern ourselves here only with the four basic schemes for *binary trees*, which are in a sense more fundamental. We list them here together with the ordering associated to the example of figure blabla:

1. Level order:	1 2 3 4 5 6 7 8 9
2. Preorder:	1 2 4 5 7 8 3 6 9
3. Inorder:	4 2 7 5 8 1 6 9 3
4. Postorder:	4 7 8 5 2 9 6 3 1

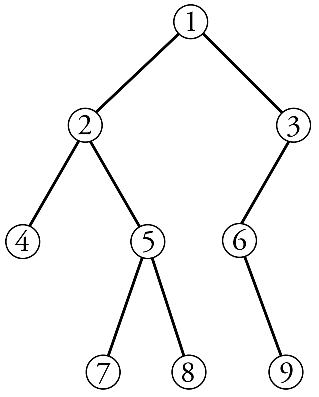


Figure 12: Our example of a binary tree, which we traverse in 4 different ways.

6.2 Level Order

Similarly to the breadth-first-search algorithm, the level order explores a tree "level by level" where a level consists of nodes a given combinatorial distance away from the root. The implementation also relies on a queue.

```
LEVEL ORDER($t$)
// The input $t$ is a pointer to a tree
MAKENULL($Q$)
ENQUEUE($t, Q$)
while $|Q| \geq 1$
    $v \longleftarrow$ DEQUEUE($Q$)
    Visit the node pointed to by $v$
    For all children $v$ of $t$, from left to right
        ENQUEUE($v, Q$)
```

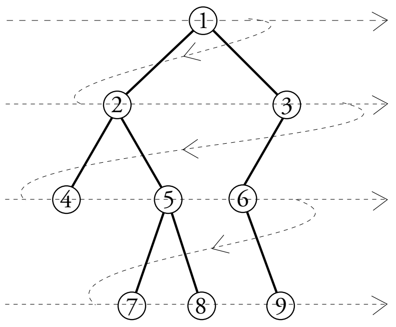


Figure 13: A heuristic picture for the level order traversal of a binary tree.

6.3 PreOrder, InOrder, PostOrder

These three traversal are all defined recursively but differ in the particular sequence of recursive calls.

Preorder	Inorder	Postorder
Visit the root node	Recurse on the left subtree	Recurse on the left subtree
Recurse on the left subtree	**Visit the root node**	Recurse on the right subtree
Recurse on the right subtree	Recurse on the left subtree	**Visit the root node**

The code for each traversal is exact in all accounts save for the order of the visit of the root. We provide the code for the Preorder traversal.

```
PREORDER($t$)
if $t \neq nil$
    Visit $t$
    $v \longleftarrow$ Left child of $t$
    PREORDER($v$)
    $v \longleftarrow$ Right child of $t$
    PREORDER($v$)
```

Remark 10. Seeing a tree as a coast line, one can see the Preorder traversal as the labelling that a boat leaving from the root and going along the coast line before returning to the root would obtain (see figure blabla).

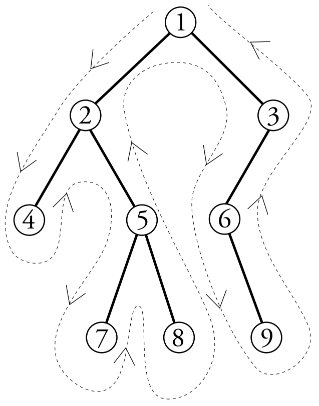


Figure 14: One can look at the preorder traversal of a binary tree as the listing of a boat sailing along the coast line defined by the tree.

6.4 *Traversals in Non-Binary Ordered Trees*

The Level Order traversal can be used without changes on any ordered tree (in line 7 of the example, there is no specification on the number of children). Likewise, the Preorder and Postorder traversals can also be generalised to ordered tree by making the following adjustments:

PREORDER	POSTORDER
Visit the root node	Recurse on the children subtrees from left to right
Recurse on the children subtrees from left to right	**Visit the root node**

For the Inorder traversal however, there is no longer a canonical choice of splitting of the children subtrees. In the case of a binary tree we can visit the root in between recursive calls on left and right subtrees, but for a k-ary tree for example, there would be $k-1$-choices of Inorder traversals as we may visit the root in between any of the k recursive calls on the children nodes.

6.5 *Ancestor Information*

Given the preorder and postorder numbers associated to each node in an ordered tree, one can retrieve all the information about the ancestors and the descendants of a given node. This is because for a given node v, another node u has a preorder either strictly less or strictly greater than that of v and likewise for its postorder. There is then a partition of the nodes in four sets corresponding to the 2 possible categories that a node falls in for each of the 2 orders (see Figure 15):

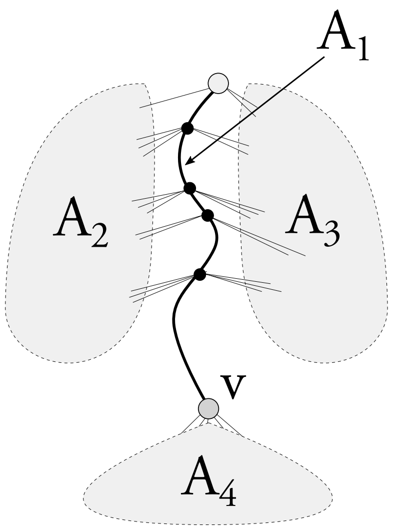


Figure 15: Retrieving the ancestry of a given node using preorder and postorder traversals.

	PREORDER(w)<PREORDER(v)	PREORDER(w)>PREORDER(v)
POSTORDER(w)<POSTORDER(v)	$w \in A_2$	$w \in A_4$
POSTORDER(w)>POSTORDER(v)	$w \in A_1$	$w \in A_3$

The **Ancestors** of v correspond precisely to the set of nodes with lesser preorder and greater postorder than that of v while the **descen-**

dants of v correspond to the nodes with greater preorder and lesser postorder than that of v.

6.6 Non-Recursive, Stack-Based Algorithms for Traversals

We have already seen recursive algorithms for the preorder and postorder traversals in *binary trees*. They can, alternatively, be executed without the use of recursion with the use of a stack to store the successive nodes under consideration. Just like before, the difference between the two traversals will then consist in the order in which we visit a node and push its children on the stack. In a preorder traversal, the root node is visited and children are pushed on the stack from right to left. In a postorder traversal, if the root node is unmarked, it is marked and pushed on the stack before its children (again, from right to left; they are unmarked); if the root node is marked, it is visited and nothing is pushed on the stack. So, an extra bit for marking suffices.

Remark 11. There is a subtlety in the order in which children are being pushed on the stack: it is *reversed* compared to the order of the recursive calls which were from left to right children. This is because we want to consider the left subtree first and therefore to push it on the stack last.

More precisely, we have the following code for a **stack-based traversal** in a binary tree:

```
PREORDER($t$)
1  MAKENULL($S$), PUSH($t, S$)
2  while NOTEMPTY($S$)
3        $v \longleftarrow$ POP($S$)
4        Visit $v$
5        if RIGHT[$v$] $\neq$ NIL
6              PUSH(RIGHT[$v$], $S$)
7        if LEFT[$v$] $\neq$ NIL
8              PUSH(LEFT[$v$], $S$)
```

One possible issue with the physical implementation of stack-based traversals on a computer lies in controlling the size of the stack and ensuring some sort of upper bound to make sure either that machine limitations are met or simply that as much space is saved as possible by our algorithm. To that effect one can modify the previous algorithms to prioritise the following heuristic: *always push the largest subtree first on the stack*. This is again because that way we will traverse the smaller subtrees first and avoid adding up large subtrees onto the stack for later consideration.

With such an algorithm, which one might call a **parsimonious stack-based traversal**, we can guarantee that the size of the stack never exceeds $\log_2 n$ where n is the number of nodes of the tree, this is because we can only push one node on the stack for each level (see Figure 16).

We also have that at any given time during the execution of the algorithm, the size of the subtree under consideration under the node currently being visited is at most $\frac{n}{2^{|S|}}$ where $|S|$ is the size of the stack at that given time. This is easy to see by induction since pushing a node on the stack corresponds to a choice of right or left subtree, but since we always push the largest subtree first on the stack, it must be that the subtree which we selected under the node being visited contains less than half of the nodes in the tree rooted at that node.

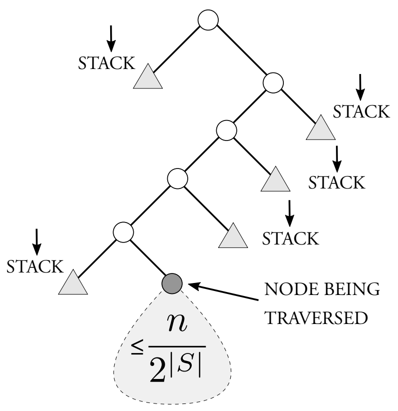


Figure 16: Parsimonious stack-based traversals of trees.

6.7 *Non-Recursive, Stackless Algorithms for Traversals*

In a way, the use of the stack in the previous algorithms was solely to encode parent information for nodes and allow us to climb back up the tree until the parent node of the subtree we were just traversing. Assuming we are being given *parent pointers* for a binary tree, one might expect to be able to get rid of recursion and stacks altogether and this is indeed the case.

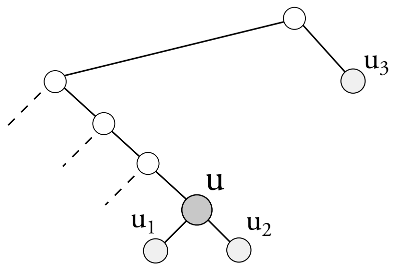


Figure 17: The NEXT function sends $u \to u_1$, $u_1 \to u_2$ and $u_2 \to u_3$.

Notice first that, for an *internal* node, the next node in the preorder will be its left child. So that, going down from the root, we can keep going down the tree, always selecting the left child and incrementing the order by one. The problem now is then to understand what we ought to do once we reach a *leaf*. Well, since we are given parent pointers, we can climb up the tree until we reach a node with two children and whose left subtree we ascended from. That node is then the next node in the preorder. We can thus define, without the use of recursion or the need for a stack the following NEXT function which will return, given a node as input, the next node in the preorder of the given tree.

```
NEXT(u)
1  if LEFT[u] ≠ nil
2      return LEFT[u]
3  elseif RIGHT[u] ≠ nil
4      return RIGHT[u]
5  else
6      v ⟵ PARENT(u)
7      while v ≠ nil and (RIGHT[v] = u or nil)
8          u ⟵ v
9          v ⟵ PARENT[v]
10     if v = nil
11         return nil
12     else
13         return RIGHT[v]
```

The new stackless preorder algorithm is then simply:

```
PREORDER(t)
1  u ⟵ t
2  while u ≠ nil
3      Visit u
4      u ⟵ NEXT(u)
```

7 *Compact Representation of Trees*

Consider the problem of encoding binary trees using the minimal number of bits possible. Recall that they are $\frac{1}{n+1}\binom{2n}{n}$ binary trees on n nodes. Recall as well that given a set S of N objects, there is a theoretical lowerbound of $\log_2 N$ bits required to represent any single object in S so that it may be differentiated from any other element in S. This tells us that the minimal number of bits required to store binary trees on n nodes is not less than:

$$\begin{aligned}\log_2\left[\frac{1}{n+1}\binom{2n}{n}\right] &\geq \log_2\left[\frac{1}{n+1}\left(\frac{\sum_{i=0}^{2n}\binom{2n}{i}}{2n+1}\right)\right]\\ &= \log_2\left[\frac{2^{2n}}{(n+1)(2n+1)}\right]\\ &= 2n - o(n)\end{aligned}$$

where the first inequality comes from the fact that the binomial $\binom{2n}{i}$ seen as a function of i for n fixed has its maximum at $i = n$ (see Figure 18).

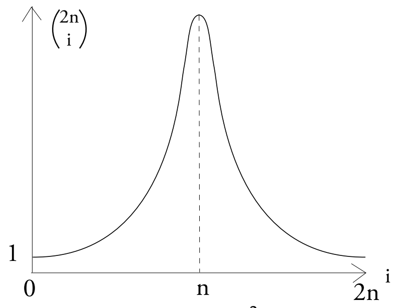


Figure 18: The binomial $\binom{2n}{i}$, seen as a function of i with n fixed reaches a maximum fo $i = n$.

As it turns out, we can in fact devise a way of encoding binary trees which uses exactly $2n$ bits! Notice that any traversal encodes all the information we need save for the type of nodes being traversed, i.e the traversal does not tell us whether we the nodes are leaves, nodes with two children or nodes with a left/right child. Without this information there is no way of differentiating for example the two trees on two nodes (the first with a root and a left child and the second with a root and a right child) since they both yield the same preorder traversal.

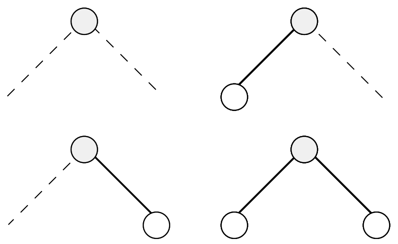


Figure 19: The four different types of nodes in a binary tree, to which we associate the unique binary *signatures* 00, 01, 10 and 11 respectively.

Observe then that there are exactly 4 types of nodes (see Figure bla), to which we can then associate a binary *signature* of 2 digits. Our encoding is then simply the string resulting from substituting the number of a node by its signature in the traversal (see Figure bla)!

Exercise 12. *Write an algorithm to collect the signatures of each node in a preorder traversal.*

Exercise 13. *Reconstruct the binary tree in linear time from each of the three $2n$-bits representations associated with preorder, inorder and postorder traversals.*

Exercise 14. *How many bits does one need to represent the shape of an ordered tree on n nodes?*

Exercise 15. *Give a linear time algorithm for computing each representation of a binary tree.*

8 Counting Binary Trees & Catalan Numbers

We first establish a 1-to-1 correspondence between binary trees on n nodes and walks with steps 1 of -1 on the *positive* integers, starting at 0 and ending at 0 after $2n$ steps. An equivalent way to look at these walks which we will adopt is to consider lattice paths from $(0,0)$ to $(2n,0)$ with steps $(1,1)$ and $(1,-1)$ which never fall under the x-axis[3]. In order to make this correspondence clear, we first associate to every node in a binary tree the markings consisting of a "$+$" sign, respectively of a "$-$" sign on its left, respectively bottom side (See Figure 21). That way, looking back at remark 10 we can perform a preorder traversal and collect the signs of each node as the traversal is executed. We can then associate to this traversal the lattice path resulting from the following correspondence:

[3] Such lattice paths are called *Dyck paths*. See an example in Figure bla. They can be viewed alternatively as "staircase" lattice paths joining $(0,0)$ to (n,n) with steps $(0,1)$ and $(1,0)$ which do not cross the main diagonal of points (i,i), $i \leq n$.

1. If a "$+$" sign is collected, execute the $(1,1)$ step.
2. If a "$-$" sign is collected, execute the $(1,-1)$ step.

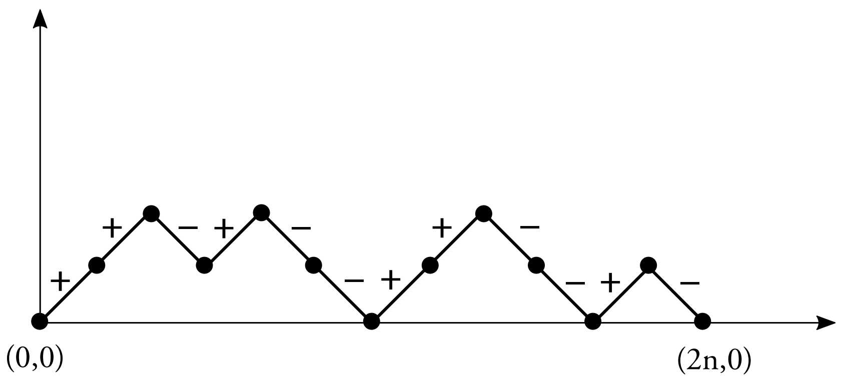


Figure 20: Seeing binary trees as Dyck paths.

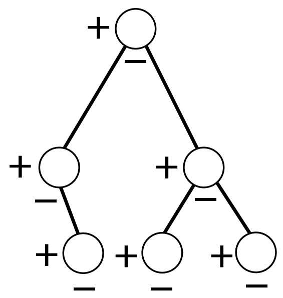


Figure 21: The markings describing the bijection between binary trees and Dyck paths. A preorder traversal of this tree would collect the markings in the following order: $++-+--++--+-$.

This clearly defines a lattice path from $(0,0)$ to $(2n,0)$ since at the end of the traversal, all the signs have been collected and "$+$" and "$-$" signs are present exactly in equal amount. It is not hard either to see that the path stays positive (does not cross the x-axis) since for each single node, the preorder traversal necessarily collects the "$+$" sign first (we start going along the coastline down from the left side of the root).

We have thus established that:

$$\#\big(\text{BIN. TREES ON } n \text{ NODES}\big) = \#\big(\text{POSITIVE PATHS: } (0,0) \to (2n,0)\big)$$

Since all the lattice paths which fail to be positive necessarily cross the line ℓ with y-coordinate -1 we can rewrite the number of positive lattice paths as the following difference:

$$\#\big(\text{PATHS: } (0,0) \to (2n,0)\big) - \#\big(\text{PATHS: } (0,0) \to (2n,0) \text{ MEETING } \ell\big)$$

The number of (all the) lattice paths from $(0,0)$ to $(2n,0)$ is simply $\binom{2n}{n}$. Indeed for such a path, we need to go up exactly n times and down exactly n times as well, there are then $\binom{2n}{n}$ different ways of choosing at what step we ought to go up or down (the steps at which we go up, resp. down steps constitute a subset of size n inside the set of all $2n$ steps).

Now all that is left to do is count the number of lattice paths starting at $(0,0)$ and ending at $(2n,0)$ which cross the line ℓ. Consider such a lattice path and note that either the path *crosses* ℓ or it stays above it.

Suppose then the path under consideration crosses ℓ and look at the first point of crossing of this path with ℓ. We can then perform a reflection of the remaining section of the path across ℓ to obtain a path which will now end at $(2n,-2)$. We can perform this operation for any given path which crosses ℓ (see Figure 22). But notice as well that any lattice path from $(0,0)$ to $(2n,-2)$ must necessarily

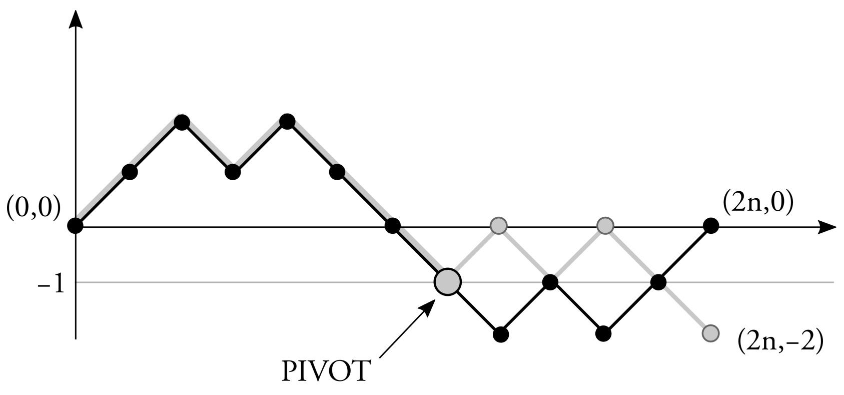


Figure 22: A lattice path crossing the line ℓ of ordinate -1 and the corresponding pivot point and reflected path (shown in grey).

cross ℓ. So this reversible process of reflection defines a bijection between paths from $(0,0)$ to $(2n,0)$ crossing ℓ and paths from $(0,0)$ to $(2n,-2)$. Following our previous reasoning it is immediate that the number of paths joining $(0,0)$ to $(2n,-2)$ is equal to $\binom{2n}{n+1} = \binom{2n}{n-1}$ (there are now $n+1$ down steps, or $n-1$ up steps).

We thus obtain that the number of lattice paths staying above the x-axis is precisely:

$$\binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1}\binom{2n}{n}$$

This number is called the *n-th Catalan number*[4].

[4] The Catalan numbers appear in a lot of counting problems in combinatorics and there is a plethora of well-known bijections between sets of objects which are counted by Catalan numbers. For a reference see [2], [5], [4] or [3] for a more introductory presentation of some of these bijections. Among the most famous ones we mention the number of balanced bracketings/parenthesizations of a given string of $n+1$ characters and the number of triangulations of a convex $(n+2)$-gon.

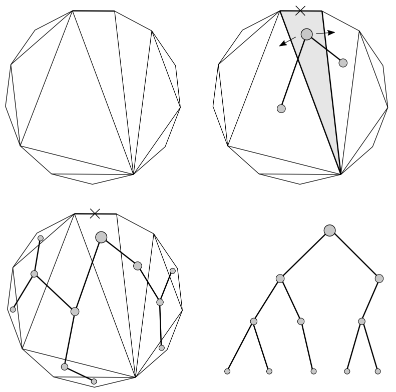


Figure 23: A pictorial description of the bijection between binary trees and triangulations on convex $(n+2)$-gons.

9 *Arithmetic Expressions*

9.1 *Expression Trees*

The problem of evaluating arithmetic expressions is an interesting example of the use of binary trees and traversals. Notice indeed that any arithmetic operation involves exactly two *operands* and we can therefore construct an *expression tree* for any arithmetic expression where the internal nodes correspond to the operators and the leaves the operands (see Figure 24). Different traversals of such an expression tree correspond to different notation systems:

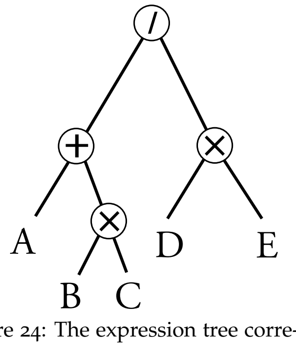


Figure 24: The expression tree corresponding to the arithmetic expression "$\frac{A+(B\times C)}{D\times E}$"

1. The *Prefix/Polish* notation corresponds to a *preorder* traversal of the expression tree.

   Example: $/ + A \times B\ C \times D\ E$

2. The *Postfix/Reverse Polish* notation corresponds to a *postorder* traversal of the expression tree[5].

   Example: $A\ B\ C \times + D\ E \times /$

[5] Postfix notation is used for example in the PostScript language.

9.2 *Evaluating Postfix Expressions*

If given a postfix expression we can evaluate it using the following simple algorithm relying on a stack:

```
EVALUATE($E$)
// The input is an arithmetic expression/string $E$ in postfix
// notation, consisting of a concatenation of characters $x \in E$
// which are thus all either operands or (binary) operators

MAKENULL($S$)
for $x \in E$ (read from left to right)
    if $x \in \{$Operands$\}$
        PUSH($x, S$)
    else
        $A \longleftarrow$ POP($S$)
        $B \longleftarrow$ POP($S$)
        $C \longleftarrow AxB$
        PUSH($C, S$)
Return POP($S$)
```

Exercise 16. *How many bits suffice to store the shape of an n-node expression tree ?*

References

[1] Donald E Knuth. *The Art of Computer Programming, Volumes 1-4A Boxed Set*. Addison-Wesley Professional, 2011. ISBN 978-0321751041.

[2] Igor Pak. Catalan numbers page. URL `https://www.math.ucla.edu/~pak/lectures/Cat/pakcat.htm`.

[3] Richard P Stanley. Catalan numbers. URL `http://www-math.mit.edu/~rstan/transparencies/china.pdf?fbclid=IwAR1LrjTdL7OZ3xeaPfJmxultM7a8EoiMXW4usSIxq2ugBO5Ck66Dg1pVGTA`.

[4] Richard P. Stanley. *Enumerative Combinatorics*, volume II. Cambridge University Press, 2nd edition, 1999. ISBN 0-521-56069-1.

[5] Richard P. Stanley. *Enumerative Combinatorics*, volume I. Cambridge University Press, 2nd edition, 2012. ISBN 978-1-107-01545-5.