5.4 *N*-ary Trees

A binary tree restricts the number of children of each node to two. A more general *N*-ary tree restricts the number of children to *N*.

5.4.1 Description

An *N*-ary tree is a tree where each node has at most *N* children where each of the children are non-overlapping *N*-ary trees. For example, a 3-ary tree or *ternary* tree restricts each node to having at most three children. A quaternary tree limits its children to four. Figure 1 shows two examples of a ternary tree and a perfect quaternary tree of height 2.

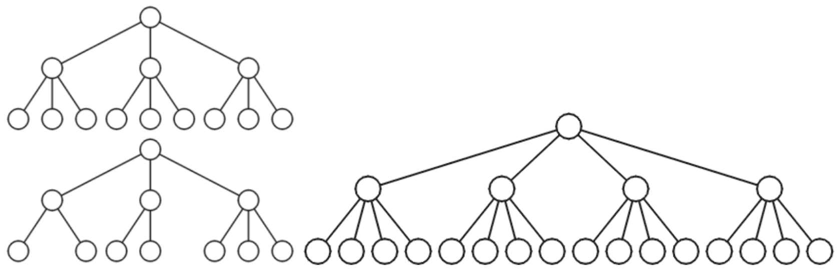
Figure 1. Ternary and a quaternary tree.

As an aside, the following terminology may be used to represent the different bases:

2	3	4	5	6	7	8	9	10	12
binary	ternary	quaternary	quinary	senary	septenary	octal	nonary	decimal	duodecimal

5.4.2 Theorems

We will now proceed to prove a number of theorems about perfect binary trees.

5.4.2.1 Nodes in a Perfect *N*-ary Tree

Theorem

A perfect binary tree of height *h* has $\frac{N^{h+1}-1}{N-1}$ nodes.

Proof: We could use recursion; however, another approach is to observe that the maximum number of children at depth k is N^k. Thus, the total number of children is $1 + N + N^2 + \cdots + N^h$. This is a geometric sum and thus we have

$$n = \sum_{k=0}^{h} N^k = \frac{N^{h+1}-1}{N-1}$$

When $N = 2$, this simplifies to our formula for perfect binary trees: $2^{h+1} - 1$.

5.4.2.2 Logarithmic Height

Solving this for *h* gives the formula $h = \log_N\left(n(N-1)+1\right)-1$; however,

$$\lim_{n\to\infty}\frac{\log_N\left(n(N-1)+1\right)-1}{\log_N(n)} = \lim_{n\to\infty}\frac{\dfrac{N-1}{\left(n(N-1)+1\right)\ln(N)}}{\dfrac{1}{n\ln(N)}} = \lim_{n\to\infty}\frac{n(N-1)}{nN-n+1} = \lim_{n\to\infty}\frac{N-1}{N-1} = 1$$

and therefore $\log_N(n)$ is a reasonable approximation of the height of a perfect *N*-ary tree with *n* nodes.

5.4.2.3 *N*-ary Trees versus Binary Trees

The ratio of the heights of a binary tree containing *n* nodes and an *N*-ary tree containing *n* nodes is

$$\frac{\log_2(n)}{\log_N(n)} = \frac{\log_2(n)}{\dfrac{\log_2(n)}{\log_2(N)}} = \log_2(N)$$

and therefore, the height of a corresponding binary tree will always be approximately a constant multiple times that of an *N*-ary tree. For example, a perfect binary tree will be approximately 3 times deeper than the corresponding octal tree.

5.4.2.4 Complete *N*-ary trees

The height of a complete *N*-ary tree containing *n* nodes is

$$h = \left\lfloor \log_N\left((N-1)n\right)\right\rfloor$$

Like complete binary trees, complete *N*-ary tree can be stored efficiently using an array:

1. Unlike a complete binary tree, we will assume the root is at index $k = 0$,
2. The parent of a node with index *k* is located at $\left\lfloor \dfrac{k-1}{N}\right\rfloor$, and
3. The children of a node with index *k* are located at $kN + j$ for $j = 1, 2, \ldots, N$.

Unlike binary trees, there is no computational advantage by placing the root at index $k = 1$.

5.4.3 Implementation

From previous projects, one might consider implementating an *N*-ary tree as follows:

```
#include <algorithm>

template <typename Type>
class Nary_tree {
    private:
        Type element;
        int N;
        Nary_tree **children;

    public:
        Nary_tree( Type const & = Type(), int = 2 );
        // ...
};

template <typename Type>
Nary_tree<Type>::Nary_tree( Type const &e, int n ):
element( e ),
N( std::max( 2, n ) ),
children( new *Nary_tree[N] ) {
    for ( int i = 0; i < N; ++i ) {
        children[i] = 0;
    }
}
```

However, this requires the allocation of dynamic memory in all cases. An alternate design is to use templates:

```
#include <algorithm>

template <typename Type, int N>
class Nary_tree {
    private:
        Type element;
        Nary_tree *children[std::max(N, 2)];   // an array of N children

    public:
        Nary_tree( Type const & = Type() )
        // ...
};

template <typename Type, int N>
Nary_tree<Type, N>::Nary_tree( Type const &e ):
element( e ) {
    for ( int i = 0; i < N; ++i ) {
        children[i] = 0;
    }
}
```

In this case, we could create an *N*-ary tree as follows

```
    Nary_tree<int, 4> i4tree( 1975 );
    std::cout << i4tree.retrieve() << std::endl;
```

The value of *N*, however, must be known at compile time. It would not be possible create *N*-ary trees without *aprior* knowledge of the expected arity.

5.4.4 Application

One application of an *N*-ary tree is to create a dictionary of valid strings. For example, if we consider only words containing the 26 letters of the English alphabet, we could let the root node represent the starting point of each word. Each of 26 children would represent those words starting with the corresponding letter. Similarly, each other node could have up to 26 children representing the next letter in a given word. In this way, the letters in any path from the root form a word. A node may be flagged as being a terminal character in a word. Such a data structure is called a *trie* from the contents of re*trie*ve. The author claims the name should be a homophone of "tree"; however, most people pronounce it as "try".

For example, consider all the words in the sentence "The fable then faded from my thoughts and memory." The words in this sentence would generate the trie in Figure 2.

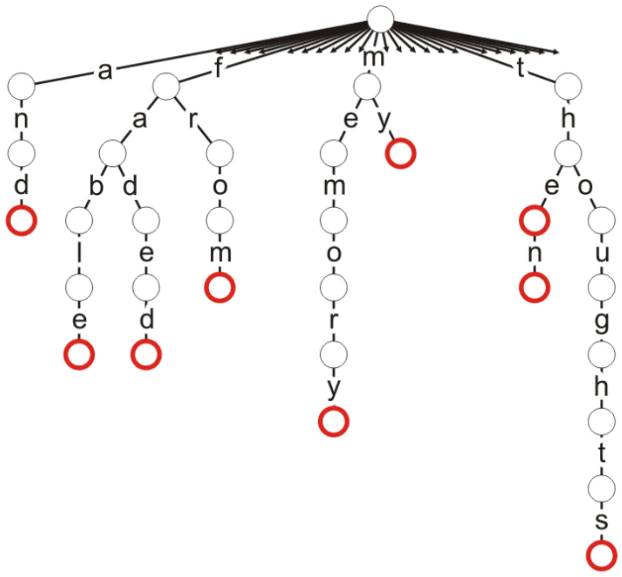

Figure 2. A trie with the words from "The fable then faded from my thoughts and memory."

You will note that "the" is a prefix to the word "then".

The terminal points in a trie could be associated with a linked list of locations within a document. For example, each word in a document could be indexed in a trie allowing for very fast searches. For example, the word "and" would be indexed with position 38, as "and" forms the substring in locations 38 through 40.