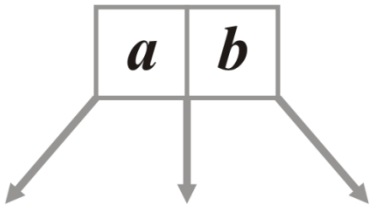
© 2011 by Douglas Wilhelm Harder. All rights reserved.

6.4.1 In-order traversals on general or *N*-ary trees

An in-order depth-first traversal does not make sense with a general tree or an *N*-ary tree with $N > 2$, as there is no obvious order between the node and its children. The in order traversal only works for binary trees because of the relative order of the left and right sub-trees.

6.4.2 Three-way trees

Suppose, however, we store two ordered items in a node. In this case, we could introduce three sub-trees and impose a relative order of these sub-trees.



In this case, we would have three sub-trees, and impose an order of these sub-trees relative to the stored items. An implementation of such a 3-way tree may be:

```
template<typename Type>
class Three_way_node {
    Three_way_node   *left_tree;
    Type              first_element;
    Three_way_node   *middle_tree;
    Type              second_element;
    Three_way_node   *right_tree;
    // ...
};
```

Now we can define such a tree to be a search tree by stipulating the requirements that

1. the first element is less than the second,
2. all sub-trees are 3-way search trees,
3. the left sub-tree contains items less than the first element,
4. the middle sub-tree contains items between the two elements, and
5. the right sub-tree contains items greater than the second element.

One issue, however, is that such a tree may have only one element, in which case we will add the following requirement:

6. A node will only have a sub-tree if both elements are filled.

© 2011 by Douglas Wilhelm Harder. All rights reserved.

To store the number of nodes, we will include a `num_elements` member variable and a member function

```
template <typename Type>
bool Three_way_node::full() const {
    return num_elements == 2;
}
```

6.4.2.1 Operations on 3-way trees

Thus, operations such as find and insertion become more complex:

```
template<typename Type>
bool Three_way_node<Type>::find( Type const &obj ) const {
    if ( this == nullptr ) {
        return false;
    } else if ( !full() ) {
        return ( first() == obj );
    }

    if ( obj < first() ) {
        return left()->find( obj );
    } else if ( obj == first() ) {
        return true;
    } else if ( obj > first() && obj < second() ) {
        return middle()->member( obj );
    } else if ( obj == second() ) {
        return true;
    } else {
        return right()->find( obj );
    }
}

template<typename Type>
bool Three_way_node<Type>::insert( Type const &obj ) {
    if ( !full() ) {
        if ( obj == first() ) {
            return false;
        } else if ( obj < first() ) {
            second_element = first();
            first_element = obj;
        } else {
            second_element = obj;
        }

        num_elements = 2;
        return true;
    }

    if ( obj == first() || obj == second() ) {
        return false;
    }

    if ( obj < first() ) {
        if ( left() == nullptr ) {
            left_tree = new Three_way_node( obj );
            return true;
        } else {
            return left() ->insert( obj );
        }
    } else // and so on...

}
```

Attempting to erase an element may be much more complex as there are other cases to consider.

© 2011 by Douglas Wilhelm Harder. All rights reserved.

6.4.2.2 Example insertions into a 3-way tree

Suppose we insert 68, 27, 91, 38, and 82 into an initially empty 3-way tree.

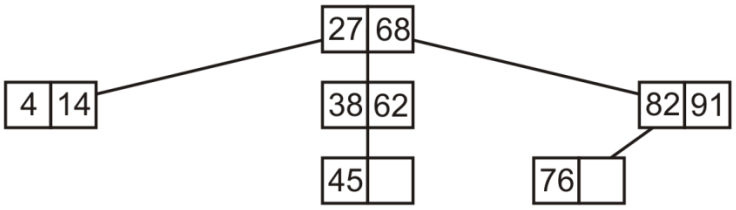
1. The first two elements are placed into the root node, which now contains (27, 68),
2. 91 > 68, so we create a new right sub-tree of the root containing 91,
3. 27 < 38 < 68, so we create a new middle sub-tree of the root containing 38, and
4. 82 > 68, so we insert 82 into the right sub-tree, which now contains (82, 91).

This gives us the 3-way tree

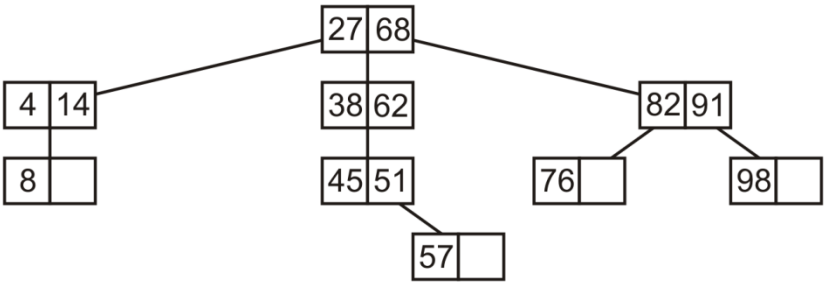
If we continue inserting 14, 62, 45, 76, and 4 we have:

5. 14 < 27, so we create a new left sub-tree of the root containing 14,
6. 27 < 62 < 68, so we insert 62 into the middle sub-tree, which now contains (38, 62),
7. 27 < 45 < 68 and 38 < 45 < 62, so we create a new middle sub-tree of that node containing 45,
8. 76 > 68 and 76 < 82, so we create a new left sub-tree of the node containing (82, 91), and
9. 4 < 27, so we insert it into the left sub-tree, which now contains (4, 14).

This gives us the 3-way tree



Finally, inserting 51, 8, 98, and 57 yields the 3-way tree:



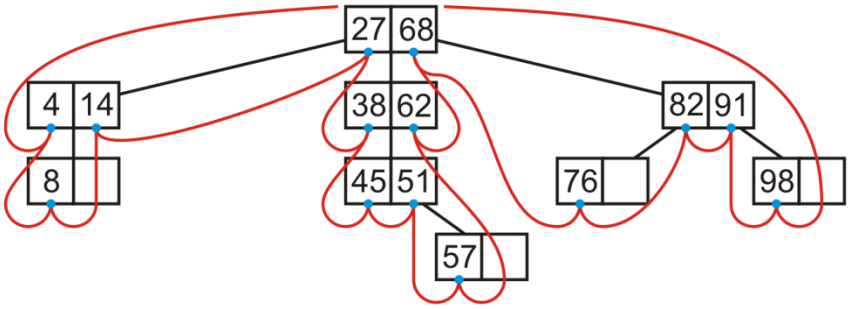
© 2011 by Douglas Wilhelm Harder. All rights reserved.

6.4.2.3 In-order depth-first traversals of 3-way trees

We may now perform an in-order depth-first traversal, where for a node with only one element, we visit that one element, and for any full node, we

1. visit the left sub-tree,
2. visit the first element,
3. visit the middle sub-tree,
4. visit the second element, and
5. visit the third sub-tree,

in that order. An in-order traversal of the tree in the previous example would yield



yielding

4, 8, 14, 27, 38, 45, 51, 57, 62, 68, 76, 82, 91, 98.

6.4.3 Generalizations to *M*-way trees

The next generalization would be to have a search tree where each node contains:

1. An array of up to $M - 1$ elements, and
2. Up to M sub-trees interleaving the $M - 1$ elements.

© 2011 by Douglas Wilhelm Harder. All rights reserved.

Such a class may be implemented as:

```
template<typename Type, int M>
class M_way_node {
    private:
        int num_elements;
        Type elements[M – 1];
        M_way_node *[M];  // an array of pointers to M-way nodes

    public:
        M_way_node( Type const & );
        // ...
};

template<typename Type, int M>
M_ way_node<Type, M>::M_ way_node( Type const &obj ):num_elements( 1 ) {
    elements[0] = obj;

    for ( int i = 0; i < M; ++i ) {
        subtrees[i] = nullptr;
    }
}

template<typename Type, int M>
bool M_ way_node<Type, M>::full() const {
    return ( num_elements == M - 1 );
}
```

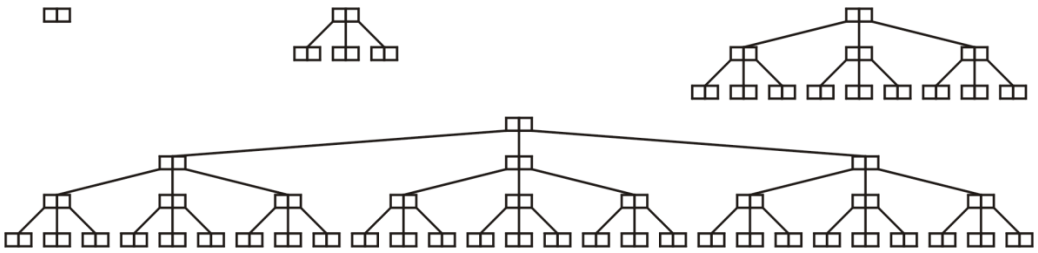
An in-order traversal printing of such a tree could be implemented as follows:

```
template <typename Type, int M>
void M_way_node<Type, M>::in_order_traversal() const {
    if ( empty() ) {
        return;
    } else if ( !full() ) {
        for ( int i = 0; i < num_elements; ++i ) {
            cout << elements[i];
        }
    } else {
        for ( int i = 0; i < M - 1; ++i ) {
            subtrees[i]->in_order_traversal();
            cout << elements[i];
        }
        subtrees[M - 1]->in_order_traversal();
    }
}
```

© 2011 by Douglas Wilhelm Harder. All rights reserved.
ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 6.4.
Assistances and comments will be acknowledged.

6.4.3.1 Size of *M*-way trees

We may define a perfect *M*-way tree in a manner similar to previous cases. If we count the number of elements stored in perfect 3-way trees of height $h = 0$, 1, 2, and 3, shown here



we get an obvious pattern arising in the size, or number of nodes:

h	Size	Formula
0	2	$3^1 - 1$
1	8	$3^2 - 1$
2	26	$3^3 - 1$
3	80	$3^4 - 1$

Thus, we may assume that the number of nodes in a perfect *M*-way tree of height *h* is $M^{h+1} - 1$. This formula also holds for perfect binary trees, where such a tree of height *h* has $2^{h+1} - 1$ nodes.

To prove this, we will make one observation: the number of nodes in a perfect *M*-way tree is the same as the number of nodes in a perfect *M*-ary tree. Thus, we have $\frac{M^{h+1} - 1}{M - 1}$ nodes; however, each node stores $M - 1$ elements, so by multiplying these two, we get the desired result: $M^{k+1} - 1$.

6.4.3.2 Proportion of elements in leaf nodes

Recall that for a perfect binary tree, approximately half of the nodes are leaf nodes. As there are M^h leaf nodes in a perfect *M*-way tree of height *h*, the number of elements stored in the leaf nodes is therefore $M^h(M-1)$ and consequently, it appears that as *M* gets larger, a greater and greater proportion of the elements appear in the leaf nodes:

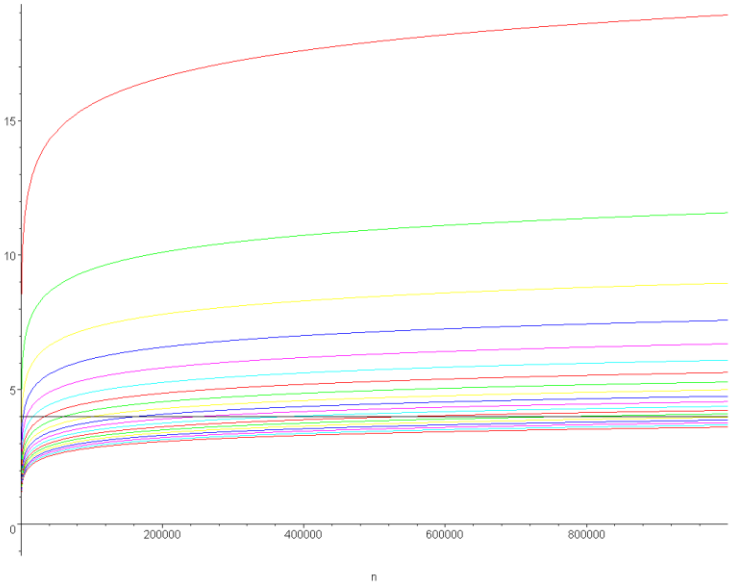
$$\frac{M^h(M-1)}{M^{h+1} - 1} \approx \frac{M^h(M-1)}{M^{h+1}} = \frac{M-1}{M}.$$

For example, in an 8-way search tree, approximately 87½ % of the elements are in the leaf nodes, and in a 100-way tree, approximately 99 % of the elements would appear in the leaf nodes.

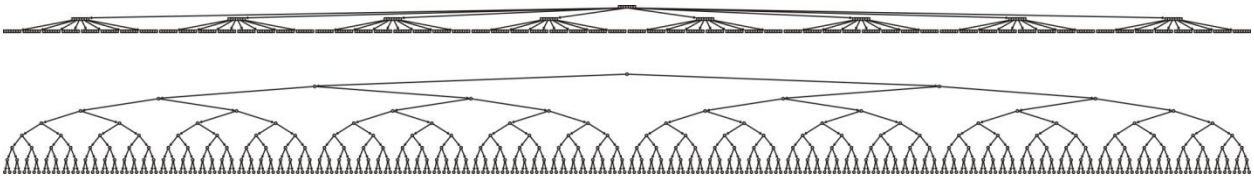
© 2011 by Douglas Wilhelm Harder. All rights reserved.

6.4.3.3 Minimum height of *M*-way trees with *n* nodes

Notice that the minimum height of an *M*-way tree storing *n* nodes is therefore $\lfloor \log_M (n) \rfloor$. For large values of *M*, this is potentially significantly smaller than that for a binary tree. This is shown in the following graph of the minimum height of an *M*-way tree for up to one million elements.

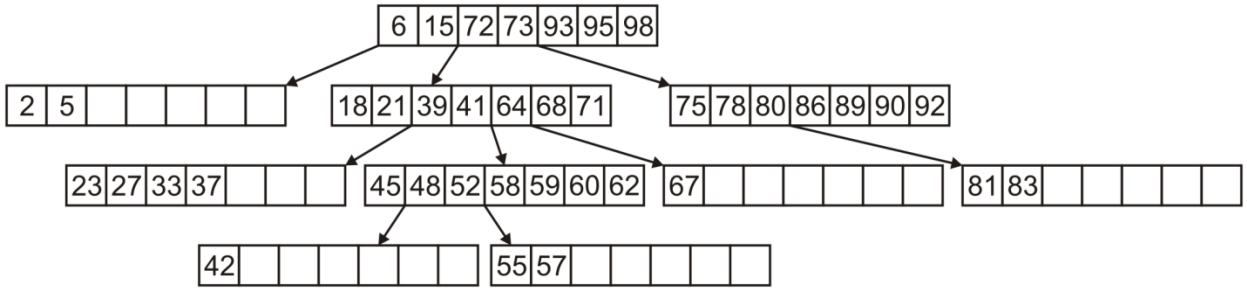


For example, a perfect 8-way tree with height $h = 2$ stores 511 elements in 73 nodes. A perfect binary tree with the same number of elements would have 511 nodes with a height of 8:



6.4.3.4 Advantages and disadvantages

Consider the following 8-way search tree:



Suppose we are searching for 43. First we must perform a binary search on the root node to determine that if 43 exists at all, it is in the sub-tree between 15 and 17; that is, the 3rd sub-tree. We follow this link and we must now perform a second binary search of that array to find that if it exists, it must be in the

© 2011 by Douglas Wilhelm Harder. All rights reserved.

Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 6.4.
Assistances and comments will be acknowledged.

sub-tree between 41 and 64; that is, the 5^{th} sub-tree. We must repeat this process again at the next node, and then finally, in the last node, we note there is only the value 42, so 43 does not exist in this 8-way search tree.

The advantage of an *M*-way search tree over a binary search tree is that the height is significantly less with larger values of *M*. Unfortunately, this also makes the tree significantly more complex and there may be many nodes that are only partially filled—potentially wasting space. Consequently, an *M*-way tree is likely only to be used if memory is cheap and it is expensive to access a particular node. We will come across such a scenario in the next topic.