Multi-way tree

**I. Introduction:**

A*multi-way tree* or *m-way tree* is a generalized version of the binary search tree. We know that in a binary search tree (BST), each node contains a distinct key (value) and two pointers point to its left and right child nodes.

Each node in m-way tree:

- Can contain at most m – 1 keys and m addresses of m child nodes.

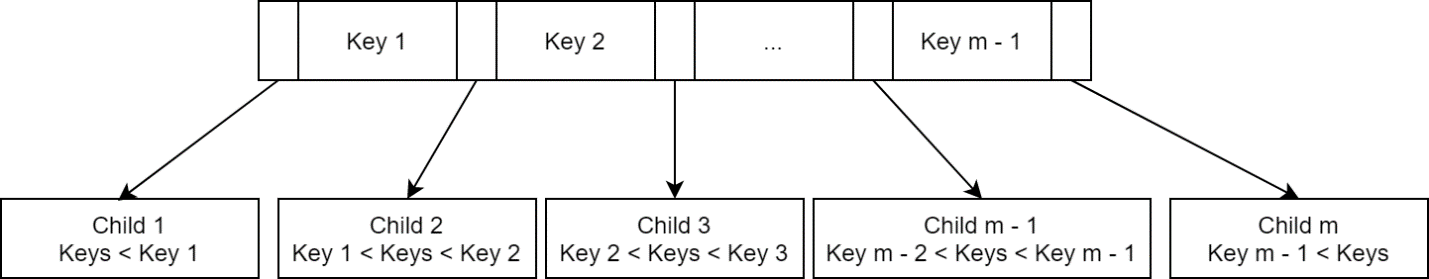
- The keys are distinct and in ascending order.

- The keys in the 1stchild node must be smaller than the 1st key.

- The keys in the mth child node must be larger than the (m – 1)th key.

- The keys in the ith child node must be smaller than the ith key and larger than the (i – 1)th key (2 ≤ i ≤ m - 1).

- The child nodes can be empty.

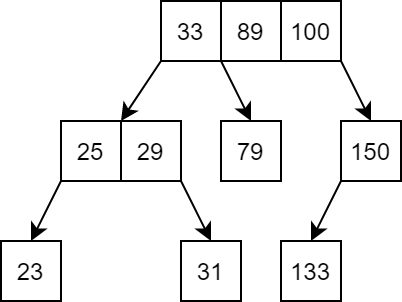
Imagine a node in this form:

We call child[i] left child of key[i], and child[i +1] its right child.

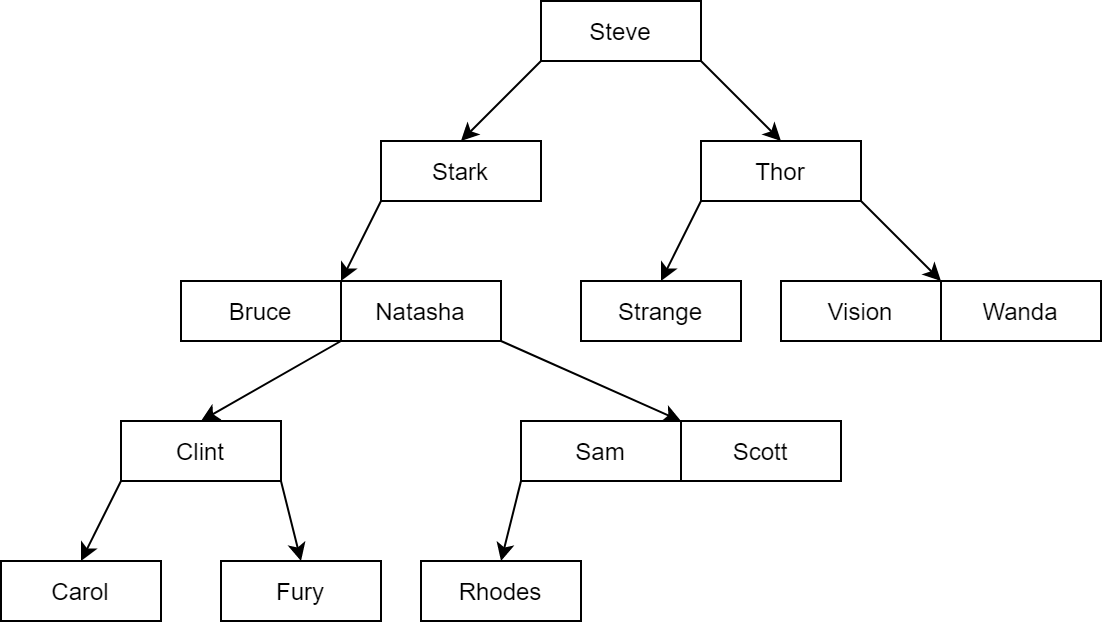
A BST is basically a 2-way tree. But an unbalanced BST can waste a lot of memory, and even if balanced, the height of the tree can still be very high if the number of keys is large. Also, the implementation of a balanced BST is very complex. The m-way tree reduces height of the tree but take more time in traversal, in other word it slows down the data accessing process because the branching system is not as simple as in BST. In particular, we only make a 2-way branching decision at each node in BST, but in m-way tree, we must make a multiway branching decision.

**\*Example 1:**

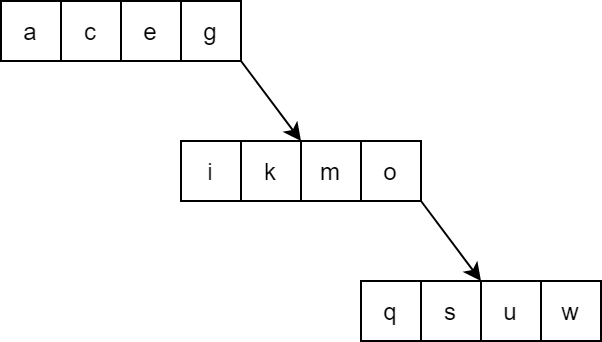
4-way tree example (integer keys):



**\*Example 2:**

3-way tree example (string keys): using string compare to

**\*Example 3:**

5-way tree example (character keys):

This is an example of an unbalanced m-way tree. This case usually happens when we add keys to the tree in sorted order.

Let WAY be the maximum number of branch in a m-way tree node, the node structure will be define below:

struct node

{

int count;

int value[WAY – 1];

struct node\* child[WAY];

};

Here, *count* store the number of value stored in the current node. All values in current node are stored in the array *value,* and the array *child* store addresses of every child nodes of the current node.

Let’s also define a function to check if a node is full and another to create a new node, which will help later:

bool isFull(node\* curr)

{

return (curr->count == WAY - 1) ? 1 : 0;

}

node\* createnode()

{

node\* newnode = new node;

newnode->count = 0;

for (int i = 0; i < WAY; i++) newnode->child[i] = NULL;

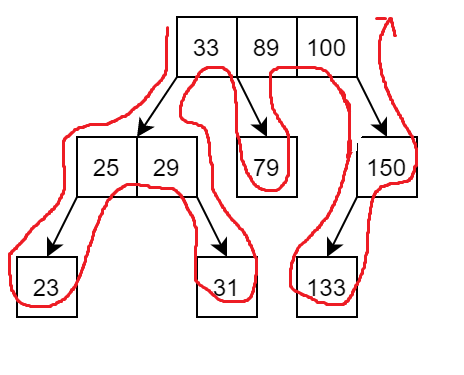
return newnode;

}

**II. Algorithms:**

**1. Traversing and searching algorithms:**

Traversing an m-way tree is similar to in-order traversing in BST. We start at the left-most child, then the key and the next child, repeat for all remaining keys and child.

**\*Example:**

Searching in an m-way tree consist of two processes: traversing between nodes and searching in node. Traversing between nodes is similar to BST traversal, the difference is now we have m option to branch instead of two. Searching in node is to consider every value in the current nodes.

We consider a second function called *search\_in\_node*, which will determine if the key we looking for is in the current node or not, and for both case, return value *pos* which is either the position of key in current node if found, or position of the child node that might contain key. For example *pos* will be i if value[i – 1] < key < value[i], or i if value[i] = key. The main search function will then call recursive to search the corresponding child node.

bool search\_in\_node(int key, node\* curr, int& pos)

{

pos = 0;

// find the first key in node that is >= key,

while ((pos < curr->count) && (key > curr->value[pos])) pos++;

// if key > all values in node return 0, pos is position of the last child in node

if (pos == curr->count) return 0;

// if key is found, return 1,pos is now position of key in current node;

if (key == curr->value[pos]) return 1;

// else return 0, pos is now position of the child node that might contain key

return 0;

}

node\* search(int key, node\* curr, int& pos)

{

// reach NULL node mean the key is not found in the tree, return NULL

if (curr == NULL) return NULL;

// if key is found in the current node return the current node

// pos will be position of key in current node

if (search\_in\_node(key, curr, pos)) return curr;

// child[pos] is now the possible child to contain key, call recursive to this child

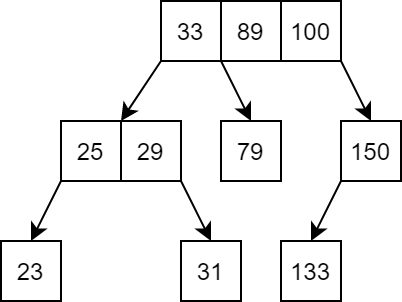
return search(key, curr->child[pos]);

}

Calling the search function for the root node of a m-way tree will return address of the node contain the key being look for, or NULL if the key is not found.

Note that if the searching algorithm above is used, it can only return the node holding the key, not its exact position in that node.

**\*Example:** Let’s search for 31 in example 1.

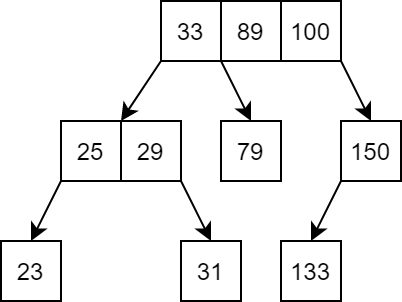


Start at root node

31 is not found,

31 < 33, pos is now 0

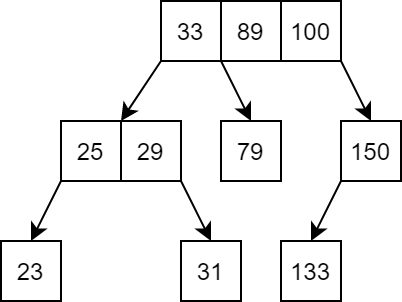
Child[0] path



31 is not found,

29 < 31, pos is now 2

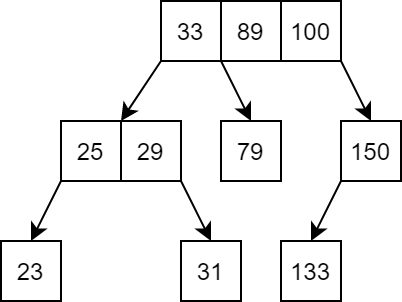
Child[2] path



31 is found, return

this node address

Let’s try again with 29.

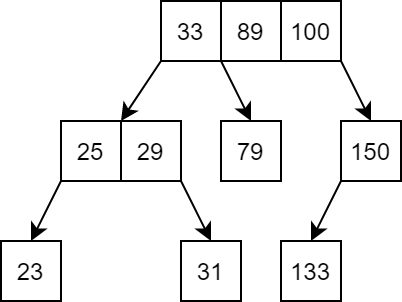


Start at root node

29 is not found,

29 < 33, pos is now 0

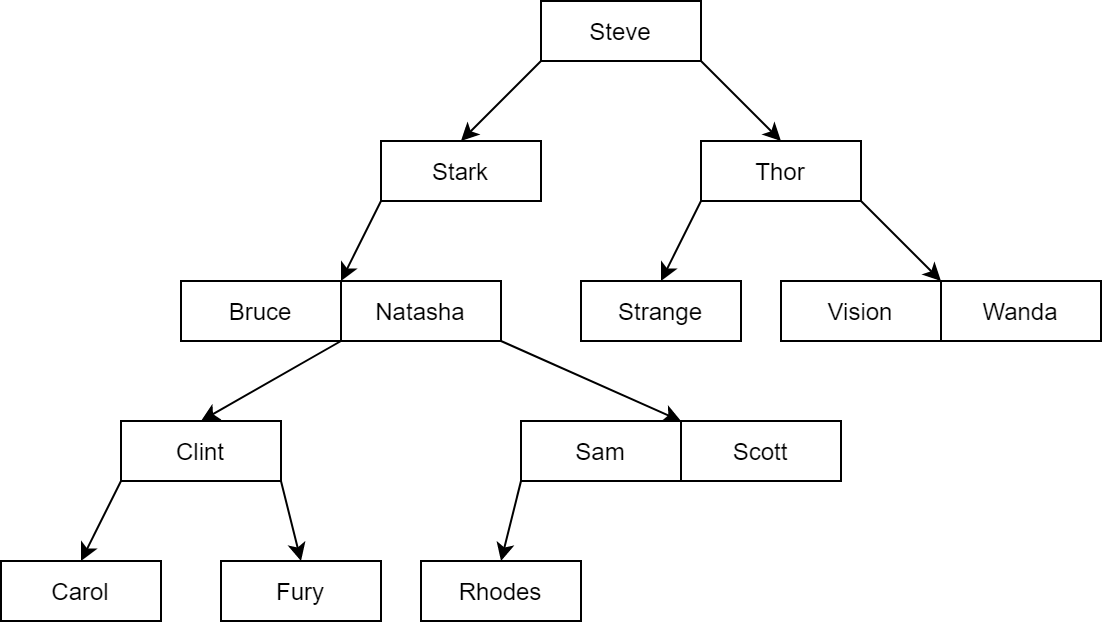
Child[0] path



29 is found, return

this node address

Now let’s look for Logan in example 2:

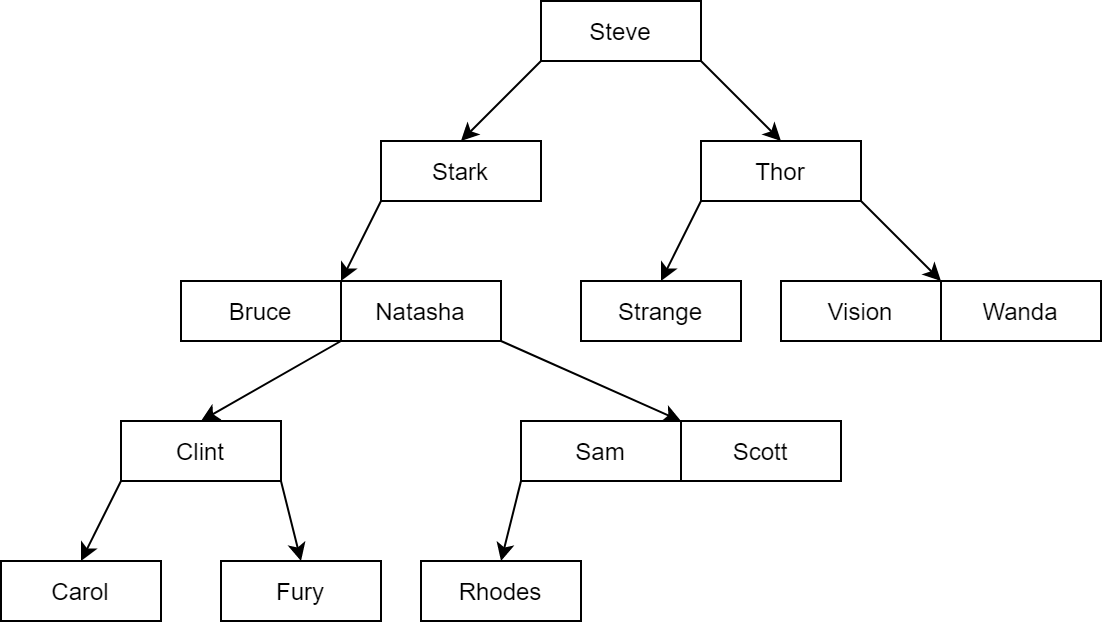


Start at root node

Logan is not found,

Logan < Steve, pos is now 0

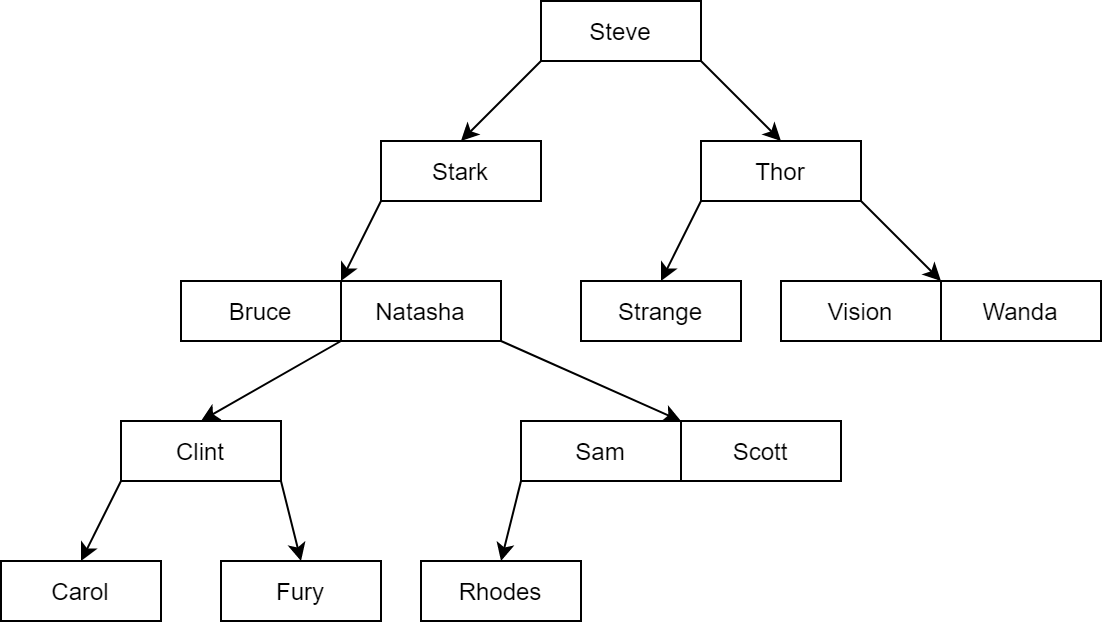
Child[0] path



Logan is not found,

Logan < Stark, pos is now 0

Child[0] path

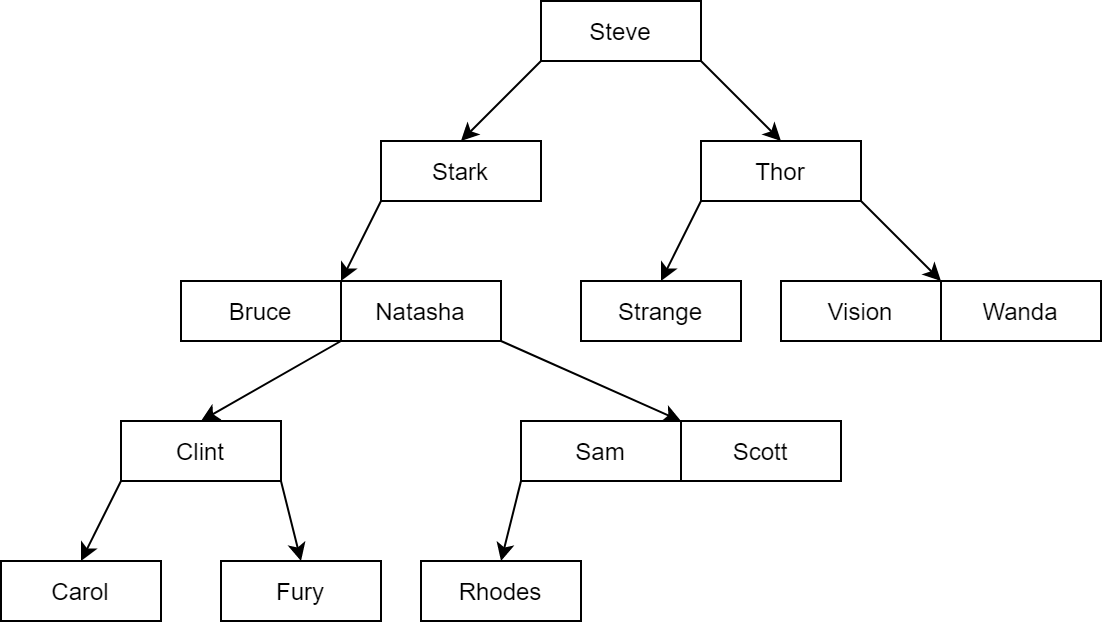


Logan is not found,

Bruce < Logan < Natasha,

pos is now 1

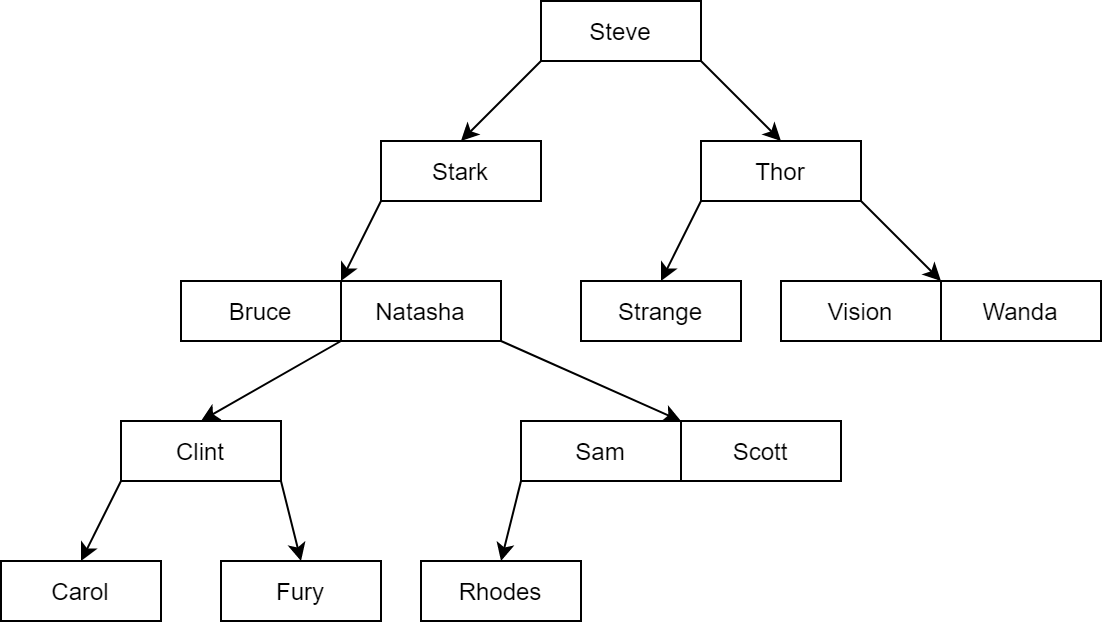
Child[1] path



Logan is not found,

Clint < Logan, pos is now 1

Child[1] path



Logan is not found,

Fury < Logan, pos is now 1

Child[1] path

NULL node is reach, Logan can’t be found in the tree, return 0

**2. Inserting algorithms:**

To insert a new value into an m-way tree, we follow these steps:

- Traverse the tree until reaching a NULL node. We reuse the *search\_in\_node* function and the recursive method in searching algorithm.

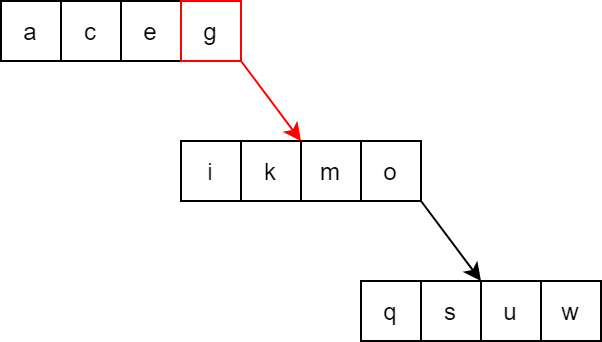
- If the parent of the NULL node is not full, insert the new key into the parent node.

- Otherwise, create a new node and insert the new key to this new node.

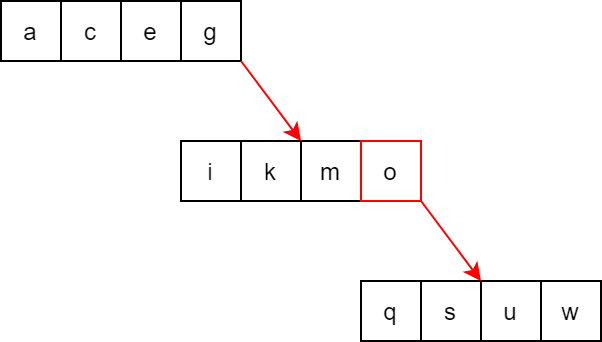
**\*Example:**

Let’s try inserting r into example 3:

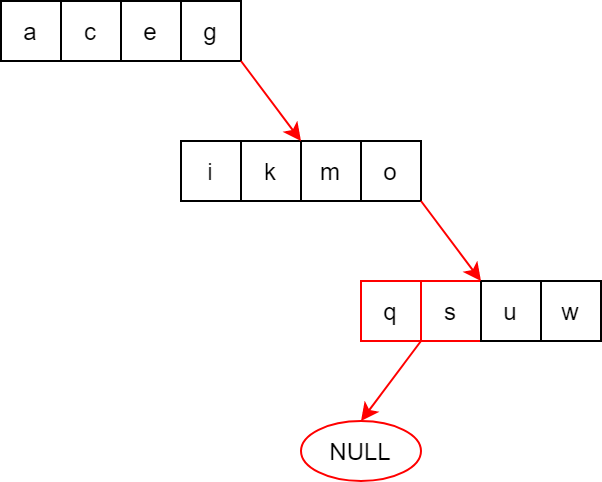
First we traverse the tree to find a NULL node. This step is done by “searching” for r in the tree. With that, we traverse the tree following the red path:



g < r



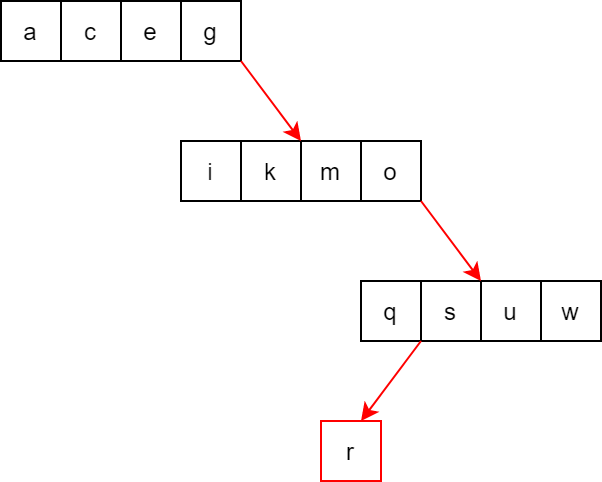
o < r



NULL node found

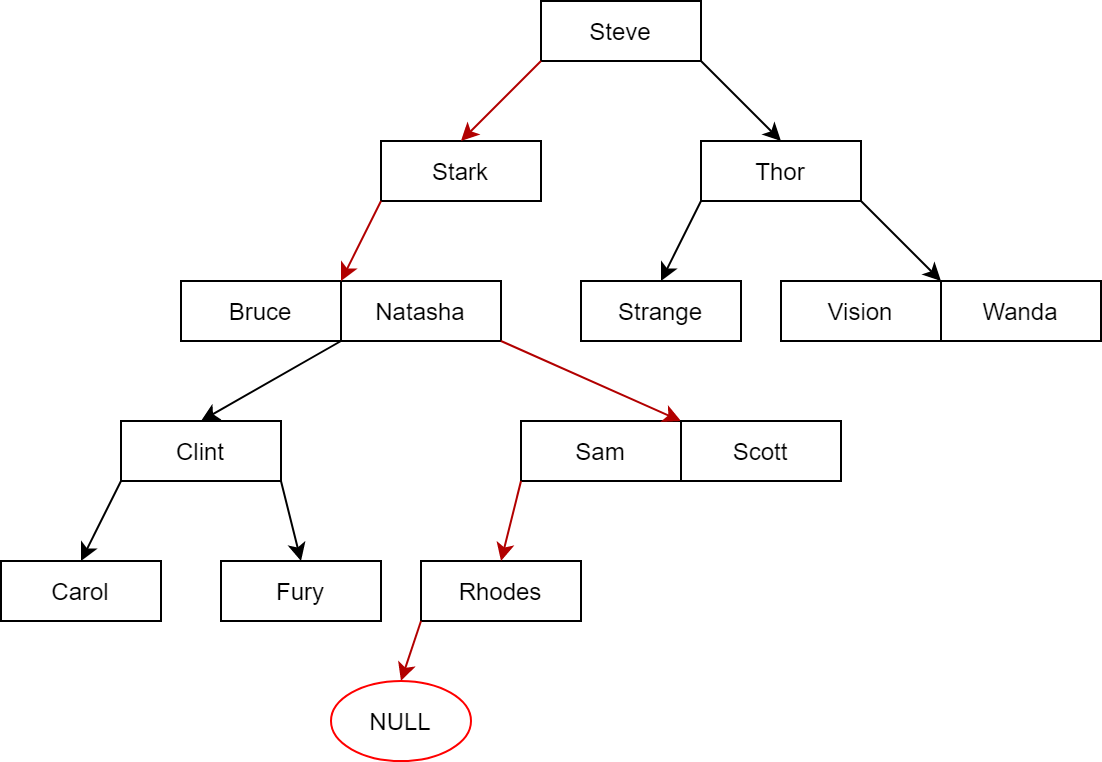
Parent node

q < r < s

The parent node is full, so we create a new node in position of the NULL node, and add r to it. The new node only have one value.

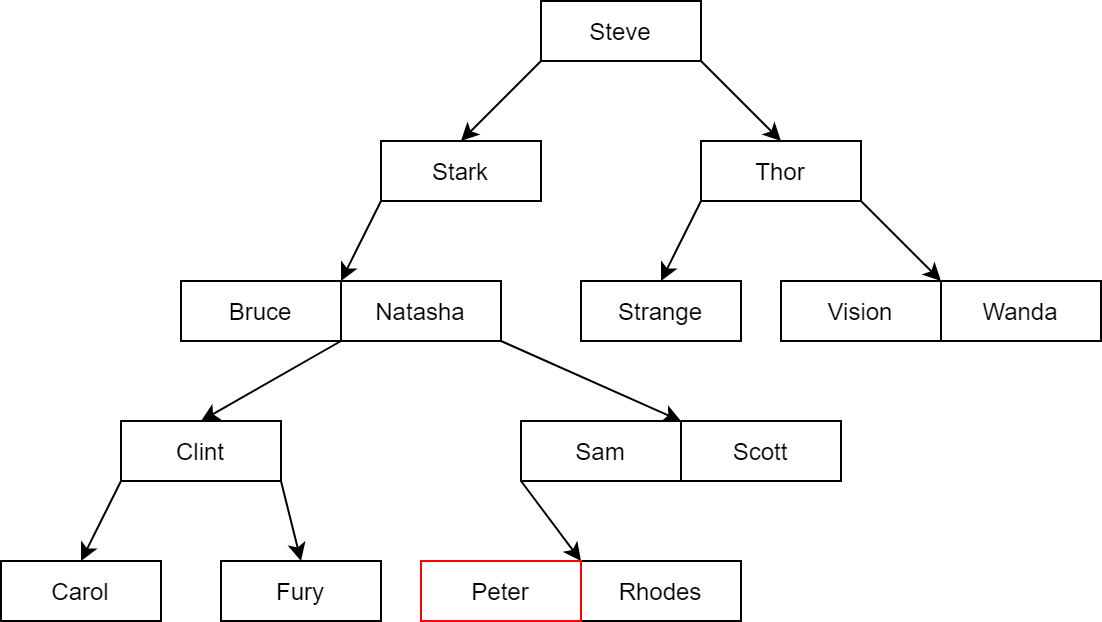
Let’s add Peter to example 2:

First traverse the tree following the red path to reach a NULL node.



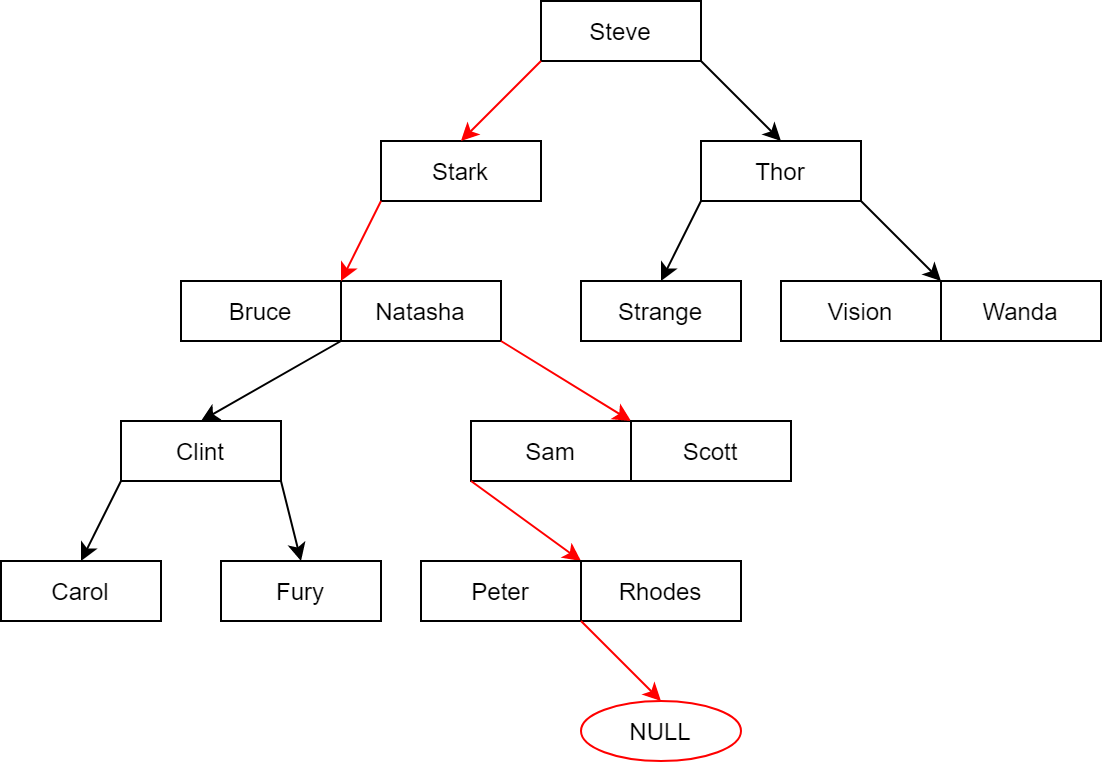
NULL node found

Parent node

Now we have reached a NULL node, but its parent node is not full, so we insert Peter into the parent node. Peter < Rhodes, so he is inserted in the left of Rhodes.

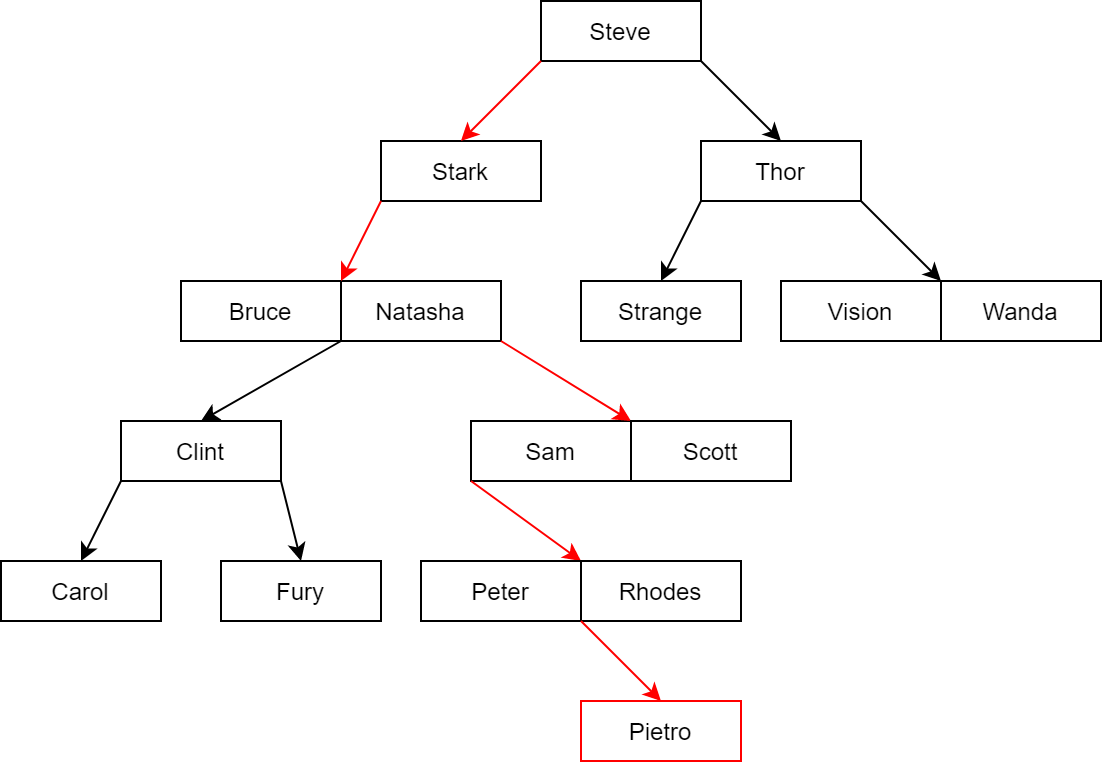
Also add Pietro to example 2:

Again, we traverse the tree to reach a NULL node.



Parent node

NULL node found

The NULL node is found at the same level with the same parent node in the previous example, but now, the parent node is full, so we must create a new node for Pietro.

The inserting algorithm is implement below:

bool insert\_node(node\*& curr, int key)

{

// if tree is empty

if (curr == NULL)

{

curr = createnode();

curr->value[0] = key;

curr->count = 1;

}

int pos;

// if the new key is already in current node, which mean in the tree, do not insert

if (search\_in\_node(key, curr, pos)) return 0;

// reach a NULL node

if (curr->child[pos] == NULL)

{

if (isFull(curr))

{

// if parent node is full

curr->child[pos] = createnode();

curr->child[pos]->count = 1;

curr->child[pos]->value[0] = key;

}

else

{

// otherwise insert key into parent node

// shift all value > key to the right to create a slot

for (int i = curr->count; i > pos; i--)

{

curr->value[i] = curr->value[i - 1];

curr->child[i + 1] = curr->child[i];

}

curr->child[pos + 1] = curr->child[pos];

// insert key into created slot

curr->value[pos] = key;

curr->child[pos] = NULL;

// update count value

curr->count++;

}

// successfully insert

return 1;

}

else

{

// call recursive to the child node

return insert\_node(curr->child[pos], key);

}

}

**3. Deleting algorithms:**

To remove a key from an m-way tree, we follow these steps:

- Traverse the tree to find the node holding key.

- If key doesn’t have both left and right child, we remove key from current node by moving all value after key and their child one slot forward.

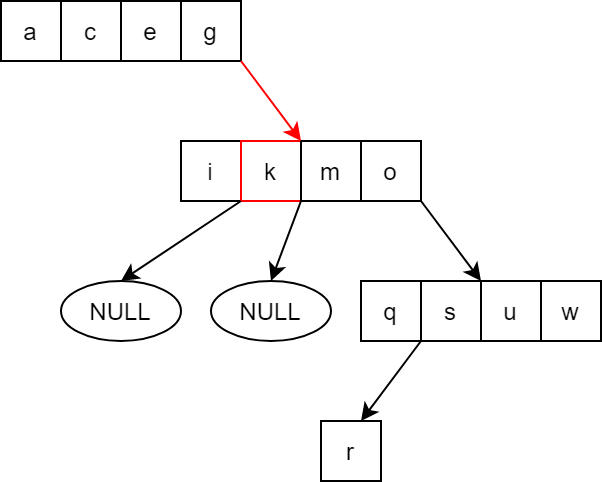
- If key has either left or right child, we replace key with the largest value in its left sub tree, or the smallest value in its right sub tree. Then, we call recursive to remove the replaced value from its node.

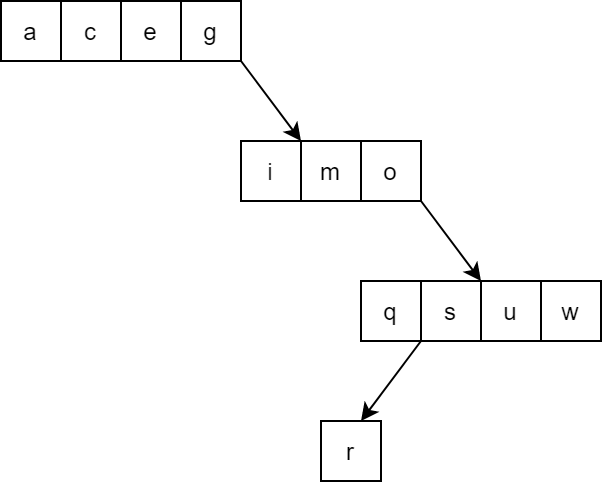
- If a node become empty, we delete it.

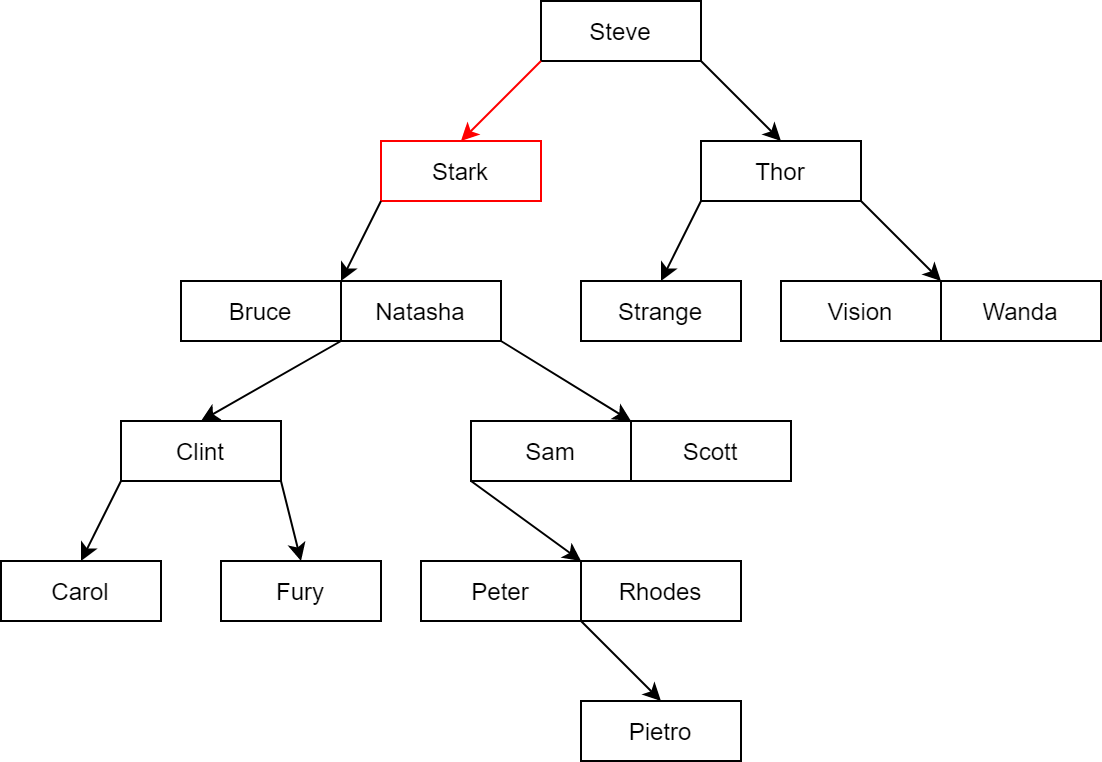
**\*Example:**

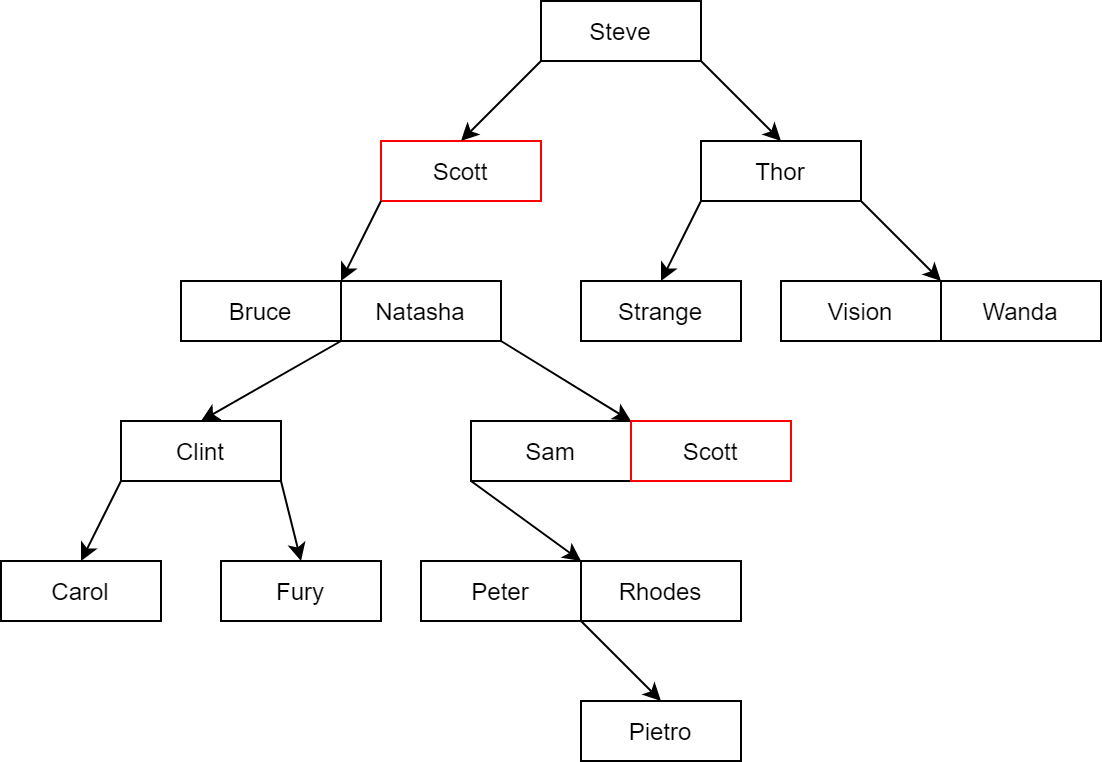
Let’s remove k from example 3:

First we find the node holding k:

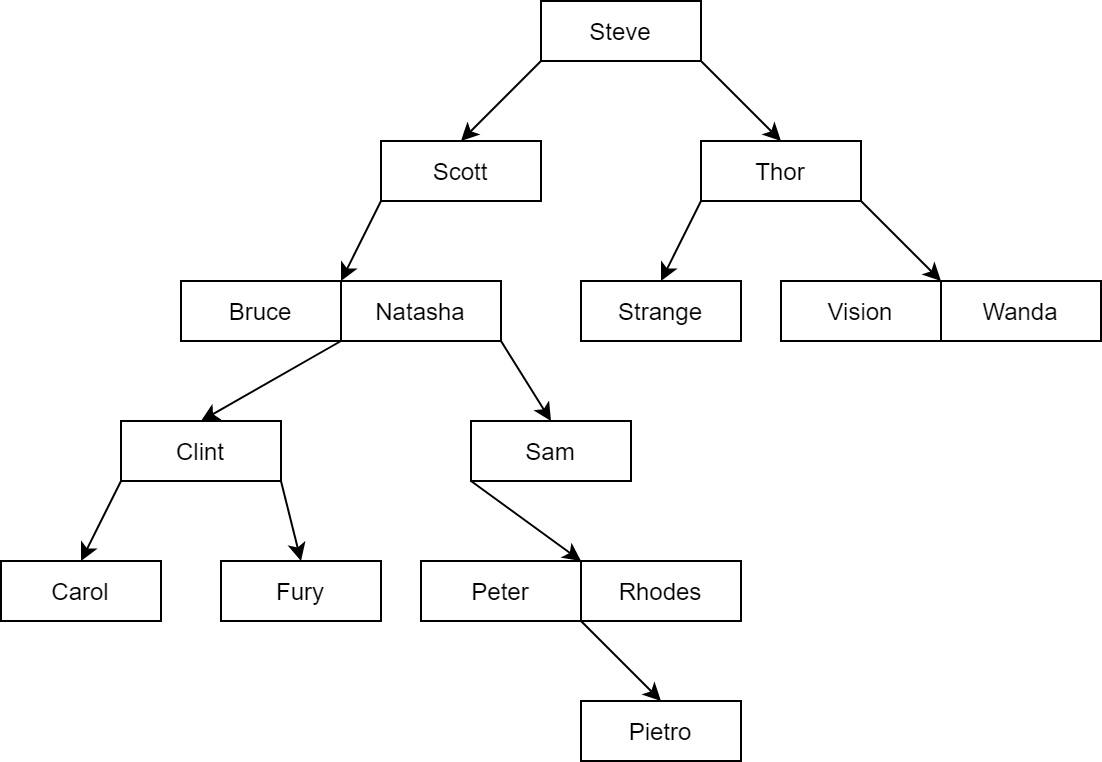


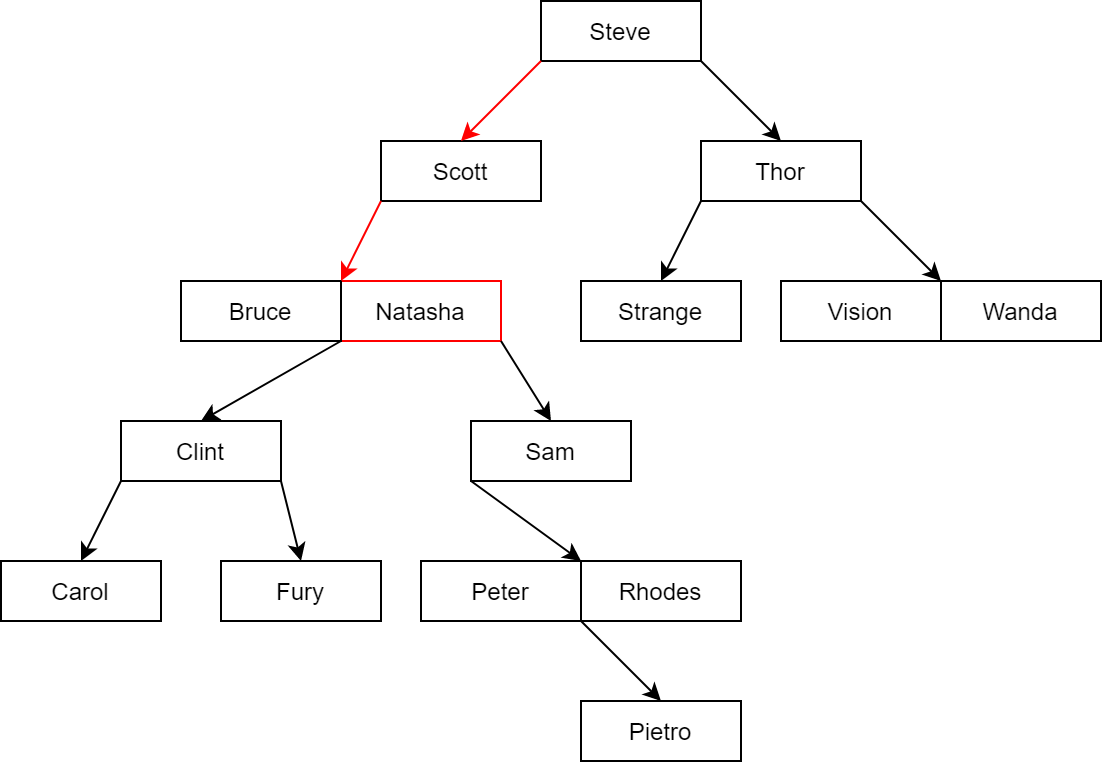
We see that k doesn’t have both left and right child, so we just remove it from its node.

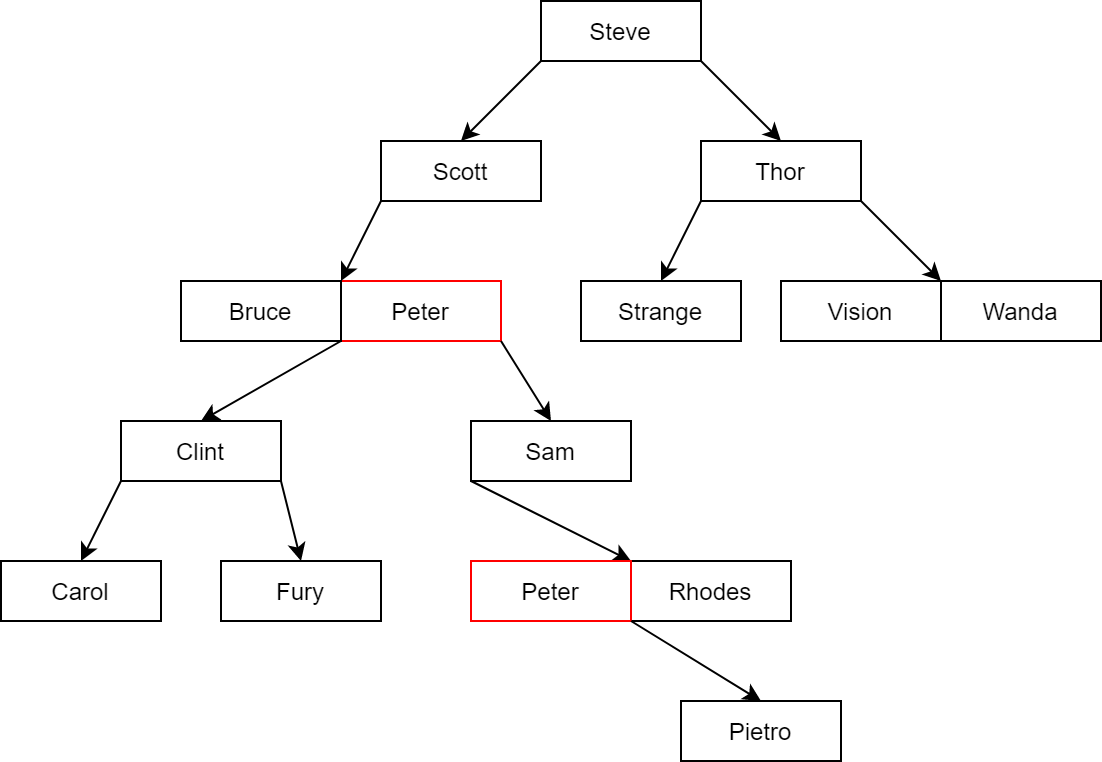
Remove Stark from example 2: First we find Stark.

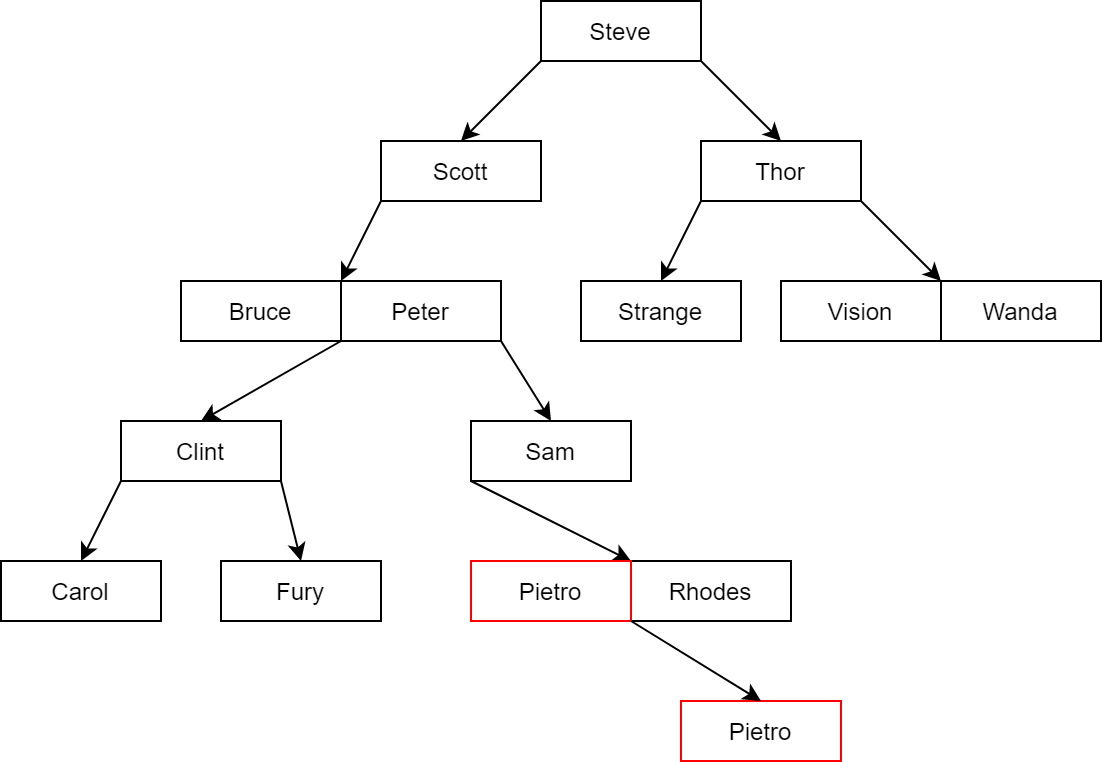
We notice that Stark has left child, so we replace Stark with the largest value in the left sub tree. Here, that value is Scott.

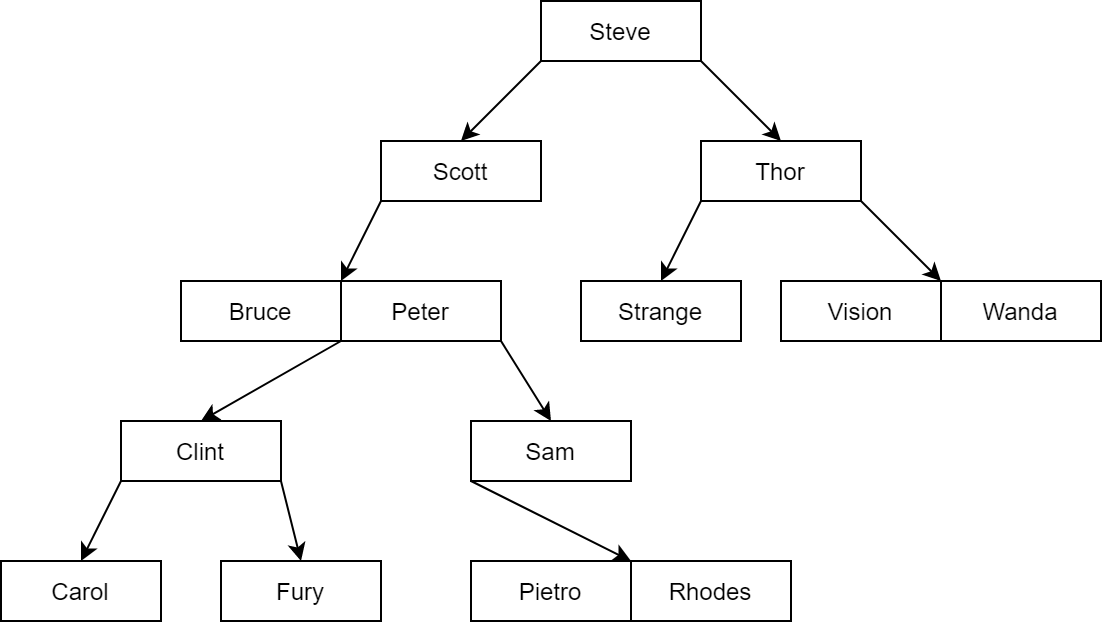
Now that Scott has been moved to Stark position, we remove Scott from his old position. Old Scott doesn’t have any child, so we just remove him.



Let’s try again with Natasha.

Natasha has both left and right child, we can choose either to replace her with the largest value in left sub tree, or the smallest one in right sub tree. This time, let follow the right path. The smallest value in the right sub tree is Peter.

Now we have to remove old Peter, but old Peter still has a right child, so we repeat and replace old Peter which the smallest and also the only child in right sub tree, Pietro.

Moving to remove old Pietro, he doesn’t have any child and is also the only value in his node. After removing him, his node become empty, so we delete it.

The deleting algorithm will be implement below:

node\* left\_replace(node \*curr,node \*&parent, int &replacekey)

{

// find the largest value in this sub tree

// get the position of the largest value in current node

int pos = curr->count - 1;

// if value[pos] still have right child, it is not the largest value

if (curr->child[pos + 1] != NULL)

{

parent = curr;

return left\_replace(curr->child[pos + 1],parent,replacekey);

}

else

{

// else return the current node

replacekey = curr->value[pos];

return curr;

}

}

node\* right\_replace(node\* curr,node \*&parent, int& replacekey)

{

// the first value is the smallest value in current node

// if the first value still have left child, it is not the smallest value

if (curr->child[0] != NULL)

{

parent = curr;

return right\_replace(curr->child[0],parent, replacekey);

}

else

{

// else return the current node

replacekey = curr->value[0];

return curr;

}

}

bool delete\_node(node\*& curr, int key)

{

// if the function reach a NULL node, mean the key being look for is not in the tree,

// or this is an empty tree

if (curr == NULL) return 0;

int pos;

// if key is not in current node, call recursive to child node

if (!search\_in\_node(key, curr, pos))

{

return delete\_node(curr->child[pos], key);

}

// if key is found, value[pos] is now holding key

if (curr->child[pos] != NULL)

{

// if key has a left child, replace key with the largest value in left sub tree

int replacekey;

node\* parent = curr;

node\* alter\_node = left\_replace(curr->child[pos], parent, replacekey);

// replace the current key need to be deleted

curr->value[pos] = replacekey;

// remove the replaced key in the alternative node

bool temp = delete\_node(alter\_node, replacekey);

// if alter node is empty, delete it

if (alter\_node == NULL)

{

if (parent == curr) parent->child[pos] = NULL;

else parent->child[parent->count - 1] = NULL;

delete alter\_node;

}

return temp;

}

else if (curr->child[pos + 1] != NULL)

{

// if key has a right child, replace key with the smallest value in right sub trees

int replacekey;

node\* parent = curr;

node\* alter\_node = right\_replace(curr->child[pos + 1],parent, replacekey);

// replace the current key need to be deleted

curr->value[pos] = replacekey;

// remove the replaced key in the alternative node

bool temp = delete\_node(alter\_node, replacekey);

// if alter node is empty, delete it

if (alter\_node == NULL)

{

if (parent == curr) parent->child[pos + 1] = NULL;

else parent->child[0] = NULL;

delete alter\_node;

}

return temp;

}

else

{

// if key doesn't have both left and right child, remove key from current node

// shift all value after key to the left

while (pos < curr->count - 1)

{

curr->value[pos] = curr->value[pos + 1];

curr->child[pos] = curr->child[pos + 1];

pos++;

}

curr->child[pos] = curr->child[pos + 1];

curr->child[pos + 1] = NULL;

// update count value

curr->count--;

if (curr->count == 0) curr = NULL;

// successfully delete

return 1;

}

}