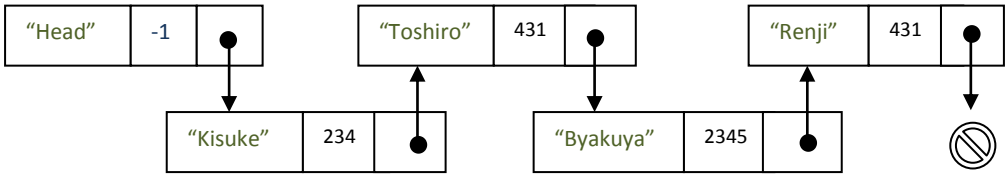
Recursion: Additional Material

Recursion is a method of superimposing code on itself in such a way that it can repeat itself an infinite amount of times. It is literally a recurring piece of code, much like we would use a method to replace common code; however in this context, it refers to recurrence within itself. For example, we click a link on a web page; most of us will continue clicking the link until it opens. This recursive behaviour can be looked at as:

```
clicklink(URL)
while(URL hasn't opened yet) {
clicklink(URL)
}
```

Take a single linked list (those who read the additional material for Linked Lists will find this familiar):

Each element comprises of a String, an Integer and a pointer to the next element – the names /values are unimportant



Traditionally starting from the head, we would print out each of the elements by moving to the next, printing its value, moving to the next, printing that, and so on, until there are no more nodes. If you look at the notes for SLinkedList and CLinkedList, the code will look similar to:

```
CNode curnode = head;
while(curnode != null) {
    System.out.println(curnode.name);
    curnode = curnode.next;
}
```

So working through:

- Curnode = "head", output "head", curnode = curnode.next which is "Kisuke"
- Curnode = "Kisuke", output "Kisuke", curnode = curnode.next which is "Toshiro"
- Curnode = "Toshiro", output "Toshiro", curnode = curnode.next which is "Byakuya"
- Curnode = "Byakuya", output "Byakuya", curnode = curnode.next which is "Renji"
- Curnode = "Renji", output "Renji", but curnode.next is null, so we end the loop

Which will output:

```
Head
Kisuke
Toshiro
Byakuya
Renji
```

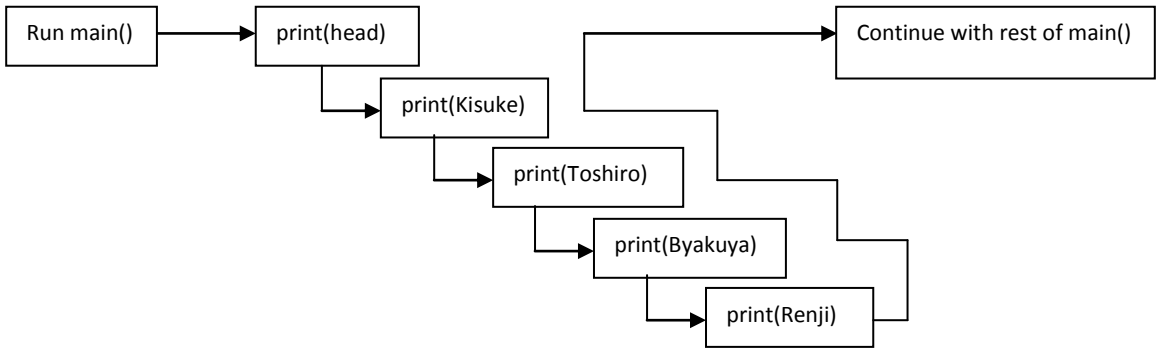
Recursion works in a similar way to this; start at the head node and move to the next until the end, but instead of a while loop, we stick the code in a method and call the method from within itself:

```
public static void printNew(CNode node) {
    System.out.println(node.name);

    if(node.next != null) {
        printNew(node.next);
    }

}
```

So assuming the head node is passed as the parameter: `printNew(head);`

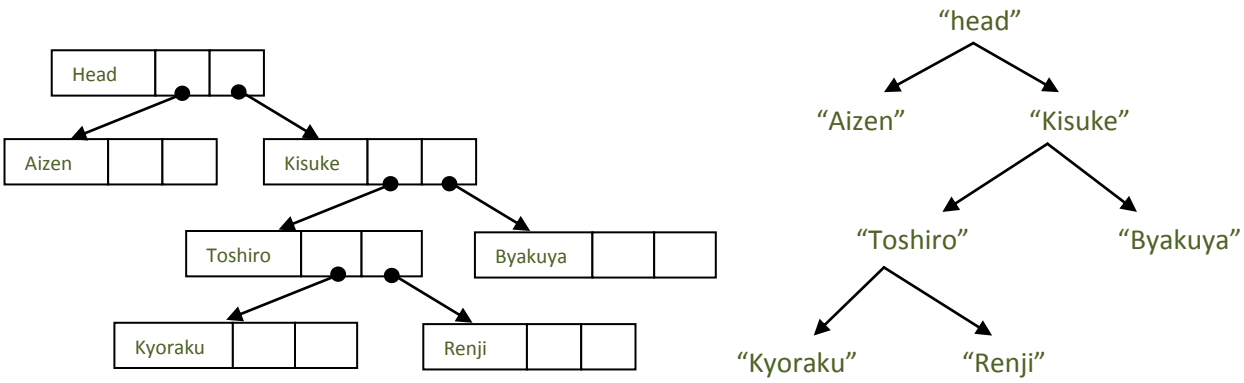
- Output (head.name), “head”; if there is next element, call the method on itself for that element, in this case, next is “Kisuke”
- The program would then run through the method again, outputting “Kisuke”, and calling itself for the next node, which would be “Toshiro”
- Output “Toshiro”, call itself again for the next node, “Byakuya”
- Output “Byakuya”, call itself again for the next node, “Renji”
- Output “Renji”, but now the next node is null, so the method will end

A more graphical representation might be:



After being called, the method will always return to that point (same as when calling a method in main(); when finished, the application will continue to run from the line directly after). The graphical representation above attempts to show you that the method will call itself until the end of the data, and then return to the method which called it until it returns back to the first print() in main().

A better example for this would be to look at a tree. Using a slightly altered version of the CLinkedList used above (so now each CNode points to two elements), we create the following tree:



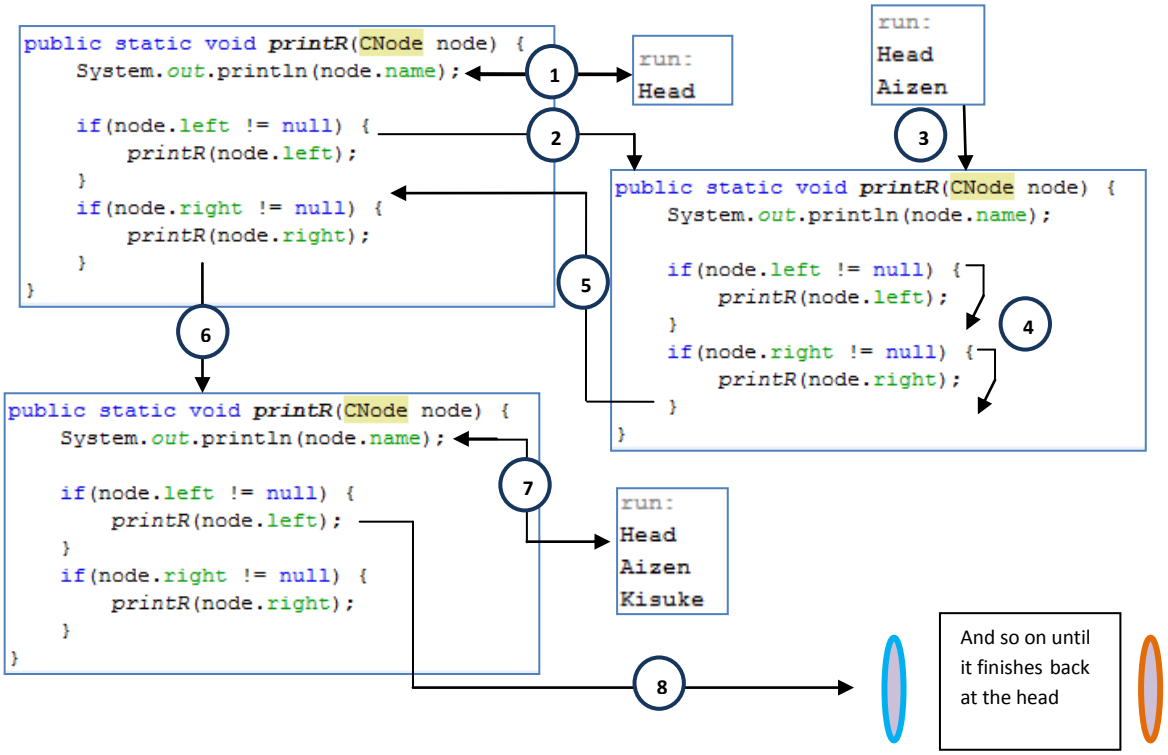
Even with the number of children is increased by one, it is no longer so simple to iterate through the data structure using a while loop (think about it, how do you reach every element?).This is where recursion is the greater method of traversing the data structure.

Following the same procedure as before, we will traverse the tree using recursion. The code is kept simple; for any element, print its name, then if there is an element stored in the 'left' variable, call the print method on that element, after, check if there is an element in the 'right' variable too.

```
public static void printR(CNode node) {
    System.out.println(node.name);

    if(node.left != null) {
        printR(node.left);
    }
    if(node.right != null) {
        printR(node.right);
    }
}
```

So when we call the method printR() on a node, for example, head: `printR(head);`

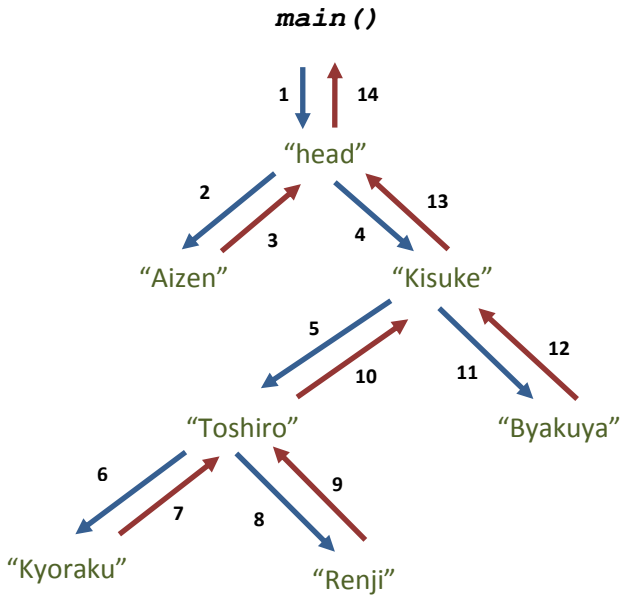
The following will happen (note: numbers are ordering, see explanation below):



NB: Only the steps to print the first 3 elements are shown here, but this continues until all elements are printed, and returns the user back to the original method call (in the main() method) and ends the program.

1. The name of the node is printed out; as the node is the head node, “head” is printed, the method then continues as normal
2. We check to see if the left variable is equal to null; we find that it isn’t, and so we call the method in itself, this time with a different element as its parameter
3. Print the name of the element; in this case “Aizen” is printed, and the method continues to the next piece of code
4. Check if the left variable is equal to null; it is, so we move past this if statement to the next; so we check if the right variable is equal to null; it is as well, so we move past this if statement too
5. The method has no more code left, so it finishes and returns back to its call place; the first printf method where the node is the head element
6. As we finished the first if statement, we move to the next and check if the right variable is a null value, it isn’t, so we call the method in itself again, this time passing the right element as its parameter
7. We print the element name; in this case “Kisuke” is printed, and the method continues to the first if statement
8. We check if the left variable is equal to null, it isn’t, so we call the method on itself again with the left variable as its parameter
9. This continues until the methods are complete and the computer is back to its main() method

A simpler view of the system is as follows, blue lines are when the method printf() is called, and red when it is returning to its call place, each element is represented by its output:




```
run:
Head
Aizen
Kisuke
Toshiro
Kyoraku
Renji
Byakuya
```

Advanced

Above we attempt to show you a recursive method of traversing a table and outputting its contents, instead of the while loop approach you should be familiar with. Here, we will show you how to write a method of traversing the tree and outputting its contents, without recursion.

Unlike the Linked List's used previously, and as discussed above, trees have multiple pointers pointing to their child nodes, and they themselves point to their children and so on, making the normal "while this is not null" approach useless. You could argue that an 'if' statement inside the while loop, or even multiple while loops, could check over all possibilities, but this is unlikely to work for all cases.

Instead, we would use a queue to temporarily store each element, so we can visit them all (turning a dynamic data structure into a linear data structure, AKA cheating).

```
public static void printL(CNode node) {

    Queue<CNode> qu = new LinkedList<CNode>();
    qu.add(node);

    while(!qu.isEmpty()) {
        CNode temp = qu.poll();
        System.out.println(temp.name);

        if(temp.left != null) {
            qu.add(temp.left);
        }
        if(temp.right != null) {
            qu.add(temp.right);
        }
    }

}
```

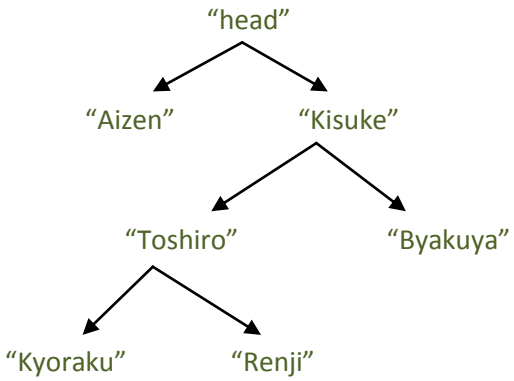
This code essentially moves through the list in a breadth first search fashion (one level at a time), and outputs it as it goes.

- Add the head to the queue
- Begin looking over the Queue, polling the first element (polling is the same as get() but removes the item from the queue)
- Output the elements data, then check if it has any children, if so, add them to the list
- If there were no children, the algorithm would finish as the queue would now be empty, but as luck would have it, there are other elements
- The next element in the array will be polled, printing its name and check if it has children, and if any, add them to the bottom of the queue
- This is repeated until all elements are polled, leaving the queue empty

This method ensures all elements are printed by adding them all to the queue (and any element in the queue will have its children added to the queue).

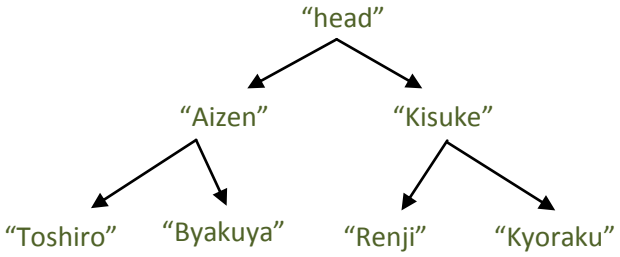
For the LinkedList/Tree example given before, the queue would look a bit like: (NB: the code works from left to right and works through each level at a time, like a printer).

"Head"
"Aizen"
"Kisuke"
"Toshiro"
"Byakuya"
"Kyoraku"
"Renji"



And a slightly different tree to compare it to:

"Head"
"Aizen"
"Kisuke"
"Toshiro"
"Byakuya"
"Renji"
"Kyoraku"



Even though the structure is different, the only change is "Renji" for "Kyoraku", which would be the same as before if I did not switch them. This is just because of the order. Play with the code and try to get it printing from right to left instead.