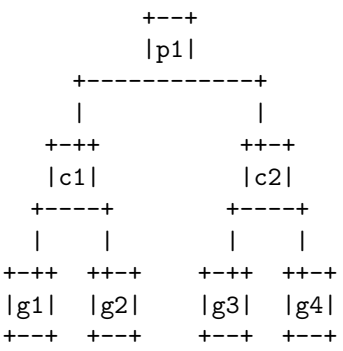
Prepared by Linan Qiu <lq2137@columbia.edu>

Tree Traversal

The tree from our previous example was pretty useless. We have to hold on to references to the parent, the children and all the grand children in order to print out each of them. That just feels intuitively wrong, since I should be able to go from just the parent and print out all of its children and grand children into a giant family tree right?

This process of "walking" the tree and printing out of each node (or doing something else with each of the nodes) is called **tree traversal** and evidently it is super importantly.

Before we do that, let's visualize our tree:



Prefix Traversal

Let's say that I want the list to print out like this: for each node

- Print out the node itself
- Run this procedure on its left child (if it has any)
- Run this procedure on its right child (if it has any)

This is called prefix because the node **precedes** the children.

This is clearly recursive, and since you're 5 weeks into data structures, let's dive into the Java code for this:

```
public void prefix(AwsmNode<T> node) {
  System.out.print(node);
```

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

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

We can run this on our tree earlier by putting this in a test class and we indeed get the following string: `p1c1g1g2c2g3g4`. This is what happens if you trace out our procedure above on the tree we have.

Since we just made a useful structure out of our nodes, let's make this into a class: `AwsmTree`.

```
// AwsmTree.java

public class AwsmTree<T> {
  private AwsmNode<T> parent;

  public AwsmTree(AwsmNode<T> parent) {
    this.parent = parent;
  }

  public void prefix() {
    prefix(parent);
  }

  private void prefix(AwsmNode<T> node) {
    System.out.print(node);

    if(node.left != null) {
      prefix(node.left);
    }

    if(node.right != null) {
      prefix(node.right);
    }
  }

  public static void main(String[] args) {
     // let's make grandchildren!
    AwsmNode<String> grandChild1 = new AwsmNode<>("g1", null, null);
    AwsmNode<String> grandChild2 = new AwsmNode<>("g2", null, null);
    AwsmNode<String> grandChild3 = new AwsmNode<>("g3", null, null);
```

```
    AwsmNode<String> grandChild4 = new AwsmNode<>("g4", null, null);

    // let's make children, and assign them their children
    AwsmNode<String> child1 = new AwsmNode<>("c1", grandChild1, grandChild2);
    AwsmNode<String> child2 = new AwsmNode<>("c2", grandChild3, grandChild4);

    // let's make the parent, and assign it its children
    AwsmNode<String> parent = new AwsmNode<>("p1", child1, child2);

    AwsmTree<String> tree = new AwsmTree<>(parent);
    tree.prefix();
  }
}
```

`AwsmTree` is an entire tree of nodes. However, it only needs to hold on to the top most node – the parent node since every other node in the tree is accessible from the parent node. Notice that we also used the public method / private method pattern from previous chapters in designing the `prefix()` method. Essentially, an external user calls the `prefix()` method, which in turn calls an internal `prefix` method that has an argument: the node to start with. With this, we are not restricted to starting with the top most parent: we can even print subsections of trees starting with a non-parent node (called subtrees).

What if we wanted `prefix()` to return a string instead of just printing it? That gives us more flexibility: after all we can choose to print the string or pass it on to some other method. We can redesign it like this:

```
// AwsmTree.java

public class AwsmTree<T> {
  private AwsmNode<T> parent;

  public AwsmTree(AwsmNode<T> parent) {
    this.parent = parent;
  }

  public String prefix() {
    // create an empty stringbuilder
    StringBuilder sb = new StringBuilder();
    // populate the stringbuilder usin the recursive method
    prefix(parent, sb);
    // now the stringbuilder should be full of node strings
    return sb.toString().trim();
  }

  private void prefix(AwsmNode<T> node, StringBuilder sb) {
```

```
    sb.append(node);
    // add a space in between nodes for clarity.
    sb.append(" ");

    if (node.left != null) {
      prefix(node.left, sb);
    }

    if (node.right != null) {
      prefix(node.right, sb);
    }
  }
}
```

Essentially, what we are doing in the public `prefix()` method is to instantiate a `StringBuilder`. You can think of this as a linked list of strings. What it does is to collect strings that users add to it. We give the private `prefix()` method this `StringBuilder`, and at each step of the recursion, it simply adds onto itself the current node. Then, we return the `toString()` of the `StringBuilder` (which is the concatenated `toStrings()`s of all the nodes in the tree). We call a `trim()` method at the end just to trim the the trailing space (since the last node appended would have added an extra space from the recursive method).

If you need a refresher on why you should use `StringBuilder`s whenever you manipulate strings, please do refer to the String Manipulation note in the 3rd chapter of these notes on Practical Coding. Essentially, if you use `+` or `+=` for strings in Java, the operation is $O(N)$. Java has to copy over each character in both strings being added and create a new string. Remember how arrays cannot be expanded? Strings are arrays of `char`s, hence they cannot be "expanded". You have to create new ones if you are concatenating two strings. Hence, if you do

```
String a = "han solo";

for(int i = 0; i < n; i++) {
  a += n;
}

System.out.println(a);
```

The for loop runs `n` times, making the entire operation $O(N^2)$. However, if you use `StringBuilder`s, this operation would be much more efficient:

```
StringBuilder sb = new StringBuilder();
sb.append("han solo");
```

```java
for(int i = 0; i < n; i++) {
  sb.append(n);
}

System.out.println(sb.toString());
```

`StringBuilder`, being a linked list of strings, has an $O(1)$ `append` operation. This entire operation becomes $O(N)$.

Then, for our tree traversal, it is much more efficient to use `StringBuilder`s than to use string addition.

Now we have a tree with working `prefix()` string output! The tree now prints the prefix: `p1 c1 g1 g2 c2 g3 g4`

Infix Traversal

Infix traversal is very similar to prefix traversal. For each node:

- Run this procedure on its left child (if it has any)
- Print out the node itself
- Run this procedure on its right child (if it has any)

This is called infix because the node is **in between** the children.

You'll probably realize that we are just swapping the first two steps in a prefix traversal. In fact, that's exactly what it is.

Then, we only need a slight modification of the prefix method to make infix work as well.

```java
// AwsmTree.java

public class AwsmTree<T> {
  private AwsmNode<T> parent;

  public AwsmTree(AwsmNode<T> parent) {
    this.parent = parent;
  }

  public String prefix() {
    StringBuilder sb = new StringBuilder();
    prefix(parent, sb);
    return sb.toString().trim();
  }
```

```
private void prefix(AwsmNode<T> node, StringBuilder sb) {
  sb.append(node);
  sb.append(" ");

  if (node.left != null) {
    prefix(node.left, sb);
  }

  if (node.right != null) {
    prefix(node.right, sb);
  }
}

public String infix() {
  StringBuilder sb = new StringBuilder();
  infix(parent, sb);
  return sb.toString().trim();
}

private void infix(AwsmNode<T> node, StringBuilder sb) {
  if (node.left != null) {
    infix(node.left, sb);
  }

  sb.append(node);
  sb.append(" ");

  if (node.right != null) {
    infix(node.right, sb);
  }
}

public static void main(String[] args) {
  // let's make grandchildren!
  AwsmNode<String> grandChild1 = new AwsmNode<>("g1", null, null);
  AwsmNode<String> grandChild2 = new AwsmNode<>("g2", null, null);
  AwsmNode<String> grandChild3 = new AwsmNode<>("g3", null, null);
  AwsmNode<String> grandChild4 = new AwsmNode<>("g4", null, null);

  // let's make children, and assign them their children
  AwsmNode<String> child1 = new AwsmNode<>("c1", grandChild1, grandChild2);
  AwsmNode<String> child2 = new AwsmNode<>("c2", grandChild3, grandChild4);

  // let's make the parent, and assign it its children
  AwsmNode<String> parent = new AwsmNode<>("p1", child1, child2);
```

```
    AwsmTree<String> tree = new AwsmTree<>(parent);
    System.out.println(tree.prefix()); // p1 c1 g1 g2 c2 g3 g4
    System.out.println(tree.infix()); // g1 c1 g2 p1 g3 c2 g4
  }
}
```

Postfix Traversal

As you'd have guessed by now, there's a third variation:

- Run this procedure on its left child (if it has any)
- Run this procedure on its right child (if it has any)
- Print out the node itself

This is caleld postfix traversal, and it is named so because the node **goes after** (hence post) the children.

The implementation is trivial again:

```
// AwsmTree.java

public class AwsmTree<T> {
  private AwsmNode<T> parent;

  public AwsmTree(AwsmNode<T> parent) {
    this.parent = parent;
  }

  public String prefix() {
    StringBuilder sb = new StringBuilder();
    prefix(parent, sb);
    return sb.toString().trim();
  }

  private void prefix(AwsmNode<T> node, StringBuilder sb) {
    sb.append(node);
    sb.append(" ");

    if (node.left != null) {
      prefix(node.left, sb);
    }

    if (node.right != null) {
      prefix(node.right, sb);
```

```
  }
}

public String infix() {
  StringBuilder sb = new StringBuilder();
  infix(parent, sb);
  return sb.toString().trim();
}

private void infix(AwsmNode<T> node, StringBuilder sb) {
  if (node.left != null) {
    infix(node.left, sb);
  }

  sb.append(node);
  sb.append(" ");

  if (node.right != null) {
    infix(node.right, sb);
  }
}

public String postfix() {
  StringBuilder sb = new StringBuilder();
  postfix(parent, sb);
  return sb.toString().trim();
}

private void postfix(AwsmNode<T> node, StringBuilder sb) {
  if (node.left != null) {
    postfix(node.left, sb);
  }

  if (node.right != null) {
    postfix(node.right, sb);
  }

  sb.append(node);
  sb.append(" ");
}

public static void main(String[] args) {
  // let's make grandchildren!
  AwsmNode<String> grandChild1 = new AwsmNode<>("g1", null, null);
  AwsmNode<String> grandChild2 = new AwsmNode<>("g2", null, null);
  AwsmNode<String> grandChild3 = new AwsmNode<>("g3", null, null);
```

```
    AwsmNode<String> grandChild4 = new AwsmNode<>("g4", null, null);

    // let's make children, and assign them their children
    AwsmNode<String> child1 = new AwsmNode<>("c1", grandChild1, grandChild2);
    AwsmNode<String> child2 = new AwsmNode<>("c2", grandChild3, grandChild4);

    // let's make the parent, and assign it its children
    AwsmNode<String> parent = new AwsmNode<>("p1", child1, child2);

    AwsmTree<String> tree = new AwsmTree<>(parent);
    System.out.println(tree.prefix()); // p1 c1 g1 g2 c2 g3 g4
    System.out.println(tree.infix()); // g1 c1 g2 p1 g3 c2 g4
    System.out.println(tree.postfix()); // g1 g2 c1 g3 g4 c2 p1
  }
}
```

(Bonus) Pretty `toString()`

So far the output string has been pretty unsatisfying: it is simply a line of text. What if we wanted something like this:

```
p1
  c1
    g1
    g2
  c2
    g3
    g4
```

Just like how a directory of files on your computer would be listed. This clearly shows the hierarchy. Well, turns out we can easily do this. Take a look at the `toString()` method of `AwsmTree` that I implemented. I challenge you to figure out the code on your own. It should be quite comprehensible.

```
// AwsmTree.java

public class AwsmTree<T> {
  private AwsmNode<T> parent;

  public AwsmTree(AwsmNode<T> parent) {
    this.parent = parent;
  }

  public String prefix() {
```

```
  StringBuilder sb = new StringBuilder();
  prefix(parent, sb);
  return sb.toString().trim();
}

private void prefix(AwsmNode<T> node, StringBuilder sb) {
  sb.append(node);
  sb.append(" ");

  if (node.left != null) {
    prefix(node.left, sb);
  }

  if (node.right != null) {
    prefix(node.right, sb);
  }
}

public String infix() {
  StringBuilder sb = new StringBuilder();
  infix(parent, sb);
  return sb.toString().trim();
}

private void infix(AwsmNode<T> node, StringBuilder sb) {
  if (node.left != null) {
    infix(node.left, sb);
  }

  sb.append(node);
  sb.append(" ");

  if (node.right != null) {
    infix(node.right, sb);
  }
}

public String postfix() {
  StringBuilder sb = new StringBuilder();
  postfix(parent, sb);
  return sb.toString().trim();
}

private void postfix(AwsmNode<T> node, StringBuilder sb) {
  if (node.left != null) {
    postfix(node.left, sb);
```

```
    }

    if (node.right != null) {
      postfix(node.right, sb);
    }

    sb.append(node);
    sb.append(" ");
  }

  public String toString() {
    StringBuilder sb = new StringBuilder();
    toString(parent, sb, 0);
    return sb.toString().trim();
  }

  private void toString(AwsmNode<T> node, StringBuilder sb, int level) {
    for (int i = 0; i < level; i++) {
      sb.append("  ");
    }
    sb.append(node);
    sb.append("\n");

    if (node.left != null) {
      toString(node.left, sb, level + 1);
    }

    if (node.right != null) {
      toString(node.right, sb, level + 1);
    }
  }

  public static void main(String[] args) {
    // let's make grandchildren!
    AwsmNode<String> grandChild1 = new AwsmNode<>("g1", null, null);
    AwsmNode<String> grandChild2 = new AwsmNode<>("g2", null, null);
    AwsmNode<String> grandChild3 = new AwsmNode<>("g3", null, null);
    AwsmNode<String> grandChild4 = new AwsmNode<>("g4", null, null);

    // let's make children, and assign them their children
    AwsmNode<String> child1 = new AwsmNode<>("c1", grandChild1, grandChild2);
    AwsmNode<String> child2 = new AwsmNode<>("c2", grandChild3, grandChild4);

    // let's make the parent, and assign it its children
    AwsmNode<String> parent = new AwsmNode<>("p1", child1, child2);
```

```
    AwsmTree<String> tree = new AwsmTree<>(parent);
    System.out.println(tree.prefix());
    System.out.println(tree.infix());
    System.out.println(tree.postfix());
    System.out.println(tree);
  }
}
```

Stack Prefix Traversal

Now you learned earlier that every recursive algorithm can be transformed into an iterative version. It is just a matter of programming convenience. In this case, it is not too hard to translate the recursive prefix traversal into an iterative one.

The iterative version looks like this:

```
// in AwsmTree.java

public String prefixStack() {
  StringBuilder sb = new StringBuilder();

  LinkedList<AwsmNode<T>> stack = new LinkedList<>();
  stack.push(parent);

  while (!stack.isEmpty()) {
    AwsmNode<T> current = stack.pop();
    sb.append(current);
    sb.append(" ");

    if (current.left != null) {
      stack.push(current.left);
    }

    if (current.right != null) {
      stack.push(current.right);
    }
  }

  return sb.toString().trim();
}
```

Essentially, we start with the same `StringBuilder`. This time, we use a stack. We push the parent node onto the stack, then enter a while loop. The while loop tests if the stack is empty. If it is, it means that we have no more nodes left. If it isn't, we pop the top one and make it our current node. Here is where

we do our prefix routine - we append the current node's `toString()`, then push the left and right children onto the stack.

This ensures that the next node to be popped off will be the left node, means that we descend into the left child, exactly what we want for prefix traversal.

Breadth First Traversal

Now what if we wanted to print the tree in level order? In other words, the parents will get printed first, then all the children, then all the grand children. Our current methods won't work – it dives in deep and on the left. In fact, it is the opposite of what we want. Prefix, infix and postfix traversal are **depth first traversals**. They go deep first. What we need to print the tree in level order is **breadth first traversal**: going level by level.

Turns out this is not too hard at all. We simply take the algorithm we have above for iterative depth first traversal of the tree and **change the stack to a queue**.

```
// in AwsmTree.java

public String breadthFirst() {
  StringBuilder sb = new StringBuilder();

  LinkedList<AwsmNode<T>> queue = new LinkedList<>();
  queue.addFirst(parent);

  while (!queue.isEmpty()) {
    AwsmNode<T> current = queue.removeLast();
    sb.append(current);
    sb.append(" ");

    if (current.left != null) {
      queue.addFirst(current.left);
    }

    if (current.right != null) {
      queue.addFirst(current.right);
    }
  }

  return sb.toString().trim();
}
```

This produces us the string: `p1 c1 c2 g1 g2 g3 g4`, a level order traversal of the tree.

The reason why this works is that nodes that get visited first gets dequeued first. Remember that a queue is first in first out (FIFO). Since nodes near the top get visited first, their children get enqueued at the front of the queue. As opposed to the stack that would get the latest child out first, the queue dequeues the earliest child out even as more grandchildren are being enqueued onto the back. The stack would have popped out the grand children, causing a further descent.

Iterating a Tree

Now that we are good at traversing trees, we can construct an iterator for the tree. Let's make an inner class in `AwsmTree` called `AwsmTreeIterator`:

```
// in AwsmTree.java

public class AwsmTreeIterator<Q> implements Iterator<Q> {

    private LinkedList<AwsmNode<Q>> list;

    public AwsmTreeIterator(AwsmNode<Q> parent) {
      list = new LinkedList<>();
      prefixTraverse(parent);
    }

    public void prefixTraverse(AwsmNode<Q> node) {
      list.add(node);
      if (node.left != null) {
        prefixTraverse(node.left);
      }
      if (node.right != null) {
        prefixTraverse(node.right);
      }
    }

    @Override
    public boolean hasNext() {
      return !list.isEmpty();
    }

    @Override
    public Q next() {
      return list.removeFirst().data;
    }
  }
}
```

This iterator takes in a node to start iterating from, and populates a linked list using the recursive prefix traversal method. Then, we simply use the linked list to determine `hasNext` and `next`.

To return an instance of this iterator, we instantiate an instance of `AwsmTreeIterator` with the parent node. The entire `AwsmTree` now looks like this:

```
// AwsmTree.java

import java.util.Iterator;
import java.util.LinkedList;

public class AwsmTree<T> implements Iterable<T> {
  private AwsmNode<T> parent;

  public AwsmTree(AwsmNode<T> parent) {
    this.parent = parent;
  }

  public String prefix() {
    StringBuilder sb = new StringBuilder();
    prefix(parent, sb);
    return sb.toString().trim();
  }

  private void prefix(AwsmNode<T> node, StringBuilder sb) {
    sb.append(node);
    sb.append(" ");

    if (node.left != null) {
      prefix(node.left, sb);
    }

    if (node.right != null) {
      prefix(node.right, sb);
    }
  }

  public String infix() {
    StringBuilder sb = new StringBuilder();
    infix(parent, sb);
    return sb.toString().trim();
  }

  private void infix(AwsmNode<T> node, StringBuilder sb) {
```

```
  if (node.left != null) {
    infix(node.left, sb);
  }

  sb.append(node);
  sb.append(" ");

  if (node.right != null) {
    infix(node.right, sb);
  }
}

public String postfix() {
  StringBuilder sb = new StringBuilder();
  postfix(parent, sb);
  return sb.toString().trim();
}

private void postfix(AwsmNode<T> node, StringBuilder sb) {
  if (node.left != null) {
    postfix(node.left, sb);
  }

  if (node.right != null) {
    postfix(node.right, sb);
  }

  sb.append(node);
  sb.append(" ");
}

public String prefixStack() {
  StringBuilder sb = new StringBuilder();

  LinkedList<AwsmNode<T>> stack = new LinkedList<>();
  stack.push(parent);

  while (!stack.isEmpty()) {
    AwsmNode<T> current = stack.pop();
    sb.append(current);
    sb.append(" ");

    if (current.left != null) {
      stack.push(current.left);
    }
```

```
    if (current.right != null) {
      stack.push(current.right);
    }
  }

  return sb.toString().trim();
}

public String breadthFirst() {
  StringBuilder sb = new StringBuilder();

  LinkedList<AwsmNode<T>> queue = new LinkedList<>();
  queue.addFirst(parent);

  while (!queue.isEmpty()) {
    AwsmNode<T> current = queue.removeLast();
    sb.append(current);
    sb.append(" ");

    if (current.left != null) {
      queue.addFirst(current.left);
    }

    if (current.right != null) {
      queue.addFirst(current.right);
    }
  }

  return sb.toString().trim();
}

public String toString() {
  StringBuilder sb = new StringBuilder();
  toString(parent, sb, 0);
  return sb.toString().trim();
}

private void toString(AwsmNode<T> node, StringBuilder sb, int level) {
  for (int i = 0; i < level; i++) {
    sb.append("  ");
  }
  sb.append(node);
  sb.append("\n");

  if (node.left != null) {
    toString(node.left, sb, level + 1);
```

```
  }

  if (node.right != null) {
    toString(node.right, sb, level + 1);
  }
}

@Override
public Iterator<T> iterator() {
  return new AwsmTreeIterator<T>(parent);
}

public class AwsmTreeIterator<Q> implements Iterator<Q> {

  private LinkedList<AwsmNode<Q>> list;

  public AwsmTreeIterator(AwsmNode<Q> parent) {
    list = new LinkedList<>();
    prefixTraverse(parent);
  }

  public void prefixTraverse(AwsmNode<Q> node) {
    list.add(node);
    if (node.left != null) {
      prefixTraverse(node.left);
    }
    if (node.right != null) {
      prefixTraverse(node.right);
    }
  }

  @Override
  public boolean hasNext() {
    return !list.isEmpty();
  }

  @Override
  public Q next() {
    return list.removeFirst().data;
  }
}

public static void main(String[] args) {
  // let's make grandchildren!
  AwsmNode<String> grandChild1 = new AwsmNode<>("g1", null, null);
  AwsmNode<String> grandChild2 = new AwsmNode<>("g2", null, null);
```

```
    AwsmNode<String> grandChild3 = new AwsmNode<>("g3", null, null);
    AwsmNode<String> grandChild4 = new AwsmNode<>("g4", null, null);

    // let's make children, and assign them their children
    AwsmNode<String> child1 = new AwsmNode<>("c1", grandChild1, grandChild2);
    AwsmNode<String> child2 = new AwsmNode<>("c2", grandChild3, grandChild4);

    // let's make the parent, and assign it its children
    AwsmNode<String> parent = new AwsmNode<>("p1", child1, child2);

    AwsmTree<String> tree = new AwsmTree<>(parent);
    System.out.println(tree.prefix());
    System.out.println(tree.infix());
    System.out.println(tree.postfix());
    System.out.println(tree);

    System.out.println(tree.breadthFirst());

    for (String element : tree) {
      System.out.println(element);
    }
  }

}
```

Notice that now we can do an enhanced for loop over the entire tree. Isn't that sweet?