Traversing Binary Trees Iteratively

Nitin Verma
mathsanew.com

November 5, 2020

The *Preorder*, *Inorder* and *Postorder* traversals of *Binary Trees* are very simple to implement in their recursive form. In fact, their recursive methods very closely resemble their very definitions. In this article, we will be creating iterative variants of these methods, which use no recursive calls.

All methods shown below will be written in C language. A node of a binary tree will be represented by a structure called *node*, having members *lc*/*rc* which are pointers to the left/right child of the node. It can have other members also depending upon the application's needs. We assume that the node has no pointer to its parent node.

When the nodes have no pointer to their parents, these traversals will need to remember the parent nodes somehow, as the child and descendant nodes are traversed. In the recursive methods, this remembering automatically happens due to the use of program-stack. But in our iterative methods, we will use our own stack to track the nodes in various stages of their traversal. The stack will not only serve as a collection of nodes' pointers to get back to them later, but also to maintain them in a specific order by virtue of its *Last-In-First-Out* ordering.

Resume-Points

We will first define *Resume Points* with the help of the below method as an example. This is the recursive method for Preorder traversal:

```
void pre(node *x)
{
  /* RP 0 */
  process(x);               /* step A */
  if(x->lc) pre(x->lc);     /* step B */
  /* RP 1 */
  if(x->rc) pre(x->rc);     /* step C */
  /* RP 2 */
}
```

Observe that the execution of $pre(x)$ becomes active multiple times. It starts at step A. The recursive call at step B suspends its execution. Upon return of this recursive call, it becomes active again at location marked with "RP 1". Similarly, after recursive call of step C returns, it becomes active again at location "RP 2". We will refer to such instruction locations just after return from a recursive call as "Resume Point", abbreviated as "RP". In above method, there are two resume-points, marked "RP 1" and "RP 2". The very first instruction is not a resuming location, but still we will refer it as a special resume-point 0, and mark it "RP 0". So, the execution of $pre(x)$ becomes active at three different times at resume-points 0, 1 and 2 respectively. Note, if the lc/rc pointers are null, the execution simply continues without making the recursive call.

Basic Structure of Iterative Methods

All of the three iterative methods which we will write have some common basic structure. In this section, we use method name $f()$ to refer to the recursive method of any of the three traversals. It takes a single argument $(node *x)$, which is the root of the tree to traverse.

As said earlier, we will maintain a stack in the iterative methods. Each element of this stack will represent one particular invocation of the recursive method $f()$ with its argument node-pointer x. Since the invocation of $f()$ starts or resumes at the resume-points, we will associate a resume-point to each of these elements. It indicates the point where the invocation will start/resume when it becomes active. So, each element can be represented as $(f(x)\colon r)$, where x is the argument node, and r is the resume-point number, which will be 0 for fresh calls (not resuming). Note that both x and r can vary across the elements of the stack. For convenience, a short notation

$(x\colon r)$ will be used, since the method name is understood as per context. Thus, the stack consists of elements representing invocations of $f()$ at its different execution stages identified by resume-points.

Before we start the iterations in our iterative methods, this stack will be initialized with a single element representing the initial call, i.e. $(root\colon 0)$, where *root* is the pointer to root of the tree. In each iteration, we will read the topmost element of the stack, which will be called the "current" element of that iteration. The iteration will perform its execution based on its current element, and this can involve pushing of new elements for the recursive calls, and popping the current element if it is no longer needed (for example, if its execution is complete and has to return). The loop will terminate when there are no more elements on the stack to execute.

Simulating Recursion

One way to create iterative variants of these recursive methods is to just simulate the recursive calls exactly as they would happen on the program-stack, but using our own stack. Our stack can maintain its elements to represent the recursive invocations similar to how program-stack maintains the call-frames. New elements will be pushed to simulate each recursive call, and the topmost element will be popped when a call returns. The elements (representing invocations) in our stack will pass through each of their resume-points as they execute. These resume-points can be stored as a number within each stack element, to indicate where to resume the execution. In an earlier article, *Maintaining the Stack for Recursion*, we had discussed about simulating a general recursive method on an stack.

But as we will see below, we can do some optimizations according to the specific traversal, and need not make an invocation element pass through all the resume-points. We will also derive the resume-point of an element as needed, instead of storing it.

Now, having understood some general details, we will go through each traversal individually.

Preorder Traversal

In the *pre*() method (page 2), we can note that once we execute *process*(x) and have saved both $x{\rightarrow}lc$ and $x{\rightarrow}rc$ somewhere for performing the recursive calls, we no longer need to remember node x. We can benefit from this fact in our iterative method. Consider the iteration where current element is (x: 0). It can be handled as:

1. *process*(x)
2. pop (x: 0)
3. if $x{\rightarrow}rc$ is not null, push ($x{\rightarrow}rc$: 0)
4. if $x{\rightarrow}lc$ is not null, push ($x{\rightarrow}lc$: 0).

We popped the element in Step 2 as it will no longer be needed.

Note that in this traversal's case, the stack will consist of only one kind of elements, of the form (x: 0). So, we need not store the resume-point for each element as it is implied to be 0 always. Thus, each element in the stack can simply be a node pointer. As there are only one kind of elements on the stack, so the logic for handling the elements in an iteration becomes simpler.

In this and other iterative methods, the *stack* is a fixed-size array, but other ways to store the stack can be used. Code for bound-checking the *stack* array has not been shown. The code portion of an iteration, where a specific type of stack element is handled is marked with a comment mentioning the element's type, e.g. `/* type (x: 0) */`.

The iterative method *pre_i*() can be written as below:

```
void pre_i(node *root)
{
  node *stack[1000];
  int top;

  stack[0] = root;
  top = 0;

  while(top >= 0)
  {
    node *curr = stack[top];

    /* type (x: 0) */
    process(curr);
    top--;

    if(curr->rc) stack[++top] = curr->rc;

    if(curr->lc) stack[++top] = curr->lc;
  }
}
```

Inorder Traversal

Its recursive method can be written as:

```
void in(node *x)
{
  /* RP 0 */
  if(x->lc) in(x->lc);
  /* RP 1 */
  process(x);
  if(x->rc) in(x->rc);
  /* RP 2 */
}
```

Consider the iteration where current element is $(x\colon 0)$. We cannot do $process(x)$ without completing the traversal of the left child subtree. So we push $(x{\rightarrow}lc\colon 0)$ on the stack (if $x{\rightarrow}lc$ is not null) over the existing $(x\colon 0)$ element, and update $(x\colon 0)$ to $(x\colon 1)$ to indicate that $in(x)$ will later resume execution at resume-point 1. Next iteration and possibly its following ones

will complete the traversal of the left-child subtree. If $x{\rightarrow}lc$ is null, the current element can effectively be treated as $(x\colon 1)$.

When an iteration will find $(x\colon 1)$ as the current (topmost) element, it will do $process(x)$, pop this element, and push $(x{\rightarrow}rc\colon 0)$ if $x{\rightarrow}rc$ is not null. The current element could be popped because we now don't need to remember anything for invocation of $in(x)$, and so can get rid of it from the stack.

Thus, for this traversal, the stack will consist of two kinds of elements $(x\colon 0)$ and $(x\colon 1)$. The first kind can only be found on the top of the stack.

We may keep the resume-point number within each element of the stack. But instead of that, we can derive its value for the current element. If the last iteration pushed an element on the stack, we derive the current element's resume-point as 0. Otherwise, we default to the only remaining possibility of 1. Thus, each element in the stack can simply be a node pointer.

The iterative method $in\_i()$ can be written as below:

```
void in_i(node *root)
{
  node *stack[1000];
  int top;
  char pushed;

  stack[0] = root;
  top = 0;
  pushed = 1;

  while(top >= 0)
  {
    node *curr = stack[top];

    if(pushed)
    {
      /* type (x: 0) */
      if(curr->lc)
      {
        stack[++top] = curr->lc;
        continue;
      }
    }

    /* type (x: 1) */
    pushed = 0;
    process(curr);
    top--;

    if(curr->rc)
    {
      stack[++top] = curr->rc;
      pushed = 1;
    }
  }
}
```

Postorder Traversal

Its recursive method can be written as:

```
void post(node *x)
{
  /* RP 0 */
  if(x->lc) post(x->lc);
  /* RP 1 */
  if(x->rc) post(x->rc);
  /* RP 2 */
  process(x);
}
```

Consider the iteration where current element is $(x\colon 0)$. This will be handled as:

1. if $x{\rightarrow}rc$ is not null, push $(x{\rightarrow}rc\colon 0)$
2. if $x{\rightarrow}lc$ is not null, push $(x{\rightarrow}lc\colon 0)$
3. update $(x\colon 0)$ to $(x\colon 2)$ leaving it where it was on the stack; it is now below the elements pushed by above steps.

If both child pointers are null, the current element can effectively be treated as $(x\colon 2)$.

By pushing both the *lc* and *rc* pointers together, we have removed the need of keeping the $post(x)$ invocation at resume-point 1 while the $post(x{\rightarrow}lc)$ completes its execution. We made sure that both recursive invocations are taken care of by pushing their elements and so could directly shift $(x\colon 0)$ to $(x\colon 2)$. Again, similar to earlier traversals, this optimization helped in reducing the kinds of elements on the stack to two: $(x\colon 0)$ and $(x\colon 2)$. So, the logic for handling the elements in an iteration becomes simpler.

When an iteration finds $(x\colon 2)$ as the current element, it can simply do $process(x)$, and then pop this element.

We may keep the resume-point number within each element of the stack, but again we can try deriving it like earlier traversals. Note that elements of the form $(x\colon 0)$ and $(x\colon 2)$ can be anywhere on the stack in this case. When an iteration has current element $(x\colon 2)$ and pops it, the next iteration's element need not be for the parent of x; it can be for the sibling node of x (since left/right child pointers are pushed together). To derive the resume-point of current element, we can check if the last iteration popped a child

(left or right) of the current node. If so, the resume-point is 2. Otherwise, we default to the only remaining possibility of 0. Thus, like earlier traversals, each element in the stack can simply be a node pointer.

The iterative method *post_i*() can be written as below:

```
void post_i(node *root)
{
  node *stack[1000];
  int top;
  node *popped;

  stack[0] = root;
  top = 0;
  popped = NULL;

#define POPPED_A_CHILD() \
  (popped && (popped == curr->lc || popped == curr->rc))

  while(top >= 0)
  {
    node *curr = stack[top];

    if(!POPPED_A_CHILD())
    {
      /* type (x: 0) */
      if(curr->rc || curr->lc)
      {
        if(curr->rc) stack[++top] = curr->rc;

        if(curr->lc) stack[++top] = curr->lc;

        popped = NULL;
        continue;
      }
    }

    /* type (x: 2) */
    process(curr);
    top--;
    popped = curr;
  }
}
```