DS 5^{th} homework

58121102 Jiang Yuchu

November 13, 2022

P_{267} 4. Implement a forward iterator for *Tree*. Your iterator should traverse the tree in inorder.

Answer:

Quite simple, see *p267-4.cpp*. I implemented two kinds of iterators, which are `tree_const_iterator` and `tree_iterator`. Both of them are completely satisfied with requirements of c++ standard for the forward iterators.

`tree.begin()` should return the "leftmost" child of the tree, which needs to maintain an extra field to record, and `tree.end()` should return `nullptr`. This works for forward iterator, but not for bidirectional iterator. I'll describe a better implementation about tree in experiment of this chapter.

P_{267} 6. Write a nonrecursive version of function *Preorder* (Program 5.2).

Answer:

Quite simple, see *p267-6.cpp*. By using stack, its space overhead is $O(log(n))$.

So now, the question is, is it possible to traverse a binary tree without maintaining the field `parent` and using less space? Yes, it is. Next, I will introduce the Morris traversal algorithm in appendix, and here I gave a preorder version.

```
template <class Tp>
void morrisPreorder(TreeNode<Tp>* root) {
    if(root == nullptr)
        return;
    TreeNode<Tp>* cur = root, *pre;
    // pre represents the predecessor of cur in inorder traversal
    while (cur){
        pre = cur->left;
        if(pre){
            while (pre->right && pre->right != cur)
                pre = pre->right;
            if(pre->right == nullptr){
                pre->right = cur;
                Visit(cur);
                cur = cur->left;
                continue;
            }
            else
                pre->right = nullptr;
        }
        else
            Visit(cur);
        cur = cur->right;
    }
}
```

If using the implementation of `Preorder` I used at first to develop a forward iterator, `tree.begin()` should return the root of the tree, and `tree.end()` should return `nullptr`. the operator++ could be like this:

```
constexpr MyIterator& operator++() {
    if (node->left)
        node = node->left;
    else if (node->right)
        node = node->right;
    else {
        auto* suc = node->parent;
        while (node != root) {
            if (node == suc->left && suc->right) {
                node = suc->right;
                return;
            }
            node = suc, suc = suc->parent;
        }
        // How to set node to be nullptr after accessing ??
    }
    return *this;
}
```

The problem occurs, how to set node to be `nullptr` after using? I solved this by developing a new structure of tree, see experiment.

P_{273} 4. [Destructor] Write a recursive function to delete all nodes in a binary tree. What is the complexity of your function?

Answer:

Quite simple. It can be done by post-order traversal, which is the most classical method.

```
void destroy(TreeNode<Tp>* node) { // in class Tree
    if (node) {
        destroy(node->left);
        destroy(node->right);
        delete std::exchange(node, nullptr);
    }
}
~Tree() noexcept { // in class Tree
    destroy(root);
}
```

The complexity is $O(n)$, where n represents the number of nodes, because destroy operation needs to travel the whole tree.

To reduce recursion overhead, `destroy` can also be written as follows:

```
void destroy(TreeNode<Tp>* node) { // in class Tree
     while (node) {
         destroy(node->right);
         delete std::exchange(node, node->left);
     }
}
```

P_{296} 1. Write a C++ function to delete the pair with key k from a binary search tree. What is the time complexity of your function?

Answer:

See *p296-1.cpp*.The time complexity of remove operation is $O(log(n))$, where n represents the number of nodes.

n	$height$	$height/log_2(n)$
100	15	2.26
500	18	2.01
1000	19	1.91
2000	27	2.46
3000	30	2.60
4000	27	2.26
5000	28	2.28
6000	31	2.47
7000	30	2.35
9000	30	2.28
10000	31	2.33

Table 1: Running result table

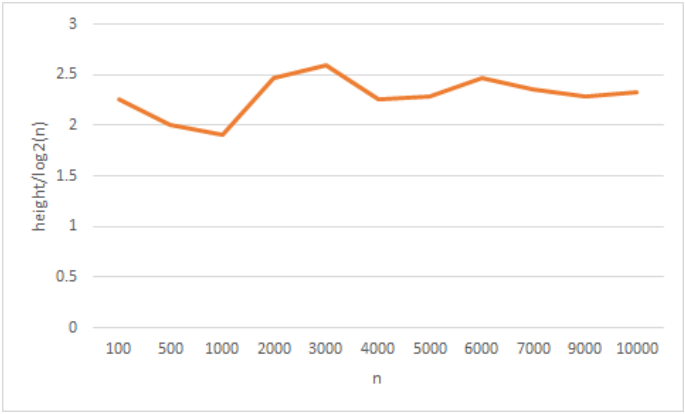

Figure 1: Running result figure

P_{296} 2. Write a program to start with an initially empty binary search tree and make n random insertions. Use a uniform random number generator to obtain the values to be inserted. Measure the height of the resulting binary search tree and divide this height by $log_2 n$. Do this for $n = 100, 500, 1000, 2000, 3000, \cdots, 10,000$. Plot the ratio $height/log_2 n$ as a function of n. The ratio should be approximately constant (around 2). Verify that this is so.

Answer:

See *p296-2.cpp*. I tested insertion when $n = 100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 9000, 10000$, then here are results. The following is the running result table and the graph drawn accordingly.

As shown in figure,the ratio is approximately constant (around 2).

P_{277} 1. Write a C++ function to insert a new node l as the left child of node s in a threaded binary tree.The left subtree of s becomes the left subtree of l

Answer:

The case of inserting the left subtree is similar to the case of inserting the right subtree, except that the left and right are exchanged in `InsertRight`.

```
1  void InsertLeft(ThreadedNode<T>* s,  ThreadedNode<T>* l) {
2  // in class ThreadedTree
3      l->leftChild = s->leftChild;
4      l->leftThread = s->leftThread;
5      l->rightChild = s;
6      l->rightThread = true;
7      s->leftChild=l;
8      s->leftThread=false;
9      if (!l->leftThread) {
10         ThreadedNode<T>* temp = InorderPred(l);
11         temp->leftChild = l;
12     }
13 }
```

P_{278} 4. Write a function to traverse a threaded binary tree in pre-order.What are the time and space requirements of your method?

Answer:

```
1  void preorder(ThreadedNode<T>* node) { // in class ThreadedTree<T>
2      if (node) {
3          Visit(node);
4          if (!node->leftThread)
5              preorder(node->left);
6          if (!node->rightThread)
7              preorder(node->right);
8      }
9  }
10
11 //use
12 preorder(root);
```

The time complexity is $O(n)$, where n represents the number of nodes. And I didn't use extra space to travel the threaded binary tree, so the space complexity is $O(1)$.

In fact, for the most part, traversing a threaded tree is not much different from a normal binary tree, except for the way it determines whether it is null or not.

P_{287} 2. Write a C++ abstract class similar to ADT 5.2 for the ADT `MinPQ`, which defines a min priority queue. Now write a C++ class `MinHeap` that derives from this abstract class and implements all the virtual functions of `MinPQ` The complexity of each function should be the same as that for the corresponding function of `MaxHeap`.

Answer:

See *p287-2.cpp*. Notice that the beginning index of my `MinHeap` is 0 instead of 1.

P_{287} 3. The worst-case number of comparisons performed during an insertion into a max heap can be reduced to $O(loglogn)$ by performing a binary search on the path from the new leaf to the root This does not affect the number of data moves though. Write an insertion algorithm that uses this strategy. Redo Exercise 1 using this insertion algorithm. Based on your experiments, what can you say about the value of this strategy over the one used in Program 5.16?

Answer:

According to the requirements of exercise 1, I tested the running performance of implementing priority queue with max heap, ordered list and unordered list respectively. First, randomly generate n ($n = 100, 500, 1000, 2000, 3000, 4000$) data and insert them into the priority queue, and then generate m ($m = 1000$) operation sequences (push or pop) to apply to the three priority queues , and calculate their average running time.

The following are my experimental results (as shown in Table 2) and the graph drawn according to the experimental results (as shown in Figure 2).

average time(us) / queue size	unorderd list	ordered list	max heap
100	2.7963	5.3763	4.8245
500	11.8832	26.3603	6.0658
1000	33.4758	73.3945	7.5881
2000	75.6631	161.364	9.4102
3000	121.38	272.443	7.5615
4000	197.523	443.986	30.7587

Table 2: Performance table of the three priority queues

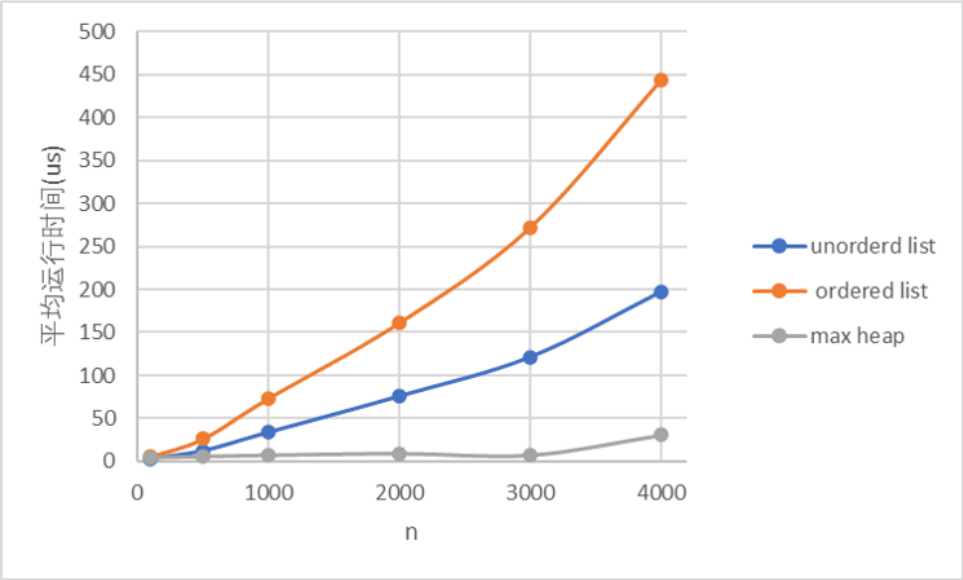

Figure 2: Performance graphs for three priority queues

If comparisons are expensive, then there are certainly reasons to use this strategy. However, if it's just an ordinary comparison, it doesn't seem to make much sense to use $O(logn)$ extra space in order to reduce the number of worst-case comparisons.

P_{301} 1. Write C++ class definitions for winner and loser trees.

Answer:

P_{301} 4. Write a function to construct a loser tree for k records.Use position 0 of your loser-tree array to store a pointer to the overall winner. Show that this construction can be carried out in time $O(k)$. Assume that k is a power of

Answer:

Experiment

P_{267}10.

1 Experiment Target

1.1 Requirements

Develop a complete C++ template class for binary trees. You must include a constructor, copy constructor, destructor, the four traversal methods of this section together with forward iterators for each. Include also the remaining functions specified in ADT 5.1.

1.2 Improvements

Based on the requirements of the topic, I have made the following improvements

1. I implemented the **bidirectional** iterator instead of forward iterator. Besides, four traversal methods mentioned in this section can be switched by defining different specific macros(`_PREORDER_ITERATOR_`, `_INORDER_ITERATOR_`, `_POSTORDER_ITERATOR_`, `_LEVEL_ORDER_ITERATOR_`), and in-order iterator is default.

2. With the help of c++ meta-programming techniques such as CRTP, Mixin, SFINAE and so on, containers such `set`,`map`,`multiset`,`multimap` can be developed using `BinaryTree` as the underlying data structure by simply specifying different template parameters. Besides, by implementing

ADT 5.1

```
template <class T>
class BinaryTree {
// objects: A finite set of nodes either empty or consisting of a
// root node, left BinaryTree and right BinaryTree.
public:
BinaryTree();
// creates an empty binary tree
bool IsEmpty();
// return true iff the binary tree is empty
BinaryTree(BinaryTree <T>& bt1, T& item, BinaryTree <T>& bt2);
// creates a binary tree whose left subtree is bt1,
// whose right subtree is bt2, and whose root node contains item
BinaryTree <T> LeftSubtree();
// return the left subtree of *this
BinaryTree <T> RightSubtree();
// return the right subtree of *this
T RootData();
// return the data in the root node of *this
};
```

different traits (including some functions that maintain the nature of balanced binary trees such as `insert_fixup`, `erase_fixup` called after insertion and deletion), it is easy to get containers with different underlying data structures.

3. In terms of memory management, use allocator instead of `new`/`delete`.
4. Since the allocator is used, the SCARY iterator should be implemented. See *iter.h*.
5. Implement more functions to meet set/map usage as required by the c++ standard. Of course, not all of functions.
6. Use **static dispatching** technique to implement polymorphism in compile time.

1.3 Complete Interfaces

To make the naming convention uniform, I changed the names of the functions in requirements. Then, I also marked every required function with a comment.

```
template <class Traits, template <class, class> class... MixIn>
class BinaryTree :
  public MixIn<Traits, BinaryTree<Traits, MixIn...>>... {
    public:
    // begin()s and end()s ...

    // required
    BinaryTree();

    BinaryTree(const BinaryTree& other);
    BinaryTree(BinaryTree&& other);

    // required
    BinaryTree(BinaryTree& bt1,
        const_reference item, BinaryTree& bt2);

    template <std::input_iterator Iter>
    BinaryTree(Iter first, Iter last);
    BinaryTree(std::initializer_list<value_type> l);
```

```
20
21
22      template <bool IsMulti = Multi,
23          std::enable_if_t<IsMulti, int> = 0>
24      iterator insert(const_reference value);
25      template <bool IsMulti = Multi,
26      std::enable_if_t<IsMulti, int> = 0>
27      iterator insert(value_type&& value);
28
29      template <bool IsMulti = Multi,
30          std::enable_if_t<!IsMulti, int> = 0>
31      std::pair<iterator, bool> insert(const_referencevalue);
32      template <bool IsMulti = Multi,
33          std::enable_if_t<!IsMulti, int> = 0>
34      std::pair<iterator, bool> insert(value_type&&value);
35
36      void insert(std::initializer_list<value_type> l);
37      template <std:input_iterator Iter>
38      void insert(Iter first, Iter last);
39
40
41      iterator lower_bound(const key_type& key);
42      const_iterator lower_bound(const key_type& key)const;
43      template <class Key, class Cmpr = key_compare,
44          class = typename Cmpr::is_transparent>
45      iterator lower_bound(const Key& key);
46      template <class Key, class Cmpr = key_compare,
47          class = typename Cmpr::is_transparent>
48      const_iterator lower_bound(const Key& key) const;
49
50      // the similar interfaces for upper_bound() and equal_range()
51
52      iterator erase(const_iterator position) noexcept;
53      size_type erase(const key_type& value) noexcept(equal_range(value));
54      iterator erase(const_iterator first, const_iterator last);
55
56      size_type size() const noexcept;
57
58      // required
59      bool empty() const noexcept;
60      BinaryTree left_subtree() const;
61      BinaryTree right_subtree() const;
62      value_type root_data() const;
63
64      void swap(BinaryTree& tree);
65
66      key_compare key_comp() const;
67      value_compare value_comp() const;
68
69      void clear() noexcept;
70
71      ~BinaryTree();
72
73      BinaryTree& operator=(const BinaryTree&);
74      BinaryTree& operator=(BinaryTree&&);
75  }
```

2 Basic Idea

2.1 Structure Design

1. In order to implement bidirectional iterator, I added a sentinel root. Its parent pointer points to the real root, and the left and right pointers point to the leftmost and rightmost leaves, respectively. The benefit of using a sentinel node is not only to act as a node representing "the element after the last", but also to provide additional information. In engineering practice, we often need to access the maximum and minimum values of an ordered sequence. In this way, we can make it possible in $O(1)$.

2. My tree node looks like:

```
template <class Tp, class Property = int>
struct TreeNode {
    using Node    = TreeNode<Tp>;
    using Nodeptr = Node*;

    bool is_nil;
    Property property; // for balanced tree
    Nodeptr left;
    Nodeptr right;
    Nodeptr parent;
    Tp value;

    //functions
};
```

   The problem is, we can determine if a pointer is null by `ptr == nullptr`, but why I created a new field called `is_nil`? Well, for purpose of reducing null judgements, I made all of null pointers point to the sentinel root. When I transplant trees(such as splitting a tree into two subtrees), it's impossible to make every null pointer point to a new sentinel root. Luckily, the field `is_nil` for each sentinel root is undoubtedly `true`. So according to this, it can achieve the purpose of null pointer judgment.

Finally, the figure 3 shows the structure of the tree I designed.

2.2 Implementation

This section is used to illustrate how I implement 2^{nd} point in improvements mentioned before.

As shown in figure 4, class `BinaryTree` has two template arguments. `Traits` contains meta info used in tree operations, while `MixIn` is a variable template argument representing function packages, such as `MapPack`. If there is no specific `MixIn`, `BinaryTree` will lack of some map operations.

`Traits`(can be `TreeTraits`) has 5 template arguments. `Cate`, the abbreviation for category, can be specified as `SetCate` or `MapCate`, which determines the way to extract keys in tree operations. `Alloc`, i.e. memory allocator. `Mfl`, which stands for multi flag, determines whether allows the same key is allowed to be inserted in the tree. `true` for implementing `multiset` or `multimap`. `Property` and `Derived` is used for implementing the balanced tree.

3 Experimental Procedure

I run the following code to test. By the way, you can test `bs_map<>`, `bs_multiset` and `bs_multimap`, too.

```
bs_set<int> b{ 5, 3, 6, 2, 4, 7 };
    for (auto& i : b)
        cout << i << " ";
    cout << endl;
    cout << "root data is " << b.root_data() << endl;
```

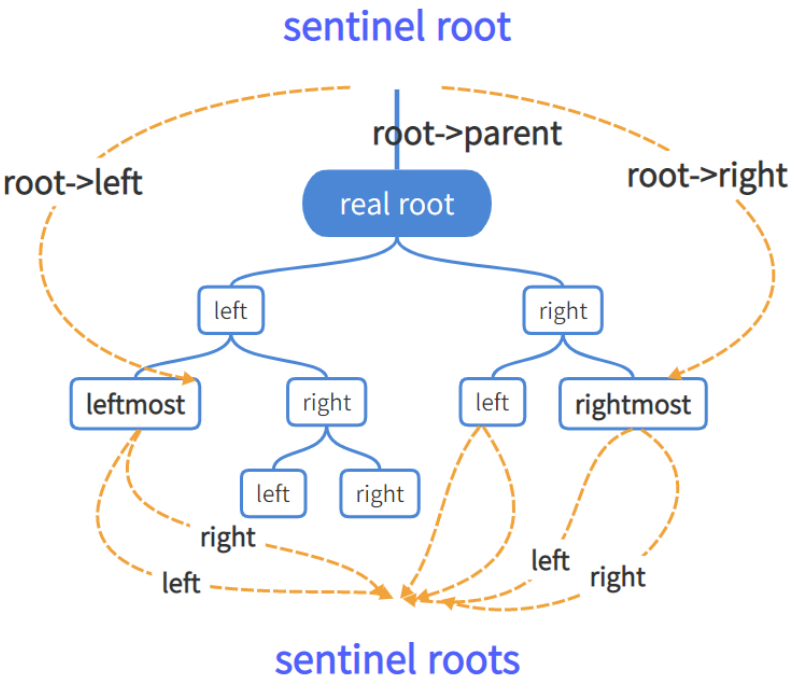

Figure 3: Tree structure

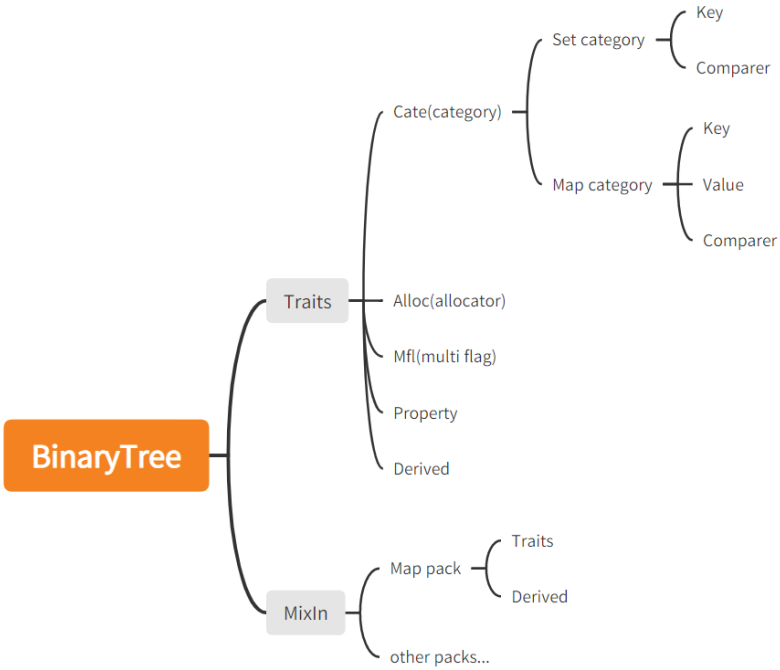

Figure 4: Template arguments

```
6       bs_set<int> l, r;
7       l = b.left_subtree();
8       r = b.right_subtree();
9       for (auto& i : l)
10          cout << i << " ";
11      cout << endl;
12      for (auto& i : r)
13          cout << i << " ";
14      cout << endl;
15      bs_set<int> a(l, 5, r);
16      for (auto& i : a)
17          cout << i << " ";
```

4 Results and Analysis

The program outputs

```
1  2 3 4 5 6 7
2  root data is 5
3  2 3 4
4  6 7
5  2 3 4 5 6 7
```

When `_PREORDER_ITERATOR_` is defined, it will travel the tree in pre-order.

```
1  5 3 2 4 6 7
2  root data is 5
3  3 2 4
4  6 7
5  5 3 2 4 6 7
```

When `_POSTORDER_ITERATOR_` is defined, it will travel the tree in post-order.

```
1  2 4 3 7 6 5
2  root data is 5
3  2 4 3
4  7 6
5  2 4 3 7 6 5
```

When `_POSTORDER_ITERATOR_` is defined, it will travel the tree in post-order.

```
1  2 4 3 7 6 5
2  root data is 5
3  2 4 3
4  7 6
5  2 4 3 7 6 5
```

When `_LEVEL_ORDER_ITERATOR_` is defined, it will travel the tree in level-order.

```
1  5 3 6 2 4 7
2  root data is 5
3  3 2 4
4  6 7
5  5 3 6 2 4 7
```

Appendix

A Scary Iterator

A.1 Background

Scary mechanism is used to solve a semantic problem. Imagine the following situation:

```
1 // the list below uses an allocator other than the default
2 list<int, my_allocator<int>> l;
3
4 // Is the following line valid? (i.e. does it compile?)
5 list<int>::iterator iter(l.begin());
```

Does it compile or not? Well, if you tried to compile that code using a pre-v11 Visual C++ compiler, it failed.

The reason you got this error is that in the last line of the code you're trying to initialize `iter` (an iterator over a list of int using the default list allocator `std::allocator<int>`) with an iterator which looks similar excepts that it corresponds to a list allocated by the alternative `my_allocator<int>`. Sad but true, these are different specializations of list of `int`s; therefore, conversions between them can't be handled by default.

From a compiler perspective there is nothing wrong here. From a practical standpoint, however, there isn't any semantic dependence between list iterators and list allocators. And, in general for all STL containers, iterators only depend (semantically speaking) on the container element type.

A.2 Definition

Based on research paper N2911 "Minimizing Dependencies within Generic Classes for Faster and Smaller Programs", the acronym SCARY describes

> Assignments and initializations that are **S**eemingly erroneous (appearing**C**onstrained by conflicting generic parameters), but **A**ctually work with the **R**ight implementation (unconstrained b**Y** the conflict due to minimized dependencies)

B Morris Traversal Algorithm

B.1 Background

The Morris traversal algorithm, also known as dynamic threaded tree, was invented by James H. Morris(also is one of inventors of pattern string matching algorithm KMP). Its basic idea is to use the null pointer of the full binary tree child node (which is similar to the idea of the threaded tree) to achieve the purpose of reducing the traversal space overhead. Compare with (non-)recursive traversal, it only costs $O(1)$ extra space instead of $O(h)$, where h denotes the height of the tree.

B.2 Principle

The following is pseudo-code of Morris traversal algorithm. Here I only use in-order traversal as an example.

Write as words. Denote the current node as `cur`.

1. If `cur` has no left child, move `cur` to the right sub-tree.
2. If `cur` has a left child, find predecessor of cur in in-order traversal of `cur`, denoted as `pre`.
   a. If the `pre` has no right child, let it point to `cur`, and `cur` moves to the left.
   b. If the right child of `pre` points to `cur`, set it to be nil, and `cur` moves to the right.
3. Repeat 1,2 until `cur` is nil.

I prepared a figure to illustrate this algorithm.

Morris Traversal Algorithm

Require: the root of tree

```
1:  function MORRIS-INORDER-TRAVEL(root)
2:      cur = root
3:      while cur ≠ nil do
4:          if cur.left == nil then
5:              cur = cur.right
6:              Visit(cur)
7:          else
8:              pre = cur.left
9:              while pre.right ≠ nil do
10:                 pre = pre.right
11:             end while
12:             if pre.right ≠ nil then
13:                 pre.right = nil
14:                 cur = cur.right
15:             else
16:                 pre.right = cur
17:                 cur = cur.left
18:             end if
19:         end if
20:     end while
21: end function
```

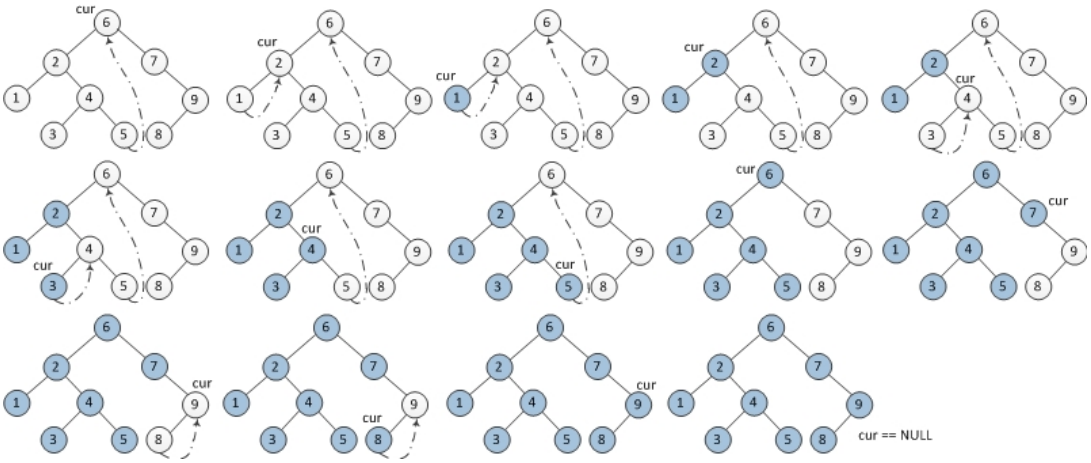

Figure 5: Example of Morris in-order traversal

B.3 Drawbacks

Its advantage is obvious, but it also have some shortcomings.

1. Modify the original data structure during the traversal process;
2. Based on the former, there needs to be strong restrictions on the data structure of the storage tree.
3. It cannot support multi-threading, or even two iterators traversing one after the other.