Binary search tree, the 'hello world' data structure

Liu Xinyu *

August 17, 2012

1 Introduction

It's typically considered that Arrays or Lists are the 'hello world' data structure. However, we'll see they are not so easy to implement actually. In some procedural settings, Arrays are the elementary representation, and it is possible to realize linked list by array (section 10.3 in [1]); While in some functional settings, Linked list are the elementary bricks to build arrays and other data structures.

Considering these factors, we start with Binary Search Tree (or BST) as the 'hello world' data structure. Jon Bentley mentioned an interesting problem in 'programming pearls' [2]. The problem is about to count the number of times each word occurs in a big text. And the solution is something like the below C++ code.

```
int main(int, char** ){
  map<string, int> dict;
  string s;
  while(cin>>s)
    ++dict[s];
  map<string, int>::iterator it=dict.begin();
  for(; it!=dict.end(); ++it)
    cout<<it->first<<": "<<it->second<<"\n";
}
```

And we can run it to produce the word counting result as the following [1].

```
$ g++ wordcount.cpp -o wordcount
$ cat bbe.txt | ./wordcount > wc.txt
```

*Liu Xinyu
Email: liuxinyu95@gmail.com

[1] This is not UNIX unique command, in Windows OS, it can be achieved by: *type bbe.txt|wordcount.exe > wc.txt*

The map provided in standard template library is a kind of balanced binary search tree with augmented data. Here we use the word in the text as the key and the number of occurrence as the augmented data. This program is fast, and it reflects the power of binary search tree. We'll introduce how to implement BST in this post and show how to solve the balancing problem in later post.

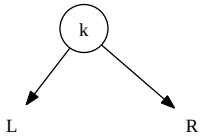
Before we dive into binary search tree. Let's first introduce about the more general binary tree.

The concept of Binary tree is a recursive definition. Binary search tree is just a special type of binary tree. The Binary tree is typically defined as the following.

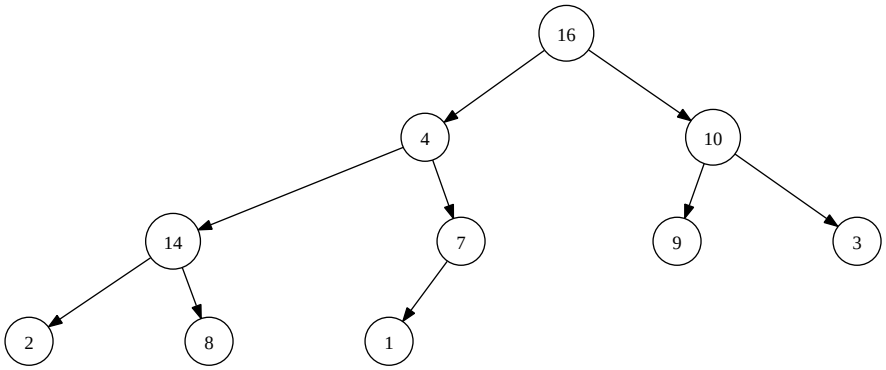
A binary tree is

- either an empty node;
- or a node contains 3 parts, a value, a left child which is a binary tree and a right child which is also a binary tree.

Figure 1 shows this concept and an example binary tree.



(a) Concept of binary tree



(b) An example binary tree

Figure 1: Binary tree concept and an example.

A binary search tree is a binary tree which satisfies the below criteria. for each node in binary search tree,

- all the values in left child tree are less than the value of of this node;
- the value of this node is less than any values in its right child tree.

Figure 2 shows an example of binary search tree. Compare with Figure 1 we can see the difference about the key ordering between them.

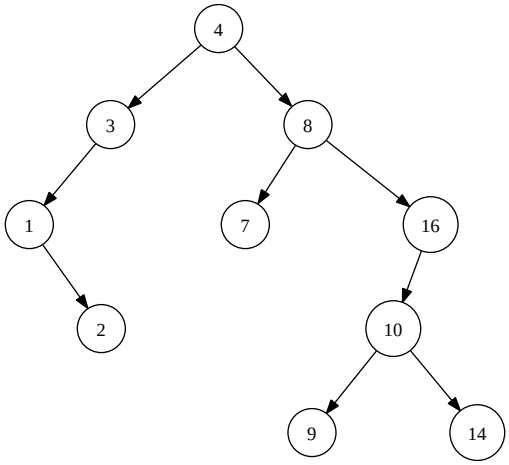


Figure 2: A Binary search tree example.

2 Data Layout

Based on the recursive definition of binary search tree, we can draw the data layout in procedural setting with pointer supported as in figure 3.

The node contains a field of key, which can be augmented with satellite data; a field contains a pointer to the left child and a field point to the right child. In order to back-track an ancestor easily, a parent field can be provided as well.

In this post, we'll ignore the satellite data for simple illustration purpose. Based on this layout, the node of binary search tree can be defined in a procedural language, such as C++ as the following.

```
template<class T>
struct node{
  node(T x):key(x), left(0), right(0), parent(0){}
  ~node(){
```

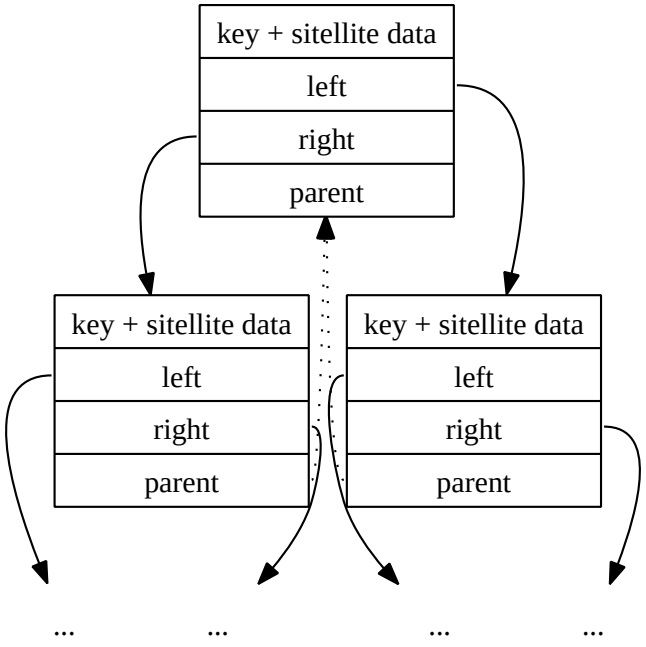


Figure 3: Layout of nodes with parent field.

```
    delete left;
    delete right;
  }

  node* left;
  node* right;
  node* parent; //parent is optional, it's helpful for succ/pred
  T key;
};
```

There is another setting, for instance in Scheme/Lisp languages, the elementary data structure is linked-list. Figure 4 shows how a binary search tree node can be built on top of linked-list.

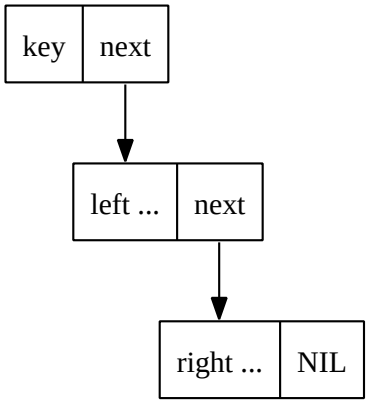


Figure 4: Binary search tree node layout on top of linked list. Where 'left...' and 'right ...' are either empty or binary search tree node composed in the same way.

Because in pure functional setting, It's hard to use pointer for back tracking the ancestors, (and typically, there is no need to do back tracking, since we can provide top-down solution in recursive way) there is not 'parent' field in such layout.

For simplified reason, we'll skip the detailed layout in the future, and only focus on the logic layout of data structures. For example, below is the definition of binary search tree node in Haskell.

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
```

3 Insertion

To insert a key k (may be along with a value in practice) to a binary search tree T, we can follow a quite straight forward way.

- If the tree is empty, then construct a leave node with key=k;
- If k is less than the key of root node, insert it to the left child;
- If k is greater than the key of root, insert it to the right child;

There is an exceptional case that if k is equal to the key of root, it means it has already existed, we can either overwrite the data, or just do nothing. For simple reason, this case is skipped in this post.

This algorithm is described recursively. It is so simple that is why we consider binary search tree is 'hello world' data structure. Formally, the algorithm can be represented with a recursive function.

$$insert(T, k) = \begin{cases} node(\phi, k, \phi) & : \quad T = \phi \\ node(insert(L, k), Key, R) & : \quad k < Key \\ node(L, Key, insert(R, k)) & : \quad otherwise \end{cases} \tag{1}$$

Where

$$\begin{aligned} L &= left(T) \\ R &= right(T) \\ Key &= key(T) \end{aligned}$$

The node function creates a new node with given left sub-tree, a key and a right sub-tree as parameters. ϕ means NIL or Empty. function $left$, $right$ and key are access functions which can get the left sub-tree, right sub-tree and the key of a node.

Translate the above functions directly to Haskell yields the following program.

```
insert::(Ord a) ⇒ Tree a → a → Tree a
insert Empty k = Node Empty k Empty
insert (Node l x r) k | k < x = Node (insert l k) x r
                      | otherwise = Node l x (insert r k)
```

This program utilized the pattern matching features provided by the language. However, even in functional settings without this feature, for instance, Scheme/Lisp, the program is still expressive.

```
(define (insert tree x)
  (cond ((null? tree) (list '() x '()))
        ((< x (key tree))
         (make-tree (insert (left tree) x)
                    (key tree)
                    (right tree)))
        ((> x (key tree))
         (make-tree (left tree)
                    (key tree)
                    (insert (right tree) x)))))
```

It is possible to turn the algorithm completely into imperative way without recursion.

```
1: function INSERT(T, k)
2:     root ← T
3:     x ← CREATE-LEAF(k)
4:     parent ← NIL
5:     while T ≠ NIL do
6:         parent ← T
7:         if k < KEY(T) then
8:             T ← LEFT(T)
9:         else
10:            T ← RIGHT(T)
11:    PARENT(x) ← parent
12:    if parent = NIL then                          ▷ tree T is empty
13:        return x
14:    else if k < KEY(parent) then
15:        LEFT(parent) ← x
16:    else
17:        RIGHT(parent) ← x
18:    return root

19: function CREATE-LEAF(k)
20:    x ← EMPTY-NODE
21:    KEY(x) ← k
22:    LEFT(x) ← NIL
23:    RIGHT(x) ← NIL
24:    PARENT(x) ← NIL
25:    return x
```

Compare with the functional algorithm, it is obviously that this one is more complex although it is fast and can handle very deep tree. A complete C++ program and a python program are available along with this post for reference.

4 Traversing

Traversing means visiting every element one by one in a binary search tree. There are 3 ways to traverse a binary tree, pre-order tree walk, in-order tree walk, and post-order tree walk. The names of these traversing methods highlight the order of when we visit the root of a binary search tree.

Since there are three parts in a tree, as left child, the root, which contains the key and satellite data, and the right child. If we denote them as $(left, current, right)$, the three traversing methods are defined as the following.

- pre-order traverse, visit $current$, then $left$, finally $right$;
- in-order traverse, visit $left$, then $current$, finally $right$;

- post-order traverse, visit $left$, then $right$, finally $current$.

Note that each visiting operation is recursive. And we see the order of visiting $current$ determines the name of the traversing method.

For the binary search tree shown in figure 2, below are the three different traversing results.

- pre-order traverse result: 4, 3, 1, 2, 8, 7, 16, 10, 9, 14;
- in-order traverse result: 1, 2, 3, 4, 7, 8, 9, 10, 14, 16;
- post-order traverse result: 2, 1, 3, 7, 9, 14, 10, 16, 8, 4;

It can be found that the in-order walk of a binary search tree outputs the elements in increase order, which is particularly helpful. The definition of binary search tree ensures this interesting property, while the proof of this fact is left as an exercise of this post.

In-order tree walk algorithm can be described as the following:

- If the tree is empty, just return;
- traverse the left child by in-order walk, then access the key, finally traverse the right child by in-order walk.

Translate the above description yields a generic map function

$$map(f, T) = \begin{cases} \phi & : \quad T = \phi \\ node(l', k', r') & : \quad otherwise \end{cases} \tag{2}$$

where

$$\begin{aligned} l' &= map(f, left(T)) \\ r' &= map(f, right(T)) \\ k' &= f(key(T)) \end{aligned}$$

If we only need access the key without create the transformed tree, we can realize this algorithm in procedural way lie the below C++ program.

```
template<class T, class F>
void in_order_walk(node<T>* t, F f){
  if(t){
    in_order_walk(t→left, f);
    f(t→value);
    in_order_walk(t→right, f);
  }
}
```

The function takes a parameter f, it can be a real function, or a function object, this program will apply f to the node by in-order tree walk.

We can simplified this algorithm one more step to define a function which turns a binary search tree to a sorted list by in-order traversing.

$$toList(T) = \begin{cases} \phi & : \quad T = \phi \\ toList(left(T)) \cup \{key(T)\} \cup toList(right(T)) & : \quad otherwise \end{cases} \tag{3}$$

Below is the Haskell program based on this definition.

```
toList::(Ord a)⇒Tree a → [a]
toList Empty = []
toList (Node l x r) = toList l ++ [x] ++ toList r
```

This provides us a method to sort a list of elements. We can first build a binary search tree from the list, then output the tree by in-order traversing. This method is called as 'tree sort'. Let's denote the list $X = \{x_1, x_2, x_3, ..., x_n\}$.

$$sort(X) = toList(fromList(X)) \tag{4}$$

And we can write it in function composition form.

$$sort = toList.fromList$$

Where function $fromList$ repeatedly insert every element to a binary search tree.

$$fromList(X) = foldL(insert, \phi, X) \tag{5}$$

It can also be written in partial application form like below.

$$fromList = foldL \quad insert \quad \phi$$

For the readers who are not familiar with folding from left, this function can also be defined recursively as the following.

$$fromList(X) = \begin{cases} \phi & : \quad X = \phi \\ insert(fromList(\{x_2, x_3, ..., x_n\}), x_1) & : \quad otherwise \end{cases}$$

We'll intense use folding function as well as the function composition and partial evaluation in the future, please refer to appendix of this book or [6] [7] and [8] for more information.

Exercise 1

- Given the in-order traverse result and pre-order traverse result, can you reconstruct the tree from these result and figure out the post-order traversing result?

  Pre-order result: 1, 2, 4, 3, 5, 6; In-order result: 4, 2, 1, 5, 3, 6; Post-order result: ?

- Write a program in your favorite language to re-construct the binary tree from pre-order result and in-order result.
- Prove why in-order walk output the elements stored in a binary search tree in increase order?
- Can you analyze the performance of tree sort with big-O notation?

5 Querying a binary search tree

There are three types of querying for binary search tree, searching a key in the tree, find the minimum or maximum element in the tree, and find the predecessor or successor of an element in the tree.

5.1 Looking up

According to the definition of binary search tree, search a key in a tree can be realized as the following.

- If the tree is empty, the searching fails;
- If the key of the root is equal to the value to be found, the search succeed. The root is returned as the result;
- If the value is less than the key of the root, search in the left child.
- Else, which means that the value is greater than the key of the root, search in the right child.

This algorithm can be described with a recursive function as below.

$$lookup(T, x) = \begin{cases} \phi & : \quad T = \phi \\ T & : \quad key(T) = x \\ lookup(left(T), x) & : \quad x < key(T) \\ lookup(right(T), x) & : \quad otherwise \end{cases} \tag{6}$$

In the real application, we may return the satellite data instead of the node as the search result. This algorithm is simple and straightforward. Here is a translation of Haskell program.

```
lookup::(Ord a)⇒ Tree a → a → Tree a
lookup Empty _ = Empty
lookup t@(Node l k r) x | k == x = t
                        | x < k = lookup l x
                        | otherwise = lookup r x
```

If the binary search tree is well balanced, which means that almost all nodes have both non-NIL left child and right child, for N elements, the search algorithm takes $O(\lg N)$ time to perform. This is not formal definition of balance. We'll show it in later post about red-black-tree. If the tree is poor balanced, the worst case takes $O(N)$ time to search for a key. If we denote the height of the tree as h, we can uniform the performance of the algorithm as $O(h)$.

The search algorithm can also be realized without using recursion in a procedural manner.

```
1: function SEARCH(T, x)
2:     while T ≠ NIL∧ KEY(T) ≠ x do
3:         if x < KEY(T) then
4:             T ← LEFT(T)
5:         else
6:             T ← RIGHT(T)
7:     return T
```

Below is the C++ program based on this algorithm.

```
template<class T>
node<T>* search(node<T>* t, T x){
  while(t && t→key!=x){
    if(x < t→key) t=t→left;
    else t=t→right;
  }
  return t;
}
```

5.2 Minimum and maximum

Minimum and maximum can be implemented from the property of binary search tree, less keys are always in left child, and greater keys are in right.

For minimum, we can continue traverse the left sub tree until it is empty. While for maximum, we traverse the right.

$$min(T) = \begin{cases} key(T) & : & left(T) = \phi \\ min(left(T)) & : & otherwise \end{cases} \tag{7}$$

$$max(T) = \begin{cases} key(T) & : & right(T) = \phi \\ max(right(T)) & : & otherwise \end{cases} \tag{8}$$

Both function bound to $O(h)$ time, where h is the height of the tree. For the balanced binary search tree, min/max are bound to $O(\lg N)$ time, while they are $O(N)$ in the worst cases.

We skip translating them to programs, It's also possible to implement them in pure procedural way without using recursion.

5.3 Successor and predecessor

The last kind of querying, to find the successor or predecessor of an element is useful when a tree is treated as a generic container and traversed by using iterator. It will be relative easier to implement if parent of a node can be accessed directly.

It seems that the functional solution is hard to be found, because there is no pointer like field linking to the parent node. One solution is to left 'breadcrumbs' when we visit the tree, and use these information to back-track or even reconstruct the whole tree. Such data structure, that contains both the tree and 'breadcrumbs' is called zipper. please refer to [**?**] for details.

However, If we consider the original purpose of providing *succ*/*pred* function, 'to traverse all the binary search tree elements one by one' as a generic container, we realize that they don't make significant sense in functional settings because we can traverse the tree in increase order by *mapT* function we defined previously.

We'll meet many problems in this series of post that they are only valid in imperative settings, and they are not meaningful problems in functional settings at all. One good example is how to delete an element in red-black-tree[3].

In this section, we'll only present the imperative algorithm for finding the successor and predecessor in a binary search tree.

When finding the successor of element x, which is the smallest one y that satisfies $y > x$, there are two cases. If the node with value x has non-NIL right child, the minimum element in right child is the answer; For example, in Figure 2, in order to find the successor of 8, we search it's right sub tree for the minimum one, which yields 9 as the result. While if node x don't have right child, we need back-track to find the closest ancestors whose left child is also ancestor of x. In Figure 2, since 2 don't have right sub tree, we go back to its parent 1. However, node 1 don't have left child, so we go back again and reach to node 3, the left child of 3, is also ancestor of 2, thus, 3 is the successor of node 2.

Based on this description, the algorithm can be given as the following.

```
1: function SUCC(x)
2:     if RIGHT(x) ≠ NIL then
3:         return MIN(RIGHT(x))
4:     else
5:         p ← PARENT(x)
6:         while p ≠ NIL and x = RIGHT(p) do
7:             x ← p
8:             p ← PARENT(p)
9:         return p
```

The predecessor case is quite similar to the successor algorithm, they are symmetrical to each other.

```
1: function PRED(x)
2:     if LEFT(x) ≠ NIL then
```

```
3:          return MAX(LEFT(x))
4:      else
5:          p ← PARENT(x)
6:          while p ≠ NIL and x = LEFT(p) do
7:              x ← p
8:              p ← PARENT(p)
9:          return p
```

Below are the Python programs based on these algorithms. They are changed a bit in while loop conditions.

```
def succ(x):
    if x.right is not None: return tree_min(x.right)
    p = x.parent
    while p is not None and p.left != x:
        x = p
        p = p.parent
    return p

def pred(x):
    if x.left is not None: return tree_max(x.left)
    p = x.parent
    while p is not None and p.right != x:
        x = p
        p = p.parent
    return p
```

Exercise 2

- Can you figure out how to iterate a tree as a generic container by using $pred()/succ()$? What's the performance of such traversing process in terms of big-O?
- A reader discussed about traversing all elements inside a range $[a, b]$. In C++, the algorithm looks like the below code:

  for_each$(m.lower\_bound(12), m.upper\_bound(26), f)$;

  Can you provide the purely function solution for this problem?

6 Deletion

Deletion is another 'imperative only' topic for binary search tree. This is because deletion mutate the tree, while in purely functional settings, we don't modify the tree after building it in most application.

However, One method of deleting element from binary search tree in purely functional way is shown in this section. It's actually reconstructing the tree but not modifying the tree.

Deletion is the most complex operation for binary search tree. this is because we must keep the BST property, that for any node, all keys in left sub tree are less than the key of this node, and they are all less than any keys in right sub tree. Deleting a node can break this property.

In this post, different with the algorithm described in [1], A simpler one from SGI STL implementation is used.[4]

To delete a node x from a tree.

- If x has no child or only one child, splice x out;
- Otherwise (x has two children), use minimum element of its right sub tree to replace x, and splice the original minimum element out.

The simplicity comes from the truth that, the minimum element is stored in a node in the right sub tree, which can't have two non-NIL children. It ends up in the trivial case, the the node can be directly splice out from the tree.

Figure 5, 6, and 7 illustrate these different cases when deleting a node from the tree.

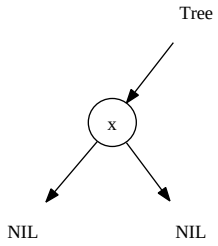


Figure 5: x can be spliced out.

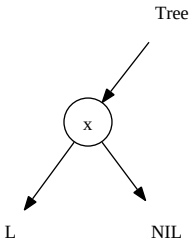
Based on this idea, the deletion can be defined as the below function.

$$delete(T, x) = \begin{cases} \phi & : & T = \phi \\ node(delete(L, x), K, R) & : & x < K \\ node(L, K, delete(R, x)) & : & x > K \\ R & : & x = K \wedge L = \phi \\ L & : & x = K \wedge R = \phi \\ node(L, y, delete(R, y)) & : & otherwise \end{cases} \tag{9}$$

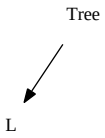
Where

$$\begin{aligned} L &= left(T) \\ R &= right(T) \\ K &= key(T) \\ y &= min(R) \end{aligned}$$

Translating the function to Haskell yields the below program.

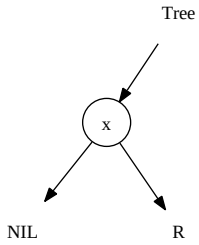


(a) Before delete x

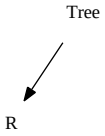


(b) After delete x

x is spliced out, and replaced by its left child.



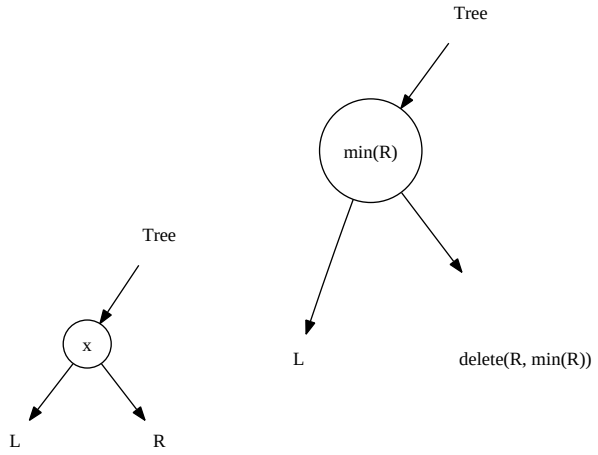
(c) Before delete x



(d) Before delete x

x is spliced out, and replaced by its right child.

Figure 6: Delete a node which has only one non-NIL child.



(a) Before delete x (b) After delete x

x is replaced by splicing the minimum element from its right child.

Figure 7: Delete a node which has both children.

```
delete::(Ord a)⇒ Tree a → a → Tree a
delete Empty _ = Empty
delete (Node l k r) x | x < k = (Node (delete l x) k r)
                      | x > k = (Node l k (delete r x))
                      -- x == k
                      | isEmpty l = r
                      | isEmpty r = l
                      | otherwise = (Node l k' (delete r k'))
                          where k' = min r
```

Function 'isEmpty' is used to test if a tree is empty (ϕ). Note that the algorithm first performs search to locate the node where the element need be deleted, after that it execute the deletion. This algorithm takes $O(h)$ time where h is the height of the tree.

It's also possible to pass the node but not the element to the algorithm for deletion. Thus the searching is no more needed.

The imperative algorithm is more complex because it need set the parent properly. The function will return the root of the result tree.

```
1: function DELETE(T, x)
2:     root ← T
3:     x' ← x                                          ▷ save x
4:     parent ← PARENT(x)
5:     if LEFT(x) = NIL then
6:         x ← RIGHT(x)
7:     else if RIGHT(x) = NIL then
```

```
8:          x ← LEFT(x)
9:      else                                          ▷ both children are non-NIL
10:         y ← MIN(RIGHT(x))
11:         KEY(x) ← KEY(y)
12:         Copy other satellite data from y to x
13:         if PARENT(y) ≠ x then                     ▷ y hasn't left sub tree
14:             LEFT(PARENT(y)) ← RIGHT(y)
15:         else                                      ▷ y is the root of right child of x
16:             RIGHT(x) ← RIGHT(y)
17:         Remove y
18:         return root
19:     if x ≠ NIL then
20:         PARENT(x) ← parent
21:     if parent = NIL then                          ▷ We are removing the root of the tree
22:         root ← x
23:     else
24:         if LEFT(parent) = x' then
25:             LEFT(parent) ← x
26:         else
27:             RIGHT(parent) ← x
28:     Remove x'
29:     return root
```

Here we assume the node to be deleted is not empty (otherwise we can simply returns the original tree). In other cases, it will first record the root of the tree, create copy pointers to x, and its parent.

If either of the children is empty, the algorithm just splice x out. If it has two non-NIL children, we first located the minimum of right child, replace the key of x to y's, copy the satellite data as well, then splice y out. Note that there is a special case that y is the root node of x's left sub tree.

Finally we need reset the stored parent if the original x has only one non-NIL child. If the parent pointer we copied before is empty, it means that we are deleting the root node, so we need return the new root. After the parent is set properly, we finally remove the old x from memory.

The relative Python program for deleting algorithm is given as below. Because Python provides GC, we needn't explicitly remove the node from the memory.

```
def tree_delete(t, x):
    if x is None:
        return t
    [root, old_x, parent] = [t, x, x.parent]
    if x.left is None:
        x = x.right
    elif x.right is None:
        x = x.left
    else:
```

```
        y = tree_min(x.right)
        x.key = y.key
        if y.parent != x:
            y.parent.left = y.right
        else:
            x.right = y.right
        return root
    if x is not None:
        x.parent = parent
    if parent is None:
        root = x
    else:
        if parent.left == old_x:
            parent.left = x
        else:
            parent.right = x
    return root
```

Because the procedure seeks minimum element, it runs in $O(h)$ time on a tree of height h.

Exercise 3

- There is a symmetrical solution for deleting a node which has two non-NIL children, to replace the element by splicing the maximum one out off the left sub-tree. Write a program to implement this solution.

7 Randomly build binary search tree

It can be found that all operations given in this post bound to $O(h)$ time for a tree of height h. The height affects the performance a lot. For a very unbalanced tree, h tends to be $O(N)$, which leads to the worst case. While for balanced tree, h close to $O(\lg N)$. We can gain the good performance.

How to make the binary search tree balanced will be discussed in next post. However, there exists a simple way. Binary search tree can be randomly built as described in [1]. Randomly building can help to avoid (decrease the possibility) unbalanced binary trees. The idea is that before building the tree, we can call a random process, to shuffle the elements.

Exercise 4

- Write a randomly building process for binary search tree.

8 Appendix

All programs are provided along with this post. They are free for downloading. We provided C, C++, Python, Haskell, and Scheme/Lisp programs as example.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. "Introduction to Algorithms, Second Edition". ISBN:0262032937. The MIT Press. 2001

[2] Jon Bentley. "Programming Pearls(2nd Edition)". Addison-Wesley Professional; 2 edition (October 7, 1999). ISBN-13: 978-0201657883

[3] Chris Okasaki. "Ten Years of Purely Functional Data Structures". http://okasaki.blogspot.com/2008/02/ten-years-of-purely-functional-data.html

[4] SGI. "Standard Template Library Programmer's Guide". http://www.sgi.com/tech/stl/

[5] http://en.literateprograms.org/Category:Binary_search_tree

[6] http://en.wikipedia.org/wiki/Foldl

[7] http://en.wikipedia.org/wiki/Function_composition

[8] http://en.wikipedia.org/wiki/Partial_application

[9] Miran Lipovaca. "Learn You a Haskell for Great Good! A Beginner's Guide". the last chapter. No Starch Press; 1 edition April 2011, 400 pp. ISBN: 978-1-59327-283-8