An Improved Inserting Algorithm to Binary Search Trees

Liyang ZHANG, Arata SUZUKI

July 29, 2020

Introduction

In Subsection 3.6.1, Quang Nhat Le has described several algorithms on binary search trees. However, the inserting algorithm has a drawback. If we insert a lot of new nodes into a binary search tree, for example, to insert 41, 42, 43, ..., 100 into Figure 3.31 , then the height of the tree will grow much faster than the logarithm of number of nodes, leading to long time for subsequenting searching, inserting or deleting operations.

Therefore, it seems good to put some reasonable restriction on the height h of the tree resulted by the inserting operation, for example,

$$h \leq \log_2 n + 2 \tag{1}$$

with n denoting the order of the graph after the inserting operation. This kind of restriction makes sure that the resulting binary search tree has a relatively low height, and that the searching, inserting and deleting operations are relatively fast.

We have created an inserting algorithm, improved based on the original one, to meet a given restriction on the height of the resulting binary search tree. The time complexity of this algorithm is still $O(h)$, as is the original one. However, we meet a difficulty in a situation shown in Figure 3 . In this case we have either to ignore the restriction on height, keeping the time complexity $O(h)$; or to reorder contents of $O(2^h)$ nodes, with a time complexity $O(n) = O(2^h)$, since there is only one element that is out of order. The improvement part is 100% original.

General Idea

The general idea of this algorithm is below.

- Try to insert the new node as a leaf, in the same way as the algorithm introduced by Quang Le.
- If the height of the resulting graph is less than or equal to the restriction, return the new graph.
- If the height of the resulting graph exceeds the restriction, do not insert the new node here. Instead, move up along the tree and search for a node with only one child, and insert the new node as the other child, or to say, at the "empty site".
- Then, rearrange the tree by comparing repeatively the parent and two children, and correcting their order. Return it after the order is corrected completely.

Figure 1 and Figure 2 are used as examples to help explaining the algorithm.

Notations

$\mathcal{N}$:= current node.
$^*\mathcal{N}$:= the content of current node.
$h(v)$:= height of node v. The height of the root node is 0.
$h_{\max}$:= the restriction on the height of the resulting tree.
In the algorithm below, ROMAN letters are the main texts or the algorithm, and *ITALIAN* letters are comments.

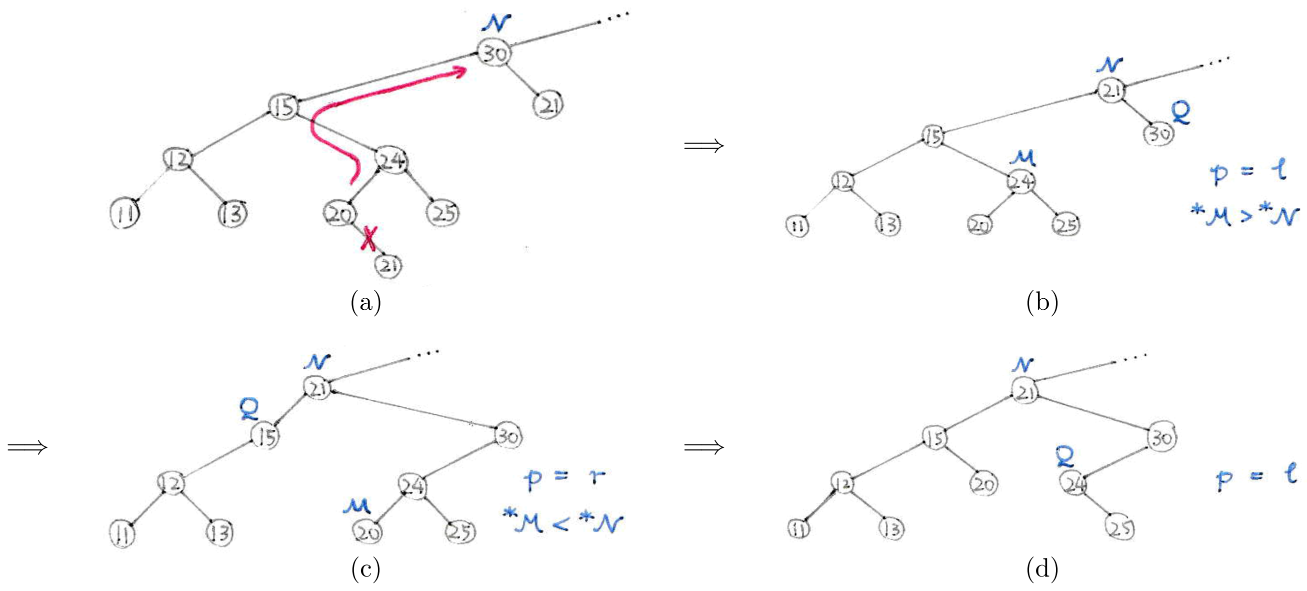


Figure 1: the first case of inserting.

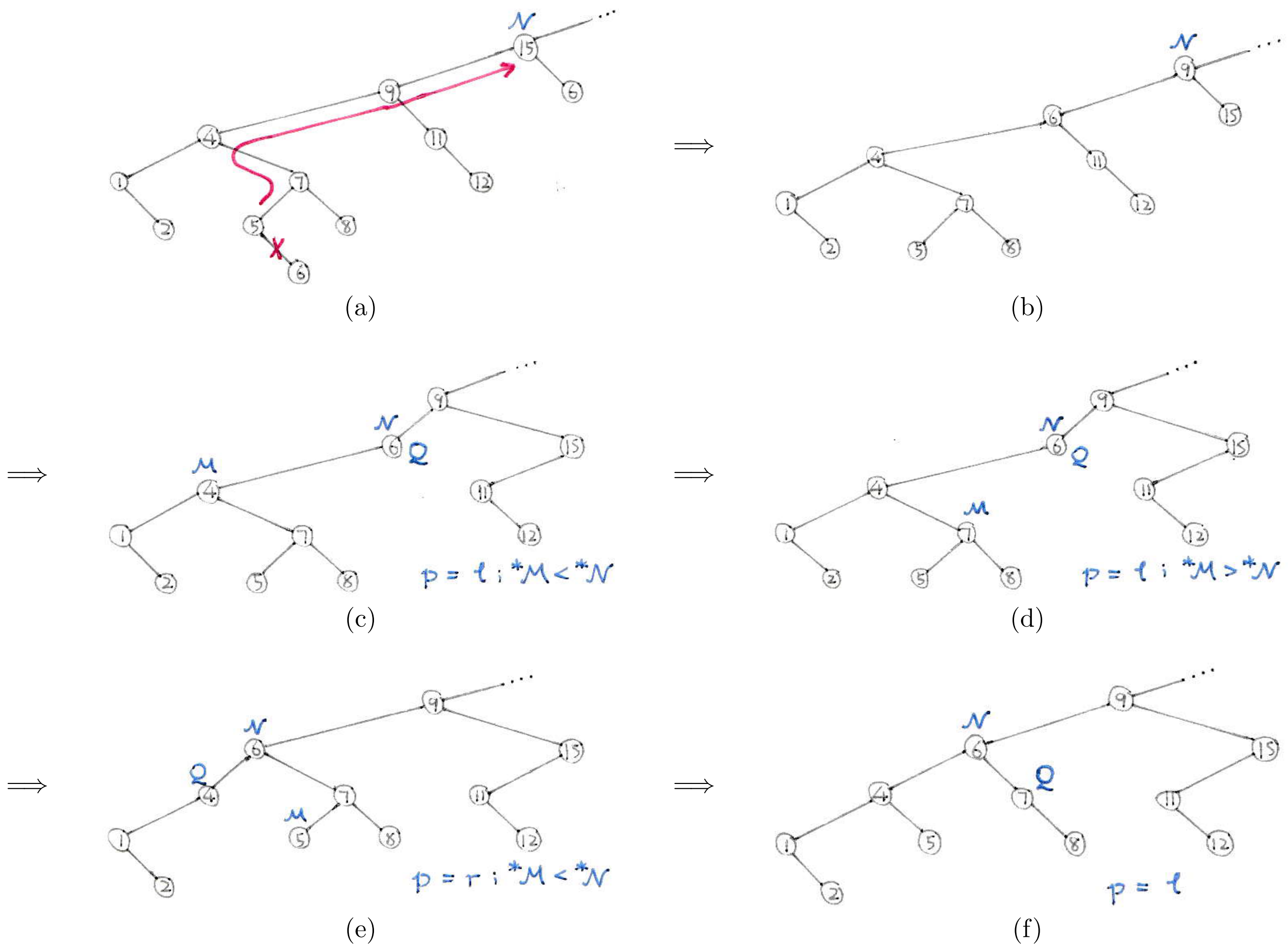


Figure 2: the second case of inserting.

Algorithm

Input: binary search tree $\mathcal{T}$ on a totally ordered set $(\mathbb{S}, \leq)$; element $x \in \mathbb{S}$ to be added into $\mathcal{T}$; restriction on the height $h_{\max}(n)$ dependent on the order n of the tree.

Output: binary search tree $\mathcal{T}_f$ resulted from the insertion.

1. Calculate $h_{\max} := h_{\max}(\text{order}(\mathcal{T}) + 1)$.
2. If $\text{order}(\mathcal{T}) \geq 2^{h_{\max}+1} - 1$, the graph is full and any insertion is impossible under this restriction.
3. Set $\mathcal{N} =$ root node.
4. While $h(\mathcal{N}) < h_{\max}$
    - If $x ==^* \mathcal{N}$
        - return $\mathcal{T}$.
    - Else if $x <^* \mathcal{N}$
        - If $\mathcal{N}$ has a left child $\mathcal{L}(\mathcal{N})$, set $\mathcal{N} = \mathcal{L}(\mathcal{N})$ and continue the loop.
        - Else, put x into a new node as the left child of $\mathcal{N}$, and return $\mathcal{T}$.
    - Else
        - If $\mathcal{N}$ has a right child $\mathcal{R}(\mathcal{N})$, set $\mathcal{N} = \mathcal{R}(\mathcal{N})$ and continue the loop.
        - Else, put x into a new node as the right child of $\mathcal{N}$, and return $\mathcal{T}$. $(*)$

   Above is the algorithm introduced by Quang Le. Below is original.

5. *When the loop above finishes before returning, we must have now* $h(\mathcal{N}) == h_{\max}$.
    - If $x ==^* \mathcal{N}$, return $\mathcal{T}$.
    - *Else, we cannot add a node as $\mathcal{N}$'s child anymore, since we have reached h_{max} already. Therefore, we add it at the closest empty space, and adjust the neiboring nodes later.*

   Now we go back to the first node we encounter with only 1 child.

6. Firstly, set $\mathcal{N} = \mathcal{P}(\mathcal{N})$ with $\mathcal{P}(\mathcal{N})$ denoting the parent of $\mathcal{N}$.
7. While $h(\mathcal{N}) \geq 0$
    - If $\mathcal{N}$ has 2 children, set $\mathcal{N} = \mathcal{P}(\mathcal{N})$.
    - Else if $\mathcal{N}$ has only 1 left child $\mathcal{L}(\mathcal{N})$
        - Create a right child of $\mathcal{N}$ called $\mathcal{R}(\mathcal{N})$, and set ${}^*\mathcal{R}(\mathcal{N}) = x$.
        - *In the downward loop before, the left child must have been chosen here, since otherwise that downward loop would have terminated and algorithm finished in $(*)$.*
          Thus, we have either ${}^*\mathcal{L}(\mathcal{N}) <^* \mathcal{R}(\mathcal{N}) <^* \mathcal{N}$ or ${}^*\mathcal{R}(\mathcal{N}) <^* \mathcal{L}(\mathcal{N}) <^* \mathcal{N}$.
        - If ${}^*\mathcal{L}(\mathcal{N}) <^* \mathcal{R}(\mathcal{N}) <^* \mathcal{N}$ *(See Figure 1 as an example.)*
            (a) Exchange ${}^*\mathcal{N}$ with ${}^*\mathcal{R}(\mathcal{N})$.
            (b) *By this construction, $\mathcal{R}(\mathcal{N})$ is a leaf, but $\mathcal{L}(\mathcal{N})$ must have children. Since $\mathcal{L}(\mathcal{N})$ is unchanged, its left subtree contain only elements smaller than ${}^*\mathcal{L}(\mathcal{N})$, and its right subtree contain only elements greater than ${}^*\mathcal{L}(\mathcal{N})$. But its right subtree may contain elements greater than ${}^*\mathcal{N}$, which leads to a problem.*
            Now we search for all such nodes and move them to be the children of $\mathcal{R}(\mathcal{N})$. Since ${}^*\mathcal{N}$ becomes ${}^*\mathcal{R}(\mathcal{N})$, elements of the right subtree of $\mathcal{L}(\mathcal{N})$ always lie between $\mathcal{L}(\mathcal{N})$ and $\mathcal{R}(\mathcal{N})$.
            (c) Create a new pointer $\mathcal{M}$, and initialize it to be $\mathcal{R}(\mathcal{L}(\mathcal{N}))$.
            $\mathcal{M}$ is the node to be examined whether to move.
            $\mathcal{N}$ is fixed, and is the root of the problematic subtree.
            (d) Create 2 new pointers $\mathcal{Q}$ and $\mathcal{Q}'$, and initialize both to be $\mathcal{R}(\mathcal{N})$.
            $\mathcal{Q}$ is the node which $\mathcal{M}$ is going to be moved to be the child of. $\mathcal{Q}'$ serves as a temporary variable.

(e) Create a flag $p \in \{l, r\}$, and initialize it to be l.
$p = l$ iff $\mathcal{M}$ is in the left subtree of $\mathcal{N}$; $p = r$ iff $\mathcal{M}$ is in the right subtree of $\mathcal{N}$.

(f) While $\mathcal{M} \neq \text{NULL}$ and $h(\mathcal{M}) \leq h_{\max}$

- If $p = l$ and ${}^*\mathcal{M} <^* \mathcal{N}$,
  $\mathcal{M}$ and its left subtree should not be moved, and we examine its right subtree.
  - Set $\mathcal{M} = \mathcal{R}(\mathcal{M})$.
- Else if $p = l$ and ${}^*\mathcal{M} >^* \mathcal{N}$,
  $\mathcal{M}$ should be at the proper place in right subtree of $\mathcal{N}$, and we examine the left subtree of $\mathcal{M}$.
  - Set $\mathcal{Q}' = \mathcal{P}(\mathcal{M})$.
  - Move the node $\mathcal{M}$ to be the left child of $\mathcal{Q}$.
  - Set $p = r$.
  - Set $\mathcal{M} = \mathcal{L}(\mathcal{M})$ and $\mathcal{Q} = \mathcal{Q}'$.
- Else if $p = r$ and ${}^*\mathcal{M} <^* \mathcal{N}$,
  $\mathcal{M}$ should be at the proper place in left subtree of $\mathcal{N}$, and we examine the right subtree of $\mathcal{M}$.
  - Set $\mathcal{Q}' = \mathcal{P}(\mathcal{M})$.
  - Move the node $\mathcal{M}$ to be the right child of $\mathcal{Q}$.
  - Set $p = l$.
  - Set $\mathcal{M} = \mathcal{R}(\mathcal{M})$ and $\mathcal{Q} = \mathcal{Q}'$.
- Else if $p = r$ and ${}^*\mathcal{M} >^* \mathcal{N}$,
  $\mathcal{M}$ and its right subtree should not be moved, and we examine its left subtree.
  - Set $\mathcal{M} = \mathcal{L}(\mathcal{M})$.
- *These are the only 4 cases.*

(g) *After this loop, everything should have been sorted out.* Return $\mathcal{T}$.

- Else if ${}^*\mathcal{R}(\mathcal{N}) <^* \mathcal{L}(\mathcal{N}) <^* \mathcal{N}$ *(See Figure 2 as an example.)*

(a) Keep the nodes, and reorder their contents such that ${}^*\mathcal{L}(\mathcal{N}) <^* \mathcal{N} <^* \mathcal{R}(\mathcal{N})$.

(b) *By this construction, $\mathcal{R}(\mathcal{N})$ is a leaf, but $\mathcal{L}(\mathcal{N})$ must have children. Since ${}^*\mathcal{L}(\mathcal{N})$ becomes ${}^*\mathcal{N}$, every element in the right subtree of $\mathcal{L}(\mathcal{N})$ is now greater than ${}^*\mathcal{N}$. And since ${}^*\mathcal{N}$ becomes ${}^*\mathcal{R}(\mathcal{N})$, every element in the right subtree of $\mathcal{L}(\mathcal{N})$ is now smaller than ${}^*\mathcal{R}(\mathcal{N})$. Therefore, we should move the whole right subtree of $\mathcal{L}(\mathcal{N})$ to be the left subtree of ${}^*\mathcal{R}(\mathcal{N})$.*

(c) Move the node $\mathcal{R}(\mathcal{L}(\mathcal{N}))$ to be the left child of $\mathcal{R}(\mathcal{N})$.

(d) *Since ${}^*\mathcal{L}(\mathcal{N})$ becomes ${}^*\mathcal{N}$, every element in the left subtree of $\mathcal{L}(\mathcal{N})$ is now smaller than ${}^*\mathcal{N}$. But since the newly added ${}^*\mathcal{R}(\mathcal{N})$ becomes ${}^*\mathcal{L}(\mathcal{N})$, the left subtree of $\mathcal{L}(\mathcal{N})$ may now contain elements greater than ${}^*\mathcal{L}(\mathcal{N})$. Therefore, we search for all such nodes and move them into the right subtree of $\mathcal{L}(\mathcal{N})$.*

(e) Create a new pointer $\mathcal{M}$, and initialize it to be $\mathcal{L}(\mathcal{L}(\mathcal{N}))$.
$\mathcal{M}$ is the node to be examined whether to move.

(f) Set $\mathcal{N} = \mathcal{L}(\mathcal{N})$.
$\mathcal{N}$ is fixed, and is the root of the problematic subtree.

(g) Create 2 new pointers $\mathcal{Q}$ and $\mathcal{Q}'$, and initialize both to be $\mathcal{N}$.
$\mathcal{Q}$ is the node which $\mathcal{M}$ is going to be moved to be the child of. $\mathcal{Q}'$ serves as a temporary variable.

(h) Create a flag $p \in \{l, r\}$, and initialize it to be l.
$p = l$ iff $\mathcal{M}$ is in the left subtree of $\mathcal{N}$; $p = r$ iff $\mathcal{M}$ is in the right subtree of $\mathcal{N}$.

(i) While $\mathcal{M} \neq \text{NULL}$ and $h(\mathcal{M}) \leq h_{\max}$

- If $p = l$ and ${}^*\mathcal{M} <^* \mathcal{N}$,
  $\mathcal{M}$ and its left subtree should not be moved, and we examine its right subtree.
  - Set $\mathcal{M} = \mathcal{R}(\mathcal{M})$.
- Else if $p = l$ and ${}^*\mathcal{M} >^* \mathcal{N}$,
  $\mathcal{M}$ should be at the proper place in right subtree of $\mathcal{N}$, and we examine the left subtree of $\mathcal{M}$.
  - Set $\mathcal{Q}' = \mathcal{P}(\mathcal{M})$.
  - If $\mathcal{Q} == \mathcal{N}$, move the node $\mathcal{M}$ to be the right child of $\mathcal{N}$.
  - Else, move the node $\mathcal{M}$ to be the left child of $\mathcal{Q}$.

- ○ Set $p = r$.
- ○ Set $\mathcal{M} = \mathcal{L}(\mathcal{M})$ and $\mathcal{Q} = \mathcal{Q}'$.

∗ Else if $p = r$ and ${}^*\mathcal{M} <^* \mathcal{N}$,
$\mathcal{M}$ should be at the proper place in left subtree of $\mathcal{N}$, and we examine the right subtree of $\mathcal{M}$.
- ○ Set $\mathcal{Q}' = \mathcal{P}(\mathcal{M})$.
- ○ If $\mathcal{Q} == \mathcal{N}$, move the node $\mathcal{M}$ to be the left child of $\mathcal{N}$.
- ○ Else, move the node $\mathcal{M}$ to be the right child of $\mathcal{Q}$.
- ○ Set $p = l$.
- ○ Set $\mathcal{M} = \mathcal{R}(\mathcal{M})$ and $\mathcal{Q} = \mathcal{Q}'$.

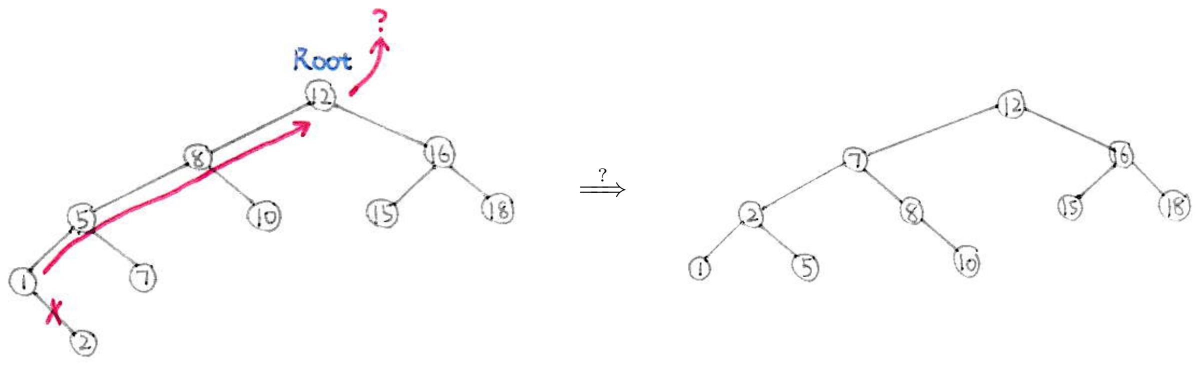
∗ Else if $p = r$ and ${}^*\mathcal{M} >^* \mathcal{N}$,
$\mathcal{M}$ and its right subtree should not be moved, and we examine its left subtree.
- ○ Set $\mathcal{M} = \mathcal{L}(\mathcal{M})$.

∗ *These are the only 4 cases.*

(j) *After this loop, everything should have been sorted out.* Return $\mathcal{T}$.

- Else, $\mathcal{N}$ *has only 1 right child* $\mathcal{R}(\mathcal{N})$. Do the same as the case of only 1 left child, while replacing every word "left" with "right", and vice versa. Then return $\mathcal{T}$.

8. If the algorithm does not return before this loop finish, it means that this algorithm has not encountered any node with only 1 child, including the root node. *(See Figure 3 as an example.)*We have not yet found a way to solve this problem within the time complexity of $O(h)$. Our choices are either to ignore the restriction, set $h_{\max} = \infty$, and carry out the algorithm (now same as the algorithm introduced by Quang Le) again; or to insert the new node at an empty place, and reorder all contents of the nodes. The time complexity of the latter choice is unfortunately $O(n) = O(2^h)$, since there is only one element that is out of order.



$\overset{?}{\Longrightarrow}$

(a) (b)

Figure 3: an example with algorithm still not found within $O(h)$.