Algorithms and Data Structures

8th Lecture: Binary Search Tree

Yutaka Watanobe, Jie Huang, Yan Pei, Wenxi Chen,
S. Semba, Deepika Saxena, Yinghu Zhou, Akila Siriweera

University of Aizu

Last Updated: 2025/01/05

Outline

- Binary Search Tree
- Querying a Binary Search Tree
- Insertion
- Deletion

Search Trees

- Search trees are data structures that support dynamic set operations, including
  - Search,
  - Minimum,
  - Successor,
  - Insert, and
  - Delete.
- Thus, a search tree can be used both as a **dictionary** and as a **priority queue**.
- Basic operations on a binary search tree take time proportional to the heigth of the tree.

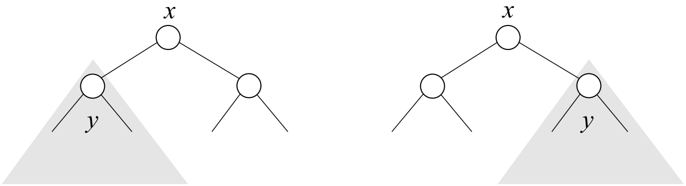
Binary Search Tree

- A binary search tree is organized in a binary tree. Such a tree can be represented by a linked data structure in which each node is an object.
- In addition to a key field and satellite data, each node contains fields **left**, **right**, and **p** that point to the nodes corresponding to its left child, its right child, and its parent, respectively.
- If a child or the parent is missing, the appropriate field contains the value NIL.
- The root node is the only node in the tree whose parent field is NIL.

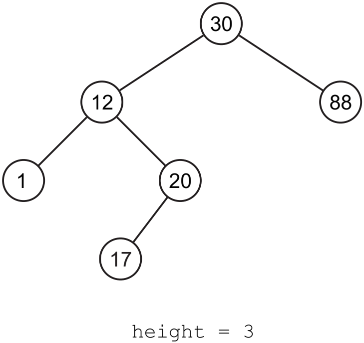
Binary Search Tree Property

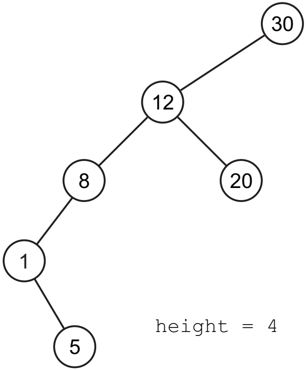
The keys in a binary search tree are always stored in such a way as to satisfy the following **binary search tree property**:

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $x.key \leq y.key$.



Binary Search Tree Property: Examples

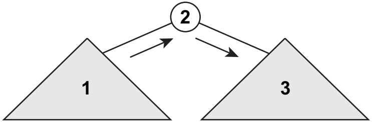




Inorder Tree Walk (1)

- The binary search tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an **inorder tree walk**.
- The algorithm prints the key of the root of a subtree between the values in its left subtree and those in its right subtree.

Inorder Tree Walk (2)




```
inorderTreeWalk(x)
    if x != NIL:
        inorderTreeWalk(x.left)
        print x.key
        inorderTreeWalk(x.right)
```

Querying a Binary Search Tree

- A common operation performed on a binary search tree is searching for a key stored in the tree.
  - Search operation
- Besides the Search operation, binary search trees can support such queries as
  - Minimum operation,
  - Maximum operation, and
  - Successor operation
- Each operation can be supported in time $O(h)$ on a binary search tree of height h.

Searching (1)

- We use the following procedure to search for a node with a given key in a binary search tree.
- Given a pointer to the root of the tree and a key *k*, treeSearch returns a pointer to a node with key *k* if one exists; otherwise, it returns NIL.

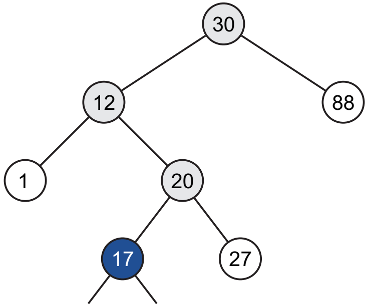
```
treeSearch(x, k)
    if x == NIL or k == x.key:
        return x
    if k < x.key:
        return treeSearch(x.left, k)
    else
        return treeSearch(x.right, k)
```

Searching (2)

- The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of treeSearch is $O(h)$, where h is the height of the tree.
- The same procedure to the treeSearch can be written iteratively by ”unrolling” the recursion into a while loop.

```
iterativeTreeSearch(x, k)
    while x != NIL and k != x.key:
        if k < x.key:
            x = x.left
        else:
            x = x.right
    return x
```

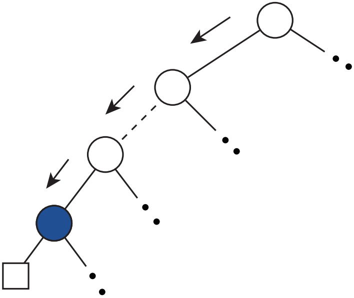
Searching: An example



treeSearch(x, 17) → 30 → 12 → 20 → 17

Minimum

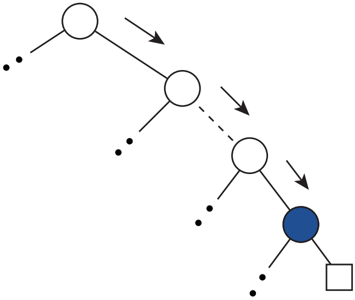
The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x.



```
treeMinimum(x)
    while x.left != NIL:
        x = x.left
    return x
```

Maximum

The pseudocode for treeMaximum is symmetric.



```
treeMaximum(x)
    while x.right != NIL:
        x = x.right
    return x
```

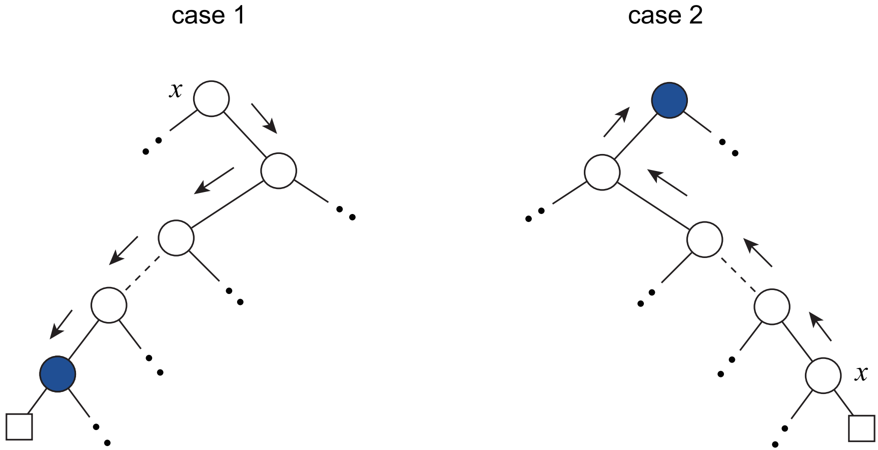
Successor (1)

- If all keys are distinct, the successor of a node x is the node with the smallest key greater than $x.key$.
- The running time of treeSuccessor on a tree of height h is $O(h)$.

```
treeSuccessor(x)
    if x.right != NIL:
        return treeMinimum(x.right) /* case 1 */

    y = x.p
    while y != NIL and x == y.right:
        x = y
        y = y.p
    return y                        /* case 2 */
```

Successor (2)



Insertion and Deletion

- The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change.
- The data structure must be modified to reflect this change, but in such a way that the **binary search tree property** continues to hold.
- As we shall see, modifying the tree to insert a new element is relatively straightforward, but handling deletion is somewhat more intricate.

Insertion

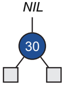
- To insert a new value v into a binary search tree T, we use the procedure treeInsert.
- The procedure is passed a node z for which $z.key = v$, $z.left = NIL$, and $z.right = NIL$.
- The procedure modifies T and some of the fields of z in such a way that z is inserted into an appropriate position in the tree.
- The procedure treeInsert runs in $O(h)$ time on a tree of height h.

The treeInsert Procedure

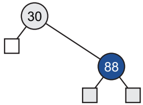
```
/* insert node z to T */
treeInsert(T, z)
    /* y is parent of x */
    y = NIL
    x = T.root
    while x != NIL:
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = x.right
    z.p = y
```

```
if y == NIL: /* Tree is empty *
    T.root = z
else if z.key < y.key:
    y.left = z
else:
    y.right = z
```

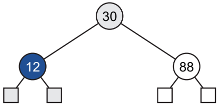
Insertion: Example



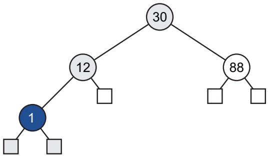
insert 30



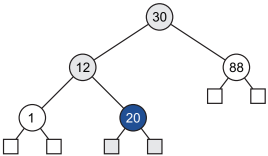
insert 88



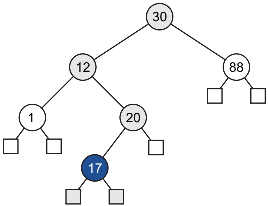
insert 12



insert 1



insert 20



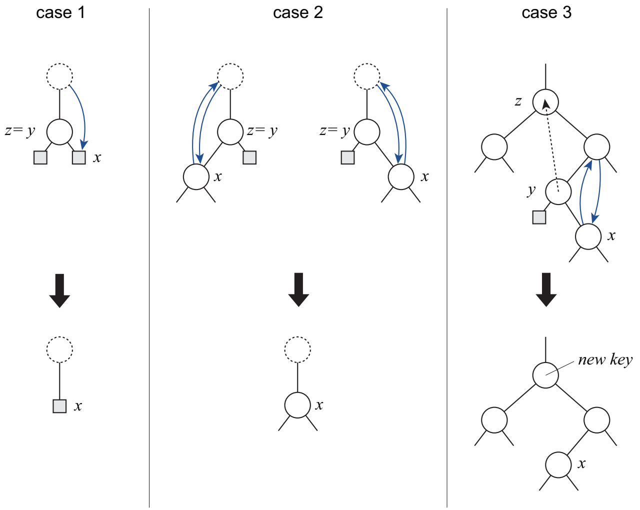
insert 17

Deletion (1)

- The procedure for deleting a given node z from a binary search tree takes as an argument a pointer to z.
- The procedure considers the three cases:
  1. If z has no children, we modify its parent $z.p$ to replace z with NIL as its child.
  2. If z has only a single child, we ”splice out” z by making a new link between its child and its parent.
  3. If z has two children, we splice out z's successor y, which has no left child and replace z's key and satellite data with y's key and satellite data.

splice (verb): to join the ends of two pieces (of rope, film etc.) so that they form one continuous piece.

Deletion (2)



The treeDelete Procedure (1)

```
/* delete node z from T */
treeDelete(T, z)
if z.left == NIL or z.right == NIL:
    y = z
else /* z have two children */
    y = treeSuccessor(z)

if y.left != NIL
    x = y.left
else
    x = y.right

if x != NIL
    x.p = y.p
```

The treeDelete Procedure (2)

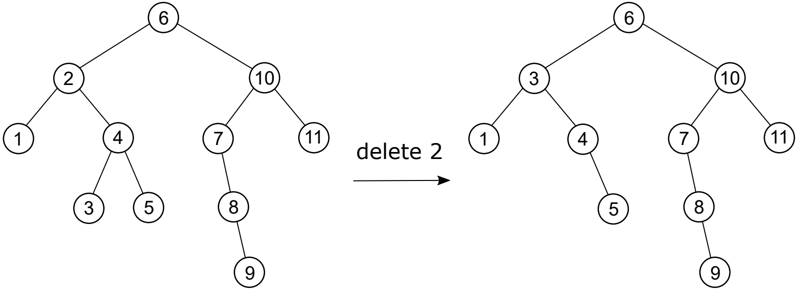
```
if y.p == NIL
    T.root = x
else if y == y.p.left
    y.p.left = x
else
    y.p.right = x

if y != z
    z.key = y.key
```

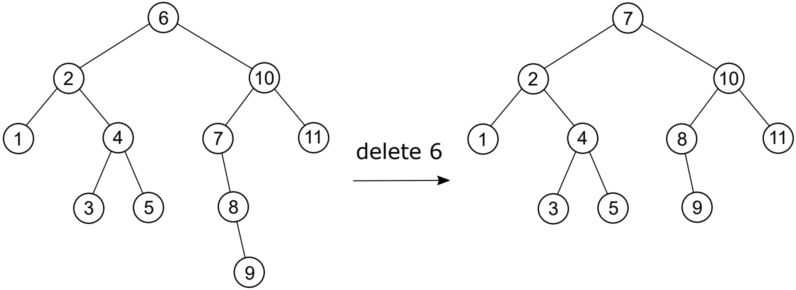
The Tree-Delete Procedure (3)

- In lines 1-4, the algorithm determines a node y to splice out. The node y is either z (if z has at most 1 child) or the successor of z (if z has two children).
- In lines 6-9, x is set to the non-NIL child of y, or to NIL if y has no children.
- In lines 11-18, node y is spliced out by modifying pointers in $y.p$ and x. Splicing out y is somewhat complicated by the need for proper handling of the boundary conditions, which occur when $x = NIL$ or when y is the root.
- In lines 20-21, if the successor of z was the node spliced out, y's key and satellite data are moved to z, overwriting the previous key and statellite data.

Deletion: Example 1



Deletion: Example 2



Complexity

- Let h be the height of the binary search tree. The computational complexity of a search operation on a binary search tree is $O(h)$.
- An insertion operation on a binary search tree can be performed in $O(1)$, but actually requires a search, so the computational complexity is $O(h)$.
- A delete operation on a binary search tree can be performed in $O(1)$, but actually requires a search, so the computational complexity is $O(h)$.

Limitations

- Let h be the height of the binary search tree and n be the number of nodes in the tree.
- The worst-case computational complexity of a search operation on a binary search tree by a naive implementation is $O(n)$.
  - If we insert elements in order $1, 2, 3, ..., n$, it is easy to imagine that the tree will become a list.
- Fortunately, there is a technique that keeps the height of the tree at $O(\log n)$ even if we repeatedly insert and delete elements.
  - Red-black tree
  - Treap
  - etc.

Reference

1. Introduction to Algorithms (third edition), Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press, 2012.