CS 253: Algorithms

Chapter 12

Binary Search Trees

* “Deletion” and “Problems”

Credit: Dr. George Bebis
Binary Search Trees

- Tree representation:
  - A linked data structure in which each node is an object

- Node representation:
  - Key field
  - Satellite data
  - Left*: pointer to left child
  - Right*: pointer to right child
  - p*: pointer to parent
    \[(p\ [root\ [T]] = NIL)\]

- Satisfies the binary-search-tree property (see the next slide)
Binary Search Tree Property

- If y is in left subtree of node x, then key [y] ≤ key [x]
- If y is in right subtree of node x, then key [y] ≥ key [x]
Binary Search Trees

- Support many dynamic set operations
  - SEARCH
  - INSERT, DELETE
  - MINIMUM, MAXIMUM
  - PREDECESSOR, SUCCESSOR

- Running time of basic operations on binary search trees
  - On average: $\Theta(lgn)$
    - The expected height of the tree is $lgn$
  - In the worst case: $\Theta(n)$
    - The tree is a linear chain of $n$ nodes
Worst Case

- If the tree is very **unbalanced**, then running time will be $O(n)$. 

![Diagram](https://via.placeholder.com/150)

- $h=n-1$  
  (no better than a linked list)
Traversing a Binary Search Tree

- **Inorder** tree walk:
  - Root is printed between the values of its left and right subtrees:
    \[
    \text{left, root, right} \quad \Rightarrow \quad \text{keys are printed in sorted order}
    \]

- **Preorder** tree walk:
  - Root printed first:
    \[
    \text{root, left, right}
    \]

- **Postorder** tree walk:
  - Root printed last:
    \[
    \text{left, right, root}
    \]

![Binary Search Tree Diagram]

- **Inorder**: 2 3 4 5 7 9
- **Preorder**: 5 3 2 4 7 9
- **Postorder**: 2 4 3 9 7 5
Traversing a Binary Search Tree

**Alg:** INORDER-TREE-WALK(x)

1. if \( x \neq \text{NIL} \)
2. then INORDER-TREE-WALK ( left \( [x] \) )
3. print key \( [x] \)
4. INORDER-TREE-WALK ( right \( [x] \) )

**E.g.:**

```
  5
 / \  
3   7
 / \  
2   5
    /  
    9
```

Output: 2 3 4 5 7 9

- Running time:
  - \( \Theta(n) \), where \( n \) is the size of the tree rooted at \( x \)
Searching for a Key

- Given a pointer to the root of a tree and a key $k$:
  - Return a pointer to a node with key $k$ if one exists
  - Otherwise return NIL

- Start at the root; trace down a path by comparing $k$ with the key of the current node $x$:
  - If the keys are equal: we have found the key
  - If $k < \text{key}[x]$ search in the left subtree of $x$
  - If $k > \text{key}[x]$ search in the right subtree of $x
Example: TREE-SEARCH

Search for key 13:

15 $\rightarrow$ 6 $\rightarrow$ 7 $\rightarrow$ 13
Searching for a Key

Alg: TREE-SEARCH(x, k)

1. if x = NIL or k = key [x]
2. then return x
3. if k < key [x]
4. then return TREE-SEARCH(left [x], k)
5. else return TREE-SEARCH(right [x], k)

Running Time:
O (h), h – the height of the tree
Finding the Minimum in a Binary Search Tree

**Goal:** find the minimum value in a BST
Following left child pointers from the root, until a NIL is encountered

**Alg:** TREE-MINIMUM(x)

1. while left \([x] \neq \text{NIL}\)
2. do \(x \leftarrow \text{left \([x]\)}\)
3. return \(x\)

Running time
\(O(h), \quad h – \text{height of tree}\)

Minimum = 2
Finding the Maximum in a Binary Search Tree

Goal: find the maximum value in a BST
Following right child pointers from the root, until a NIL is encountered

Alg: TREE-MAXIMUM(x)

1. \textbf{while} right [x] \neq \text{NIL}
2. \textbf{do} x \leftarrow \text{right}[x]
3. \textbf{return} x

Running time
$O(h)$, $h$ – height of tree

Maximum = 20
**Successor**

**Def:** \( \text{successor} \ (x) = y \), such that \( \text{key} \ [y] \) is the smallest key \( \geq \text{key} \ [x] \)

*e.g.*: \( \text{successor} \ (15) = 17 \)
\( \text{successor} \ (13) = 15 \)
\( \text{successor} \ (9) = 13 \)
\( \text{successor} \ (20) = ? \)

- **Case 1:** \( \text{right} \ (x) \) is non empty
  \( \text{successor} \ (x) \) = the minimum in \( \text{right} \ (x) \)

- **Case 2:** \( \text{right} \ (x) \) is empty
  - go up the tree until the current node is a *left child*:
    \( \text{successor} \ (x) \) is the parent of the current node
  - if you cannot go further (and you reached the root):
    \( x \) is the largest element (no successor!)
Finding the Successor

**Alg:** TREE-SUCCESSOR(x)

1. if right[x] ≠ NIL
2. then return TREE-MINIMUM(right[x])
3. y ← p[x] % parent of x
4. while y ≠ NIL and x == right[y]
5. do x ← y
6. y ← p[y]
7. return y

**Running time:**

$O(h)$, $h$ – height of the tree

**Exercise:** if $x=20$, what does this algorithm return?
**Def:** \( \text{predecessor}(x) = y \) such that key \([y]\) is the biggest key \(<\) key \([x]\)

*Example:* \( \text{predecessor}(15) = 13 \)  
\( \text{predecessor}(9) = 7 \)  
\( \text{predecessor}(7) = 6 \)

- **Case 1:** \( \text{left}^\uparrow(x) \) is non empty  
  - \( \text{predecessor}(x) = \text{the maximum in } \text{left}^\uparrow(x) \)

- **Case 2:** \( \text{left}^\uparrow(x) \) is empty  
  - go up the tree until the current node is a right child: \( \text{predecessor}(x) \) is the parent of the current node  
  - if you cannot go further (and you reached the root): \( x \) is the smallest element
Insertion

**Goal:** Insert value \( v \) into a binary search tree

**Idea:**
- If \( \text{key}[x] < v \) move to the right child of \( x \)
  - else move to the left child of \( x \)
- When \( x == \text{NIL} \), we found the correct position
- If \( v < \text{key}[y] \) insert the new node as \( y \)'s left child
  - else insert it as \( y \)'s right child

- Begining at the root, go down the tree and maintain:
  - **Pointer** \( x \): traces the downward path (current node)
  - **Pointer** \( y \): parent of \( x \) (“trailing pointer”)
Example: TREE-INSERT

Insert 13:

1. $x=\text{root}[T]$, $y=\text{NIL}$
2. $x = \text{NIL}$, $y = 15$
3. $x = \text{NIL}$, $y = 15$
4. $x = \text{NIL}$, $y = 15$
Alg: TREE-INSERT(T, z)

1. y ← NIL
2. x ← root[T]
3. while x ≠ NIL
4. do y ← x
5. if key[z] < key[x]
6. then x ← left[x]
7. else x ← right[x]
8. p[z] ← y
9. if y = NIL
10. then root[T] ← z % Tree T was empty
11. else if key[z] < key[y]
12. then left[y] ← z
13. else right[y] ← z

Running time: $O(h)$
Deletion

Goal: Delete a given node z from a binary search tree

Case 1: z has no children
- Delete z by making the parent of z point to NIL
Deletion

Case 2: z has one child
- Delete z by making the parent of z point to z’s child, instead of to z.
  And parent of z becomes the parent of z’s child.
Case 3: z has two children
- z’s successor (y) is the minimum node in z’s right subtree
- y has either no children or one right child (but no left child)
- Delete y from the tree (via Case 1 or 2)
- Replace z’s key and satellite data with y’s.
TREE-DELETE($T$, $z$)

1. \textbf{if} $\text{left}[z] = \text{NIL}$ or $\text{right}[z] = \text{NIL}$
2. \textbf{then} $y \leftarrow z$ \hspace{1cm} \% $z$ has at most one child
3. \textbf{else} $y \leftarrow \text{TREE-SUCCESSOR}(z)$ \hspace{1cm} \% $z$ has 2 children
   \hspace{1cm} \langle y \text{ will be deleted} \rangle
4. \textbf{if} $\text{left}[y] \neq \text{NIL}$
5. \textbf{then} $x \leftarrow \text{left}[y]$
6. \textbf{else} $x \leftarrow \text{right}[y]$
7. \textbf{if} $x \neq \text{NIL}$
8. \textbf{then} $p[x] \leftarrow p[y]$
<Exercise: check the correctness of the pointer movements below>

9. if p[y] = NIL
10. then root[T] ← x
11. else if y = left[p[y]]
12. then left[p[y]] ← x
13. else right[p[y]] ← x
14. if y ≠ z
15. then key[z] ← key[y]
16. copy y’s satellite data into z
17. return y

Running time: $O(h)$ due to TREE-SUCCESSOR operation
Binary Search Trees - Summary

- Operations on binary search trees:
  - SEARCH \(O(h)\)
  - PREDECESSOR \(O(h)\)
  - SUCCESSOR \(O(h)\)
  - MINIMUM \(O(h)\)
  - MAXIMUM \(O(h)\)
  - INSERT \(O(h)\)
  - DELETE \(O(h)\)

- These operations are fast if the height of the tree is small – otherwise their performance is similar to that of a linked list.
Problems

Exercise 12.1-2
What is the difference between the MAX-HEAP property and the binary search tree property?

Can the min-heap property be used to print out the keys of an $n$-node tree in sorted order in $O(n)$ time?
A: No. (sorting can not be done in $O(n)$!)

Add’l exercise:
• Can you use the heap property to design an efficient algorithm that searches for an item in a binary tree?
A: no, it will be very inefficient! (why?)
Let x be the root node of a binary search tree (BST). Write an algorithm \texttt{BSTHeight}(x) that determines the height of the tree. What would be its running time?

\textbf{Alg: BSTHeight}(x)

\begin{verbatim}
    if (x==NULL) return -1;
    else
        return \texttt{max}(BSTHeight(left[x]), BSTHeight(right[x])) + 1;
\end{verbatim}

This program should not more than O(n) time. Why?
Problems

- In a binary search tree, are the insert and delete operations *commutative*?

**Insert:**
- Try to insert 4 followed by 6, then insert 6 followed by 4

**Delete**
- Delete 5 followed by 6, then 6 followed by 5 in the following tree

```
       4
      / \
     2   6
    /   / \
   5   8   7
```

```
       4
      / \
     2   8
    /   / \
   7   7   8
```

```
       4
      / \
     2   7
```