ITS202: Algorithms and Data Structures

Advanced Data Structures

Ms. Sonam Wangmo

Gyalpozhing College of Information Technology
Royal University of Bhutan

November 16, 2020

BST Ordered Operations: Keys()

BST Traversal: Inorder Traversal

1. Print all the keys in the left subtree (which are less than the key at the root by definition of BSTs)
2. Then print the key at the root,
3. Then print all the keys in the right subtree (which are greater than the key at the root by definition of BSTs)

BST Ordered Operations: Keys()

BST Traversal: Inorder Traversal

1. Traverse Left Subtree
2. Enqueue Key
3. Traverse Right Subtree

BST Ordered Operations: Keys()

```java
public Iterable<Key> keys()
  {  return keys(min(), max());  }
public Iterable<Key> keys(Key lo, Key hi)
  {
      Queue<Key> queue = new Queue<Key>();
      keys(root, queue, lo, hi);
      return queue;
  }
private void keys(Node x, Queue<Key> queue, Key lo, Key hi)
  {
     if (x == null) return;
     int cmplo = lo.compareTo(x.key);
     int cmphi = hi.compareTo(x.key);
     if (cmplo < 0) keys(x.left, queue, lo, hi);
     if (cmplo <= 0 && cmphi >= 0) queue.enqueue(x.key);
     if (cmphi > 0) keys(x.right, queue, lo, hi);
   }
```

Figure 1: keys() method

Property: Inorder traversal of a BST yields keys in ascending order.

Analysis: How efficient are the order-based operations in BSTs ?

In a BST,all operations take time proportional to the height of the tree, in the worst case.

BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	N	$\lg N$	h
insert	N	N	h
min / max	N	1	h
floor / ceiling	N	$\lg N$	h
rank	N	$\lg N$	h
select	N	1	h
ordered iteration	$N \log N$	N	N

h = height of BST (proportional to log N if keys inserted in random order)

order of growth of running time of ordered symbol table operations

Figure 2: Summary

BST: Deletion

Predecessor and Successor Concepts

Where is the predecessor of a node in a tree, assuming all keys are distinct?

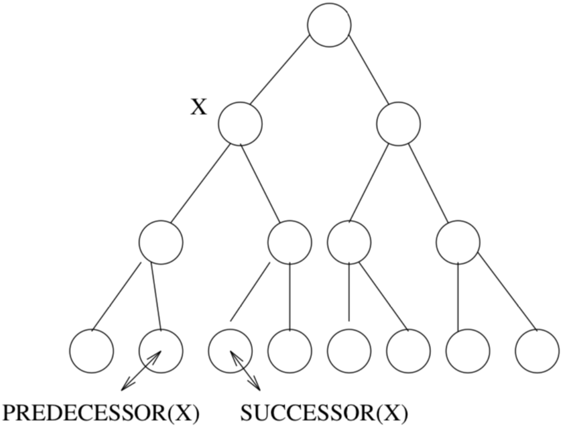

Figure 3: BST

Predecessor and Successor Concepts

If X has two children its predecessor is value in its left subtree and its successor value in its right subtree.

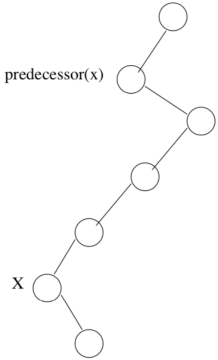

Figure 4

If it does not have a left child, a nodes predecessor is its first left

BST: Deletion

Delete the minimum/maximum

deleteMin(): Remove the key-value pair with the smallest key.

To delete the minimum key:

1. Go left until finding a node with a null left link
2. Replace that node by its right link.
3. Update subtree counts.

BST: Deletion

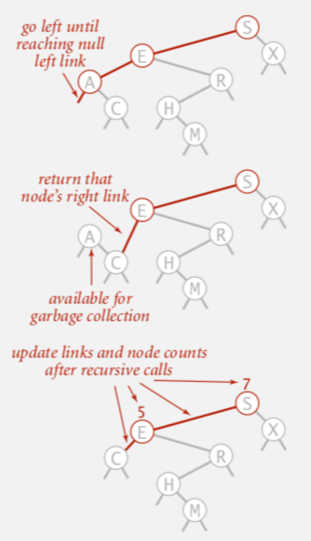

Figure 5: Deleting the minimum in a BST

BST: Deletion

```
public void deleteMin()
{  root = deleteMin(root);  }

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.count = 1 + size(x.left) + size(x.right);
   return x;
}
```

Figure 6: deleteMin() method

BST: Hibbard Deletion

To delete a node with key k: search for node t containing key k.
Case 0. [0 children] Delete t by setting parent link to null.

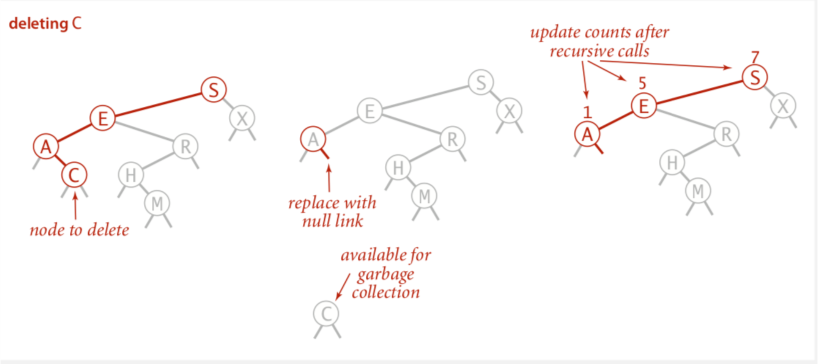

Figure 7: Deletion in a BST

BST: Hibbard Deletion

To delete a node with key k: search for node t containing key k.
Case 1. [1 child] Delete t by replacing parent link.

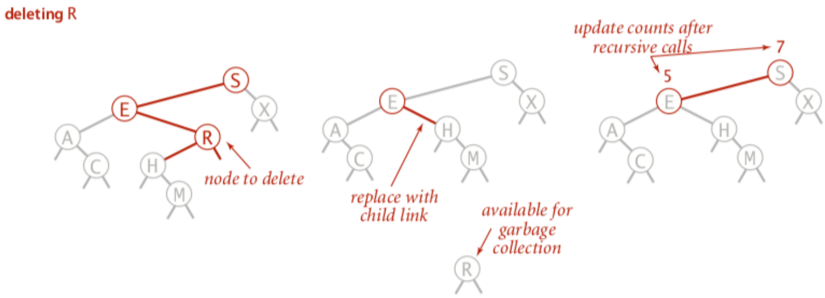

Figure 8: Deletion in BST

BST: Hibbard Deletion

To delete a node with key k: search for node t containing key k.

Case 2. [2 children]

1. Find successor x of t. <— x has no left child
2. Delete the minimum in t's right subtree. <— but don't garbage collect x
3. Put x in t's spot. <— still a BST

BST: Hibbard Deletion

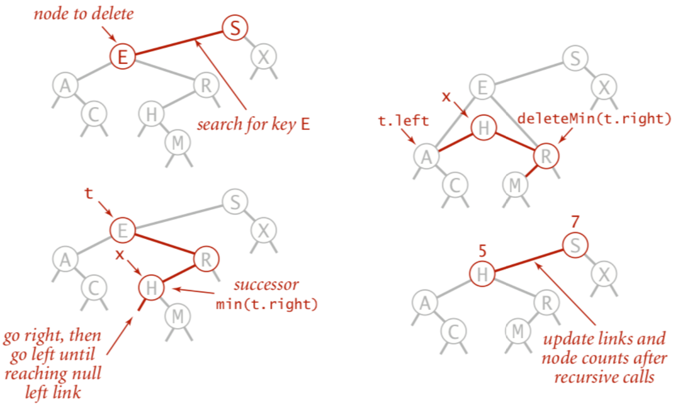

Figure 9: Deletion in BST

Hibbard deletion: Java implementation

```java
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {
      if (x.right == null) return x.left;
      if (x.left  == null) return x.right;

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.count = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

no right child

no left child

replace with successor

update subtree counts

Figure 10

BST Time Complexity

implementation	guarantee		
	search	insert	delete
sequential search (linked list)	N	N	N
binary search (ordered array)	$\lg N$	N	N
BST	N	N	N

Figure 11

WHAT IS THE BAD NEWS???