



UMC 201: Data Structures & Algorithms

Lecture 9: Static hashing; Open addressing; Binary search trees

Indian Institute of Science

Recap

- In the last lecture, we discussed hashing and showed how random hash functions can be used to implement search in $O(1)$ time in the expected (average) case.
- Such random hash functions are designed using the notion of a universal family of hash functions.

Recap

- In the last lecture, we discussed hashing and showed how random hash functions can be used to implement search in $O(1)$ time in the expected (average) case.
- Such random hash functions are designed using the notion of a universal family of hash functions.
- We also presented an explicit construction of a family of pairwise independent hash functions. A pairwise independent hash function family is universal.

Today's lecture

- In today's lecture, we'll discuss **static hashing**, which uses fixed hash functions instead of random ones.
- Although static hashing is **not** recommended due to high collision rate, it is potentially useful when the data set is small and doesn't change over time.

Today's lecture

- In today's lecture, we'll discuss **static hashing**, which uses fixed hash functions instead of random ones.
- Although static hashing is **not** recommended due to high collision rate, it is potentially useful when the data set is small and doesn't change over time.
- We'll see examples of such hash functions based on the **division method**, **multiplication method**, and the **multiply-shift method**.
- These functions are simple, fast, and easy to implement.

Today's lecture

- We'll also briefly discuss the **open addressing** method for collision resolution in a hash table.
- In open addressing, all elements occupy the hash table itself. This gives better cache performance as data sits in a continuous block of memory.

Today's lecture

- We'll also briefly discuss the **open addressing** method for collision resolution in a hash table.
- In open addressing, all elements occupy the hash table itself. This gives better cache performance as data sits in a continuous block of memory.
- With a good hash function and low load factor, open addressing can outperform chaining.
- **Recall:** Load factor is n/m , where n is the no. of keys stored and m is the no. of slots in a hash table.

Today's lecture

- We'll also briefly discuss the **open addressing** method for collision resolution in a hash table.
- In open addressing, all elements occupy the hash table itself. This gives better cache performance as data sits in a continuous block of memory.
- With a good hash function and low load factor, open addressing can outperform chaining.
- Indeed, **Python's dictionary** implementation uses a form of open addressing.

Today's lecture

- Finally, we'll discuss the **binary search tree** data structure which supports the dynamic-set operations – **Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.**
- Thus, we can use a binary search tree both as a dictionary and as a priority queue.

Static hashing

The division method

- The **division method** for creating a hash function h maps a key k into one of the m slots by taking the remainder of k divided by m .
- That is, $h(k) = k \bmod m$.

The division method

- The **division method** for creating a hash function h maps a key k into one of the m slots by taking the remainder of k divided by m .
- That is, $h(k) = k \bmod m$.
- The division method may work well when m is a prime not too close to an exact power of 2. There is **no** guarantee though that this method provides good average-case performance.

The multiplication method

- The **multiplication method** for creating a hash function h operates in two steps:
 - First, multiply the key k by a chosen constant $A \in (0,1)$ and extract the fractional part of kA .
 - Then, multiply this fractional value by m and take the floor of the result.

The multiplication method

- The **multiplication method** for creating a hash function h operates in two steps:
 - First, multiply the key k by a chosen constant $A \in (0,1)$ and extract the fractional part of kA .
 - Then, multiply this fractional value by m and take the floor of the result.
- That is, $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$.

The multiplication method

- The **multiplication method** for creating a hash function h operates in two steps:
 - First, multiply the key k by a chosen constant $A \in (0,1)$ and extract the fractional part of kA .
 - Then, multiply this fractional value by m and take the floor of the result.
- That is, $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$.
- The method has the advantage that m and A can be chosen independently.

The multiplication-shift method

- The **multiplication-shift method** is derived from the multiplication method.
- In practice, the multiplication method is best when $m = 2^\ell$, where $\ell \leq w$, the machine word size in bits.
- Assume that a key k fits into a w -bit word.

The multiplication-shift method

- The **multiplication-shift method** is derived from the multiplication method.
- In practice, the multiplication method is best when $m = 2^\ell$, where $\ell \leq w$, the machine word size in bits.
- Assume that a key k fits into a w -bit word.
- Pick a constant $A \in (0,1)$ & let $a = A2^w \in (0, 2^w)$.

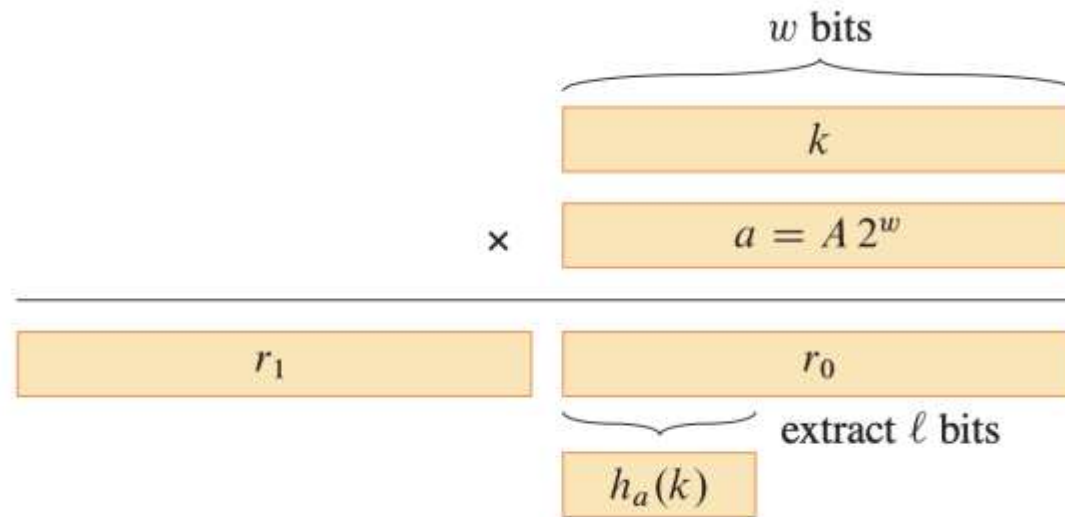
The multiplication-shift method

- The **multiplication-shift method** is derived from the multiplication method.
- In practice, the multiplication method is best when $m = 2^\ell$, where $\ell \leq w$, the machine word size in bits.
- Assume that a key k fits into a w -bit word.
- Pick a constant $A \in (0,1)$ & let $a = A2^w \in (0, 2^w)$.
- Multiply k by a & let $ka = r_1 2^w + r_0$, where r_1 & r_2 are the higher & lower order w -bit words of the product.

The multiplication-shift method

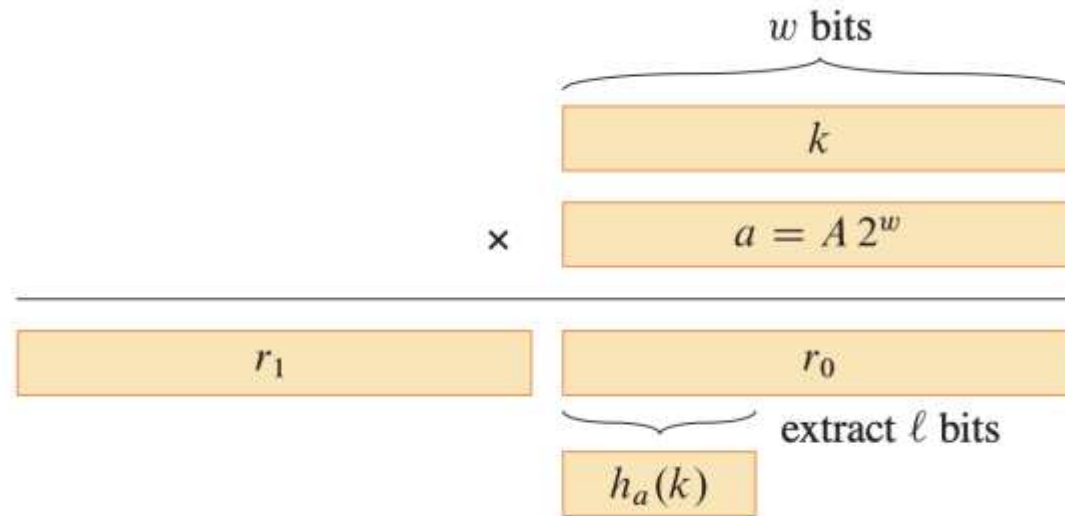
- The **multiplication-shift method** is derived from the multiplication method.
- In practice, the multiplication method is best when $m = 2^\ell$, where $\ell \leq w$, the machine word size in bits.
- Assume that a key k fits into a w -bit word.
- Pick a constant $A \in (0,1)$ & let $a = A2^w \in (0, 2^w)$.
- Multiply k by a & let $ka = r_1 2^w + r_0$, where r_1 & r_2 are the higher & lower order w -bit words of the product.
- Then, $h(k)$ is the ℓ most significant bits of r_0 .

The multiplication-shift method



- Pick a constant $A \in (0,1)$ & let $a = A2^w \in (0, 2^w)$.
- Multiply k by a & let $ka = r_1 2^w + r_0$, where r_1 & r_2 are the higher & lower order w -bit words of the product.
- Then, $h(k)$ is the ℓ most significant bits of r_0 .

The multiplication-shift method



- $h(k)$ can be computed by performing a logical right shift by $w - \ell$ bits on r_0 and ignoring r_1 .
- Thus, h can be implemented using three machine instructions: multiplication, subtraction, and shift.

Open addressing

Open addressing

- Open addressing is a method for collision resolution that, unlike chaining, does not make use of storage outside the hash table.
- Each table entry is either an element/key or *NIL*. Hence, load factor cannot exceed 1.

Open addressing

- **Open addressing** is a method for collision resolution that, unlike chaining, does not make use of storage outside the hash table.
- Each table entry is either an element/key or *NIL*. Hence, load factor cannot exceed 1.
- **Advantage.** Avoids pointers altogether. Instead, we compute a sequence of slots to be examined for insertion/search. Absence of pointer chasing makes it potentially faster than chaining for retrievals.

Open addressing: Insertion

- To perform insertion, we **probe** the hash table until we find an empty slot.
- The sequence of positions probed is computed using a hash function.

Open addressing: Insertion

- To perform insertion, we **probe** the hash table until we find an empty slot.
- The sequence of positions probed is computed using a hash function.
- The hash function takes the probe number as an additional input along with the key:

$$h: U \times \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}$$

- The **probe sequence** $\langle h(k, 0), \dots, h(k, m-1) \rangle$ need to be a permutation of $\langle 0, \dots, m-1 \rangle$ so that every table position is eventually considered.

Open addressing: Insertion

HASH-INSERT(T, k)

```
1   $i = 0$ 
2  repeat
3       $q = h(k, i)$ 
4      if  $T[q] == \text{NIL}$ 
5           $T[q] = k$ 
6          return  $q$ 
7      else  $i = i + 1$ 
8  until  $i == m$ 
9  error “hash table overflow”
```

- The procedure assumes that the input key k is not already present in the hash table.
- Computes a sequence of slots using the hash function h .
- Either returns the slot number where k is stored or flags an error if the hash table is full.

Open addressing: Search

HASH-SEARCH(T, k)

```
1   $i = 0$ 
2  repeat
3       $q = h(k, i)$ 
4      if  $T[q] == k$ 
5          return  $q$ 
6       $i = i + 1$ 
7  until  $T[q] == \text{NIL}$  or  $i == m$ 
8  return NIL
```

- The algorithm for searching a key k probes the same sequence of slots that the insertion algorithm examined when k was inserted.
- Terminates, unsuccessfully, when it finds an empty slot.

Open addressing: Deletion

- Deletion from an open-address hash table is *tricky*.
- **Issue.** When we delete a key from a slot q , it would be a mistake to mark q as NIL because we might not be able to retrieve a key k for which slot q was probed and found to be occupied when k was inserted.

Open addressing: Deletion

- Deletion from an open-address hash table is *tricky*.
- **Issue.** When we delete a key from a slot q , it would be a mistake to mark q as NIL because we might not be able to retrieve a key k for which slot q was probed and found to be occupied when k was inserted.
- **Solution (?).** One option is to mark slot q with a special value $DELETED$ instead of NIL . But then, the insert procedure should be able to treat such a slot as empty and store a new key there.

Open addressing: Deletion

- Deletion from an open-address hash table is *tricky*.
- **Issue.** When we delete a key from a slot q , it would be a mistake to mark q as NIL because we might not be able to retrieve a key k for which slot q was probed and found to be occupied when k was inserted.
- **Solution (?).** Another option is to move other keys to newly vacated slots, but this could be time consuming.

Open addressing: Deletion

- Deletion from an open-address hash table is tricky.
- **Issue.** When we delete a key from a slot q , it would be a mistake to mark q as NIL because we might not be able to retrieve a key k for which slot q was probed and found to be occupied when k was inserted.
- **Bottomline.** Open-address hash tables are not deletion-friendly.

Open addressing: Hash functions

- Double hashing offers one of the best methods available for open addressing.
- Double hashing uses a hash function of the form:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m ,$$

where h_1 and h_2 are auxiliary hash functions.

Open addressing: Hash functions

- Double hashing offers one of the best methods available for open addressing.

- Double hashing uses a hash function of the form:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m ,$$

where h_1 and h_2 are auxiliary hash functions.

- In order for the entire hash table to be searched, $h_2(k)$ must be coprime to m . One way to ensure this is to let m be a power of 2 and to design h_2 so that it always produces an odd number.

Open addressing: Hash functions

- Double hashing offers one of the best methods available for open addressing.

- Double hashing uses a hash function of the form:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m ,$$

where h_1 and h_2 are auxiliary hash functions.

- A special case of double hashing is linear probing. In this case, $h_2(k) = 1$. Linear probing excels for hierarchical memory models, as successive probes are usually to the same cache block of memory.

Open addressing: Hash functions

- Double hashing offers one of the best methods available for open addressing.

- Double hashing uses a hash function of the form:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m ,$$

where h_1 and h_2 are auxiliary hash functions.

- **Remark.** We skip a formal analysis of open addressing here as such an analysis assumes an ideal scenario: the probe sequence for each key is equally likely to be any of the $m!$ permutations of $\langle 0, \dots, m - 1 \rangle$.

Open addressing: Hash functions

- Double hashing offers one of the best methods available for open addressing.

- Double hashing uses a hash function of the form:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m ,$$

where h_1 and h_2 are auxiliary hash functions.

- **Remark.** Double hashing does not meet this assumption. It cannot generate more than m^2 different probe sequences for any key.

Open addressing vs. Chaining

Chaining	Open addressing
Uses pointers to address collisions	No pointer chasing and hence cache friendly
Handles deletions efficiently	Bad for frequent deletions
Good for high load factor	Good for low load factor
More common in standard general-purpose libraries (like Java's HashMap)	More common in high-performance systems (like Python's dict)

Binary search trees

Binary search trees

- The **binary search tree** (BST) data structure supports the dynamic-set operations – **Search**, **Minimum**, **Maximum**, **Predecessor**, **Successor**, **Insert**, and **Delete**.
- We'll see that these operations on a BST take time proportional to the height of the tree.

Binary search trees

- The **binary search tree** (BST) data structure supports the dynamic-set operations – **Search**, **Minimum**, **Maximum**, **Predecessor**, **Successor**, **Insert**, and **Delete**.
- We'll see that these operations on a BST take time proportional to the height of the tree.
- For a complete binary tree with n nodes, such operations run in $\Theta(\log n)$ time. However, if the tree is a linear chain, the same operations take $O(n)$ time.

Binary search trees

- The **binary search tree** (BST) data structure supports the dynamic-set operations – **Search**, **Minimum**, **Maximum**, **Predecessor**, **Successor**, **Insert**, and **Delete**.
- We'll see that these operations on a BST take time proportional to the height of the tree.
- For a complete binary tree with n nodes, such operations run in $\Theta(\log n)$ time. However, if the tree is a linear chain, the same operations take $O(n)$ time.
- Later, we'll see how **red-black trees** (a variation of BSTs) guarantee a height of $O(\log n)$.

The BST data structure

- A BST is organized in a binary tree, which is represented as a linked data structure.
- Each node of the tree contains a key (and possibly other satellite data), and three attributes: left (pointer to the left child), right (pointer to the right child) and p (pointer to the parent).

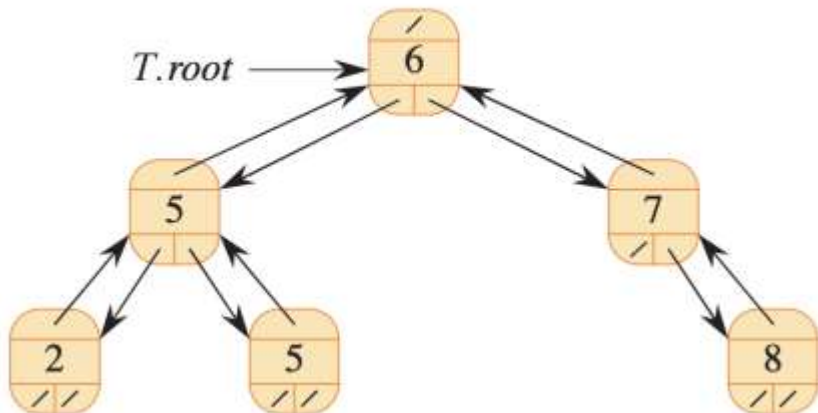
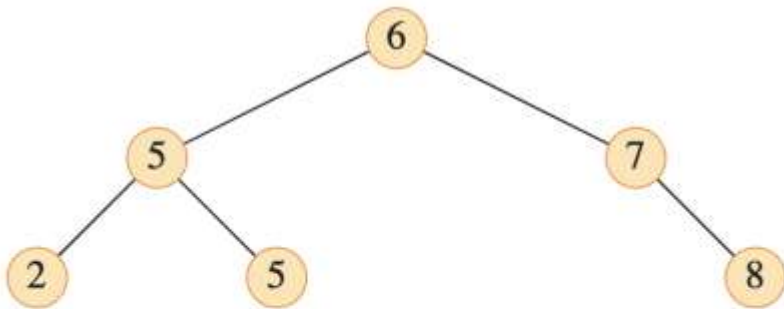
The BST data structure

- A BST is organized in a binary tree, which is represented as a linked data structure.
- Each node of the tree contains a key (and possibly other satellite data), and three attributes: left (pointer to the left child), right (pointer to the right child) and p (pointer to the parent).
- If a child or a parent is missing, the corresponding attribute contains the value NIL. The tree itself has an attribute root that points to the root node, or NIL, if the tree is empty.

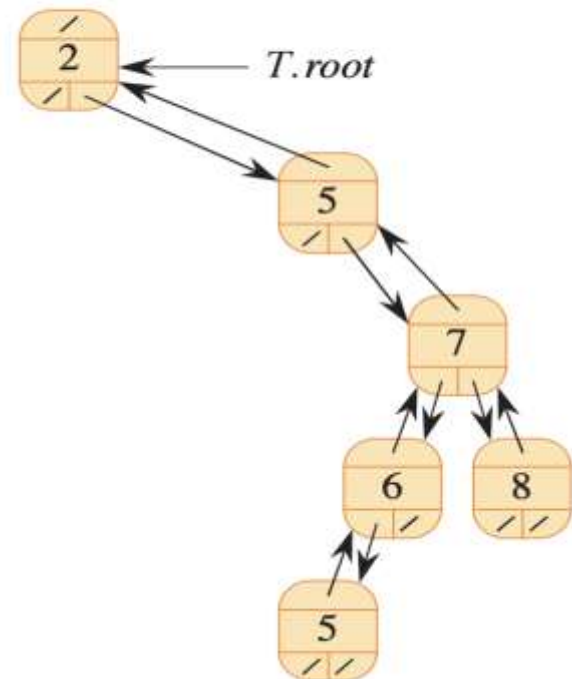
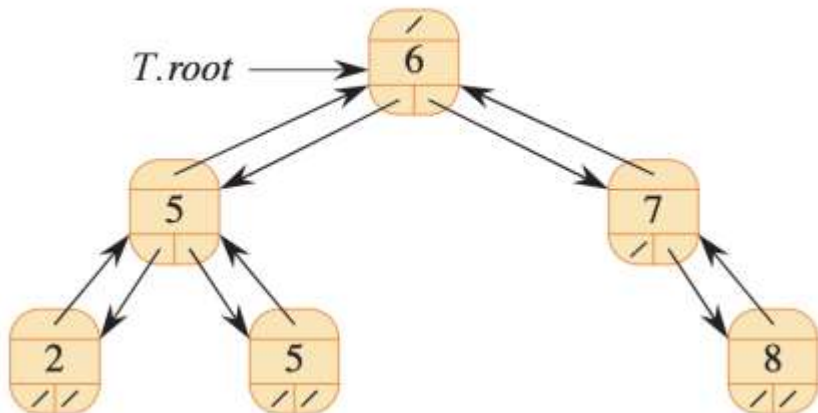
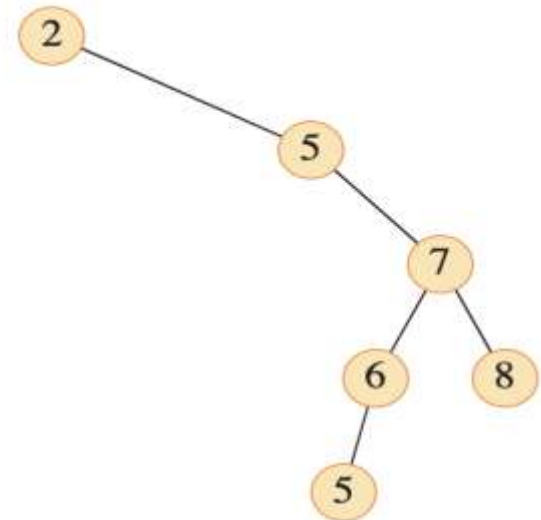
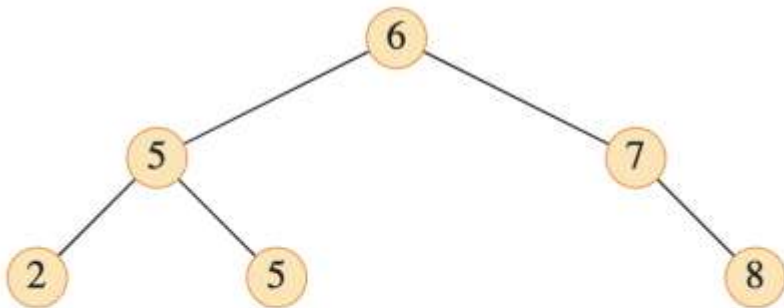
The BST property

- The keys in a BST are always stored in such a way as to satisfy the **binary-search-tree property**.
- **BST property:** Let x be a node in a BST. 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 $y.key \geq x.key$.

BST examples



BST examples



BST: Traversal and Sorting

- Owing to the BST property, we can output all the keys in a BST in sorted order by a simple recursive algorithm called **inorder walk/traversal** in $\Theta(n)$ time.

BST: Traversal and Sorting

- Owing to the BST property, we can output all the keys in a BST in sorted order by a simple recursive algorithm called **inorder walk/traversal** in $\Theta(n)$ time.

INORDER-TREE-WALK(x)

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

The algorithm is so named because it prints the key of the root of a subtree between printing the keys in its left subtree and printing those in its right subtree.

BST: Traversal and Sorting

- Owing to the BST property, we can output all the keys in a BST in sorted order by a simple recursive algorithm called **inorder walk/traversal** in $\Theta(n)$ time.

INORDER-TREE-WALK(x)

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

The correctness of using inorder walk for sorting follows from the BST property. We can define **preorder walk** and **postorder walk** similarly.

BST: Traversal and Sorting

- Owing to the BST property, we can output all the keys in a BST in sorted order by a simple recursive algorithm called **inorder walk/traversal** in $\Theta(n)$ time.

INORDER-TREE-WALK(x)

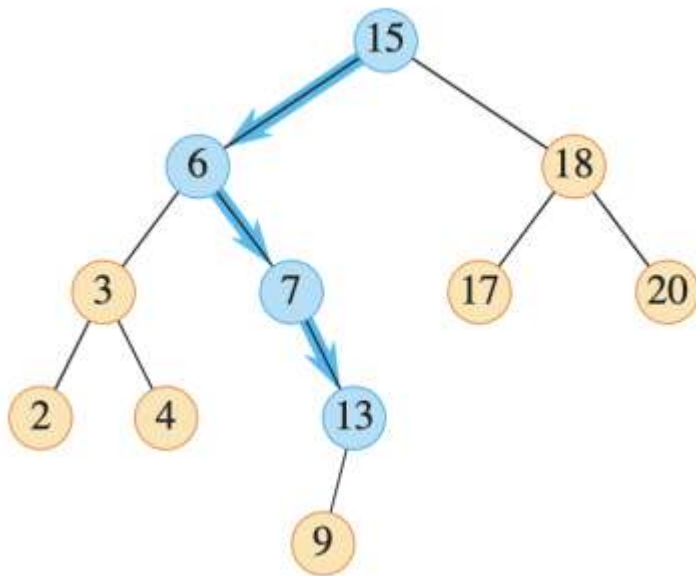
```
1  if  $x \neq \text{NIL}$ 
2      INORDER-TREE-WALK( $x.\text{left}$ )
3      print  $x.\text{key}$ 
4      INORDER-TREE-WALK( $x.\text{right}$ )
```

The correctness of using inorder walk for sorting follows from the BST property. We can define **preorder walk** and **postorder walk** similarly.

- Obs.** If x is the root of an n -node subtree, then the call **INORDER-TREE-WALK(x)** takes $\Theta(n)$ time.

BST: Search

- The search for a node with a given key in a BST begins at the root and traces a simple path downward.



In this example, a search for the node with key **13** traces the **blue** path from the root.

BST: Search

- The search for a node with a given key in a BST begins at the root and traces a simple path downward.

TREE-SEARCH(x, k)

```
1  if  $x == \text{NIL}$  or  $k == x.\text{key}$ 
2      return  $x$ 
3  if  $k < x.\text{key}$ 
4      return TREE-SEARCH( $x.\text{left}, k$ )
5  else return TREE-SEARCH( $x.\text{right}, k$ )
```

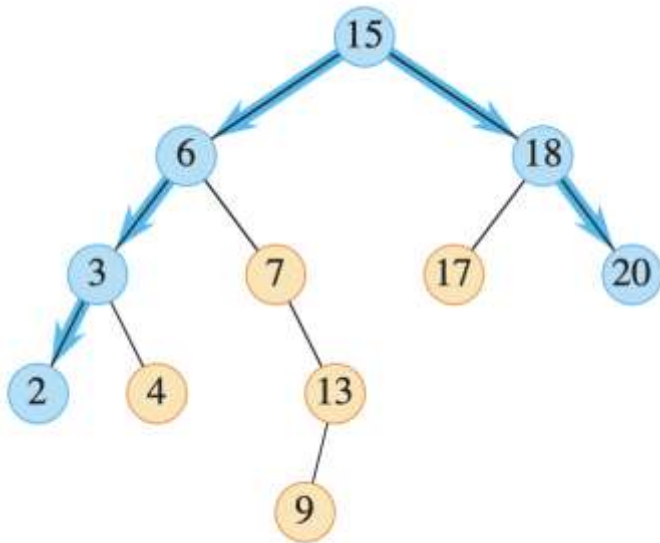
ITERATIVE-TREE-SEARCH(x, k)

```
1  while  $x \neq \text{NIL}$  and  $k \neq x.\text{key}$ 
2      if  $k < x.\text{key}$ 
3           $x = x.\text{left}$ 
4      else  $x = x.\text{right}$ 
5  return  $x$ 
```

- Although the time complexity of both procedures is $O(h)$, where h is the height of the tree, the iterative procedure is generally more efficient.

BST: Minimum and Maximum

- To find an element in a BST whose key is the minimum (maximum), just follow the left (right) child pointers from the root until we encounter a *NIL*.



In this example, the minimum key is **2** and the maximum key is **20**.

BST: Minimum and Maximum

- To find an element in a BST whose key is the minimum (maximum), just follow the left (right) child pointers from the root until we encounter a *NIL*.

TREE-MINIMUM(x)

```
1  while  $x.left \neq \text{NIL}$ 
2       $x = x.left$ 
3  return  $x$ 
```

TREE-MAXIMUM(x)

```
1  while  $x.right \neq \text{NIL}$ 
2       $x = x.right$ 
3  return  $x$ 
```

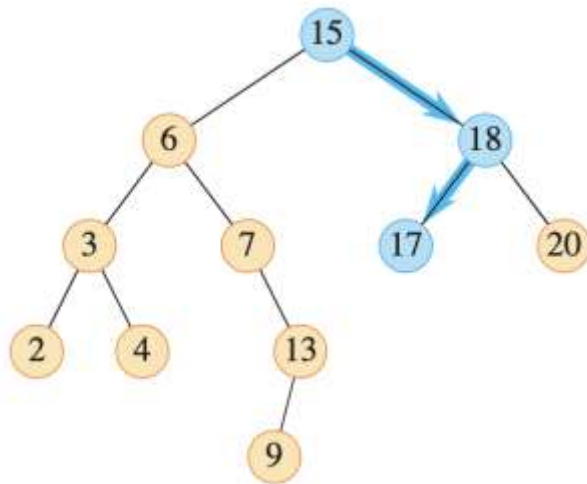
- The correctness of the above procedures follows from the BST property. The time complexity is $O(h)$.

BST: Successor and Predecessor

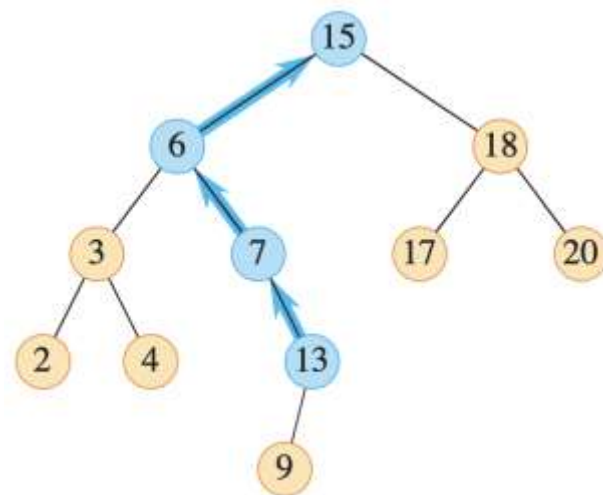
- The **successor** of a node is the next node visited in an inorder tree walk. Predecessor is defined likewise.
- If all keys are distinct, then successor of a node x is the node with the smallest key greater than $x.key$.

BST: Successor and Predecessor

- The **successor** of a node is the next node visited in an inorder tree walk. Predecessor is defined likewise.
- If all keys are distinct, then successor of a node x is the node with the smallest key greater than x . *key*.



The successor of the node with key **15** is the node with key **17**.



The successor of the node with key **13** is the node with key **15**.

BST: Successor and Predecessor

- The **successor** of a node is the next node visited in an inorder tree walk. Predecessor is defined likewise.

TREE-SUCCESSOR(x)

```
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ ) // leftmost node in right subtree
3  else // find the lowest ancestor of  $x$  whose left child is an ancestor of  $x$ 
4       $y = x.p$ 
5      while  $y \neq \text{NIL}$  and  $x == y.right$ 
6           $x = y$ 
7           $y = y.p$ 
8  return  $y$ 
```

- If the right subtree of node x is nonempty, then the successor of x is the leftmost node in x 's right subtree.

BST: Successor and Predecessor

- The **successor** of a node is the next node visited in an inorder tree walk. Predecessor is defined likewise.

```
TREE-SUCCESSOR( $x$ )
```

```
1  if  $x.right \neq \text{NIL}$   
2      return TREE-MINIMUM( $x.right$ ) // leftmost node in right subtree  
3  else // find the lowest ancestor of  $x$  whose left child is an ancestor of  $x$   
4       $y = x.p$   
5      while  $y \neq \text{NIL}$  and  $x == y.right$   
6           $x = y$   
7           $y = y.p$   
8  return  $y$ 
```

- Otherwise, the successor of x is the lowest ancestor of x whose left child is also an ancestor of x .

BST: Successor and Predecessor

- The **successor** of a node is the next node visited in an inorder tree walk. Predecessor is defined likewise.

```

TREE-SUCCESSOR( $x$ )
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ ) // leftmost node in right subtree
3  else // find the lowest ancestor of  $x$  whose left child is an ancestor of  $x$ 
4       $y = x.p$ 
5      while  $y \neq \text{NIL}$  and  $x == y.right$ 
6           $x = y$ 
7           $y = y.p$ 
8  return  $y$ 

```

- The time complexity of the above procedure is $O(h)$.

BST: Query complexity

- **Theorem.** The dynamic-set operations **Search**, **Minimum**, **Maximum**, **Successor**, and **Predecessor** can be implemented so that each one runs in $O(h)$ time on a binary tree of height h .
- Thus, if the height is $O(\log n)$, then each of these operations takes $O(\log n)$ time.

BST: Insertion and Deletion

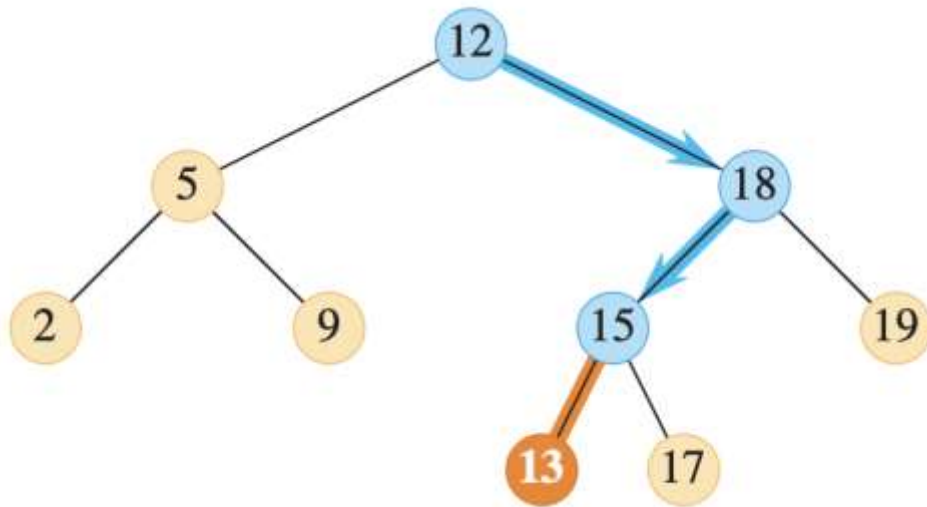
- The operations of insertion and deletion cause the dynamic set represented by a BST to change.
- The BST data structure must be modified to reflect this change, but in a way that the Binary-Search-Tree property continues to hold.
- Let us see how to modify the BST data structure for insertions and deletions.

BST: Insertion

- To insert a node z for which $z.key$ has already been filled in, $z.left = NIL$ and $z.right = NIL$, we use the following strategy:
 - Begin at the root of the tree and trace a simple path downward by comparing the key of the **current node** in the path with $z.key$ until we find a NIL that can be replaced with z .

BST: Insertion

- To insert a node z for which $z.key$ has already been filled in, $z.left = NIL$ and $z.right = NIL$, we use the following strategy:



Inserting a node with key 13 into a BST.

BST: Insertion

TREE-INSERT(T, z)

```
1   $x = T.root$            // node being compared with  $z$ 
2   $y = NIL$               //  $y$  will be parent of  $z$ 
3  while  $x \neq NIL$       // descend until reaching a leaf
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$               // found the location—insert  $z$  with parent  $y$ 
9  if  $y == NIL$ 
10      $T.root = z$         // tree  $T$  was empty
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 
```

Comparing the key of the **current node** x with $z.key$ to pick an appropriate path to descend down.

Making z a child of y .

- The procedure maintains the **trailing pointer** y as the parent of the current node x . Sets $z.p = y$ in Step 8.

BST: Insertion

TREE-INSERT(T, z)

```
1   $x = T.root$            // node being compared with  $z$ 
2   $y = NIL$               //  $y$  will be parent of  $z$ 
3  while  $x \neq NIL$       // descend until reaching a leaf
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$               // found the location—insert  $z$  with parent  $y$ 
9  if  $y == NIL$ 
10      $T.root = z$         // tree  $T$  was empty
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 
```

Comparing the key of the **current node** x with $z.key$ to pick an appropriate path to descend down.

Making z a child of y .

- **Correctness.** Follows directly from the BST property.

BST: Insertion

TREE-INSERT(T, z)

```
1   $x = T.root$            // node being compared with  $z$ 
2   $y = NIL$               //  $y$  will be parent of  $z$ 
3  while  $x \neq NIL$       // descend until reaching a leaf
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$               // found the location—insert  $z$  with parent  $y$ 
9  if  $y == NIL$ 
10      $T.root = z$        // tree  $T$  was empty
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 
```

Comparing the key of the **current node** x with $z.key$ to pick an appropriate path to descend down.

Making z a child of y .

- **Time complexity.** The procedure runs in $O(h)$ time on a tree of height h .

BST: Deletion

- The overall strategy for deleting a node z from a BST has two cases:
 - Case 1. z has at most one child.
 - Case 2. z has two children.
 - ❖ Case 2a. The successor of z is **not** z 's right child.
 - ❖ Case 2b. The successor of z is z 's right child.
- Intuitively, in Case 2, we should “replace” z by the successor (or the predecessor) of z in the tree.

BST: Deletion

- The overall strategy for deleting a node z from a BST has two cases:
 - Case 1. z has at most one child.
 - Case 2. z has two children.
 - ❖ Case 2a. The successor of z is **not** z 's right child.
 - ❖ Case 2b. The successor of z is z 's right child.
- To simplify the presentation we'll make use of an intermediate procedure, namely **TRANSPLANT**.

BST: Transplant

- The procedure below replaces the subtree rooted at node u with the subtree rooted at node v .

TRANSPLANT(T, u, v)

```
1  if  $u.p == \text{NIL}$ 
2       $T.root = v$ 
3  elseif  $u == u.p.left$ 
4       $u.p.left = v$ 
5  else  $u.p.right = v$ 
6  if  $v \neq \text{NIL}$ 
7       $v.p = u.p$ 
```

- u 's parent ends up having v as the appropriate child.
- Node u 's parent becomes node v 's parent.
- Time complexity is $O(1)$.

BST: Transplant

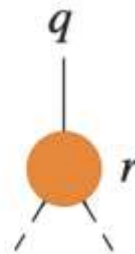
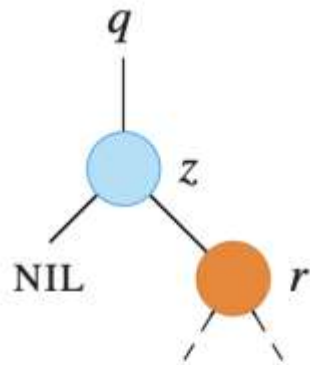
- The procedure below replaces the subtree rooted at node u with the subtree rooted at node v .

```
TRANSPLANT( $T, u, v$ )
1  if  $u.p == \text{NIL}$ 
2       $T.root = v$ 
3  elseif  $u == u.p.left$ 
4       $u.p.left = v$ 
5  else  $u.p.right = v$ 
6  if  $v \neq \text{NIL}$ 
7       $v.p = u.p$ 
```

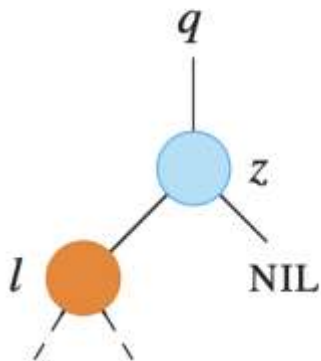


- u 's parent ends up having v as the appropriate child.
- Node u 's parent becomes node v 's parent.
- Time complexity is $O(1)$.

BST: Deletion Case I



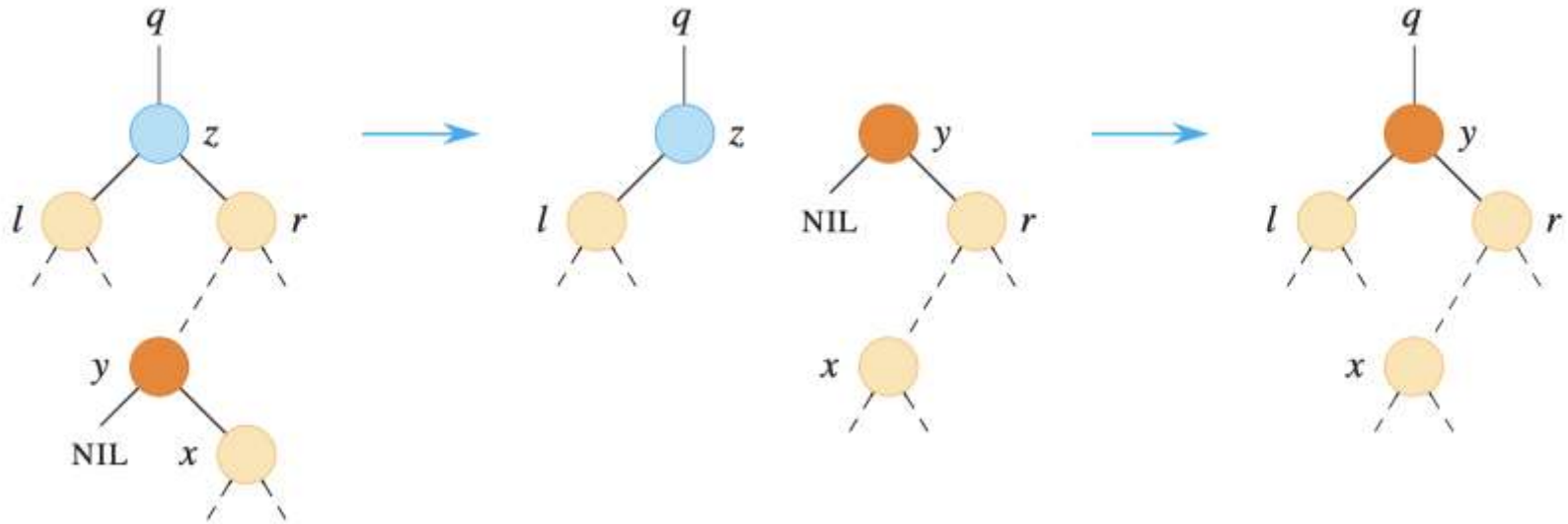
$\text{Transplant}(z, z.\text{right})^*$



$\text{Transplant}(z, z.\text{left})$

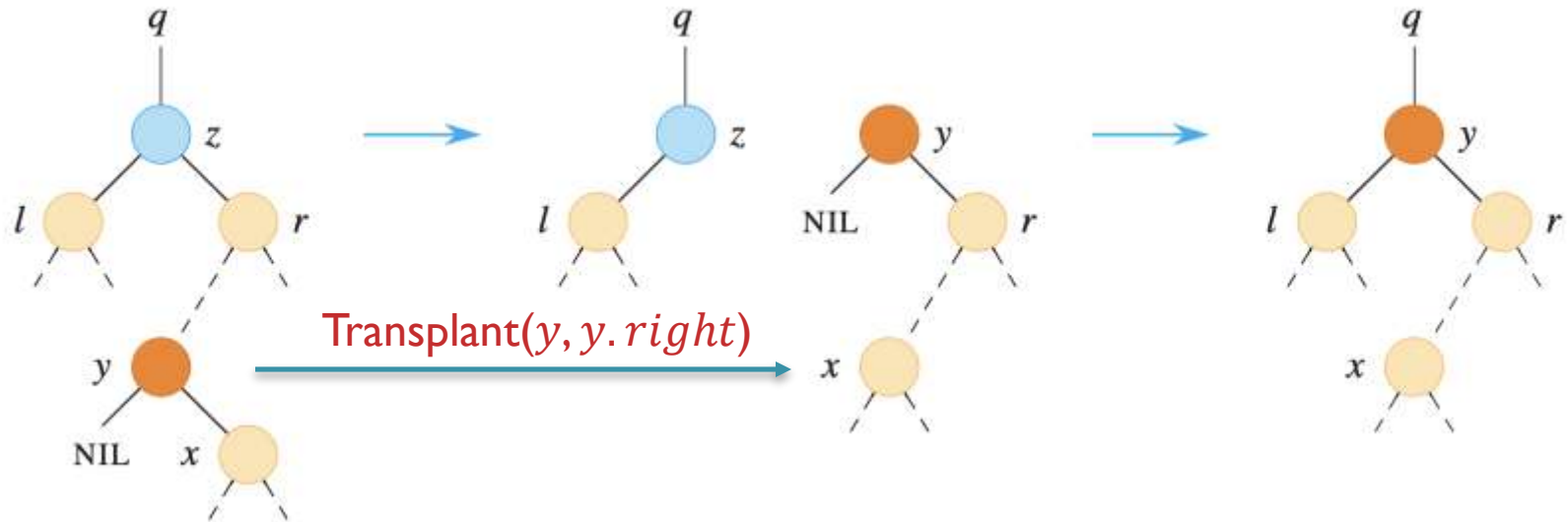
*skipping the argument T in Transplant for simplicity

BST: Deletion Case 2a



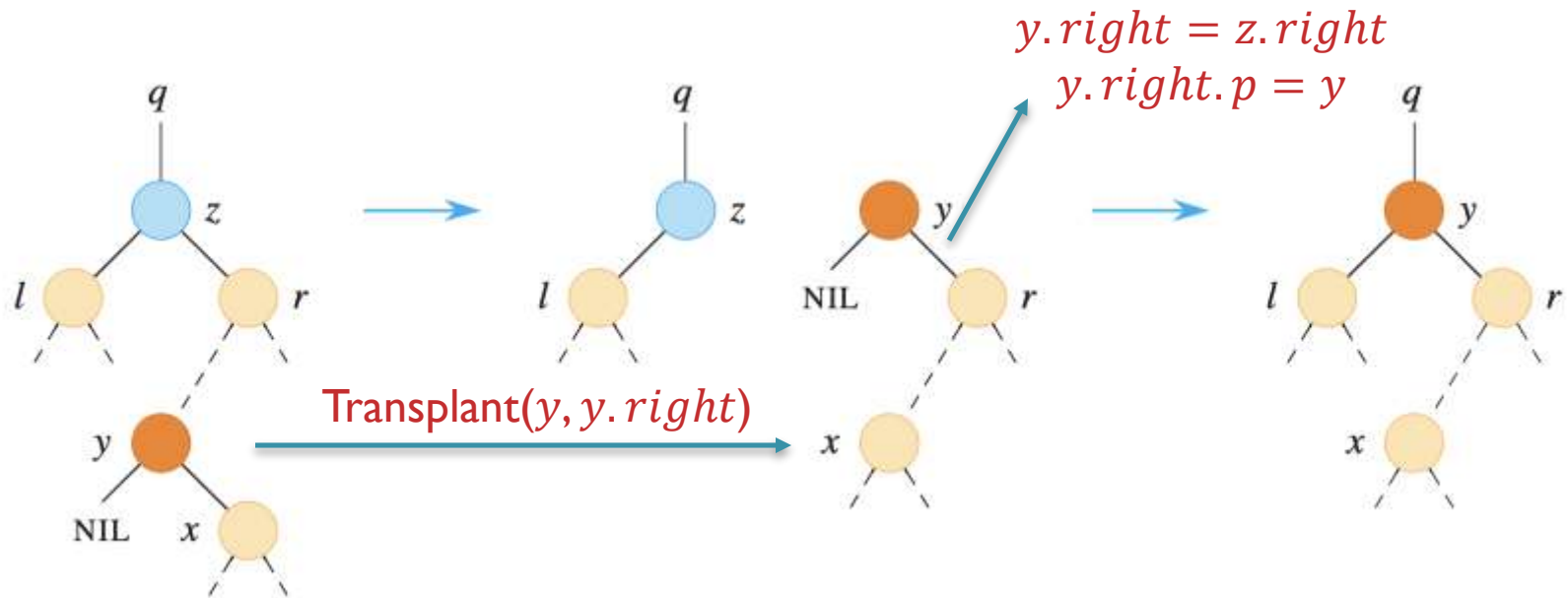
- In this case y , the successor of z , is **not** z 's right child.

BST: Deletion Case 2a



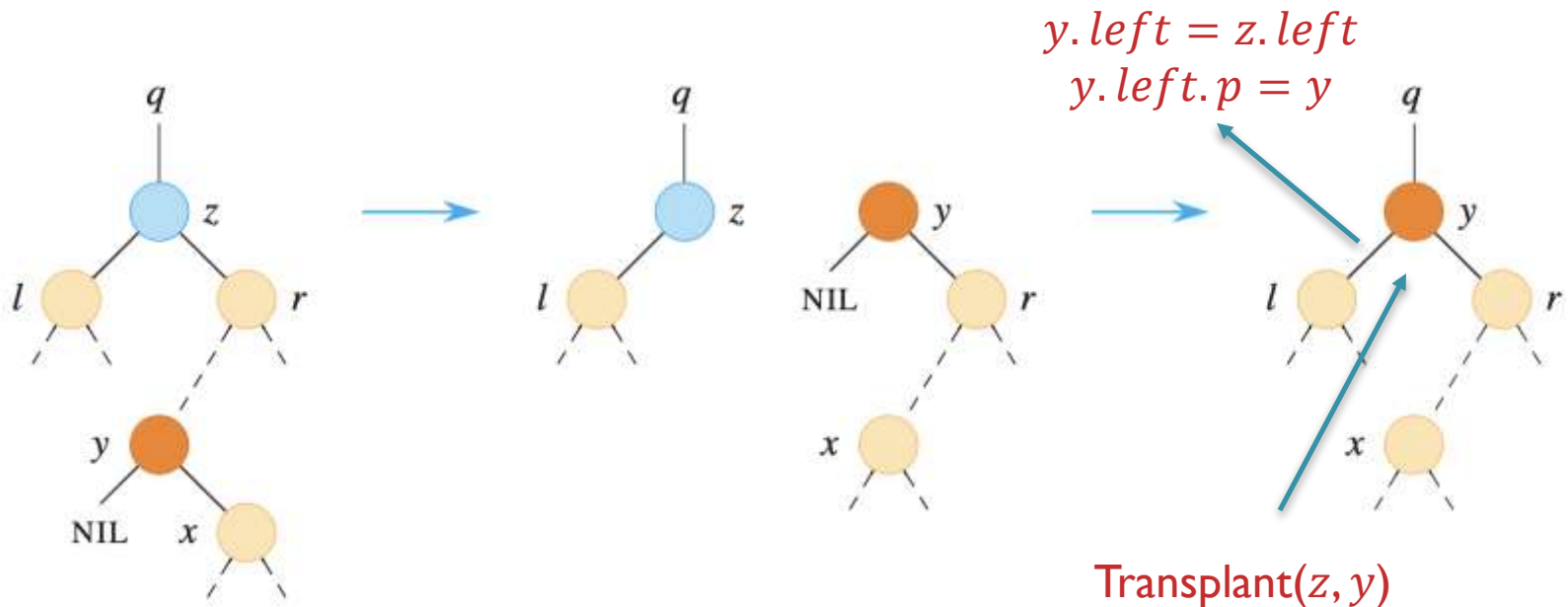
- First, we prepare a subtree rooted at y as follows:
 - Replace y by its own right child x .

BST: Deletion Case 2a



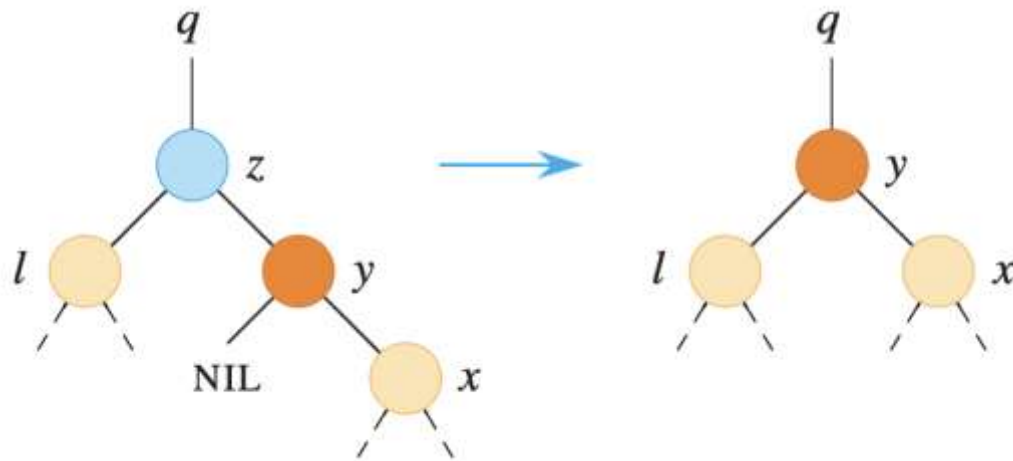
- First, we prepare a subtree rooted at y as follows:
 - Replace y by its own right child x .
 - Set y to be r 's parent.

BST: Deletion Case 2a



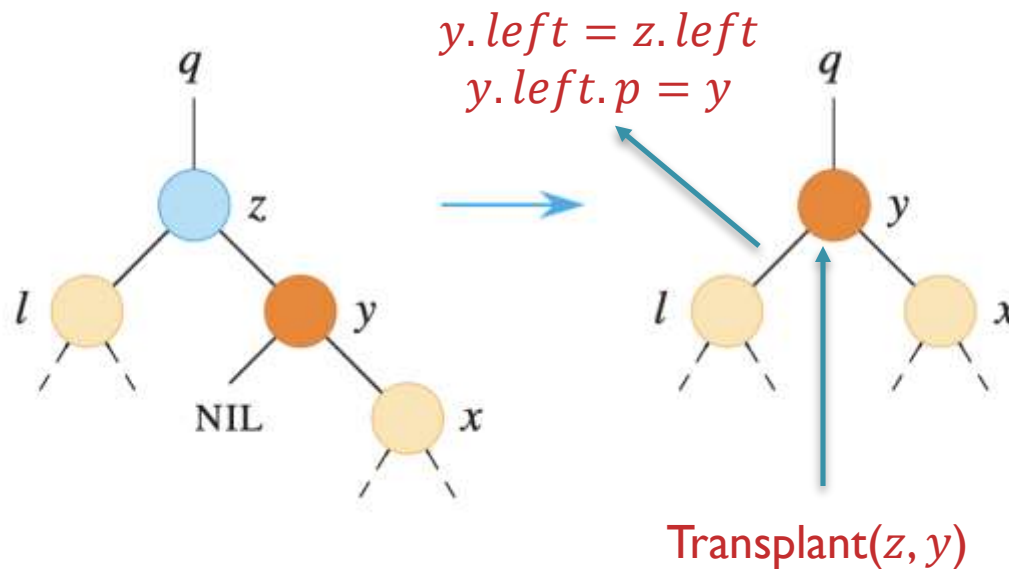
- First, we prepare a subtree rooted at y as follows:
 - Replace y by its own right child x .
 - Set y to be r 's parent.
- Then, set y to be q 's child and the parent of l .

BST: Deletion Case 2b



- In this case y , the successor of z , is z 's right child.

BST: Deletion Case 2b

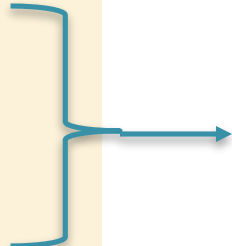


- Replace z by y , updating y 's left child to become l .

BST: Deletion

TREE-DELETE(T, z)

```
1  if  $z.left == \text{NIL}$ 
2      TRANSPLANT( $T, z, z.right$ )
3  elseif  $z.right == \text{NIL}$ 
4      TRANSPLANT( $T, z, z.left$ )
5  else  $y = \text{TREE-MINIMUM}(z.right)$ 
6      if  $y \neq z.right$ 
7          TRANSPLANT( $T, y, y.right$ )
8           $y.right = z.right$ 
9           $y.right.p = y$ 
10     TRANSPLANT( $T, z, y$ )
11      $y.left = z.left$ 
12      $y.left.p = y$ 
```



Handling Case I

BST: Deletion

TREE-DELETE(T, z)

```
1  if  $z.left == \text{NIL}$ 
2      TRANSPLANT( $T, z, z.right$ )
3  elseif  $z.right == \text{NIL}$ 
4      TRANSPLANT( $T, z, z.left$ )
5  else  $y = \text{TREE-MINIMUM}(z.right)$ 
6      if  $y \neq z.right$ 
7          TRANSPLANT( $T, y, y.right$ )
8           $y.right = z.right$ 
9           $y.right.p = y$ 
10     TRANSPLANT( $T, z, y$ )
11      $y.left = z.left$ 
12      $y.left.p = y$ 
```

Note. TREE-MINIMUM($z.right$)
returns the successor of z .

Handling Case 2

BST: Deletion

TREE-DELETE(T, z)

```
1  if  $z.left == \text{NIL}$ 
2      TRANSPLANT( $T, z, z.right$ )
3  elseif  $z.right == \text{NIL}$ 
4      TRANSPLANT( $T, z, z.left$ )
5  else  $y = \text{TREE-MINIMUM}(z.right)$ 
6      if  $y \neq z.right$ 
7          TRANSPLANT( $T, y, y.right$ )
8           $y.right = z.right$ 
9           $y.right.p = y$ 
10     TRANSPLANT( $T, z, y$ )
11      $y.left = z.left$ 
12      $y.left.p = y$ 
```

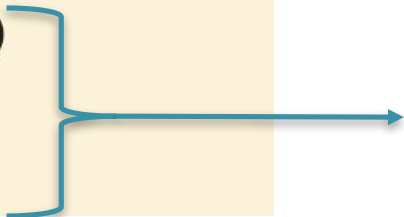
Note. TREE-MINIMUM($z.right$)
returns the successor of z .

Handling Case 2a

BST: Deletion

TREE-DELETE(T, z)

```
1  if  $z.left == \text{NIL}$ 
2      TRANSPLANT( $T, z, z.right$ )
3  elseif  $z.right == \text{NIL}$ 
4      TRANSPLANT( $T, z, z.left$ )
5  else  $y = \text{TREE-MINIMUM}(z.right)$ 
6      if  $y \neq z.right$ 
7          TRANSPLANT( $T, y, y.right$ )
8           $y.right = z.right$ 
9           $y.right.p = y$ 
10     TRANSPLANT( $T, z, y$ )
11      $y.left = z.left$ 
12      $y.left.p = y$ 
```



Note. TREE-MINIMUM($z.right$)
returns the successor of z .

Handling Case 2b

BST: Deletion

TREE-DELETE(T, z)

```
1  if  $z.left == \text{NIL}$ 
2      TRANSPLANT( $T, z, z.right$ )
3  elseif  $z.right == \text{NIL}$ 
4      TRANSPLANT( $T, z, z.left$ )
5  else  $y = \text{TREE-MINIMUM}(z.right)$ 
6      if  $y \neq z.right$ 
7          TRANSPLANT( $T, y, y.right$ )
8           $y.right = z.right$ 
9           $y.right.p = y$ 
10     TRANSPLANT( $T, z, y$ )
11      $y.left = z.left$ 
12      $y.left.p = y$ 
```

Time complexity. The procedure runs in $O(h)$ time owing to the call to **TREE-MINIMUM($z.right$)** in Step 5.

Practice problems

Practice Problem I

Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ and k is a character string interpreted in radix 2^p . Show that if string x can be converted to string y by permuting its characters, then x and y hash to the same value. ~~Give an example of an application in which this property would be undesirable in a hash function.~~

Practice Problem 2

What is the difference between the binary-search-tree property and the min-heap property ~~on page 163~~? Can the min-heap property be used to print out the keys of an n -node tree in sorted order in $O(n)$ time? Show how, or explain why not.

Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

Practice Problem 3

You are searching for the number 363 in a binary search tree containing numbers between 1 and 1000. Which of the following sequences *cannot* be the sequence of nodes examined?

- a.* 2, 252, 401, 398, 330, 344, 397, 363.
- b.* 924, 220, 911, 244, 898, 258, 362, 363.
- c.* 925, 202, 911, 240, 912, 245, 363.
- d.* 2, 399, 387, 219, 266, 382, 381, 278, 363.
- e.* 935, 278, 347, 621, 299, 392, 358, 363.

Practice Problem 4

Write the TREE-PREDECESSOR procedure.

Professor Kilmer claims to have discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up at a leaf. Consider three sets: A , the keys to the left of the search path; B , the keys on the search path; and C , the keys to the right of the search path. Professor Kilmer claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

Practice Problem 5

Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $y.key$ is either the smallest key in T larger than $x.key$ or the largest key in T smaller than $x.key$.

Practice Problem 6

Is the operation of deletion “commutative” in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x ? Argue why it is or give a counterexample.

Practice Problem 7

Let b_n denote the number of different binary trees with n nodes. In this problem, you will find a formula for b_n , as well as an asymptotic estimate.

a. Show that $b_0 = 1$ and that, for $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} .$$

b. ~~Referring to Problem 4-5 on page 121 for the definition of a generating function,~~ let $B(x)$ be the generating function

$$B(x) = \sum_{n=0}^{\infty} b_n x^n .$$

Show that $B(x) = xB(x)^2 + 1$, and hence one way to express $B(x)$ in closed form is

$$B(x) = \frac{1}{2x} (1 - \sqrt{1 - 4x}) .$$

The *Taylor expansion* of $f(x)$ around the point $x = a$ is given by

Practice Problem 7 (contd.)

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!} (x - a)^k ,$$

where $f^{(k)}(x)$ is the k th derivative of f evaluated at x .

c. Show that

$$b_n = \frac{1}{n+1} \binom{2n}{n}$$

(the n th *Catalan number*) by using the Taylor expansion of $\sqrt{1-4x}$