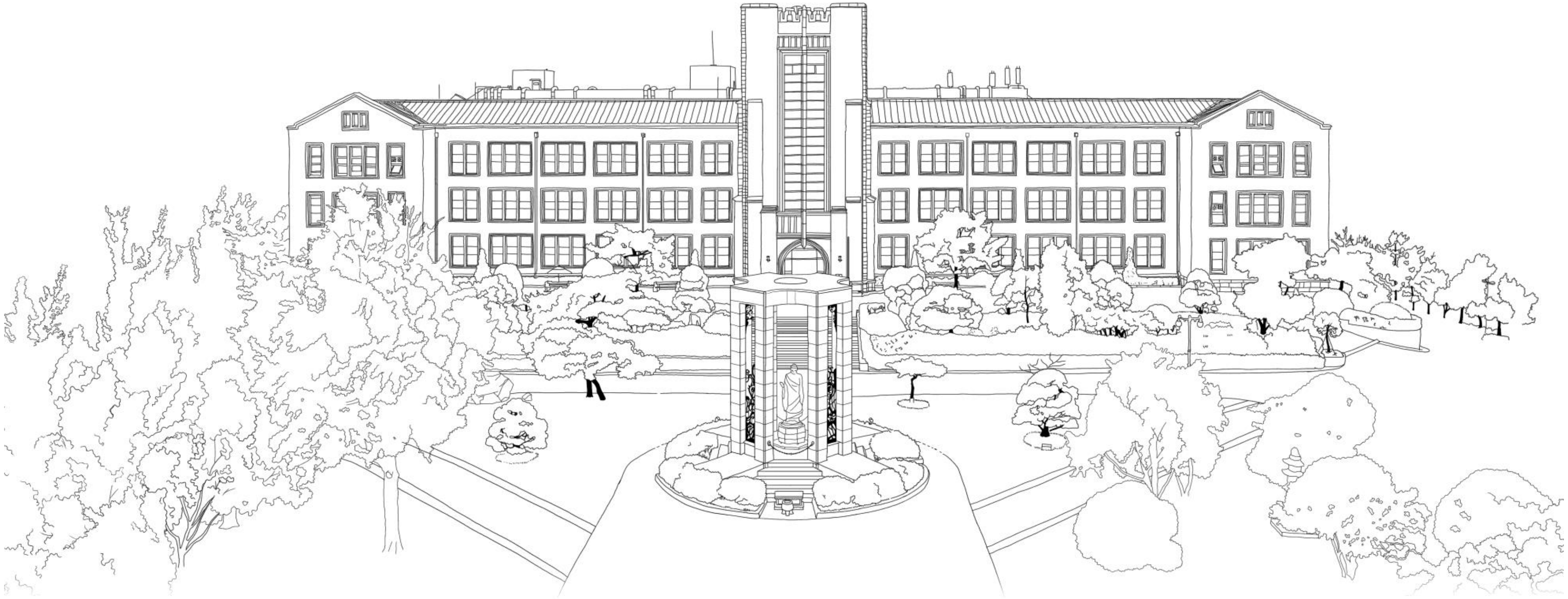


Introduction to Algorithms

DongYeon Kim
Department of Multimedia Engineering
Dongguk University

BigDataLab
Dept. of Multimedia Engineering at Dongguk University

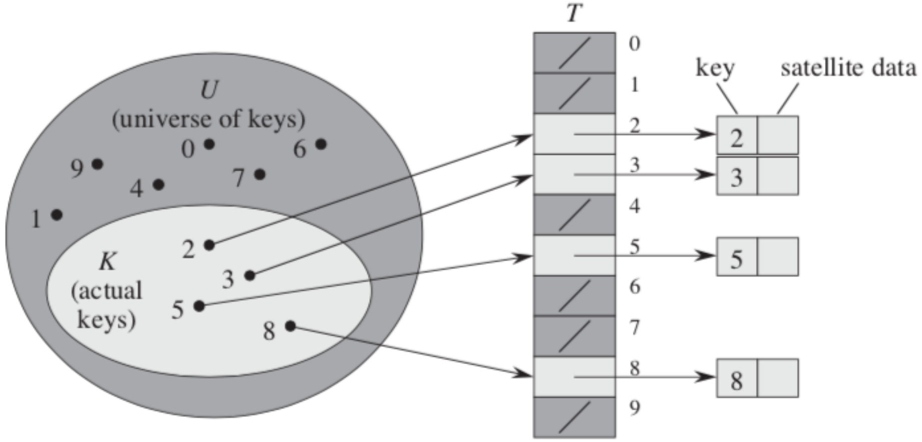
Contents

❖ Chapter.3 Data Structures

❖ Direct-address tables

- **Introduction**
  - ➢ A hash table is an effective data structure for implementing dictionaries
  - ➢ The average time to search for an element in a hash table is O(1)
    - ✓ In worst case, searching for an element in a hash table equals to linked list : $\theta(n)$

- **Definition of direct-address tables**
  - ➢ Works well when the universe U of keys is reasonably small
  - ➢ Each element has a key from the universe $U = \{0,1,..,m-1\}$ → no two elements have the same key



- ➢ Use a direct-address table denoted by $T[0,..,m-1]$
- ➢ Each slot corresponds to a key in the universe U
- ➢ Slot k points to an element in the set with the key k
- ➢ If the set contains no element with key k, then $T[k] = NIL$

❖ Direct–address tables

- The operation of table

DIRECT-ADDRESS-SEARCH(T, k)
1 **return** $T[k]$

DIRECT-ADDRESS-DELETE(T, x)
1 $T[x.key] = \text{NIL}$

DIRECT-ADDRESS-INSERT(T, x)
1 $T[x.key] = x$

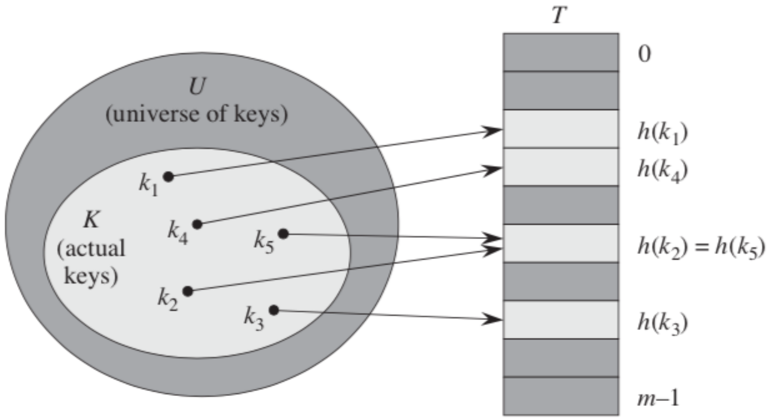
  - ➢ Each of the operations takes only $O(1)$ time

❖ Hash tables

- introduction
  - ➢ The disadvantages of direct addressing table
    - ✓ If the universe U is large, a table T of size $|U|$ may be impossible in given memory
    - ✓ The set k of keys actually stored may be small relative to U → most of the space allocated for T would be wasted

❖ Hash tables

- definition
  - ➢ The hash table requires much less storage than a direct address table
  - ➢ Reduce the storage requirement to $\theta(|k|)$, maintaining the benefit that searching for an element in only $O(1)$ time
  - ➢ With hashing, the element k is stored in slot $h(k)$
    - ✓ In direct address table, stored in slot k
    - ✓ Use hash function h to compute the slot from the key k
    - ✓ h maps the universe U of keys into slots of a hash table $T[0,..,m-1]$
    - ✓ $h : U \rightarrow \{0, 1, \ldots , m-1\}$



➢ Reduces the range of array indices and the size of array T

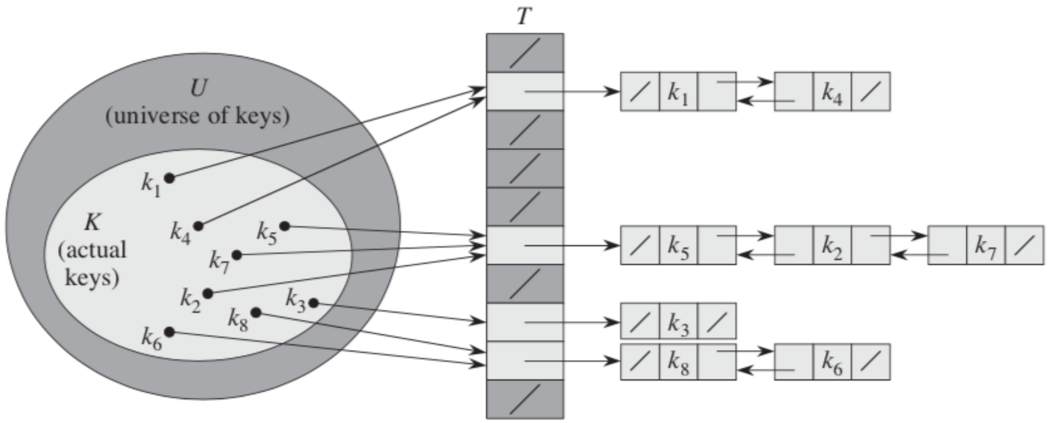
➢ there is a collision problem like $h(k_2) = h(k_5)$

➢ Introduce effective techniques for resolving the conflict created by collision

❖ Hash tables

- **problems**
  - ➢ The ideal solution would be to avoid collision
    - ✓ $|U| > m$, must be at least two keys that have the same hash value
    - ✓ Avoiding collisions altogether is impossible
    - ✓ Present the simplest collision resolution technique called **chaning**

- **Collision resolution by chaning**



- ➢ Place all elements that hash to the same slot into the same linked-list
- ➢ Slot j contains a pointer to the head of the list all stored elements that hash to j
- ➢ If there are no elements, slot j contains NIL

Hash Tables

❖ Hash tables

- The operations on hash table T
  - ➢ Easy to implement when collisions are resolved by chaning
  - ➢ Worst case running time for $INSERT$ & $DELETE$: $O(1)$

CHAINED-HASH-INSERT(T, x)
1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)
1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)
1 delete x from the list $T[h(x.key)]$

❖ Hash functions

- Introduction
  - ➢ In this chapter, we discuss some issues regarding the design of good hash functions
  - ➢ Present three methods for their creation
  - ➢ A good hash function satisfies the assumption of simple uniform hashing
    - ✓ Each key is equally likely to hash to any of the m slots, independently, evenly

❖ Hash functions

- **The division method**
  - ➢ Map a key k into one of m slots by taking the remainder of k divided by m
  - ➢ $h(k) = k \bmod m$
    - ✓ Hash table size m = 12, key k = 100
    - ✓ $h(k)$ = 4
  - ➢ Requires only a single division operation, hashing by division is quite fast

- **The multiplication method**
  - ➢ Multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA
  - ➢ Multiply this value by m and take the floor of the result
  - ➢ $h(k) = \lfloor m(\underline{kA \bmod 1}) \rfloor$

Represent the fractional part of kA, that is $kA - \lfloor kA \rfloor$

❖ Hash functions

- **Universal hashing**
  - ➢ In worst-case, n elements are hashed to the same slot
    - ✓ Retreival time is $\theta(n)$
    - ✓ Fixed hash function is vulnerable to such worst-case
  - ➢ Choose the hash function randomly in a way that is independent of the keys that are actually going to be stored

❖ Open addressing

- **Definition**
  - ➢ All elements occupy the hash table itself
    - ✓ Each table entry contains either the element or NIL value
  - ➢ no elements are stored outside the table unlike in chaining
  - ➢ The advantage of open addressing is that it avoids pointers altogether
    - ✓ Instead of following pointers, compute the sequence of slots
    - ✓ The extra memory freed by not storing pointers provide the hash table with a larger number of slots for the same amount of memory

0	700
1	50
2	85
3	92
4	NIL
5	NIL
6	76

❖Open addressing

- Insertion
  - ➢ Probe the hash table until we find an empty slot
  - ➢ Require that for every key k, there is a prob sequence
    - ✓ $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ probe number
    - ✓ Every hash table slots are considered as a slot for a new key as the table fills up

0	700
1	50
2	85 40
3	92 40
4	40
5	NIL
6	76

HASH-INSERT(T, k)

```
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == NIL
5          T[j] = k
6          return j
7      else i = i + 1
8  until i == m
9  error "hash table overflow"
```

- ➢ Each slot contains key or NIL
- ➢ Input : table T, key k
- ➢ Output : inserted index number or error message
- ➢ line 3 : get index number which is hashed
- ➢ line 4~6 : if it finds an empty slot, insert element
- ➢ line 7 : if it is not empty slot, examine the next slot
- ➢ Line 8~9 : if it examine all slots → hash overflow

❖Open addressing

- Search
  - ➢ The algorithm for searching key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted
  - ➢ Search can terminate unsuccessfully when it finds an empty slot
    - ✓ Since k would has been inserted there
    - ✓ Not later in its probe sequence

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

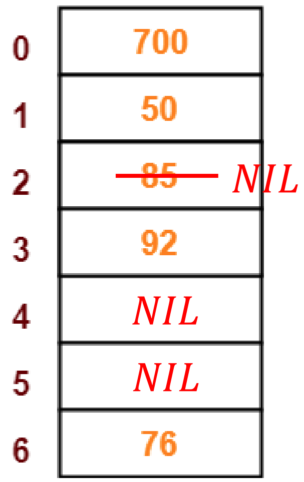
- ➢ Input : table T, key k
- ➢ Output
  - ✓ j – if it finds slot contains key k
  - ✓ NIL – if it doesn't find the element
- ➢ line 3 : get index number
- ➢ line 4~5 find the element
- ➢ line 7~8 : if there is no element, return NIL

0	700
1	50
2	85
3	92
4	
5	
6	76

Hash Tables

❖Open addressing

- Deletion
  - ➢ Deletion from an open-address hash table is difficult
  - ➢ When we delete a key from slot i, we cannot simply mark that slot as NIL
    - ✓ If we did, we might be unable to retrieve any key k
  - ➢ By marking the slot as $DELETED$, we solve this problem
    - ✓ Modify the procedure $HASH-INSERT$ to treat such slot as empty



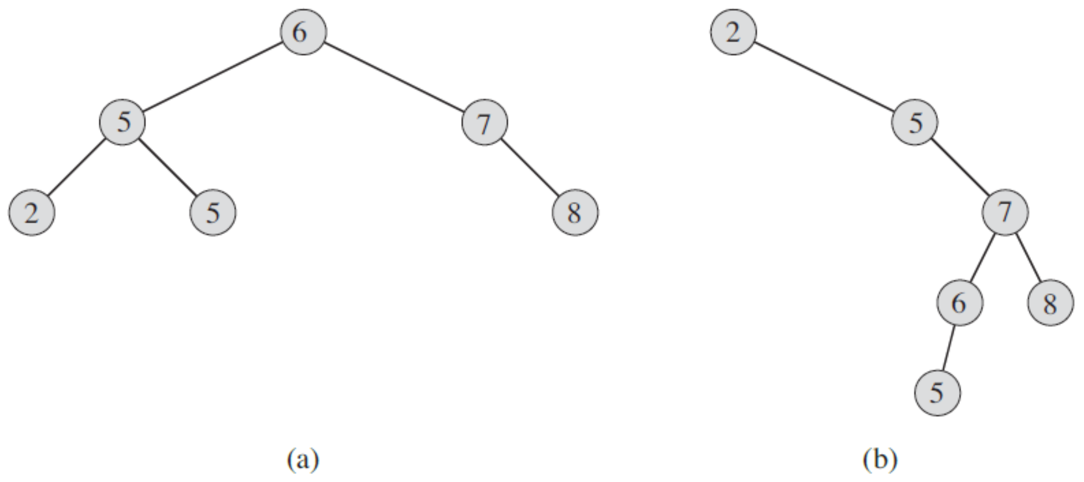
Contents

❖ Chapter.3 Data Structures

❖ What is a binary search tree?

- Introduction
  - ➢ Search tree data structure supports many operations, including $\boldsymbol{SEARCH}, \boldsymbol{MINIMUM}, \boldsymbol{MAXIMUM}, \boldsymbol{PREDECESSOR}, \boldsymbol{SUCCESSOR}, \boldsymbol{INSERT}, \boldsymbol{DELETE}$
  - ➢ Basic operations on a binary search tree take time proportional to the height of the tree $\theta(\lg n) = \theta(h)$

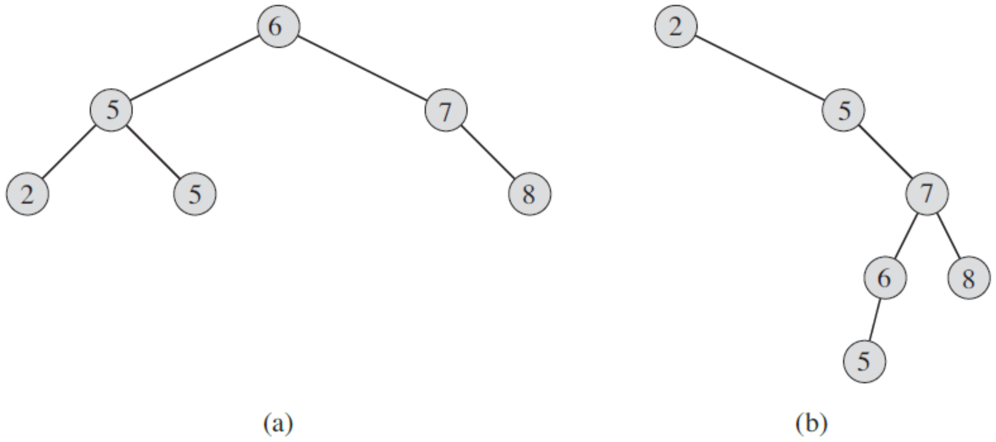
- Definition of binary search tree



- ➢ Represent such a tree by a linked data structure in which each node is an object
- ➢ Each node contains a key, satellite data, attributes left, right, and p

❖ What is a binary search tree?

- Definition of binary search tree



➢ Attributes $left, right$
  - ✓ Point to the notes corresponding to its left and right child

➢ Attributes P
  - ✓ Point to its parent nodes

➢ If child or parents is missing, the attribute contains the value NIL

➢ 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 $y.key \geq x.key$.

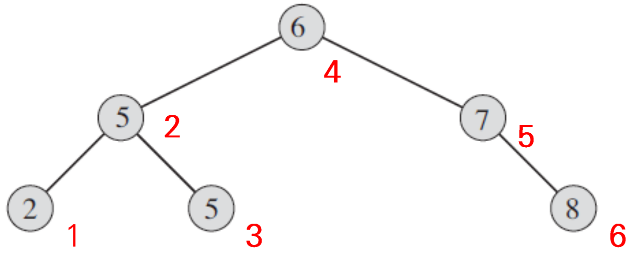
✓ This property holds for every node in tree

Binary Search Tree

❖ What is a binary search tree?

- **Inorder-tree-walk**
  - ➢ BST property allows us to print out all the key in a BST in sorted order
  - ➢ Print the root of a subtree between printing the left and right subtree (inorder)
    - ✓ Preorder : print the root before the values in either subtree
    - ✓ Postorder : print the root after the values in either subtree

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```



- ✓ Takes $\theta(n)$ time to walk an n-node BST
  - ✓ After the initial call, the procedure calls itself recursively exactly twice for each node

❖ What is a binary search tree?

- Inorder-tree-walk

Theorem 12.1

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

➢ Proof of the theorem

✓ $T(n)$: the time taken by $INORDER - TREE - WALK$

✓ If $n = 0, T(0) = c$

✓ For n>0, left subtree has k nodes, right subtree has $n - k - 1$ nodes

✓ $T(n) \leq T(k) + T(n - k - 1) + d$ for some constant $d > 0$

✓ Using substitution method, $T(n) \leq (c + d)n + c$

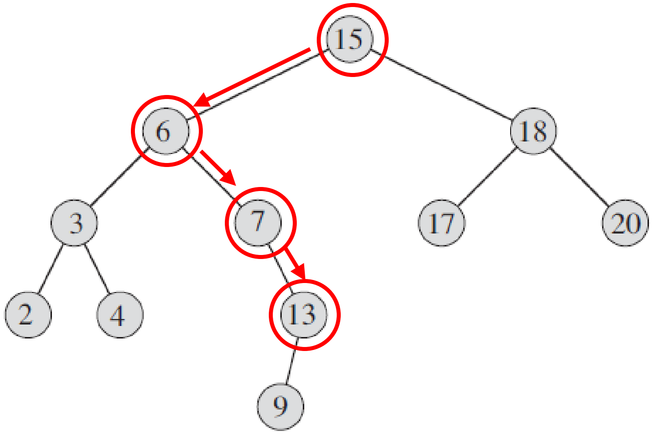
✓ For $n = 0$, $T(0) = (c + d) * 0 + c = c$

✓ For $n > 0$,
$$\begin{aligned} T(n) &\leq T(k) + T(n-k-1) + d \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ &= (c+d)n + c - (c+d) + c + d \\ &= (c+d)n + c , \end{aligned}$$

❖ Querying a BST

- **Introduction**
  - ➢ In this section, introduce $SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDESSESSOR$
  - ➢ Show how to support each one in time $O(h)$ → height of BST is h

- **Searching**
  - ➢ Search for a node with a given key in a BST
  - ➢ Input : a pointer to the root node & key k
  - ➢ Output : a pointer to a node with key k or NIL



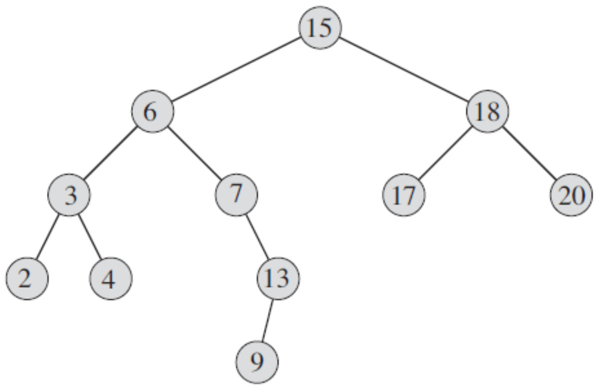
➢ To search for the key 13
  - ✓ Follow the path 15 → 6 → 7 → 13

➢ For each node x it encounters, it compares the key k with $x.key$

Binary Search Tree

❖ Querying a BST

- Searching
  - ➢ Input : a pointer to the root node & key k
  - ➢ Output : a pointer to a node with key k or NIL




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

  - ➢ Line 1~2 : if it finds key k or there are no elements, return x (NIL or $x.key$)
  - ➢ Line 3~4 : if k is smaller than $x.key$, the search continues in the left subtree of x
    - ✓ The BST property implies that k could not be stored in the right subtree
  - ✓ Line 5 : symmetrically with the line 3~4

Binary Search Tree

❖ Querying a BST

- Searching
  - ➢ Can rewrite this procedure in a iterative fashion

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

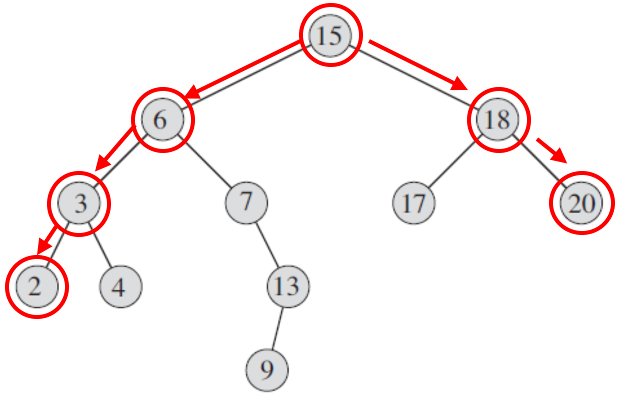
→

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

  - ➢ The running time of $TREE - SEARCH$ is $O(h)$
    - ✓ h is the height of tree
    - ✓ Encounter nodes from the root of the tree to the finding key in a simple path downward

❖ Querying a BST

- **Minimum and Maximum**
  - ➢ Can always find an element in a BST whose key is a minimum or maximum by following left or right child pointers from the root until we encounter a NIL



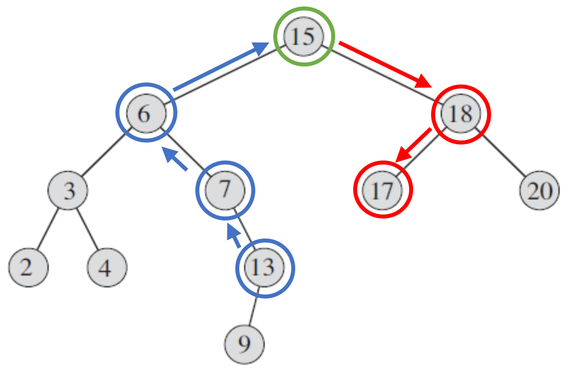

```
TREE-MINIMUM(x)
    while x.left ≠ NIL
        x = x.left
    return x
```

```
TREE-MAXIMUM(x)
    while x.right ≠ NIL
        x = x.right
    return x
```

  - ➢ Input : pointer to the root node x
  - ➢ Output : pointer to the minimum or maximum element
  - ➢ The running time of these procedure is $O(h)$
    - ✓ The sequence of nodes encountered forms a simple path downward from the root

❖ Querying a BST

- **Successor and predecessor**
  - ➢ Find node x's successor in the sorted order determined by an inorder tree walk



- ➢ The successor of a node x is the node with the smallest key greater than $x.key$
- ➢ BST allows us to determine the successor without ever comparing keys

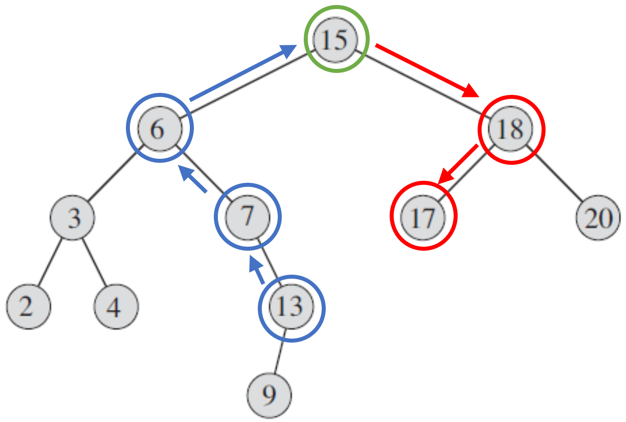
```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

- ➢ Input : pointer of node x
- ➢ Output : pointer of node x's successor in sorted order
- ➢ Divide the pseudocode in two steps
- ➢ First step : node x has right subtree
  - ✓ Line 1~2 : find the minimum value on right subtree

Binary Search Tree

❖ Querying a BST

- Successor and predecessor




```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

➢ Second step : node x has no right subtree

✓ Find the parents node which is larger than child

✓ Line 3~7 : simply go up the tree from x until encounter a node that is the left child of its parent

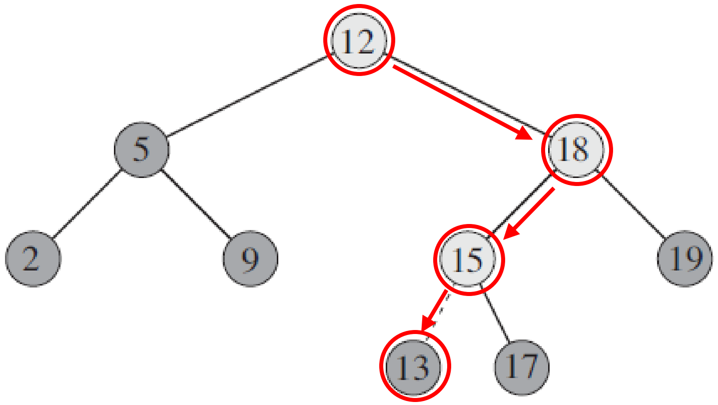
➢ The running time of tree successor on a tree of height h is $O(h)$

✓ Either follow a simple path up the tree or down the tree

❖ Insertion and deletion

- Definition
  - ➢ The operation of insertion and deletion cause the dynamic set represented by a BST to change
    - ✓ The BST property continues to hold

- Insertion
  - ➢ To insert the new value v into a BST T, we use the procedure $TREE-INSERT(T,z)$
  - ➢ Initialize the node z for which $z.key = v,\ z.left = NIL,\ z.right = NIL$

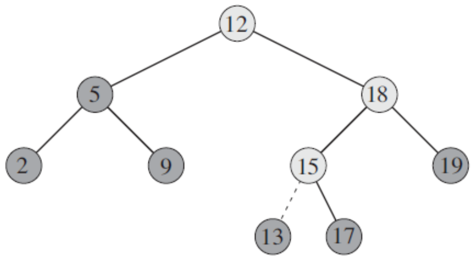


- ➢ TREE-INSERT begins at the root of the tree
- ➢ pointer x traces a simple path downward looking for a NIL
- ➢ replace NIL with the input item z

❖ Insertion and deletion

- Insertion

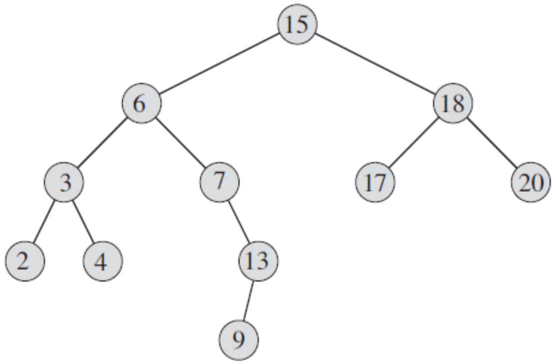
```
TREE-INSERT(T, z)
1   y = NIL
2   x = T.root
3   while x ≠ NIL
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y
9   if y == NIL
10      T.root = z          // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```



- Line 1~2 : initialize x, y
  - ✓ x : current node
  - ✓ y : parent node of x
- Line 3~7
  - ✓ While loop causes two pointers to move down
  - ✓ Going left or right depending on the comparison of $z.key$ with $x.key$ until x becomes NIL
- Line 8~13
  - ✓ Set the pointer that cause z to be inserted
- The running time is $O(h)$
  - ✓ Start from root to NIL

❖ Insertion and deletion

- Deletion
  - ➢ The overall strategy for deleting a node z from a BST T has three basic cases



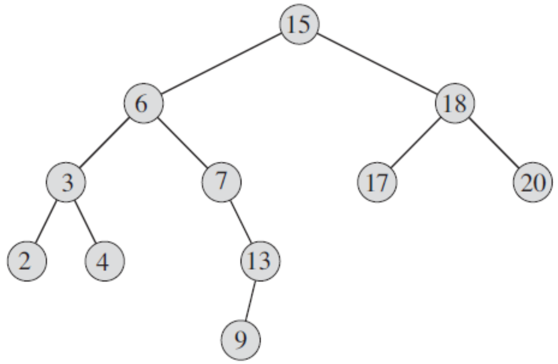
- ➢ First case
  - ✓ z has no children
  - ✓ Simply remove it by modifying its parent to replace z with NIL as its child
  - ✓ $Delete\,9 \;\rightarrow\; 13.left = NIL$

- ➢ Second case
  - ✓ z has just one child
  - ✓ Elevate that child to take z's position by modifying z's parent to replace z by z's child
  - ✓ $Delete\,13 \;\rightarrow\; 7.right = 9 \rightarrow 9.p = 7$

Binary Search Tree

❖ Insertion and deletion

- Deletion
  - ➢ The overall strategy for deleting a node z from a BST T has three basic cases



- ➢ Third case
  - ✓ z has two children
  - ✓ Find z's successor y
  - ✓ Rest of z's original right & left subtree becomes y's new right & left subtree
  - ✓ $Delete\ 15$ → find successor 17 → $17.left = 6$ & $17.right = 18$

- ➢ In order to move subtree around within the BST, we define a subroutine $TRANSPLANT(T, u, v)$
  - ✓ Replace the subtree rooted at node u with the subtree rooted at node v
  - ✓ Node u's parent becomes node v's parent

❖ Insertion and deletion

- Deletion
  - ➢ Pseudocode for $TRANSPLANT$

```
TRANSPLANT(T, u, v)
1  if u.p == 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 ≠ NIL
7      v.p = u.p
```

✓ Line 1~2 : handle the case v is root of T

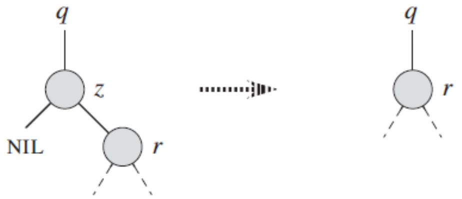
✓ Line 3~5 : examine v is left child or right child, updating $u.p.left$ or $u.p.right$

✓ Line 6~7 : update point to parent node

❖ Insertion and deletion

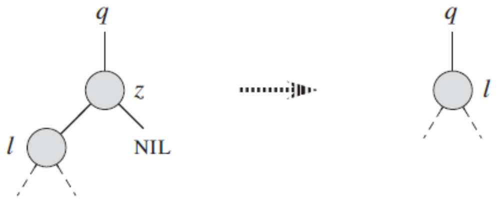
- Deletion

(a)



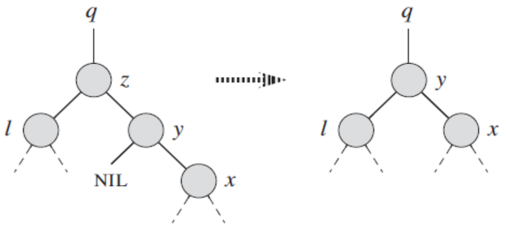
➢ (a) : z has no left child
  - ✓ Replace z with its right child

(b)



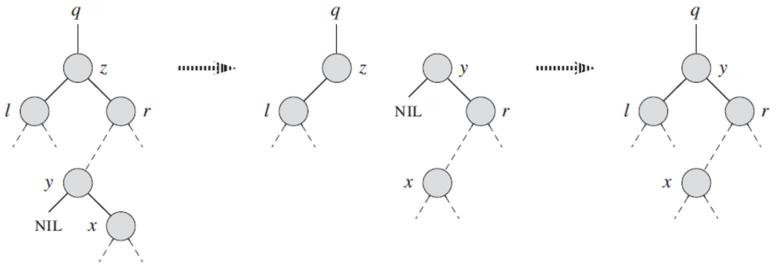
➢ (b) : z has no right child
  - ✓ Replace z with its left child

(c)



➢ (c) : z has both left & right child → successor y is z's right child
  - ✓ Replace z with its right child y

(d)



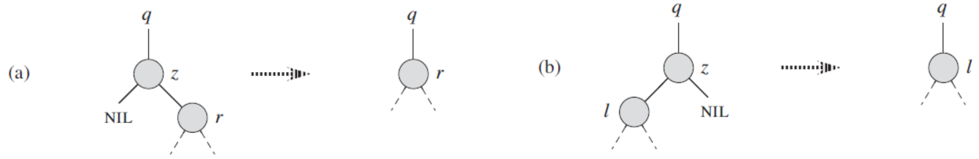
➢ (d) : z has both left & right child → successor y is not z's right child
  - ✓ Replace y by its own right child x
  - ✓ Replace z with y

❖ Insertion and deletion

- Deletion
  - ➢ Pseudocode for $TREE - DELETE$
  - ➢ Delete node z from BST T

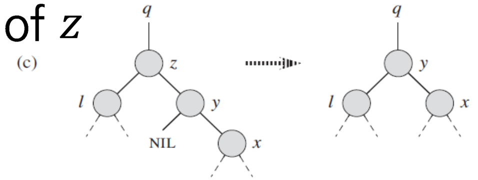
```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)
5   else y = TREE-MINIMUM(z.right)
6       if y.p ≠ z
7           TRANSPLANT(T, y, y.right)
8           y.right = z.right
9           y.right.p = y
10      TRANSPLANT(T,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

➢ Line 1~2 or line 3~4
- ✓ Only have right or left child
- ✓ Replace z with $z.right$ or $z.left$



➢ Line 5
- ✓ Find the minimum key in right subtree (successor y)

✓ Line 10~12
- ✓ If $y.p\ == z \rightarrow y$ is right child of z
- ✓ Replace z with y
- ✓ Connect $z.left$ with y

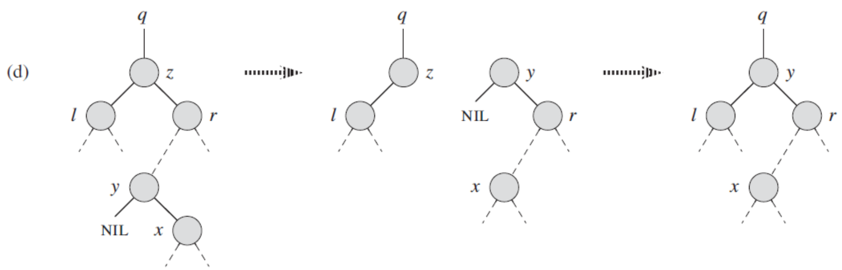


❖ Insertion and deletion

- Deletion
  - ➢ Pseudocode for $TREE - DELETE$
  - ➢ Delete node z from BST T

```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)
5   else y = TREE-MINIMUM(z.right)
6       if y.p ≠ z
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
```

- ➢ Line 6~9
  - ✓ Successor y is not a right child of z
  - ✓ Replace y with $y.right$



- ➢ The running time of $TREE - DELETE$ and $TRANSPLANT$ takes constant time $O(1)$
- ➢ $TREE - MINIMUM$ takes $O(h)$

Thank You!