Hash Tables and Binary Search Trees

Dictionaries, Hash table, Collision resolution, Chaining, Division method, Dynamic set, Binary search tree, Tree walks.

Dictionaries

- An element has a key part and a satellite data part.
- Dictionaries store elements so that they can be located quickly using **keys**
- Dictionary ADT
    - **Search(S, k)** – an access operation that returns an element where x.key = k
    - **Insert(S, x)** – a manipulation operation that adds element x to S
    - **Delete(S, x)** – a manipulation operation that removes element x from S
- Supporting order (methods such as min, max, successor, predecessor) is **not required**, thus it is enough that keys are comparable for equality.

Dictionaries: an example

- A dictionary may hold bank accounts
    - each account is an element that is identified by an **account number** (**key**)
    - each account is associated with some additional information, e.g., account holder's name, age, the amount of saving, the amount of loan, etc. (**satellite data**)
    - an application wishes to operate on an account would have to provide the account number as a search **key**

Dictionaries: a real problem

- Consider a large phone company, and they want to provide caller ID capability
    - Element: phone number (key) + name of the owner, remaining credits, etc. (satellite data).
    - Given a phone number, return the owner's name
    - n phone numbers range from 0 to r
        - E.g., 500,000 users from 99,999,999 possible mobile phone numbers.
        - r is much larger than n.
    - Wants to do this as efficiently as possible

Dictionaries: a real problem: Array implementation

- Array A[1…r]
- Direct addressing: an array indexed by key

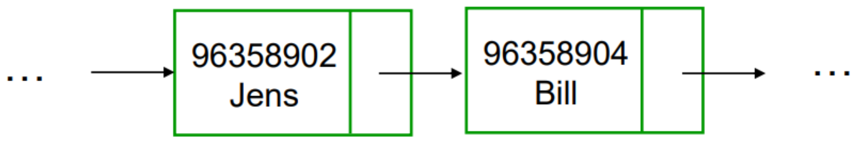
Consider a STACK ADT with the following standard operations: *init*(), *push*(*x*:*int*), *pop*():*int*, and *top*():*int*. Here, *pop*() both removes an element and returns it as a result, while *top*() just returns the element at the top of the stack. Assume an efficient implementation of this ADT, where all the aforementioned operations takes constant time. Assume that $n \geq 1$ and consider the following algorithm:

```
DoSomething(n:int):int
  sk, st: Stack
  sk.init()
  st.init()
  for i ← 1 to n do sk.push(i)
  for i ← n downto 1 do st.push(i)
  i ← n
  while sk.top() > 1 do
      for j ← 1 to i do sk.push(st.pop())
      i ← i − 1
      for j ← 1 to i do st.push(sk.pop())
  return st.top()
```

- Analysis: given a phone number, return the caller's name.
    - $\Theta(1)$ time
    - $\Theta(r)$ space - huge amount of wasted space
        - Those elements wtih "\", i.e., unused phone numbers.

Dictionaries: Linked list implementation

- A sequence of elements, where each element is with one key and one or more pointers.
- Singly linked list



- Analysis: given a phone number, return the caller's name

```
Quicksort(A, p, r)
  if p < r
      q = Partition(A, p, r)
      Quicksort(A, p, q − 1)
      Quicksort(A, q + 1, r)
```

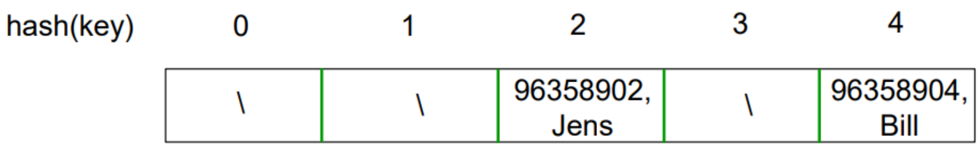
To sort an entire array A, the initial call is QUICKSORT(A, 1, $A.length$).

```
Partition(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

Hash Table

- Like an array, but come up with a **hash function** to map the large range (e.g., 0 to 99999999) into a small one which we can manage (e.g., 0 to 4).
    - e.g., take the original key, modulo the (relatively small) size of the array, and use that as an index
- Insert (96358904, Bill) into a hashed array with, say, 5 slots
    - hash(96358904) = 96358904 mod 5 = 4

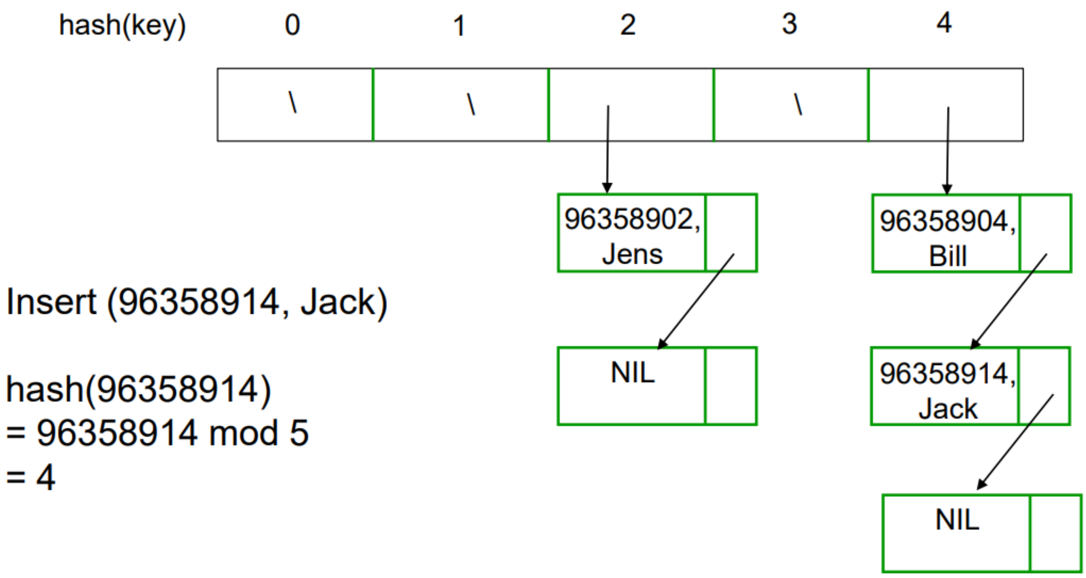
- hash(96358902) = 96358902 mod 5 = 2



- A lookup uses the same process: hash the query key, then check the array at that slot.
    - Search “96358904”, hash(96358904)=4, then “Bill”.
    - Search “96358900”, hash(96358900)=0, then “\”.
- So far so good! Constant search time and small space usage. Are there any problems?
- The problem is **collisions**!
    - Two different keys may have the same hashed value.
- Insert (96358914, Jack)
    - hash(96358914) = 96358914 mod 5 = 4
    - But Bill has taken the slot 4!

Collision Resolution

- Chaining
    - Each entry in the table is a pointer to a linked list.
    - All the elements that has the same hashed key are placed into a linked list.



Analysis of hashing

- In a hash table, an element with key k is stored in slot hash(k).
- The hash function hash maps the universe U of keys into the slots of hash table T[0...m-1]
    - hash: U → {0, 1, ..., m-1}
- Assumptions
    - **Simple uniform hashing**:
        - Each key is equally likely to be hashed into any slot;
        - Given hash table *T* with *m* slots holding n elements, the **load factor** is defined as *α=n/m*
    - The run time to compute the hash key hash(k) is Θ(1).

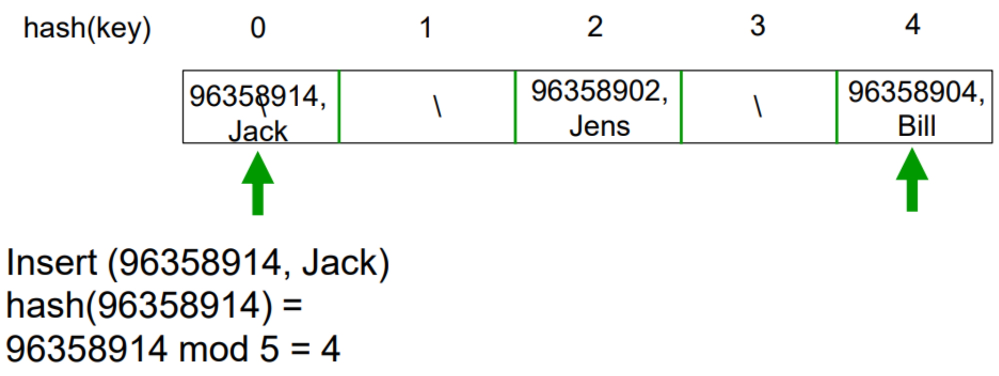
- We use chaining to solve collisions.

Analysis of Hashing with Chaining

- **Search(S, k)**
- Using the hash function to look up its slot in the table.
  - Θ(1), constant time, nothing related to n.
- Searching for the element in the linked list of the slot.
  - *uniform* hashing yields an average list length α = *n/m*.
  - expected number of elements to be examined is α.
  - search time Θ(α).
- **Assuming the number of hash table slots is proportional to the number of elements in the table**.
  - α is then a constant.
- Searching an element in a hash table with chaining is constant time Θ(1).
- Insertion: **Insert(S, x)**
  - Constant time (insertion in a linked list takes constant time)
- Deletion: **Delete(S, x)**
  - Constant time (deletion in a linked list takes constant time)
- When choosing a **simple uniform hashing**;
- When computing the hash function is done in **constant time**.
- When using **chaining** to solve collisions;
- When the number of slots m is **proportional** to the number of elements n:
- Then, a hash table can do all the 3 important dictionary operations in **CONSTANT TIME**!

Collision Resolution (2)

- Linear Probing: if the current location is used, try the next table location.
- Lookups walk along the table until the key or an empty slot is found.



Insert: First you calculate the modulo value. If that spot in hash is taken, take the next.. do this until you find a free spot. Search: get hash value. Is that value the one you are searching? If no check the next one. Do this until an empty space is found.

Open Addressing

- Step *i* from 0, 1, 2, ..., m-1 (Only when encountering a CRASH!)

- Linear probing: $h(k, i) = (h'(k) + i) \, mod \, m$
- Quadratic probing (c_1 and c_2 are constant): $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \, mod \, m$
- Double hashing: $h(k, i) = (h_1(k) + i h_2(k)) \, mod \, m$

Choosing a good hash function

- What is a good hash function?
  - Quick to compute: constant time.
  - Satisfies the simple **uniform hashing assumption**.
    - Each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to.
  - Good hash functions are very rare.
- Most hash functions assume that the universe of keys is the set of natural numbers {0, 1, 2, ...}.

- How to deal with hashing non-integer keys?
  - Find some way of turning the keys into integers
    - Remove the hyphen in string “9635-8904” to get 96358904.
    - For a string, add up the ASCII values of the characters of the string.
  - Then, use a standard hash function on the integers.

Hash Function: Division Method

- Use the remainder of division: **h(k) = k mod m**
  - k is the key, m the size of the table.
  - Fast, constant time.
- Need to carefully choose m and avoid certain values.
- $m = 2^e$ (**bad**)
  - if m is a power of 2, h(k) gives the e least significant bits of k.
  - all keys with the same ending go to the same place.
- m prime (**good**)
  - helps ensure uniform distribution.
  - primes not too close to exact powers of 2.

Example of a good Hash Function

- Hash table needs to hold for n = 2,000 keys
- Assume that we don’t mind examining 3 elements per slot.
- We can choose m = 701
  - A prime number near 2000/3
  - But not near any power of 2
- Note that m=701 is only good for this specific n=2000. If n changes, m must be also changed.

Dynamic Set

- Queries

- Search(S, k)
  - Search for the element with key k
- Minimum(S)
  - Find the element who has the smallest key.
- Maximum(S)
  - Find the element who has the largest key.
- Successor(S, x)
  - Find the element who has the next larger key to the key of element x.
- Predecessor(S, x)
  - Find the element who has the next smaller key to the key of element x.

- Modifying operations
  - Insert(S, x)
    - Insert element x into S,
  - Delete(S, x)
    - Remove element x from S.

Dictionary

- Dictionary ADT – a dynamic set with methods:
  - **Search(S, k)** – a query operation that returns element x where x.key = k
  - **Insert(S, x)** – a modifying operation that adds the element pointed to by x to S
  - **Delete(S, x)** – a modifying operation that removes the element x from S
- Constant time for the three operations under certain assumptions.

Priority Queue

- Priority Queue ADT – a dynamic set with methods:
  - **Insert(A, x)** – a modifying operation that adds the element x to A.
    - $\Theta(\lg n)$ for heap implementation
  - **Maximum(A)** – a query operation that finds the element whose key is the biggest
    - $\Theta(1)$ for heap implementation

Doubly Linked List

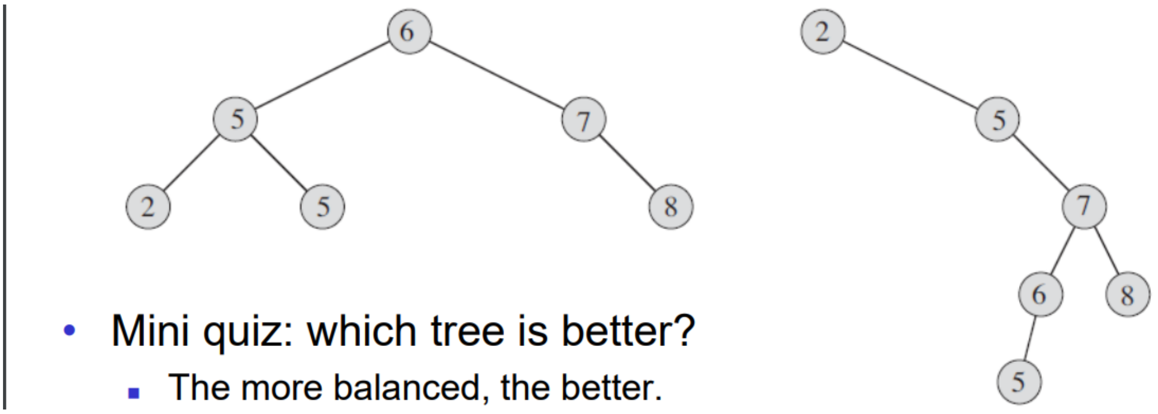
- Unordered list
  - Search, minimum, maximum, predecessor, successor: $\Theta(n)$
  - Insert, delete: $\Theta(1)$
- Ordered list
  - Search, insert: $\Theta(n)$
  - minimum, maximum, predecessor, successor, delete: $\Theta(1)$

What is a binary search tree?

- A binary search tree is a binary tree T satisfies **binarysearch-tree** property (do you still remember the max-heap property?)
  - Let x be a node in a binary search tree.

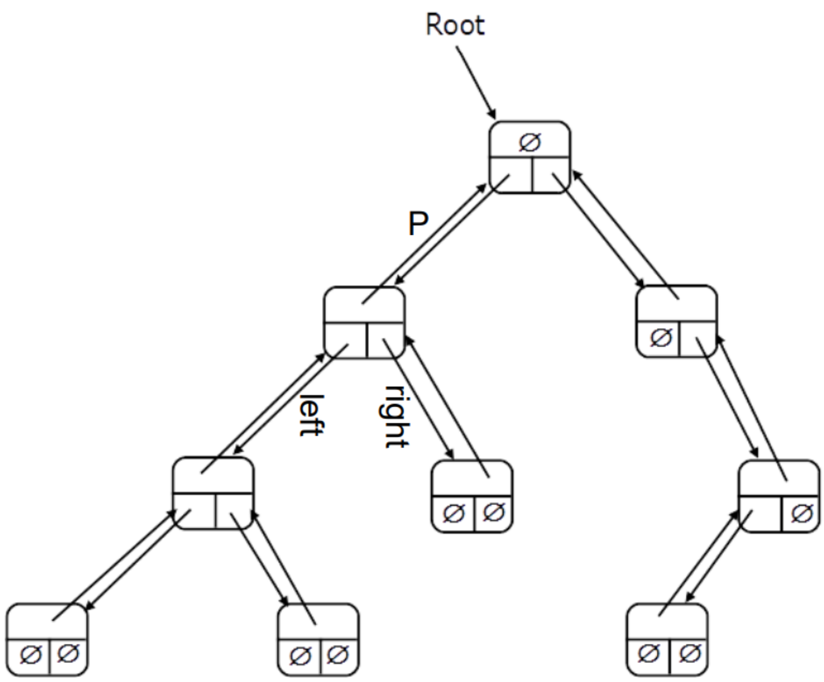
- If y is a node in the left subtree of x, then y.key≤x.key.
- If y is a node in the right subtree of x, then y.key≥x.key.

- Example: 2,5,5,6,7,8



How to represent a tree?

- Extend the idea of representing lists to representing trees.
- Each node has three pointers
  - “P”: points to its parent.
  - “Left”: points to its left child.
  - “Right”: points to its right child.
- A tree has a pointer “Root”
  - points to the root of the tree.



Tree Walks

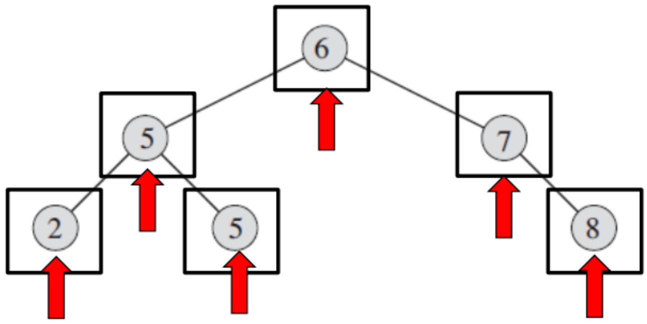
- Process of visiting each node in a tree data structure exactly once.
- Keys in the BST can be printed using “tree walks”.
- **Inorder** tree walk: The key of each node is visited (printed) between the keys in the left and right subtrees.

```
InorderTreeWalk(x)
if x ≠ NIL then
    InorderTreeWalk(x.left())
    print x.key()
    InorderTreeWalk(x.right())
```

- Divide-and conquer algorithm.

Inorder Tree Walk: example

- Can you write the output of running InorderTreeWalk(T.root) on the following tree?



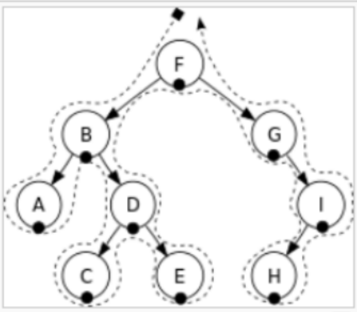

```
InorderTreeWalk(x)
if x ≠ NIL then
    InorderTreeWalk(x.left())
    print x.key()
    InorderTreeWalk(x.right())
```

- InorderTreeWalk on a BST prints all the keys in sorted order.

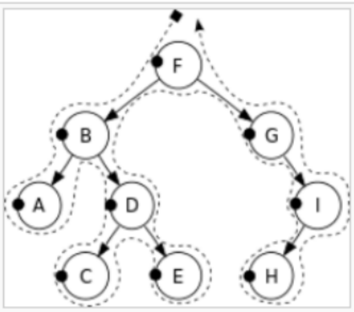
Breath first search (BFS) - left to right. (InorderTreeWalk)

Other Tree Walks

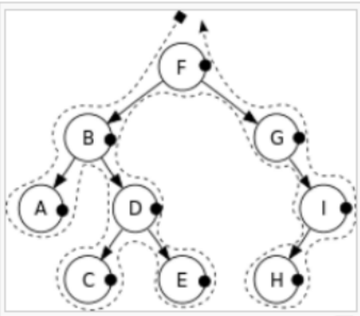
- A preorder tree walk visits each node before visiting its children. (is DFS (depth first search) - top to bottom. Not sure...)
- A postorder tree walk visits each node after visiting its children.



In-order: A, B, C, D, E, F, G, H, I



Pre-order: F, B, A, D, C, E, G, I, H



Post-order: A, C, E, D, B, H, I, G, F

Exercise 2 [5 points]

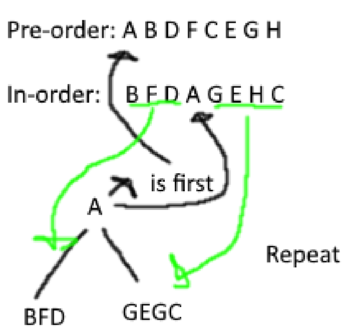
Given a binary tree T, its pre-order walk produces the following sequence:

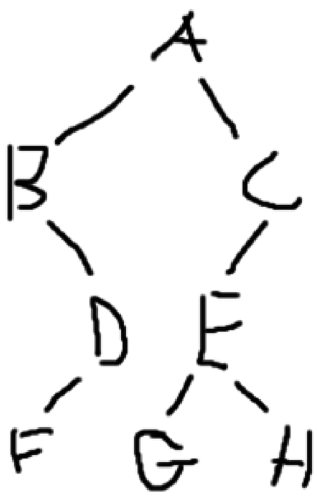
$$ABDFCEGH,$$

and its in-order walk produces the following sequence:

$$BFDAGEHC.$$

1. (*3 points*) Draw the tree T.
2. (*2 points*) Write down the tree's post-order walk sequence.





Search a BST

- To find an element with key k in a tree T
  - compare k with the root of the tree T.root.key
    - If k==T.root.key, return;
    - If k < T.root.key, search for k in x.left;
    - otherwise, search for k in x.right.

```
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)
```

A non-recursive version

```
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
```

- What shall we call in the beginning
    - (Interative-) Tree-Search(T.rood, k)

Search: Analysis

- When searching we simply go to the left child if we want a bigger number and to the right child for smaller number.
- What is the run time of searching an element with key k in a BST with n elements?
- Depending on the height of the BST h.
    - O(h).
- What is the worst case run time?
    - O(n) (When the tree is one long line of nodes)

Maximum and Minimum

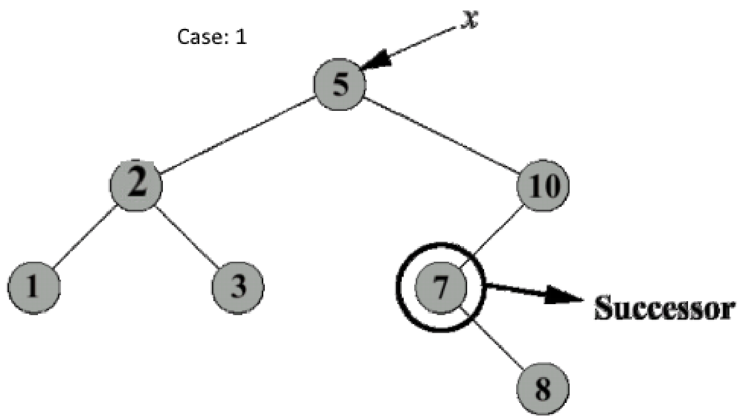
To find these we simple go all the way to the right or left. Keep going to either left or right child.

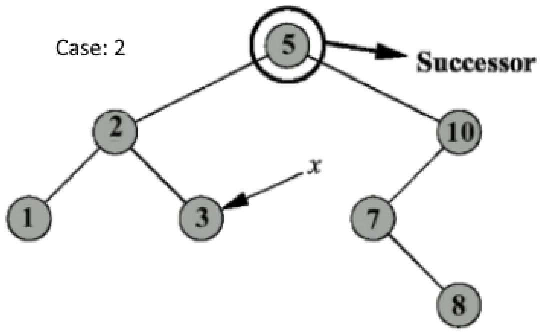
```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

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

Successor

- Given x, find the node with the smallest key greater than x.**key**
- We can distinguish two cases, depending on the right subtree of x
- Case 1: Right subtree of x is nonempty.
    - The successor is the smallest node in the right subtree.
    - This can be done by returning Minimum(x.**right**).
- Case 2: Right subtree of x is empty.
    - The successor is the lowest ancestor of x whose left child is also an ancestor of x





the smallest node in the right subtree

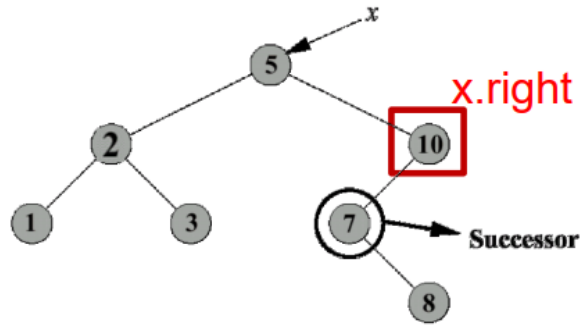
TREE-SUCCESSOR(x)

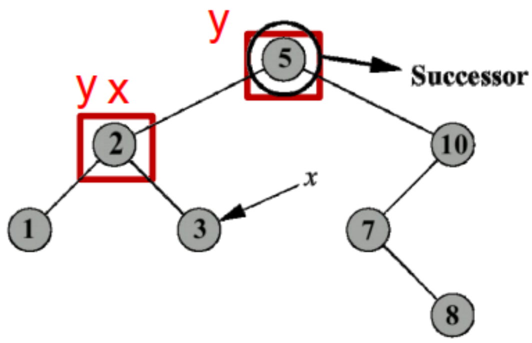
```
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

Case 1

Case 2

the lowest ancestor of *x* whose left child is also an ancestor of *x*





Insertion

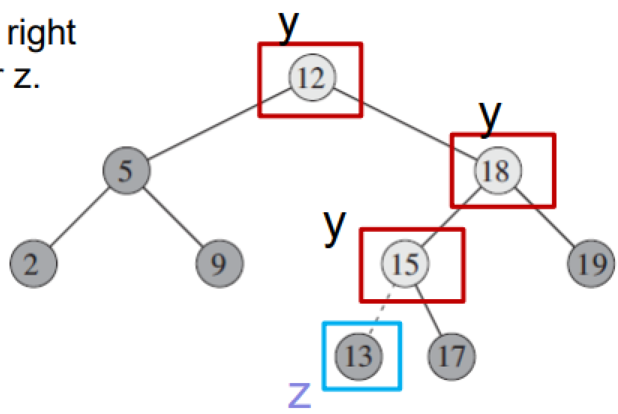
- The basic idea is similar to searching
  - Suppose we want to insert a new value v into the BST T.
  - Create a new node z (z.key=v, z.left=NIL, z.right=NIL, z.p=NIL)
  - find place in T where z belongs (as if searching for z.key),
  - And add z there
- The running on a tree of height h is O (h)

Find the right place for z.

TREE-INSERT(T, z)

```
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
    T.root = z        // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

Add z into the tree.

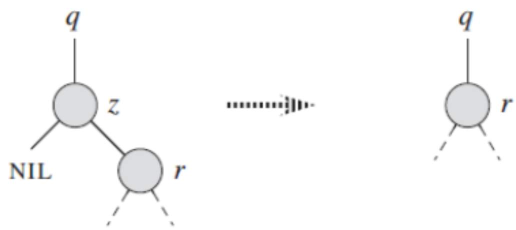


Insert 13 into the above tree.

Deletion

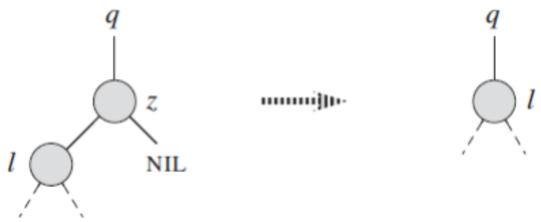
- Delete(T, z): Delete node z from BST T.

  - Case 1: If z has no left child, replace z by its right child.

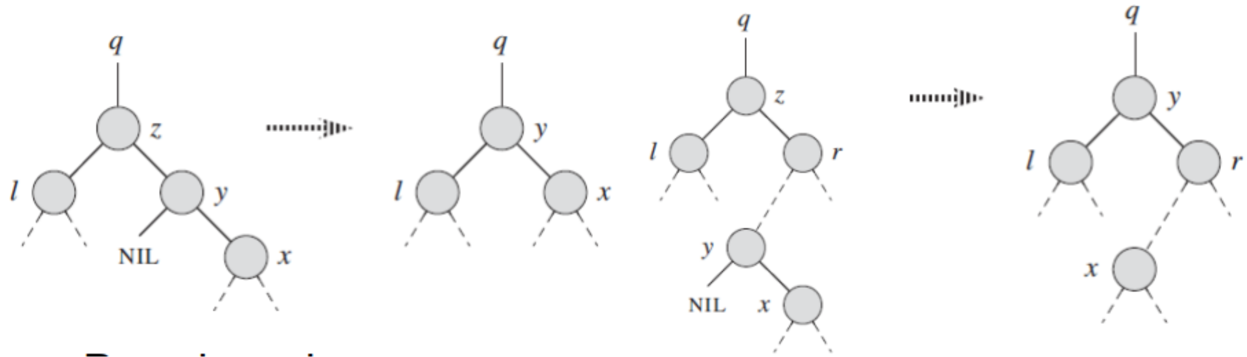


If z is the left subtree of its parent
    z.p.left=z.right
Else
    z.p.right=z.right
z.right.p=z.p

  - Case 2: if z only has left child (no right child), replace z by its left child.



- Otherwise, z has both left and right children. Find z's successor y, which must lie in z's right sub-tree and has no left child. (Recall the first case of Successor operation.)
- Case 3: if y is z's right child, replace z by y.
- Case 4: if not, replace y by its own right child, and then replace z by y.



- Pseudo code
    - Check the Transplant and Tree-Delete in CLRS, pp 296 – 298.