THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學

COMP2013
Data Structures and Algorithms

Lecture 7
Hashing and
Binary Search Tree I

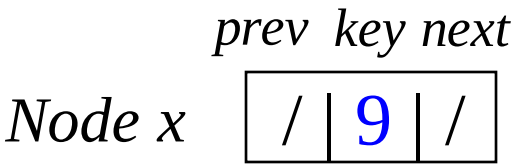
Dr. Jieming Shi

Outline

- **Revisit Linked List**
- Hash Tables
- Binary Search Tree I

Doubly Linked List

- In a doubly linked list, each node/element *x* stores:
  - *key*: key value of the node
  - *prev*: pointer to the previous node
  - *next*: pointer to the next node
- A pointer can be set to:
  - The memory address of another node in Node type, OR
  - `NIL` value (shown as '/' in examples)

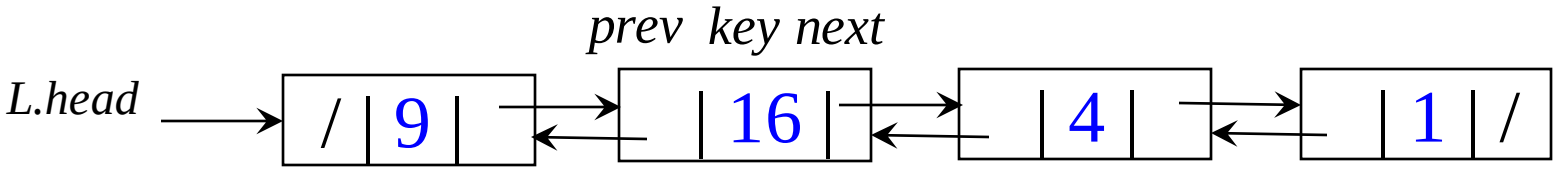



```
class Node {
  int key;
  Node* prev;
  Node* next;
}
```

Doubly Linked List *L*

- *L* is a chain of nodes
  - ***head*** is the first node in the linked list
    - *L.head* points to the head
  - First node: *prev* = `NIL`; Last node: *next* = `NIL`
  - Consecutive nodes *x* & *y* should satisfy:

$$x.next = y \quad \text{and} \quad y.prev = x$$

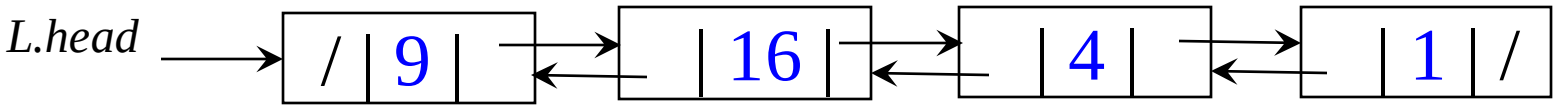


Doubly Linked List *L*

- Search key *k* in doubly linked list *L*

search (*L*, *k*)

1. $x \leftarrow L.head$
2. while $x \neq$ `NIL` and $x.key \neq k$
3. $x \leftarrow x.next$
4. return *x*

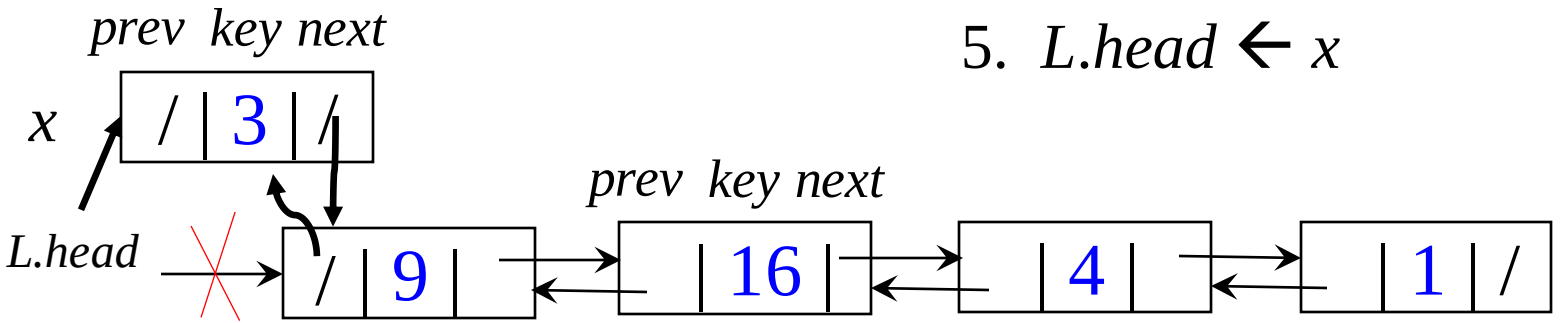


Doubly Linked List *L*

- Insert node *x* with key *k* to the front of the linked list *L*

Prepend-insert (*L*, *x*)

1. *x.prev* ← NIL
2. *x.next* ← *L.head*
3. if *L.head* ≠ NIL
4. *L.head.prev* ← *x*
5. *L.head* ← *x*

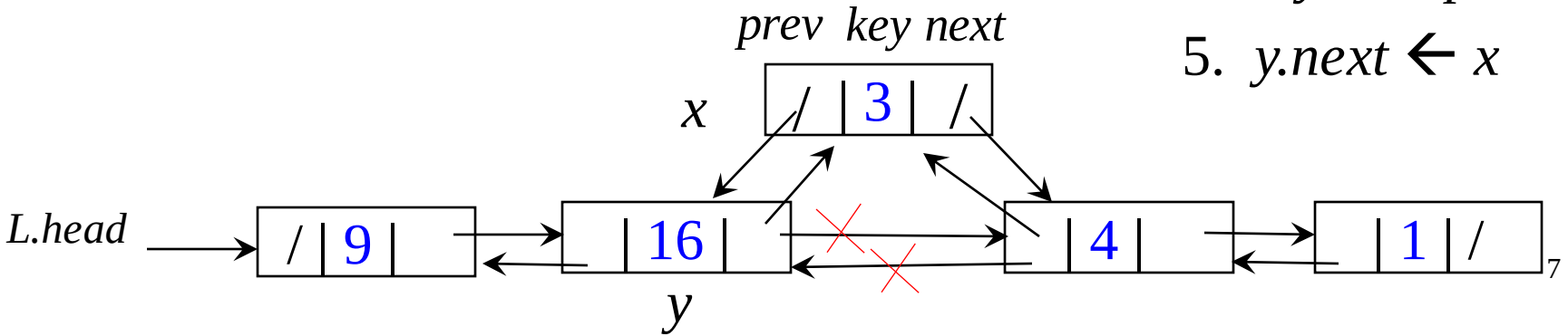


Doubly Linked List *L*

- Insert node *x* after y in the linked list *L*

Insert (*L*, *x*, *y*)

1. *x.prev* ← y
2. *x.next* ← *y.next*
3. if *y.next* ≠ NIL
4. *y.next.prev* ← *x*
5. *y.next* ← *x*

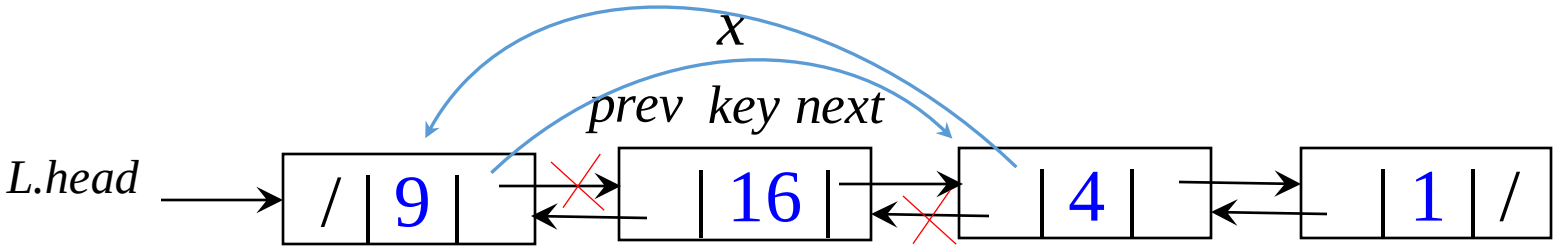


Doubly Linked List *L*

- Delete *node x* from *L*
  - Assume *x* exists in *L*
  - (If we want to remove a node with key *k*, we need to search it first)

Delete (*L*, *x*)

1. if *x.prev* ≠ `NIL`
2. *x.prev.next* ← *x.next*
3. else *L*.head ← *x.next*
4. if *x.next* ≠ `NIL`
5. *x.next.prev* ← *x.prev*

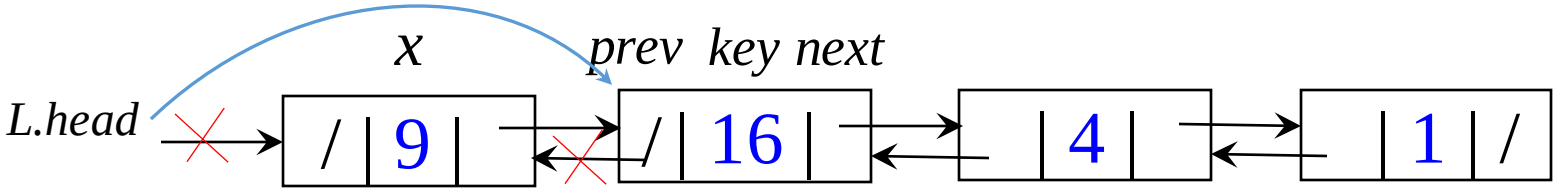


Doubly Linked List *L*

- Delete *node x* from *L*
  - Assume *x* exists in *L*
  - (If we want to remove a node with key *k*, we need to search it first)

Delete (*L*, *x*)

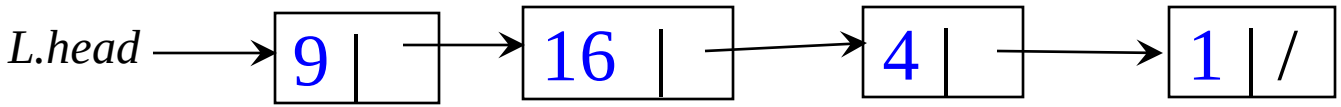
1. if *x.prev* ≠ `NIL`
2. *x.prev.next* ← *x.next*
3. else *L*.head ← *x.next*
4. if *x.next* ≠ `NIL`
5. *x.next.prev* ← *x.prev*



(Singly) Linked List

- In a singly linked list, each node stores *key*, *next* only
  - Require less space ☺
  - More complicated algorithm for deletion ☹

```
class Node {
  int key;
  Node* next;
}
```



Outline

- Revisit Linked List
- **Hash Tables**
  - A naïve way: Direct-address tables
  - Hash Table Concepts
  - Hash Table with Chaining
- Binary Search Tree

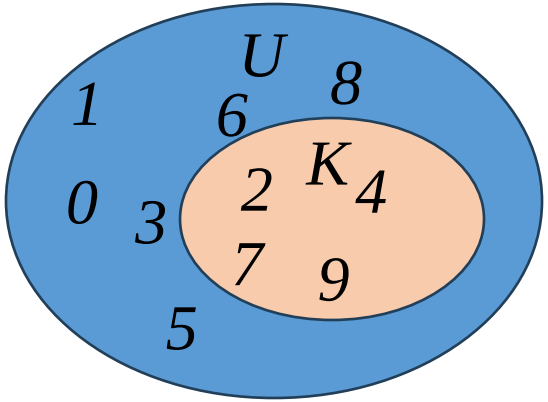
Manage a dynamic set of key values

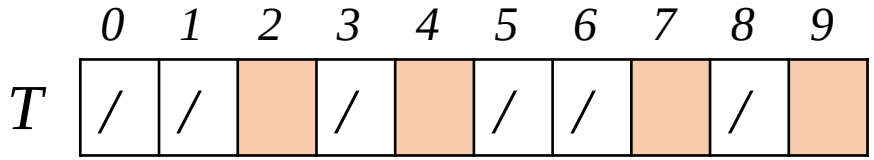
- Search and Delete and Insert
- Search in an unsorted array of size *n*
  - O(n)
- Search in a sorted array of size *n*
  - O(log n)

- *Can we have constant time search O(1)?*

Direct-address tables (A naive way)

- Universe $U = \{0, \ldots, M-1\}$
  - E.g., $U = \{0, \ldots, 9\}$
  - Any dynamic set K from the universe contains keys only from 0 to $M-1$
  - In other words, you know all possible keys in advance
- In a dynamic set K, how to quickly locate a key in O(1)?
  - Create an array (table) T of size $|U|=M$, and use the *index* to be *key*





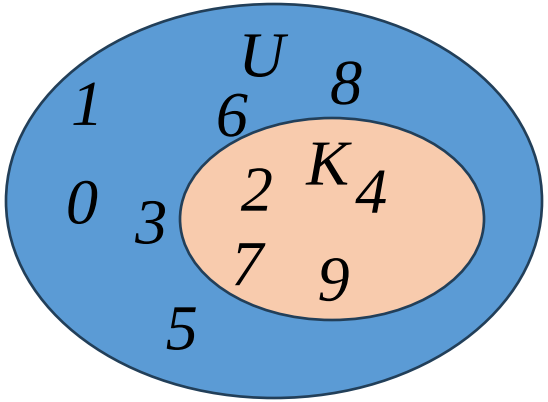
Direct-address tables (A naive way)

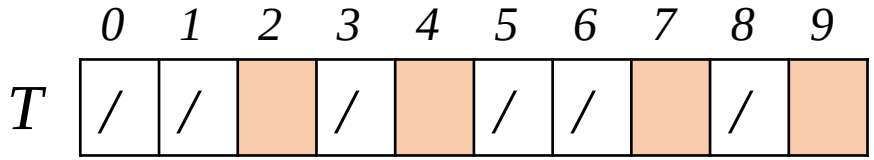
- Search(T, k): T[k]
- All in *worst-case* running time O(1)

Downside of direct addressing:

1. Universe U is large or infinite, and it is large space to store T, $O(|U|)$.
2. K is usually much smaller U: $|K| \ll |U|$

-> *A waste of space.*





Outline

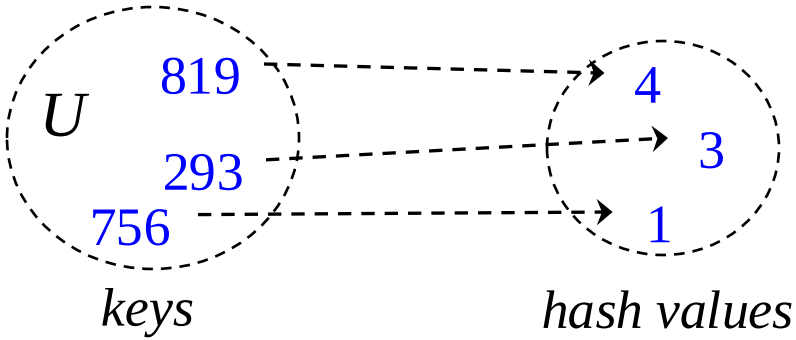
- Revisit Linked List
- Hash Tables
  - A naïve way: Direct-address tables
  - **Hash Table Concepts**
  - Hash Table with Chaining

Hash Tables

- When K is much smaller than U of all possible keys, a hash table
  - Supports *fast* search (*average-case* running time $O(1)$), while using *small space*

Hash Tables

- In a hash table $T[0, m-1]$
  - It has m positions (slots, buckets)
  - A key k's position (slot, bucket) in T is obtained by a hash function $h: U \rightarrow \{0, \ldots, m-1\}$
- An example of **hash function**: $h(k) = k \bmod m$
  - When $m=5$, what are the possible values of $h(k)$?
  - E.g., $h(293) = 293 \bmod 5 = 3$
  - Convert a key (of a large range in U) to a **hash value** (of a small range)



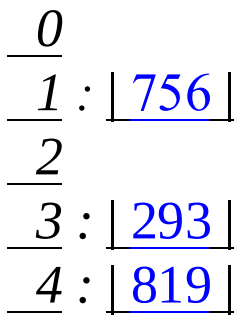
0
1 : | 756 |
2
3 : | 293 |
4 : | 819 |

hash table T with hash function $h(k)=k \bmod 5$

Hash Tables

- $h(k) = k \bmod m$, where $m=5$
- Given $k=293$, how to search it in hash table T?
  - Compute its backet (hash value) $h(293)=3$
  - $T[h(293)]=T[3]$
- Given $k=757$, how to search it in hash table T?
  - $h(757)=2$
  - $T[2]$ is empty, meaning nonexistence of 757 in T

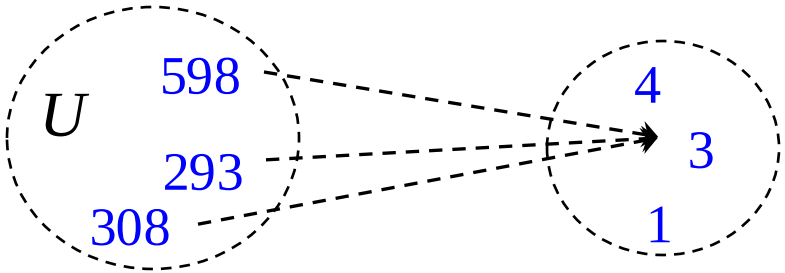
There are many different hash functions. We just use k mod m as an example



hash table T with hash function $h(k)=k \bmod 5$

Hash Tables

- Hash Table with hash function $h(k) = k \bmod 5$
- What will happen if different keys have the same hash value $h(k)$?
  - **Collision**



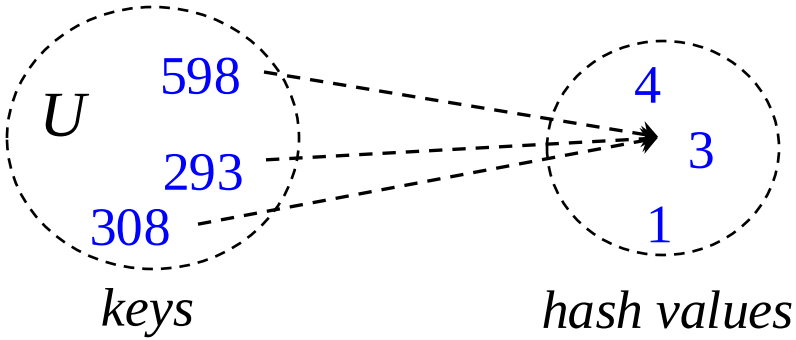
keys *hash values*

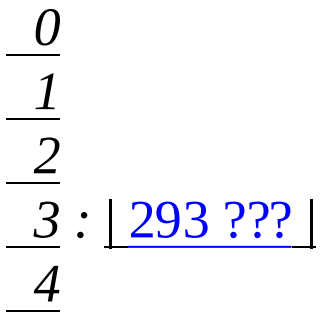
0	
1	
2	
3	293 ???
4	

hash table T

Reduce Collisions in Hash Tables

- Hash Table with hash function $h(k) = k \bmod m$
  - If $m < |U|$, collision is inevitable
- There are ways to reduce collisions
  - **Chaining**: use a linked list at each entry $T[k]$



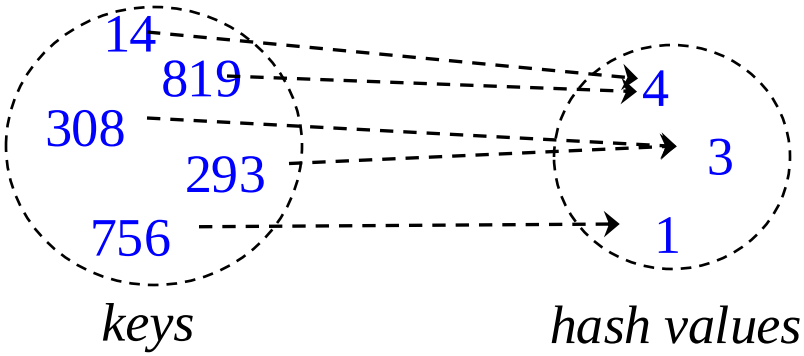


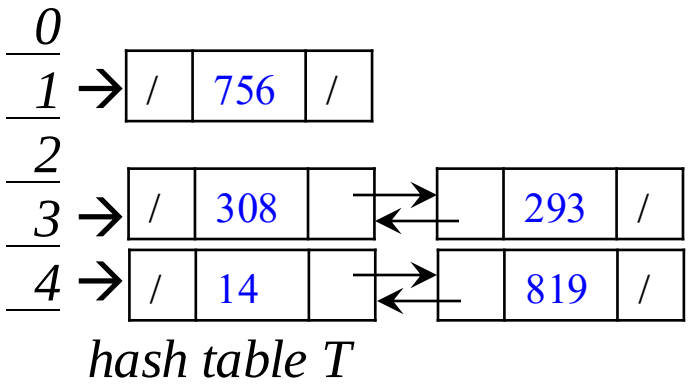
Outline

- Revisit Linked List
- Hash Tables
  - A naïve way: Direct-address tables
  - Hash Table Concepts
  - **Hash Table with Chaining**
- Binary Search Tree

Hash Table with Chaining

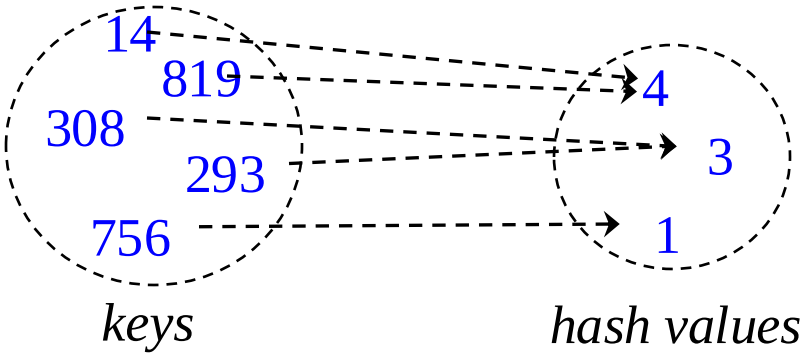
- Hash function $h(k) = k \bmod m$, e.g., $m = 5$
- An array of m buckets
- Each bucket is a *doubly linked list* of keys
  - It can also be singly
- The linked list in bucket $h(k)$ contains the collided keys

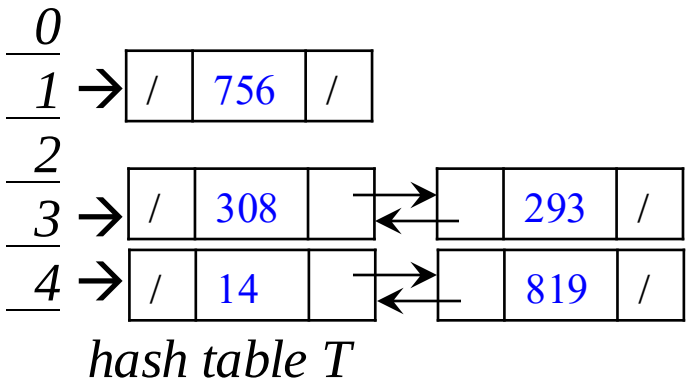




Hash Table with Chaining

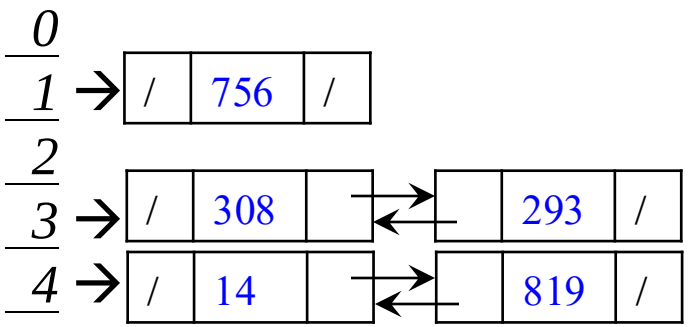
- Hash function $h(k) = k \bmod 5$
- Search k in T
  - Compute $h(k)$
  - Search in the linked list at the bucket $h(k)$
- Search 853
  - $h(853)=3$
  - Search the linked list at bucket 3
  - Key not found



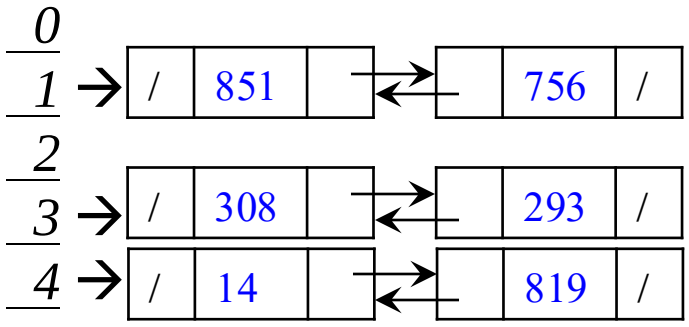


Hash Table with Chaining

- Hash function $h(k) = k \bmod 5$
- Insert k in T
  - Compute $h(k)$
  - (Prepend) Insert key k into the linked list at the bucket $h(k)$
    - Assume k does not exist in T yet
    - Without this assumption, you need to search if it exists first
- Insert *851*
  - $h(851)=1$
  - Insert 851 to the linked list at bucket 1



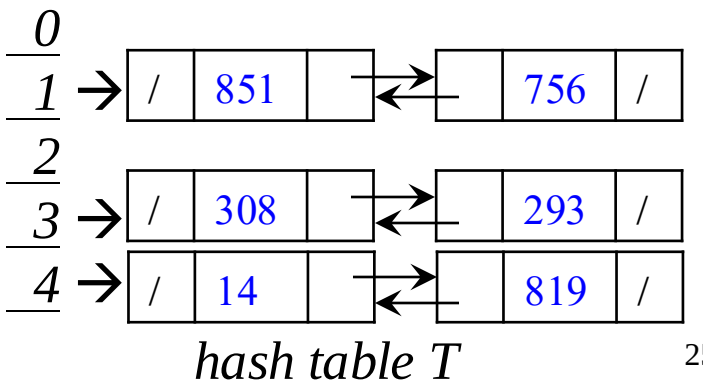
Before: hash table T



After: hash table T

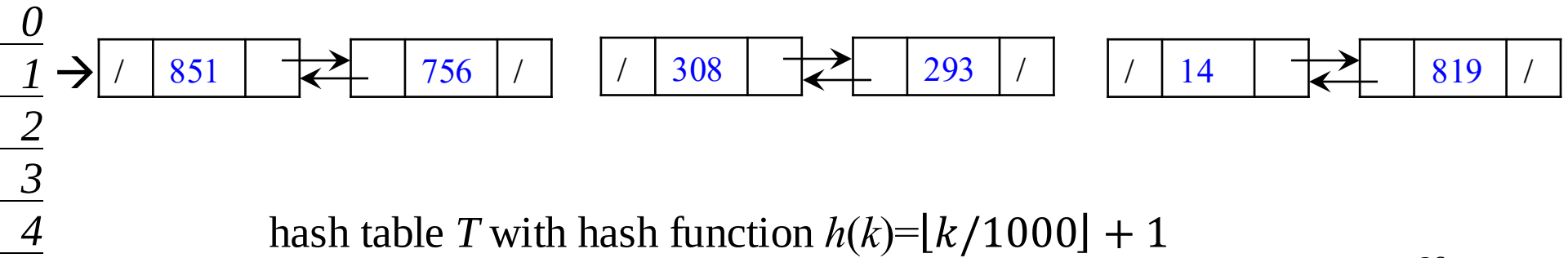
Hash Table with Chaining

- Hash function $h(k) = k \bmod 5$
- Delete k in T
  - Compute $h(k)$
  - *Search and delete* key k from the linked list at the bucket $h(k)$
- Delete 293
  - $h(293)=3$
  - In the linked list at bucket 3, search and delete the node with key 293



Hash Table with Chaining: Running time

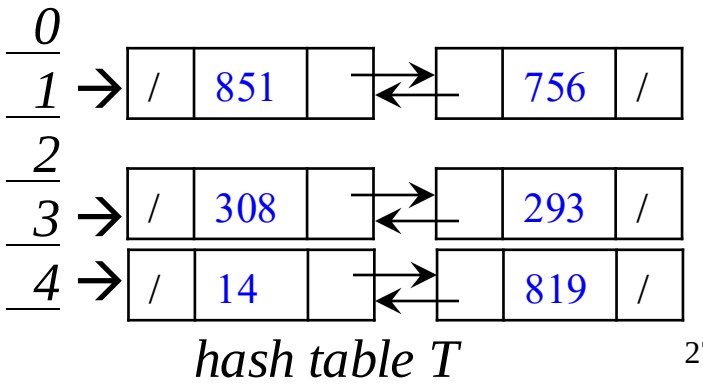
- There are *n* key values, and *m* buckets in the hash table
- **Worst case**:
  - all keys in one bucket (a terrible hash function)
  - Search O(n)
  - Delete O(n)
  - Insertion O(1) (just prepend to the linked list)



hash table T with hash function $h(k)=\lfloor k/1000 \rfloor + 1$

Hash Table with Chaining: Running time

- There are *n* key values, and *m* buckets in the hash table
- **Average case**:
  - all *n* keys evenly distributed in *m* buckets
  - **Load factor** $\alpha = n/m$
    - α can be smaller than 1
  - Search O(1+ α)
  - Delete O(1+ α)
  - Insertion O(1) (just prepend to the linked list)



Hash Tables

- Hashing is an important technique in advanced algorithm design
  - We only introduce its brief ideas

- How to design hash functions
- How to resolve collisions
- How to analyze the performance
- …

Outline

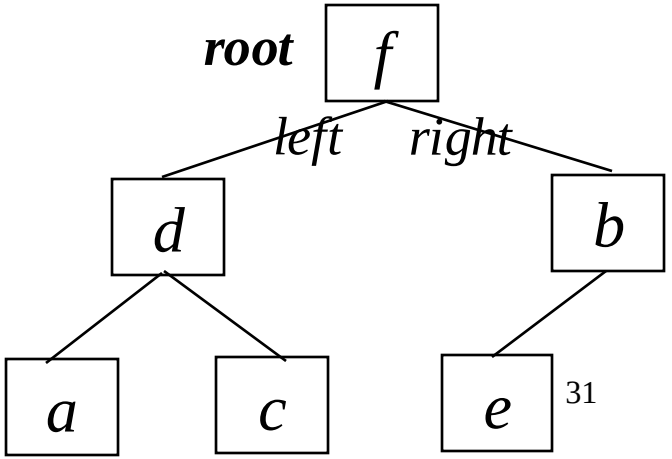
- Revisit Linked List
- Hash Tables
- Binary Search Tree
  - **Binary Tree**
  - Binary Tree Walks
  - Binary Search Tree (BST)
  - BST Operations

Binary Tree Applications

- Binary tree
  - A multi-level data structure
  - Manage a set *S* of values
    - E.g., {3,5,8,4,7,6}
- Applications
  - Searching: binary search trees
  - Binary space partition
  - Data compression: Huffman coding
  - Compilers: syntax tree
  - …

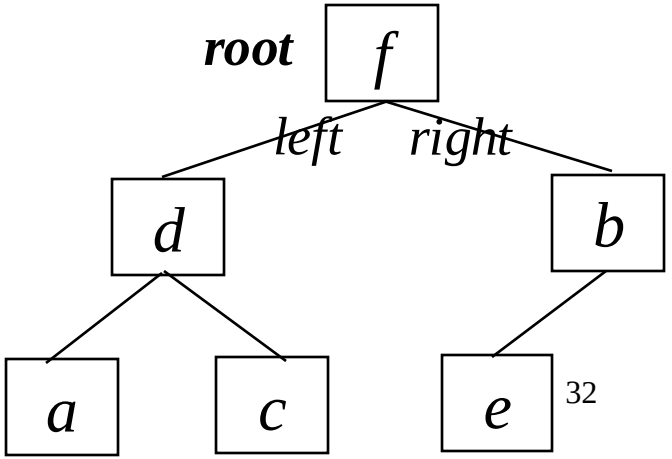
Binary Tree

- A binary tree *T* has a root node *T.root*
- Each node *x* stores these attributes:
  - *x.key*: key value in the node
  - *x.left*: pointer to left child node
  - *x.right*: pointer to right child node
  - *x.p*: pointer to the parent node
- Each node has at most 2 children
  - *x.left* = NIL, if x does not have left child;
  - *x.right* = NIL, if x does not have right child



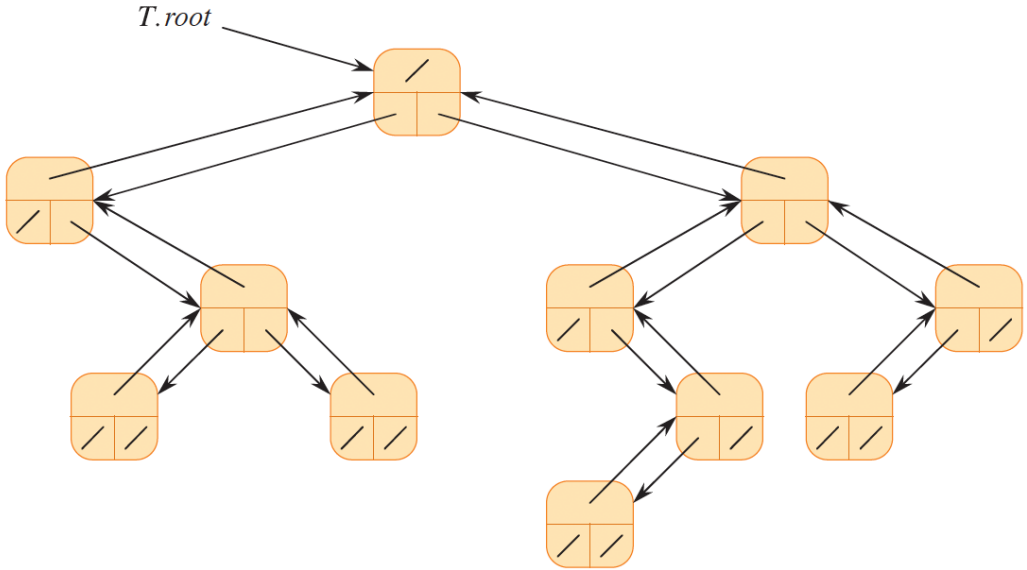
Binary Tree [Questions]

- Which key is stored in the root node?
- Find the left child of the node that stores “d”
- Find the right child of the node that stores “b”
- A *leaf* node does not have any child.
  - Which are leaf nodes?
- The left and right ***subtrees*** of *f*



Binary Tree: C++

- Binary tree



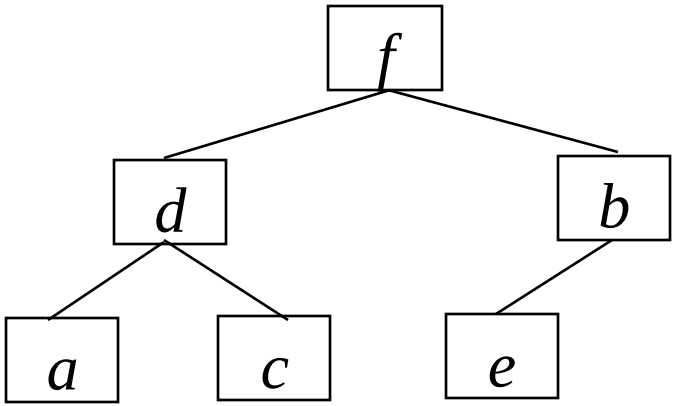

```
class Node {
   int key;
   Node* parent;
   Node* left;
   Node* right;
}
```

Outline

- Revisit Linked List
- Hash Tables
- Binary Search Tree
  - Binary Tree
  - **Binary Tree Walks**
  - Binary Search Tree (BST)
  - BST Operations

Binary Tree Walk (Traversal)

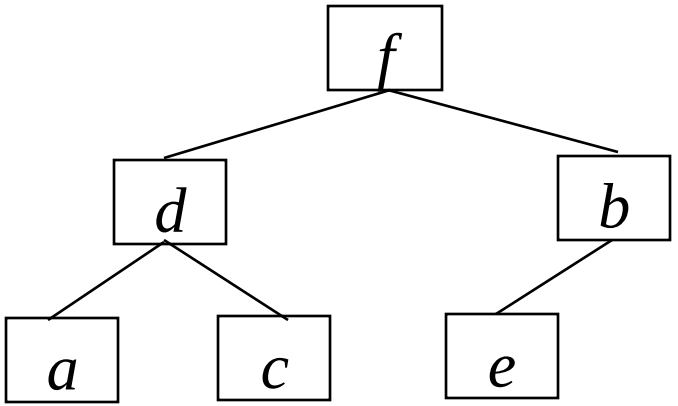
- Tree walk: visit all nodes of the tree based on a certain order
- Types of tree walk:
  - Inorder
  - Preorder
  - Postorder



Inorder Tree Walk

- Inorder: print a key between printing the key values in its left subtree and its right subtree
  - a d c f e b

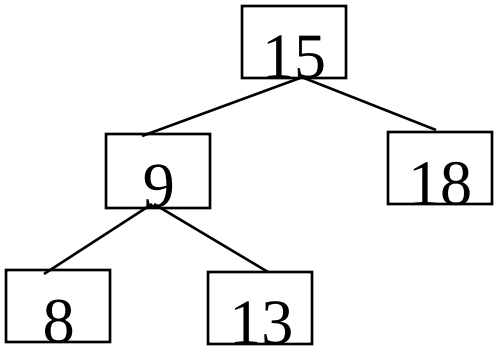
```
Inorder( x )
1. if  x ≠ NIL
2.     Inorder( x.left )
3.     print x.key
4.     Inorder( x.right )
```



Inorder Tree Walk

- Inorder: print a key between printing the key values in its left subtree and its right subtree
  - 8, 9, 13, 15, 18

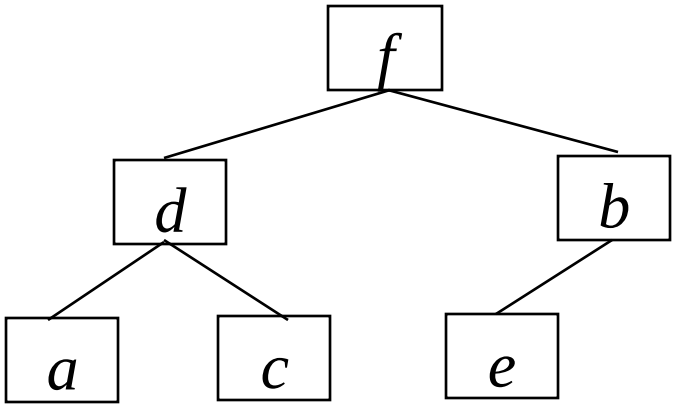
```
Inorder( x )
1. if  x ≠ NIL
2.     Inorder( x.left )
3.     print x.key
4.     Inorder( x.right )
```



Preorder Tree Walk

- Preorder: print a key before printing the key values in its left subtree and its right subtree
  - f d a c b e

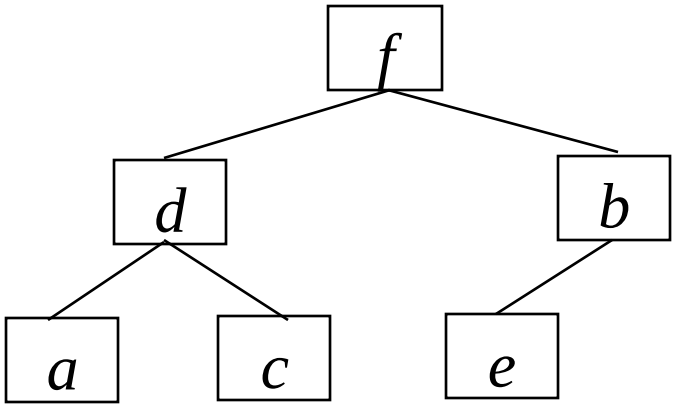
```
Preorder( x )
1. if  x ≠ NIL
2.     print x.key
3.     Preorder( x.left )
4.     Preorder( x.right )
```



Postorder Tree Walk

- Postorder: print a key after printing the key values in its left subtree and its right subtree
  - a c d e b f

```
Postorder( x )
1. if  x ≠ NIL
2.     Postorder( x.left )
3.     Postorder( x.right )
4.     print x.key
```



Inorder Tree Walk: Time Complexity $\boldsymbol{\Theta(n)}$

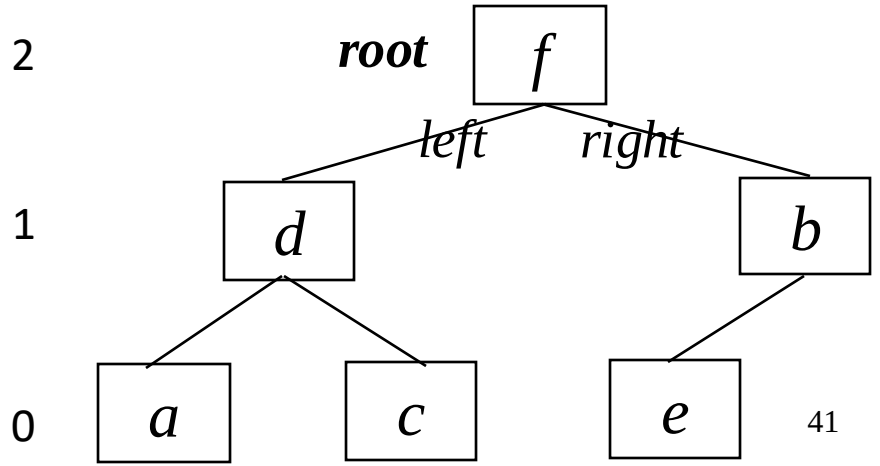
- Inorder(*T.root*) visits all *n* nodes of binary tree *T*
- Lower bound running time $\boldsymbol{T(n) = \Omega(n)}$
- We guess that time ***T(n) = O(n)*** and then prove if this is correct:
  - $T(n) = T(k) + T(n-k-1) + d$,
    - *k* is the number of nodes in left subtree, *d* is a constant cost on executing the function, exclusive the time spent in recursive calls
  - Proof: Substitution
  - Assume $T(i) \leq (c+d)\, i + c$ for all $i < n$
  - *Inductive case:*
  - ***i=n***: $Time(n)=T(k)+T(n-k-1)+d$
    $\leq (c+d)k+c + (c+d)(n-k-1)+c+d$
    $= (c+d)n+c$
  - Base case: We have $T(1) = c$. True.

```
Inorder( x )
1. if   x ≠ NIL
2.      Inorder( x.left )
3.      print x.key
4.      Inorder( x.right )
```

Tree Walk [Questions]

- How can we find the number of nodes in a tree?
- How do we find the “distance” from the root to each node?
- **Height** *h* of a binary tree (a node) is the largest downward path distance from the root (the node) to any leaf

Height = 2



Outline

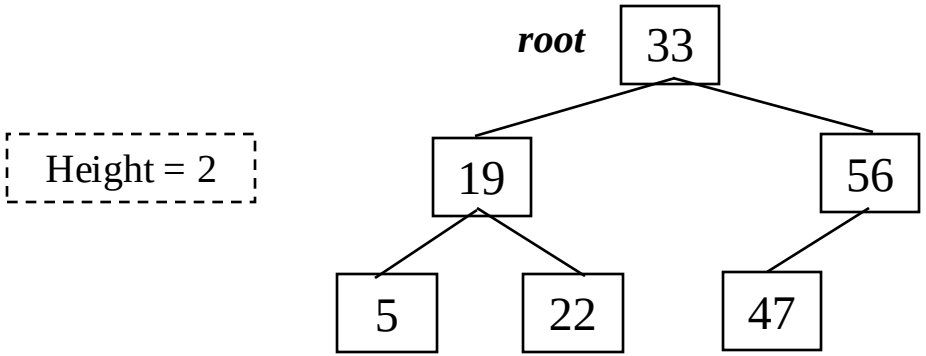
- Revisit Linked List
- Hash Tables
- Binary Search Tree
  - Binary Tree
  - Binary Tree Walks
  - **Binary Search Tree (BST)**
  - BST Operations

Binary Search Tree

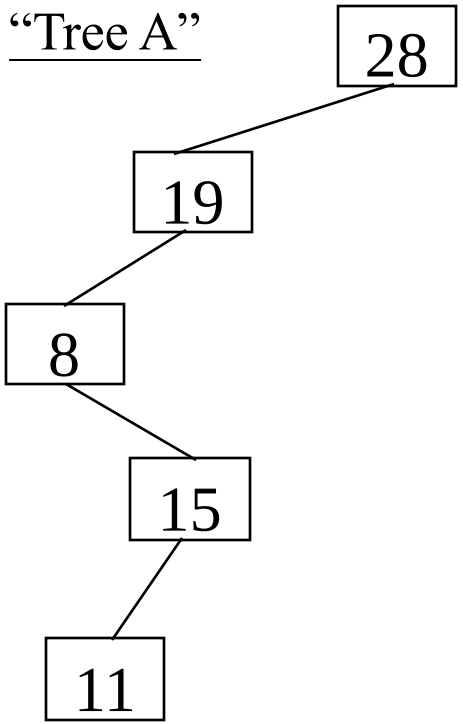
- *Efficiently* Maintain and Search a set *S* of values
  - Search
  - Minimum/Maximum
  - Insert/Delete
  - Successor/Predecessor

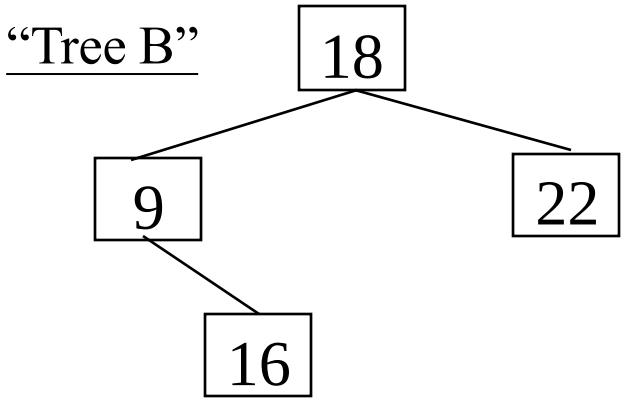
Binary Search Tree (BST)

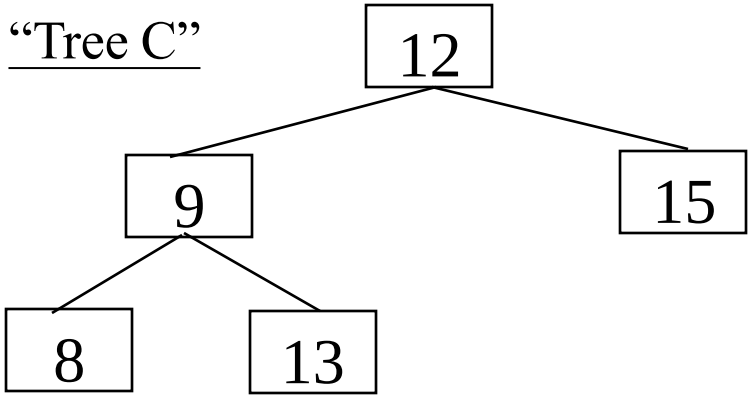
- **Binary search tree** is a binary tree with this property:
  For any node *x* in the tree,
  for any node *a* in *x*'s *left subtree*, $a.key \leq x.key$, **and**
  for any node *b* in *x*'s right subtree, $x.key \leq b.key$



Which are Binary Search Trees?

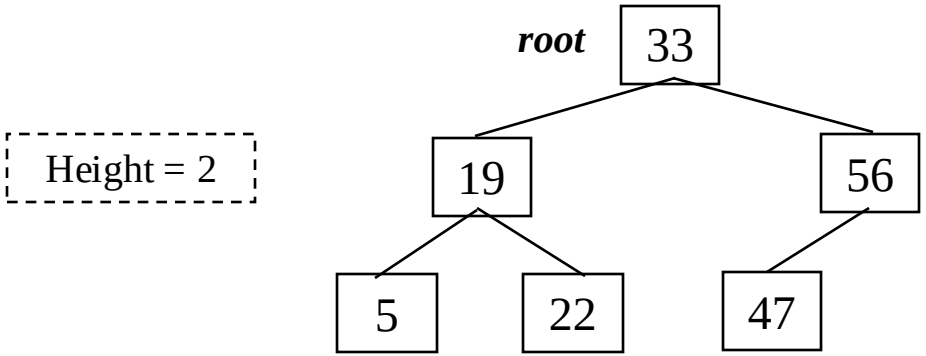






Inorder Walk of Binary Search Tree (BST)

- The inorder walk of the BST gives the key values of the nodes in sorted order.



Outline

- Revisit Linked List
- Hash Tables
- Binary Search Tree
  - Binary Tree
  - Binary Tree Walks
  - Binary Search Tree (BST)
  - **BST Operations**

BST Operations

Operation	***Complexity***	***Meaning***
Search	O(*h*)	Search a node with a key
Minimum	O(*h*)	Find the minimum node
Maximum	O(*h*)	Find the maximum node
Predecessor	O(*h*)	Find the predecessor node of a node
Successor	O(*h*)	Find the successor node of a node
Insert	O(*h*)	Insert a key
Delete	O(*h*)	Delete a key

Tree height: h, but h can be larger than $\log_2 n$

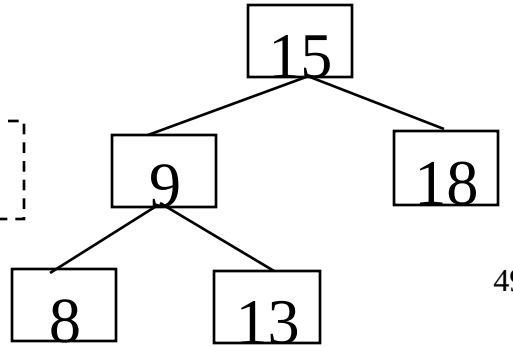
Search

- Find a node with key *k*
  - Return NIL if there is no such node
- Example: Search (*T.root*, *13*)
  - Visit the node “15”, go left
  - Visit the node “9”, go right
  - Visit the node “13”, key found!

Search (*x*, *k*)

1. if *x* = NIL or *k* = *x.key*
2. return *x*
3. if *k* < *x.key*
4. Search(*x.left* , *k*)
5. else
6. Search(*x.right* , *k*)

What are the steps for Search (*T.root* , 10)?

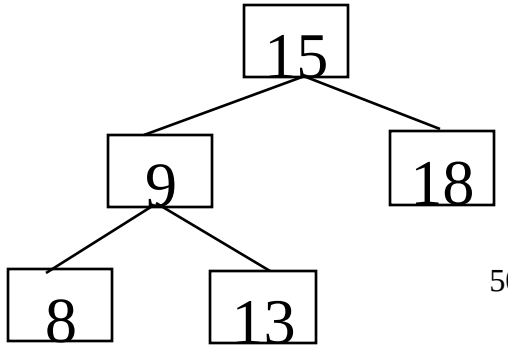


Search

- Time: O(*h*), where *h* is the tree height
- Why?
  - Each call goes down at most 1 level,
  - there are at most *h* levels

Search (*x*, *k*)

1. if *x* = NIL or *k* = *x.key*
2. return *x*
3. if *k* < *x.key*
4. Search(*x.left* , *k*)
5. else
6. Search(*x.right* , *k*)



Quiz during next lecture

- 50 minutes
  - (Second-half of the lecture)
  - Exact time can be found on blackboard
- 5 Questions
- You are allowed to bring 1 A4 paper (both sides) of reference materials
- You may need a calculator

- Thank you!