AAI2007 Introduction to Algorithms

Week 5: Priority Queue, Heap

Instructor: Jinyoung Han (jinyounghan@skku.edu)

Schedule

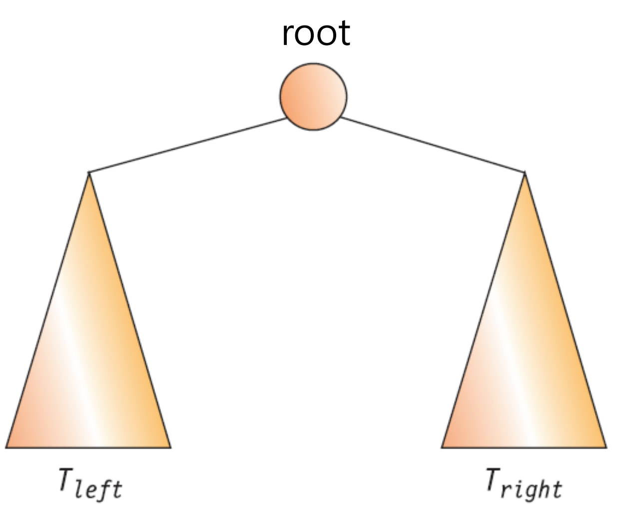
Tentative Schedule

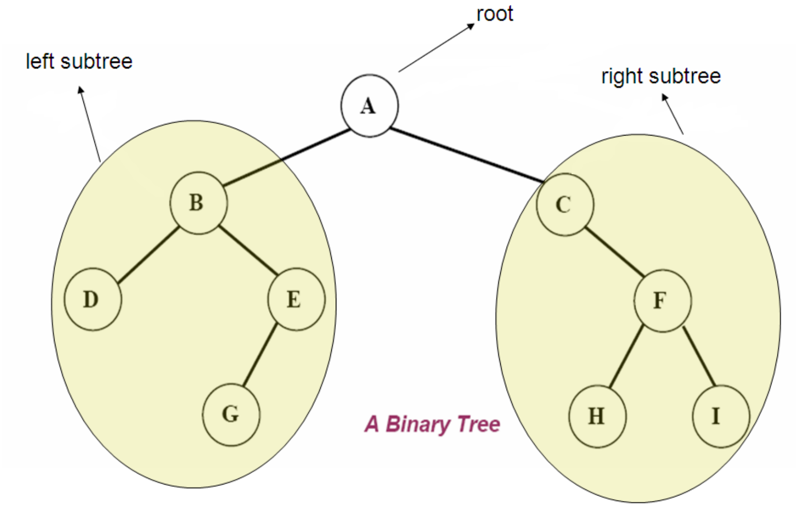
수업일	내용
9/4	Course Introduction, Algorithm Basic, Level Test
9/11	Order of Complexity, List
9/18	Stack, Queue
9/25	건학 기념일
10/2	Tree, Binary Search Tree (BST)
10/9	Priority Queue, Heap, Heap Sort 한글날
10/16	Hash Table
10/23	Graph Basic
10/30	Midterm Exam
11/6	Graph Algorithms
11/13	Sorting, Searching
11/20	Dynamic Programming (1)
11/27	Dynamic Programming (2)
12/4	Greedy Algorithms
12/11	Reserved
12/18	Final Exam

Tree Revisited

Binary Tree

- Binary tree
  - All the nodes have 2 subtrees
  - A finite set of nodes consisting of (i) empty set or (ii) root and left subtree and right subtrees
  - Degree of a node <= 2
    - Easy to implement



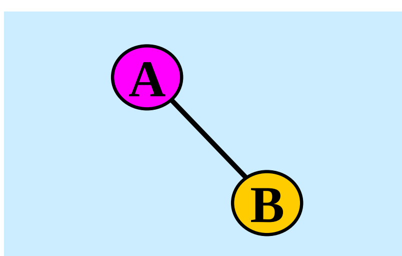


Tree Revisited

Binary Tree

- A subtree of a binary tree is either
  - (1) empty set or
  - (2) a finite set of nodes including a root, left subtree, and right subtree
  - Defined recursively
- An order exists between subtrees
  - E.g.,

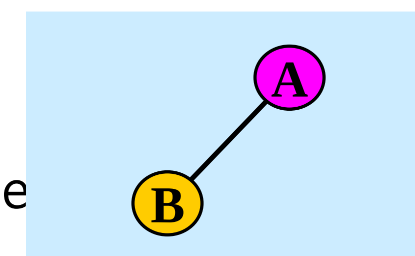
Empty left subtree



$\neq$

Different binary tree

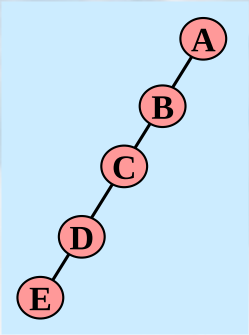
Empty right subtree

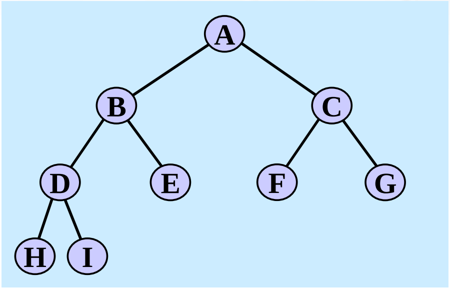


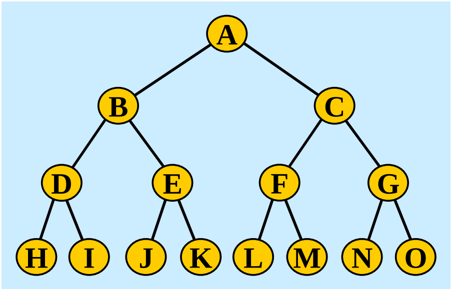
Tree Revisited

BT Types

- Skewed binary tree
  - Only have left-children – left skewed BT
  - Only have right-children – right skewed BT
- Complete binary tree
  - BT in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible
- Full binary tree
  - BT in which every node other than the leaves has two children
  - Full BT $\Rightarrow$ Complete BT
  - Full BT $\not\Leftarrow$ Complete BT





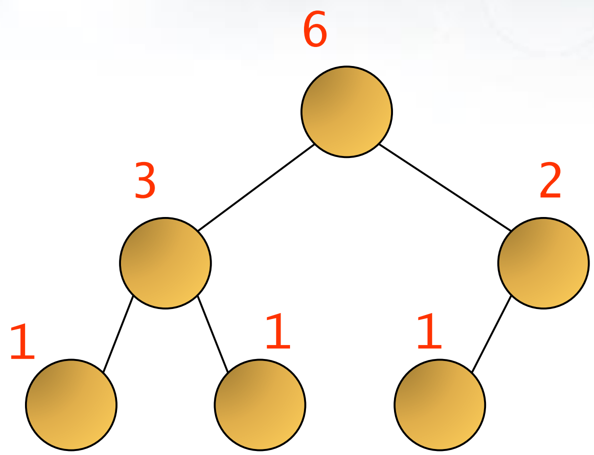


Tree Revisited

BT Operations

- Calculate the number of nodes in a tree
- Get the number of nodes in each subtree, and then summing them up plus one

```
def get_node_count(node):
   count=0
   if node is not None:
      count = 1 + get_node_count(node.left)+
                  get_node_count(node.right)
   return count
```

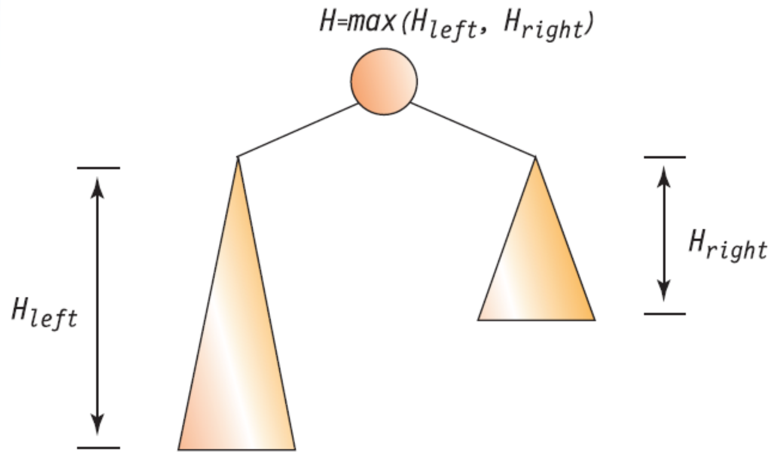


Tree Revisited

BT Operations

- Get the height of each subtree, and then return the maximum height plus one

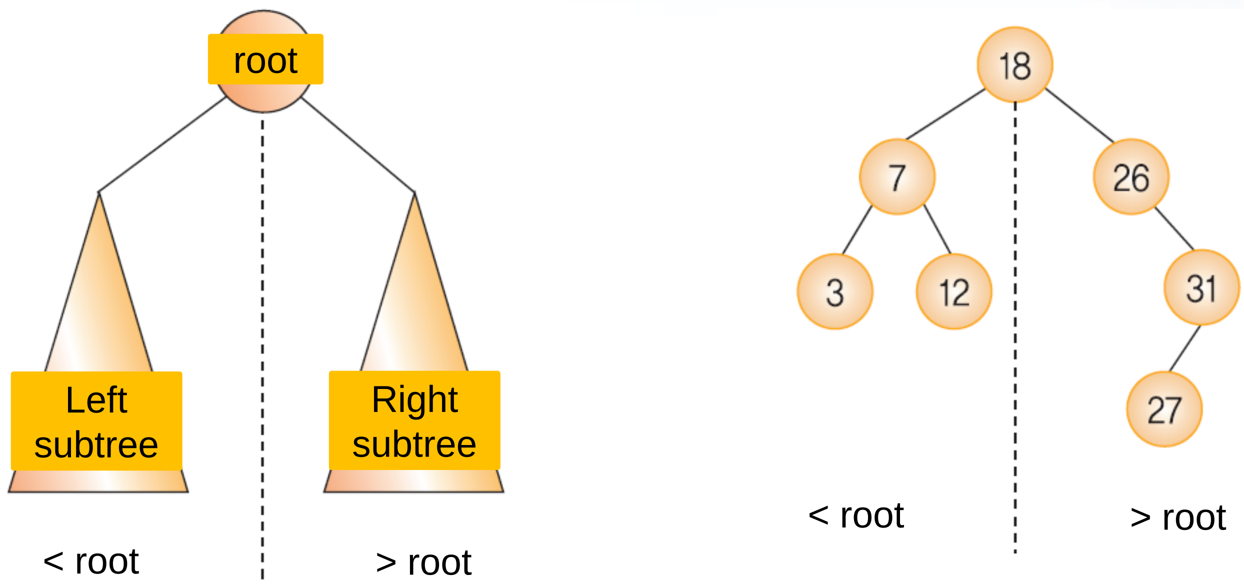
```
def get_height(node):
   height=0
   if node is not None:
      height = 1 + max(get_height(node.left),
                 get_height(node.right))
   return height
```



BST Revisited

BST

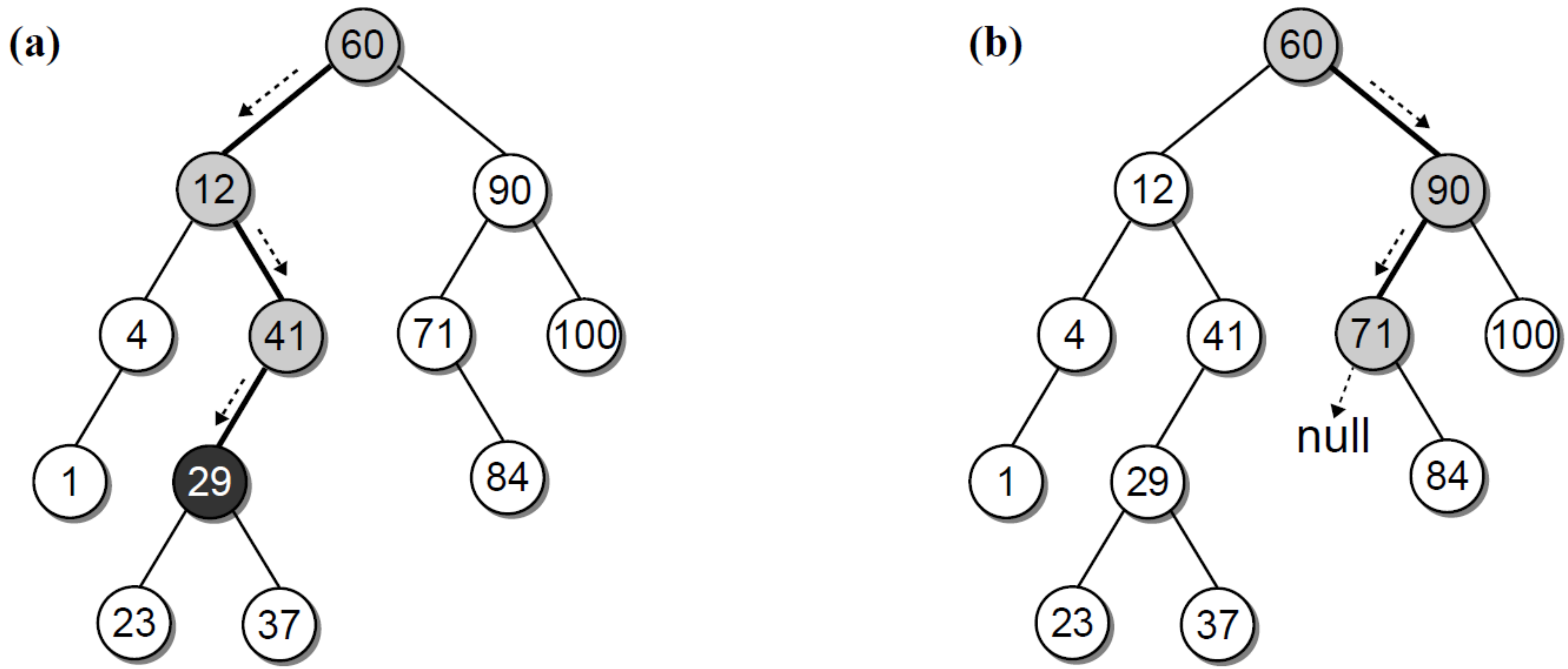
- A tree data structure for efficient “searching”
- key(left subtree)<key(root)<key(right subtree)
  - Key should be unique



BST Revisited

BST

- Searching a BST

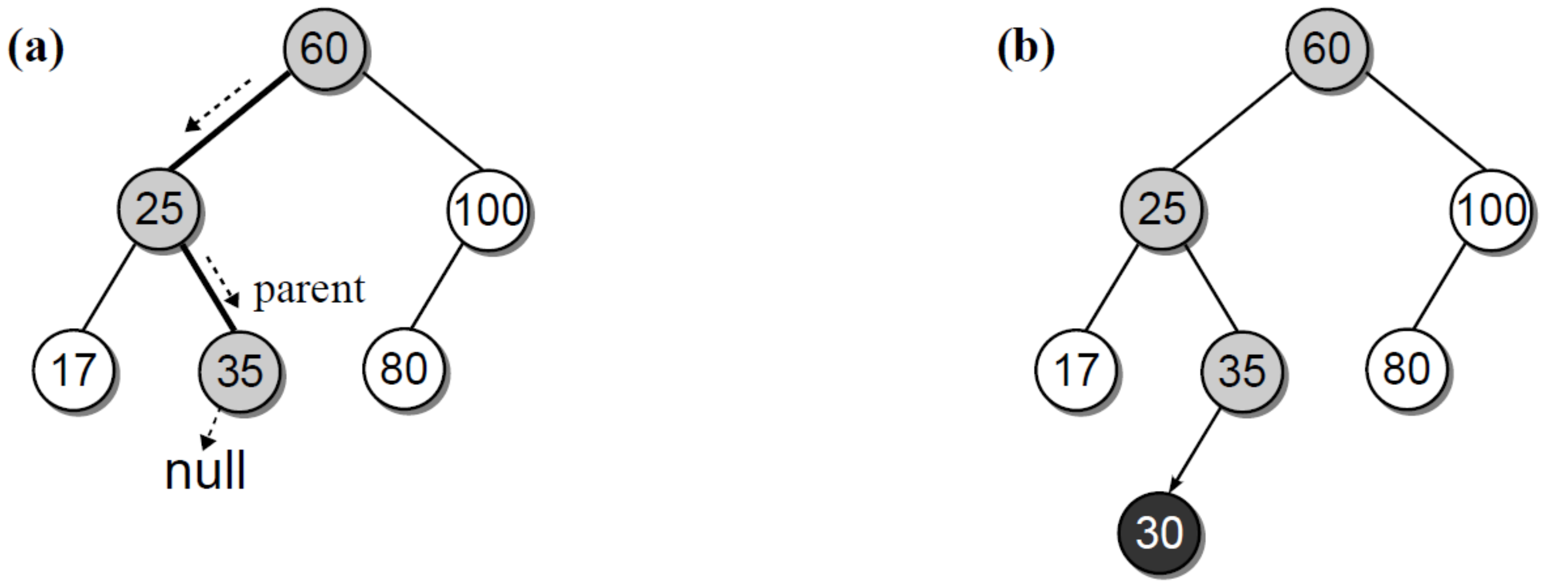


Searching a binary search tree: (a) successful search for 29 and (b) unsuccessful search for 68

BST Revisited

Insertion

- Algorithm – more illustration



Inserting a new node into a binary search tree:
(a) searching for the node's location
(b) linking the new node into the tree

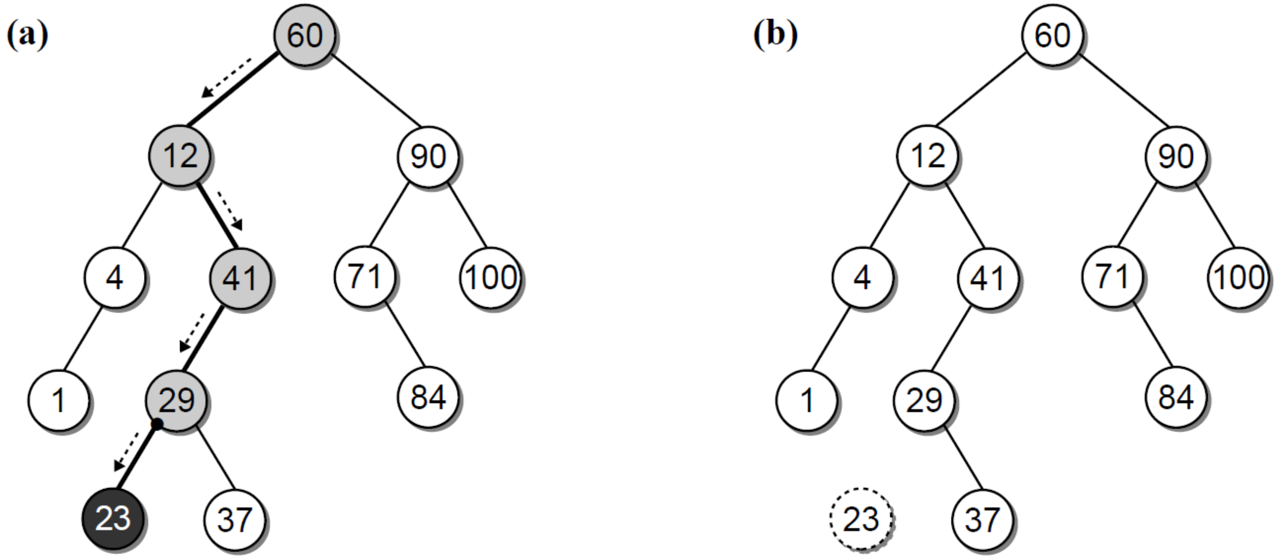
BST Revisited

Deletion

- Algorithm: Considering the location of the node
  - Case 1: The node is a leaf
  - Case 2: The node has a single child
  - Case 3: The node has two children

Removing a leaf node from a BST (a) finding the node and unlinking it from its parent; (b) the tree after removing 23

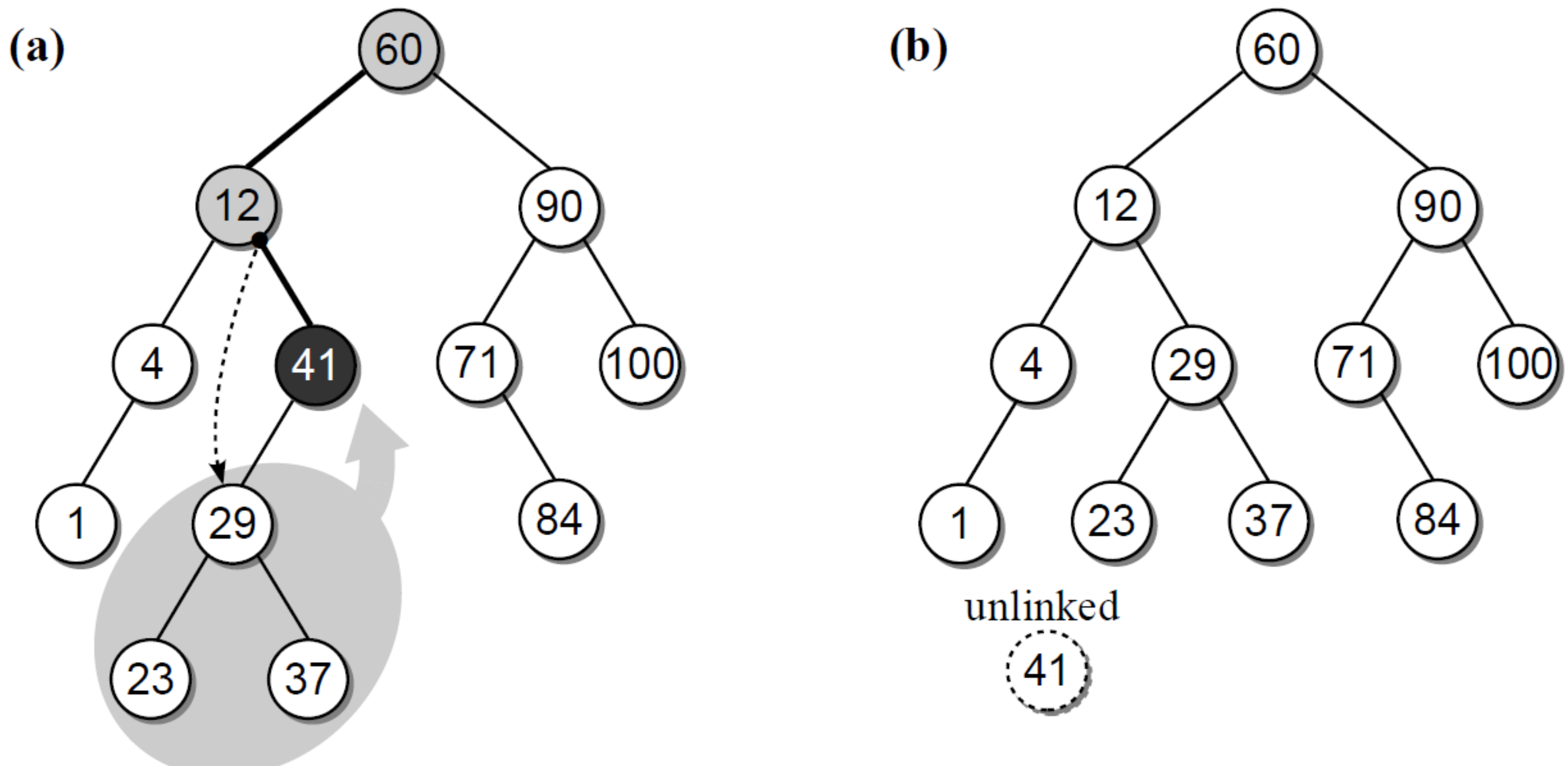
- Case 1: Removing a Leaf Node



BST Revisited

Deletion

- Case 2: Removing an Interior Node with One Child

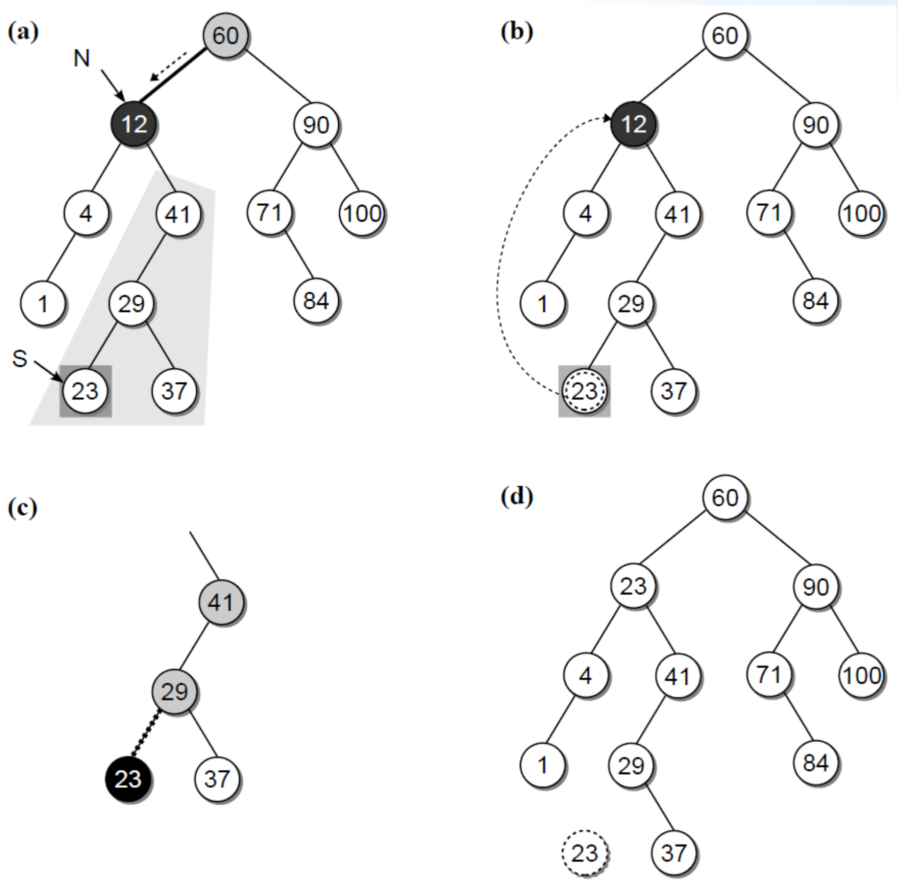


Removing an interior node (41) with one child
(a) redirecting the link from the node's parent to its child subtree
(b) the tree after removing 41

BST Revisited

Deletion

- Case 3: Removing an Interior Node with Two Children
  - Bring the most similar node to the deleted location



The steps in removing a key from a BST
(a) find the node, N, and its successor, S
(b) copy the successor key from node N to S
(c) remove the successor key from the right subtree of N
(d) the tree after removing 12

In This Lecture

Outline

1. Priority Queue
2. Heap
3. Heap Sort

01. Priority Queue

Priority Queue

- Priority queue
  - Queue with priority
    - Data with high priority is dequeued first, instead of FIFO order

High priority　　　　　Low priority

01. Priority Queue

Priority Queue

- Priority queue
  - A general queue
  - Can implement stack or FIFO queue

Data Structure	Dequeued data
Stack	Most recent data
Queue	First added data
P-queue	Data with highest priority

- Application
  - Simulation system (priority: time)
  - Network traffic control (e.g., QoS)
  - Job scheduling in OS

01. Priority Queue

ADT

·Object: a set of data with priority

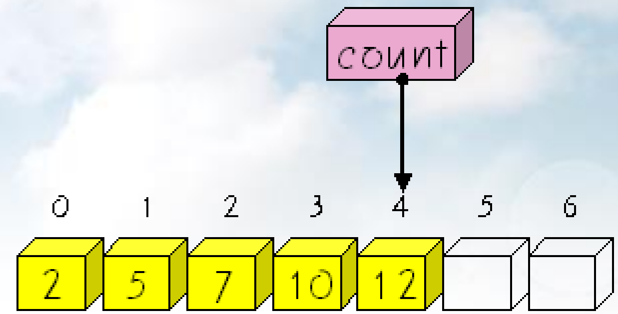
·Operations:

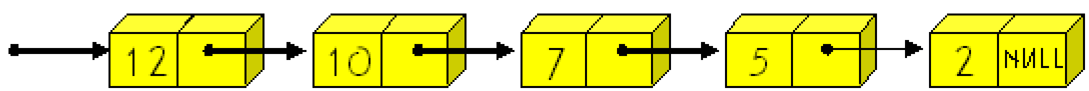
- create() ::= create p-queue ‘q’
- init(q) ::= initialize q
- is_empty(q) ::= check whether q is empty
- is_full(q) ::= check whether q is full
- insert(q, x) ::= insert x into q
- delete(q) ::= return the data with highest priority, and delete i
- find(q) ::= return the data with highest priority

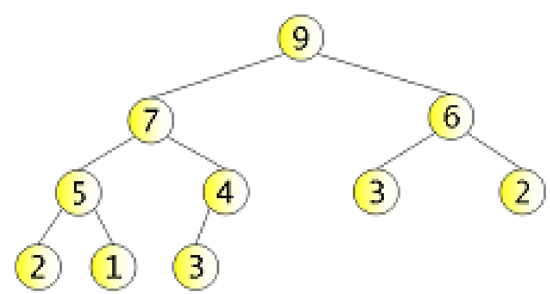
01. Priority Queue

Implementation

- Using an array
- Using a linked list
- Using a ‘Heap’







01. Priority Queue

Implementation

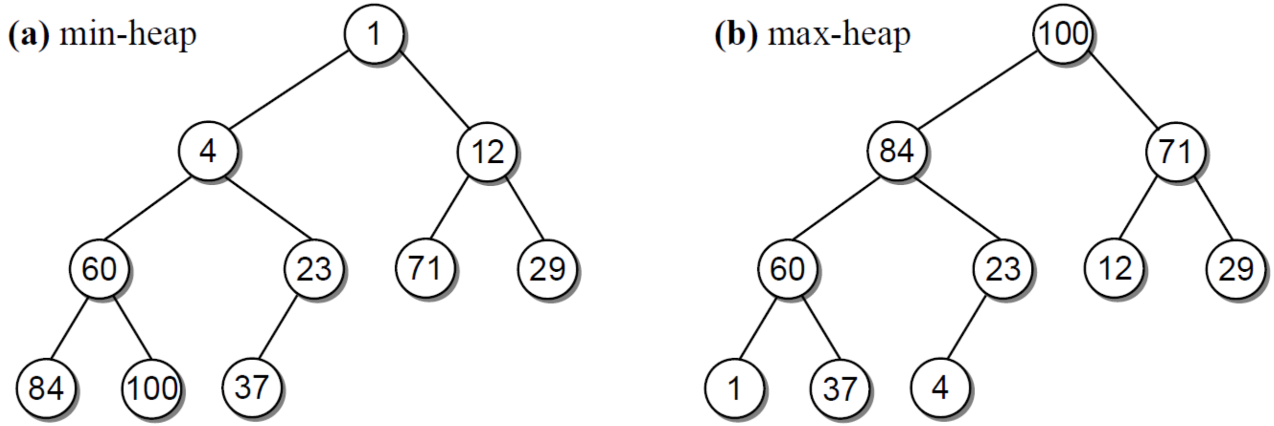
Data representation	*Insertion*	*Deletion*
unordered array	*O(1)*	*O(n)*
unordered linked list	*O(1)*	*O(n)*
ordered array	*O(n)*	*O(1)*
ordered linked list	*O(n)*	*O(1)*
heap	*O(logn)*	*O(logn)*

Heap

02. Heap

Heap

- Heap
  - A complete binary tree in which the nodes are organized based on their data entry values
- Two variants of the heap structure
  - A max-heap
    - For each non-leaf node V, the value in V is greater than the value of its two children
    - The largest value in a max-heap will always be stored in the root while the smallest values will be stored in the leaf nodes
  - A min-heap
    - For each non-leaf node V, the value in V is smaller than the value of its two children



02. Heap

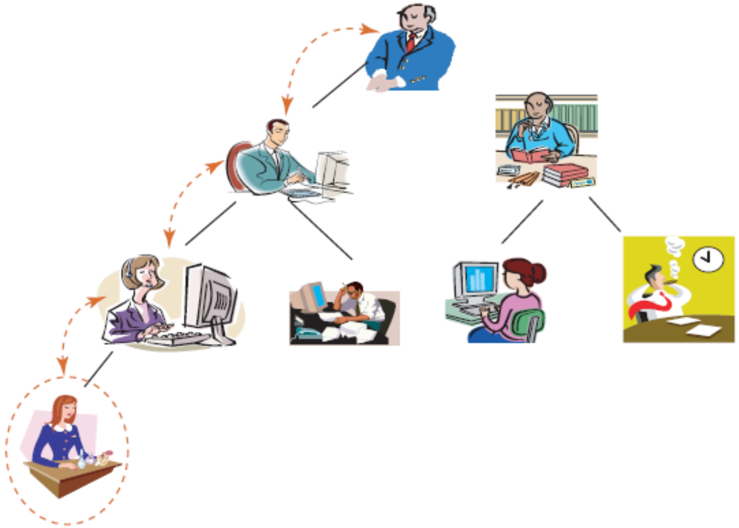
Implementation

- Using an array
  - A number (associating with an index of the array) is assigned to each node
- Locating child node is easy
  - Left-child index: i * 2
  - Right-child index: i * 2 + 1

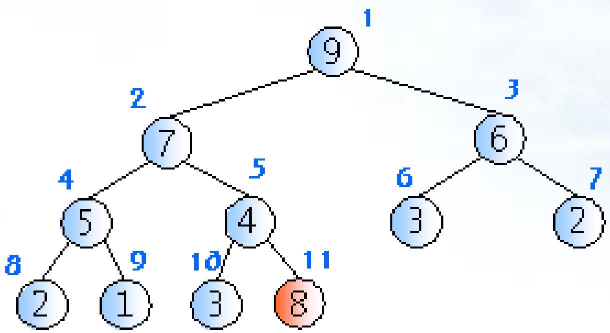
02. Heap

Insertion

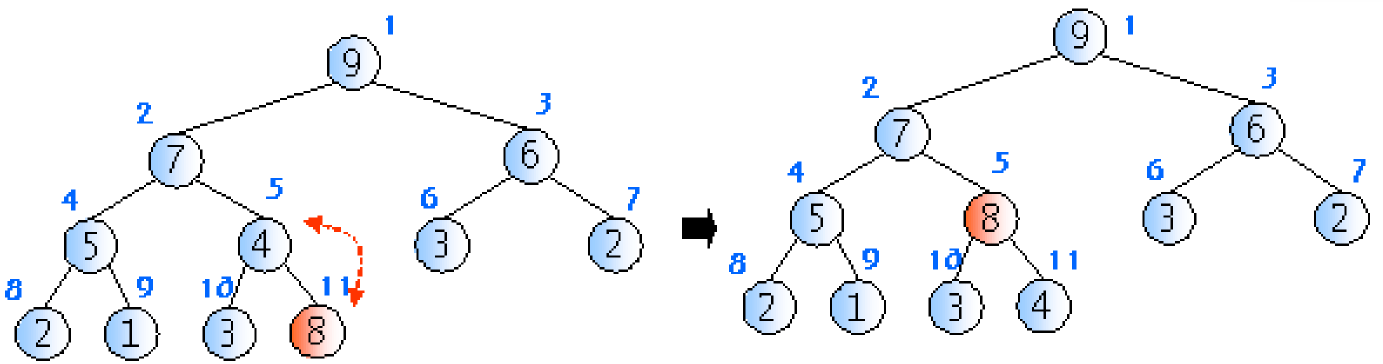
- Insertion
  - When a new value is inserted into a heap, the heap order property and the heap shape property (a complete binary tree) must be maintained
  - Similar to a promotion process from low-level employees to high-level ones
- Algorithm
  - Insert a new node into the last position
  - Exchange it with its parent nodes until heap property satisfied

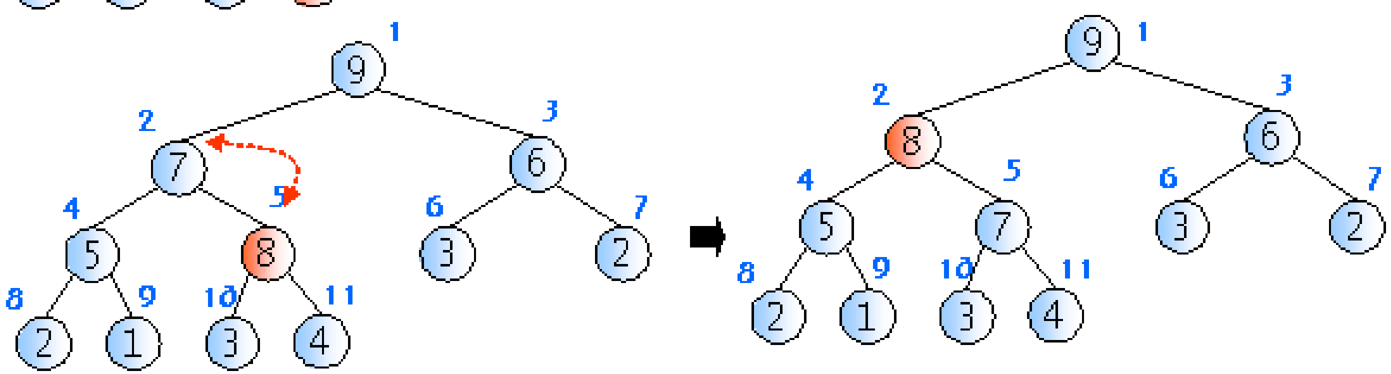


Insertion



- If new added node makes the tree non-heap-property, ‘upheap’
- ‘Upheap’: from the added node to the root, compare k and its parent nodes
- If k is smaller than its parent, finish
- The height of heap is O(log n), thus upheap takes O(log n)





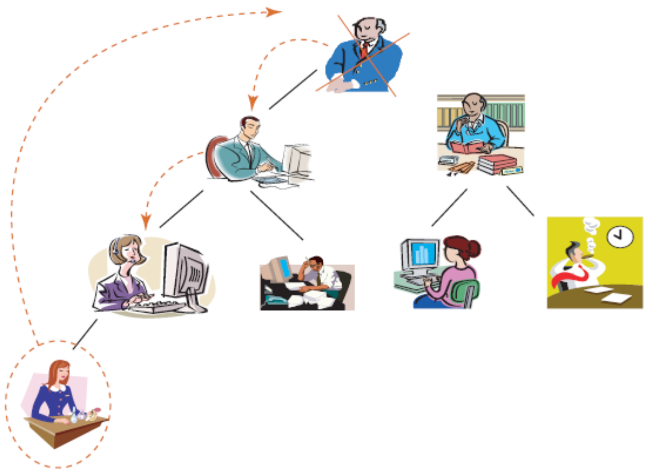
Insertion

```
insert_max_heap(A, key)



 heap_size ← heap_size + 1;
 i ← heap_size;
 A[i] ← key;
 while i ≠ 1 and A[i] > A[PARENT(i)] do
                    A[i] ↔ A[PARENT];
                    i ← PARENT(i);
```

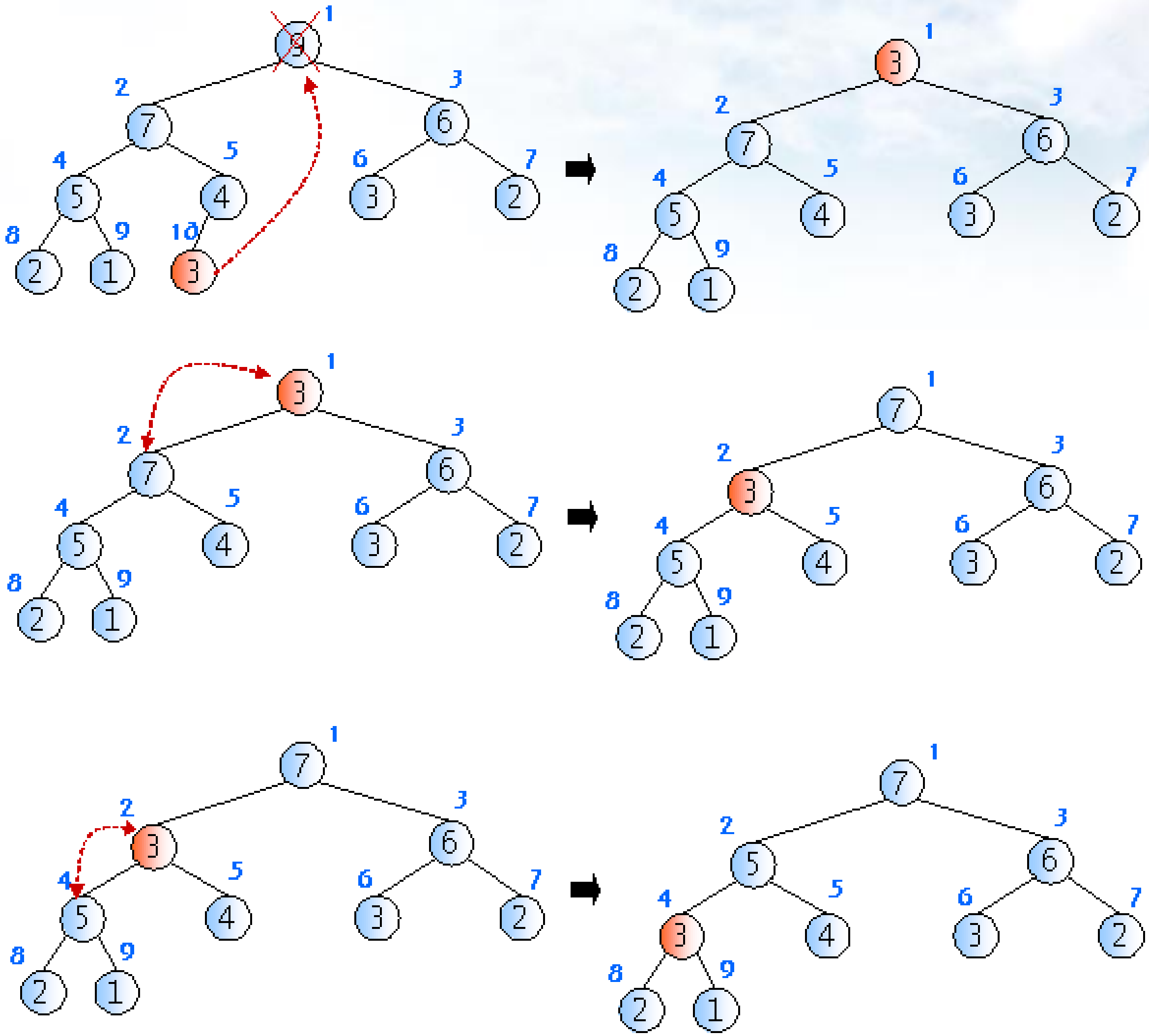
02. Heap

Deletion

- Deletion
  - When a value is extracted and removed from the heap, it can only come from the root node
    - In a max-heap, always the largest value is extracted
    - In a min-heap, the smallest value is extracted
  - After the value in the root has been removed, the binary tree is no longer a heap since there is now a gap in the root node
    - If boss position is empty, the lowest-level employee moves to the boss position, and then downgrade it
- Algorithm
  - Remove the root
  - Move the last node to the root
  - Compare it with its child



Deletion



- Downheap takes O(log n) as the height is O(log n)

Deletion

```
delete_max_heap(A)

item ← A[1];
A[1] ← A[heap_size];
heap_size←heap_size-1;
i ← 2;
while i ≤ heap_size do
        if i < heap_size and A[LEFT(i)] > A[RIGHT(i)]
                then largest ← LEFT(i);
                else largest ← RIGHT(i);
        if A[PARENT(largest)] > A[largest]
                then break;
        A[PARENT(largest)] ↔ A[largest];
        i ← CHILD(largest);
return item;
```

Time Complexity

- Insertion
  - In the worst case, the newly added node should move to the root, thus it takes O(log n)

- Deletion
  - In the worst case, the chosen node should move from the root to the lowest level, thus it takes O(log n)

Heap Sort

03. Heap Sort

Heap Sort

- Heap Sort
  - The simplicity and efficiency of the heap structure can be applied to the sorting problem
  - The heapsort algorithm builds a heap from a sequence of unsorted values and then extracts the items from the heap to create a sorted sequence
- Algorithm
  - Insert n data to a max-heap
  - Extract data from the heap, and create a sorted sequence
- Complexity
  - Insertion or deletion of a data takes O(log n)
  - N data -> O(nlogn)

Heap Sort

```
def simpleHeapSort( theSeq ):
 # Create an array-based max-heap.
 n = len(theSeq)
 heap = MaxHeap( n )

 # Build a max-heap from the list of values.
 for item in theSeq :
   heap.add( item )

 # Extract each value from the heap and store them back into the list.
 for i in range( n, 0, -1 ) :
  theSeq[i] = heap.extract()
```

What You Need to Know

Summary

- Priority Queue
  - Queue with priority
- Heap
  - insertion, deletion, …
- Heap Sort
  - O(nlogn)

Thanks

Week 5: Priority Queue, Heap
Instructor: Jinyoung Han (jinyounghan@skku.edu)