CSE 431/531: Algorithm Analysis and Design (Spring 2020)
Advanced Data Structures

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo

Outline

1. Heap: Concrete Data Structure for Priority Queue

2. Self-Balancing Binary-Search Tree
    - Counting inversions using Self-Balancing Binary-Search Tree
    - Binary Search Tree
    - Longest Increasing Subsequence using Self-Balancing BST

- Let V be a ground set of size n.

Def. A priority queue is an abstract data structure that maintains a set $U \subseteq V$ of elements, each with an associated key value, and supports the following operations:

- insert(v, key_value): insert an element $v \in V \setminus U$, with associated key value key_value.
- decrease_key(v, new_key_value): decrease the key value of an element $v \in U$ to new_key_value
- extract_min$()$: return and remove the element in U with the smallest key value
- $\cdots$

Simple Implementations for Priority Queue

- n = size of ground set V

data structures	**insert**	**extract_min**	**decrease_key**
array			
sorted array			

Simple Implementations for Priority Queue

- $n =$ size of ground set V

data structures	**insert**	**extract_min**	**decrease_key**
array	$O(1)$	$O(n)$	$O(1)$
sorted array			

Simple Implementations for Priority Queue

- $n =$ size of ground set V

data structures	**insert**	**extract_min**	**decrease_key**
array	$O(1)$	$O(n)$	$O(1)$
sorted array	$O(n)$	$O(1)$	$O(n)$

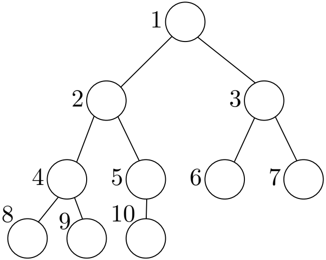
Simple Implementations for Priority Queue

- $n =$ size of ground set V

data structures	insert	extract_min	decrease_key
array	$O(1)$	$O(n)$	$O(1)$
sorted array	$O(n)$	$O(1)$	$O(n)$
heap	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$

Heap

The elements in a heap is organized using a complete binary tree:

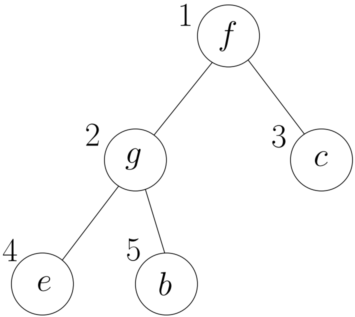


- Nodes are indexed as $\{1, 2, 3, \cdots, s\}$
- Parent of node i: $\lfloor i/2 \rfloor$
- Left child of node i: $2i$
- Right child of node i: $2i + 1$

Heap

A heap H contains the following fields

- s: size of U (number of elements in the heap)
- $A[i], 1 \leq i \leq s$: the element at node i of the tree
- $p[v], v \in U$: the index of node containing v
- $key[v], v \in U$: the key value of element v

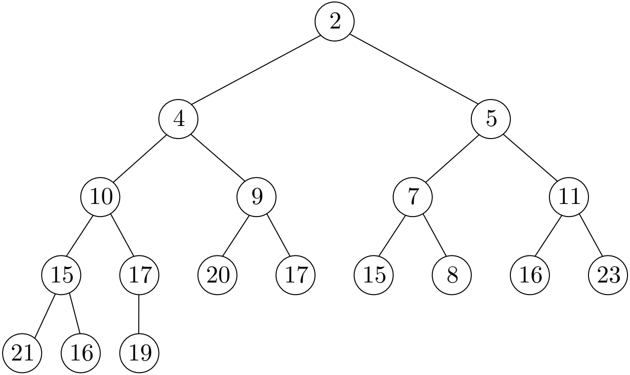


- $s = 5$
- $A = (‘f’, ‘g’, ‘c’, ‘e’, ‘b’)$
- $p[‘f’] = 1, p[‘g’] = 2, p[‘c’] = 3,$
  $p[‘e’] = 4, p[‘b’] = 5$

Heap

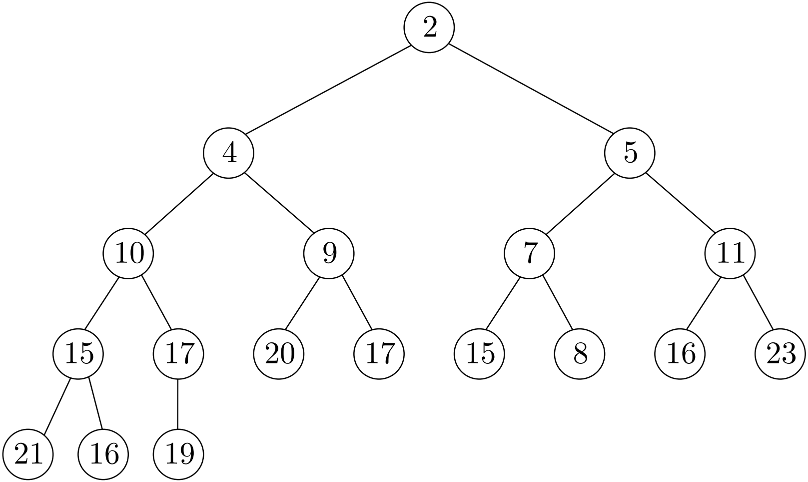
The following heap property is satisfied:

- for any two nodes i, j such that i is the parent of j, we have $key[A[i]] \leq key[A[j]]$.

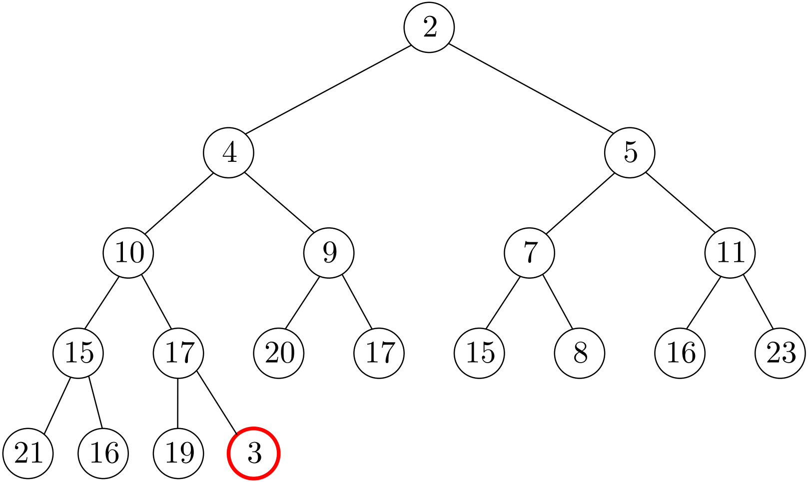


A heap. Numbers in the circles denote key values of elements.

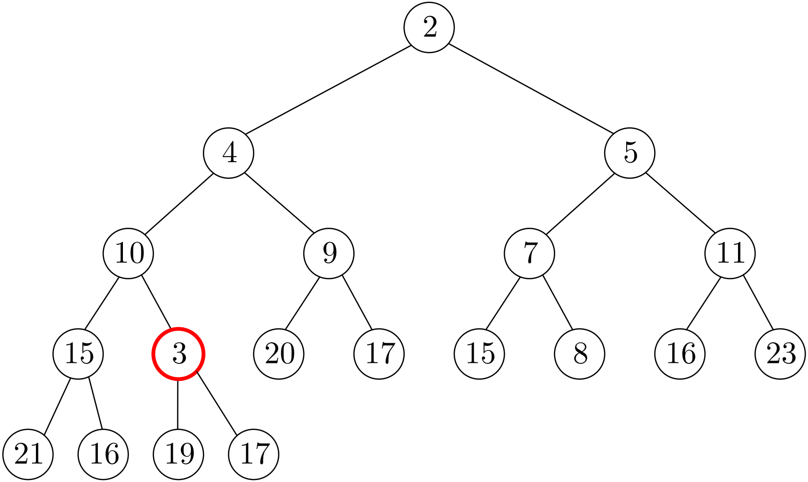
$\textsf{insert}(v, key\_value)$



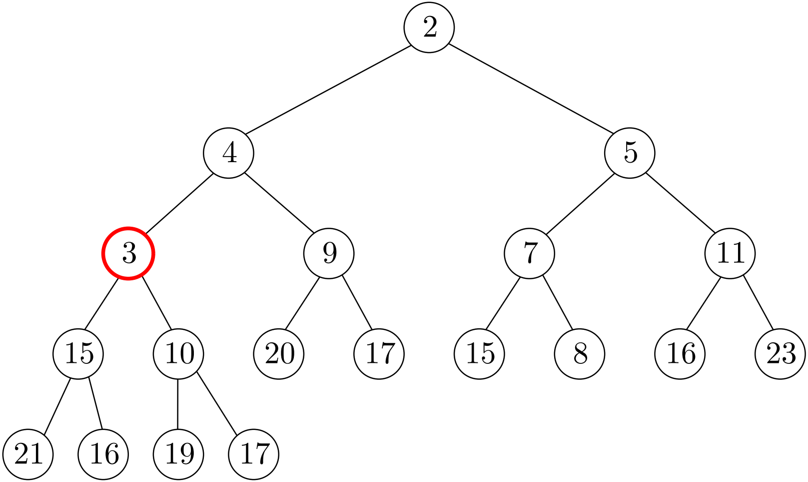
insert(v, key_value)



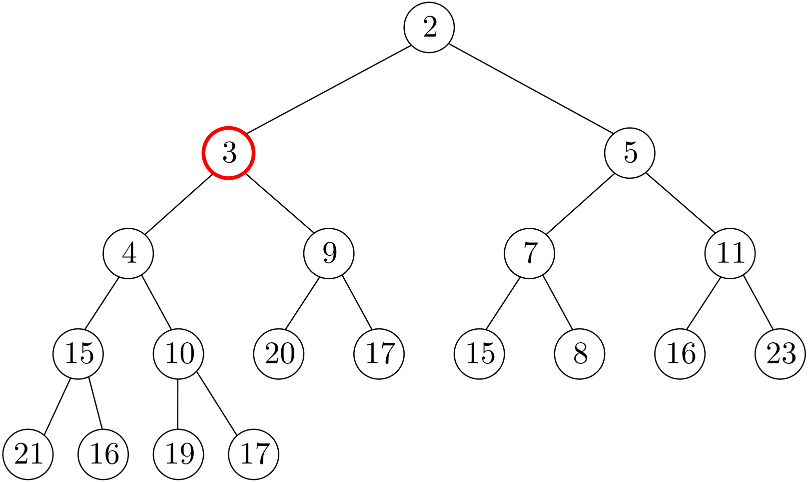
insert(v, key_value)



insert(v, key_value)



insert(v, key_value)



insert(v, key_value)

$s \leftarrow s + 1$

$A[s] \leftarrow v$

$p[v] \leftarrow s$

$key[v] \leftarrow key\_value$

heapify_up(s)

heapify-up(i)

while $i > 1$

  $j \leftarrow \lfloor i/2 \rfloor$

  if $key[A[i]] < key[A[j]]$ then

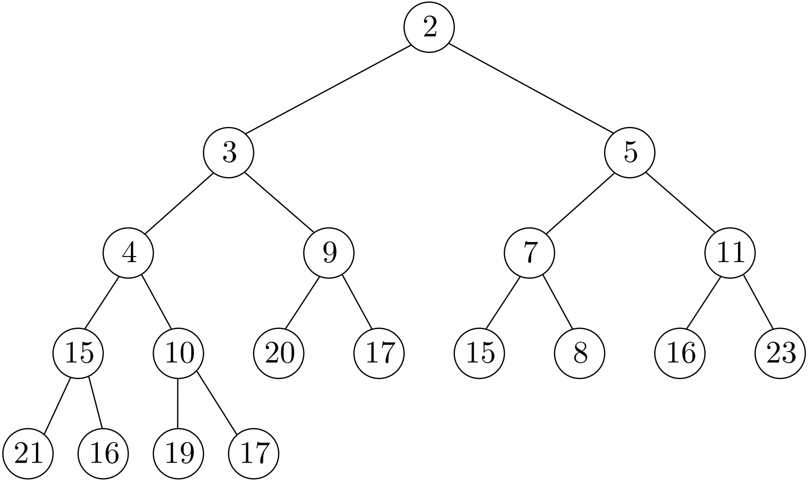
    swap $A[i]$ and $A[j]$

    $p[A[i]] \leftarrow i$, $p[A[j]] \leftarrow j$

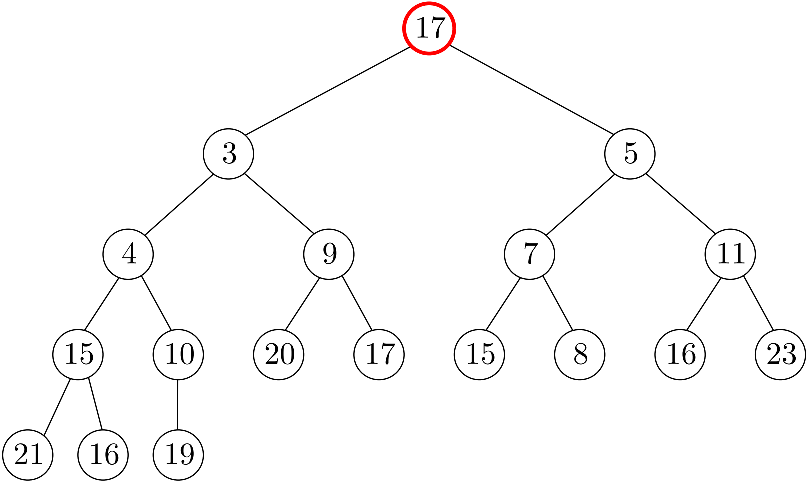
    $i \leftarrow j$

  else break

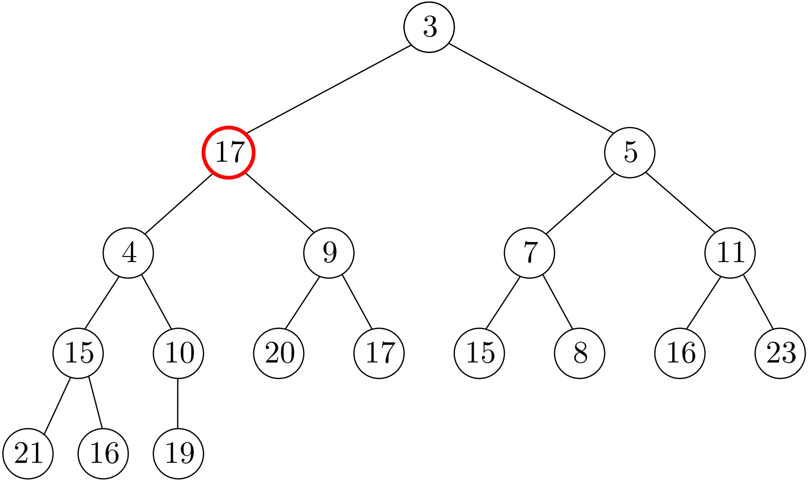
extract_min()



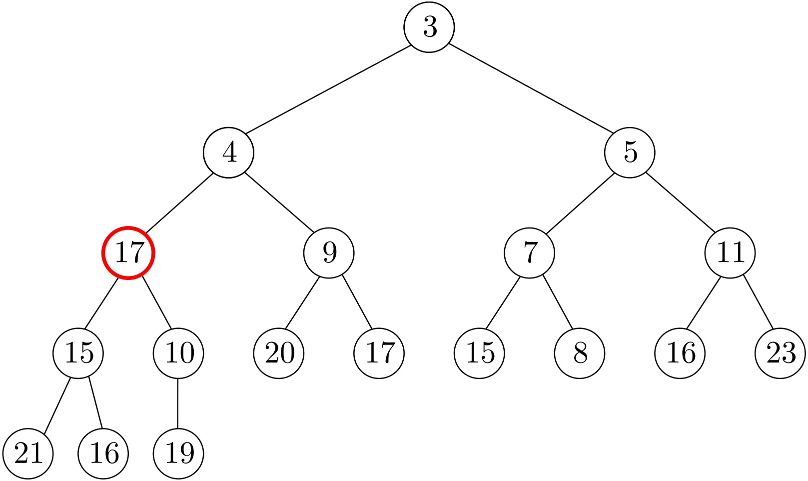
extract_min()



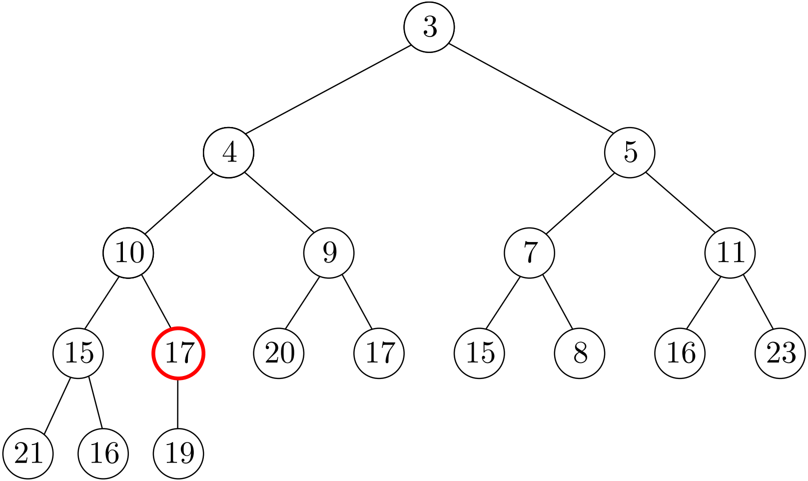
extract_min()



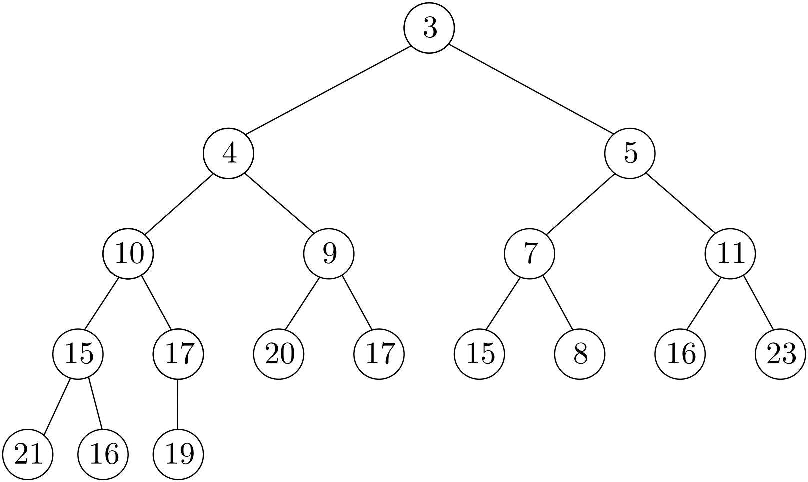
extract_min()



extract_min()



extract_min()



extract_min()

1. $ret \leftarrow A[1]$
2. $A[1] \leftarrow A[s]$
3. $p[A[1]] \leftarrow 1$
4. $s \leftarrow s - 1$
5. if $s \geq 1$ then
6.   heapify_down(1)
7. return ret

decrease_key(v, key_value)

1. $key[v] \leftarrow key\_value$
2. heapify-up$(p[v])$

heapify-down(i)

1. while $2i \leq s$
2.   if $2i = s$ or $key[A[2i]] \leq key[A[2i+1]]$ then
3. $j \leftarrow 2i$
4.   else
5. $j \leftarrow 2i + 1$
6.   if $key[A[j]] < key[A[i]]$ then
7.   swap $A[i]$ and $A[j]$
8. $p[A[i]] \leftarrow i$, $p[A[j]] \leftarrow j$
9. $i \leftarrow j$
10.   else break

- Running time of heapify_up and heapify_down: $O(\lg n)$

- Running time of heapify_up and heapify_down: $O(\lg n)$
- Running time of insert, exact_min and decrease_key: $O(\lg n)$

- Running time of heapify_up and heapify_down: $O(\lg n)$
- Running time of insert, exact_min and decrease_key: $O(\lg n)$

data structures	insert	extract_min	decrease_key
array	$O(1)$	$O(n)$	$O(1)$
sorted array	$O(n)$	$O(1)$	$O(n)$
heap	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$

Two Definitions Needed to Prove that the Procedures Maintain Heap Property

Def. We say that H is almost a heap except that $key[A[i]]$ is too small if we can increase $key[A[i]]$ to make H a heap.

Def. We say that H is almost a heap except that $key[A[i]]$ is too big if we can decrease $key[A[i]]$ to make H a heap.

Outline

1. Heap: Concrete Data Structure for Priority Queue

2. Self-Balancing Binary-Search Tree
   - Counting inversions using Self-Balancing Binary-Search Tree
   - Binary Search Tree
   - Longest Increasing Subsequence using Self-Balancing BST

Outline

Counting Inversions

inversions(A, n)

1. $T \leftarrow$ empty Binary Search Tree
2. $c \leftarrow 0$
3. for $i \leftarrow 1$ to n
4. $\quad c \leftarrow c + i - T.\text{rank}(A[i])$
5. $\quad T.\text{insert}(A[i])$
6. return c

Counting Inversions

inversions(A, n)

1. $T \leftarrow$ empty Binary Search Tree
2. $c \leftarrow 0$
3. for $i \leftarrow 1$ to n
4. $\quad c \leftarrow c + i - T.\text{rank}(A[i])$
5. $\quad T.\text{insert}(A[i])$
6. return c

15	3	16	12	32	7

Counting Inversions

inversions(A, n)

$T \leftarrow$ empty Binary Search Tree
$c \leftarrow 0$
for $i \leftarrow 1$ to n
    $c \leftarrow c + i - T$.rank$(A[i])$
    T.insert$(A[i])$
return c

15	3	16	12	32	7

$i = 1$: $\mathrm{rank}(15) = 1$

Counting Inversions

inversions(A, n)

$T \leftarrow$ empty Binary Search Tree
$c \leftarrow 0$
for $i \leftarrow 1$ to n
    $c \leftarrow c + i - T$.rank$(A[i])$
    T.insert$(A[i])$
return c

15	3	16	12	32	7

$i = 1$: $\mathrm{rank}(15) = 1$
$i = 2$: $\mathrm{rank}(\ 3) = 1$

Counting Inversions

inversions(A, n)

1. $T \leftarrow$ empty Binary Search Tree
2. $c \leftarrow 0$
3. for $i \leftarrow 1$ to n
4. $\quad c \leftarrow c + i - T.\text{rank}(A[i])$
5. $\quad T.\text{insert}(A[i])$
6. return c

15	3	16	12	32	7

$\uparrow$ (under 16)

$i = 1$: $\text{rank}(15) = 1$

$i = 2$: $\text{rank}(\ 3) = 1$

$i = 3$: $\text{rank}(16) = 3$

Counting Inversions

inversions(A, n)

1. $T \leftarrow$ empty Binary Search Tree
2. $c \leftarrow 0$
3. for $i \leftarrow 1$ to n
4. $\quad c \leftarrow c + i - T$.rank($A[i]$)
5. $\quad T$.insert($A[i]$)
6. return c

15	3	16	12	32	7

$i = 1$: $\mathrm{rank}(15) = 1$

$i = 2$: $\mathrm{rank}(\ 3) = 1$

$i = 3$: $\mathrm{rank}(16) = 3$

$i = 4$: $\mathrm{rank}(12) = 2$

Counting Inversions

inversions(A, n)

1. $T \leftarrow$ empty Binary Search Tree
2. $c \leftarrow 0$
3. for $i \leftarrow 1$ to n
4. $\quad c \leftarrow c + i - T$.rank$(A[i])$
5. $\quad T$.insert$(A[i])$
6. return c

15	3	16	12	32	7

$$\uparrow$$

$i = 1$: $\mathrm{rank}(15) = 1$

$i = 2$: $\mathrm{rank}(\ 3) = 1$

$i = 3$: $\mathrm{rank}(16) = 3$

$i = 4$: $\mathrm{rank}(12) = 2$

$i = 5$: $\mathrm{rank}(32) = 5$

Counting Inversions

inversions(A, n)

```
T ← empty Binary Search Tree
c ← 0
for i ← 1 to n
    c ← c + i − T.rank(A[i])
    T.insert(A[i])
return c
```

15	3	16	12	32	7

$i = 1$: $\mathrm{rank}(15) = 1$

$i = 2$: $\mathrm{rank}(\ 3) = 1$

$i = 3$: $\mathrm{rank}(16) = 3$

$i = 4$: $\mathrm{rank}(12) = 2$

$i = 5$: $\mathrm{rank}(32) = 5$

$i = 6$: $\mathrm{rank}(\ 7) = 2$

Counting Inversions

inversions(A, n)

```
T ← empty Binary Search Tree
c ← 0
for i ← 1 to n
    c ← c + i − T.rank(A[i])
    T.insert(A[i])
return c
```

15	3	16	12	32	7

$i = 1$: $\mathrm{rank}(15) = 1$

$i = 2$: $\mathrm{rank}(3) = 1$

$i = 3$: $\mathrm{rank}(16) = 3$

$i = 4$: $\mathrm{rank}(12) = 2$

$i = 5$: $\mathrm{rank}(32) = 5$

$i = 6$: $\mathrm{rank}(7) = 2$

$$c = (1-1) + (2-1) + (3-3) + (4-2) + (5-5) + (6-2) = 7$$

Outline

1. Heap: Concrete Data Structure for Priority Queue

2. Self-Balancing Binary-Search Tree
   - Counting inversions using Self-Balancing Binary-Search Tree
   - Binary Search Tree
   - Longest Increasing Subsequence using Self-Balancing BST

A self-balancing binary search tree T maintains a set of comparable elements and supports:

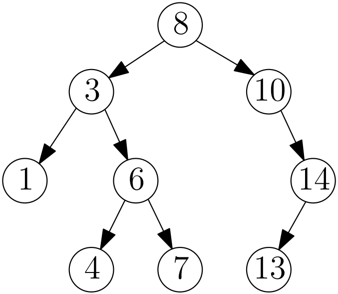
- Insertion of an element to T
- Deletion of an element from T
- Whether an element exists in T
- Return the rank of an element in T (i.e, 1 plus number of elements in T smaller than the element)
- Return the i-th smallest element in T
- ...

Each operation takes time $O(\lg n)$

Binary Search Trees

For any node v in tree:

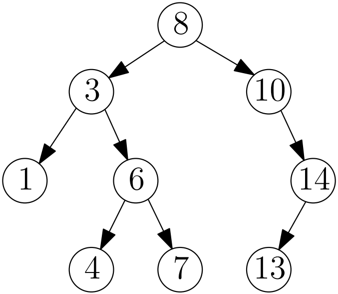
- key in v must be greater than all keys on the left-sub-tree of v
- key in v must be smaller than all keys on the right-sub-tree of v



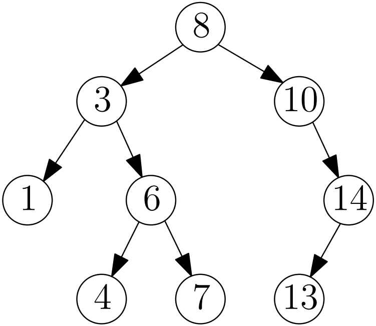
Binary Search Trees

For any node v in tree:

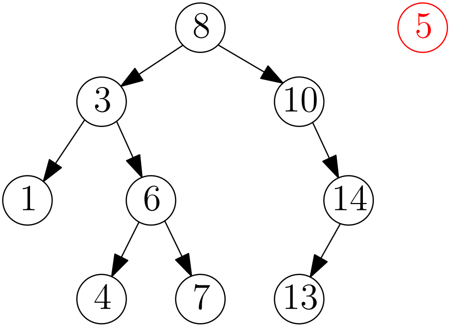
- key in v must be greater than all keys on the left-sub-tree of v
- key in v must be smaller than all keys on the right-sub-tree of v
- in-order traversal of tree gives a sorted list of keys



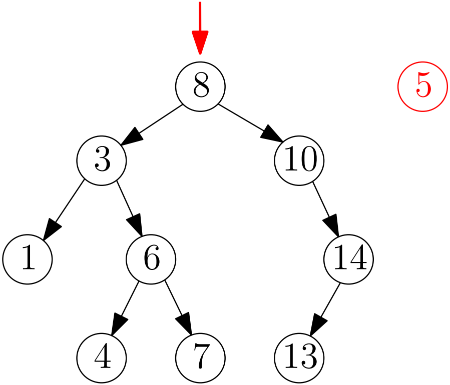
Binary Search Trees: Insertition



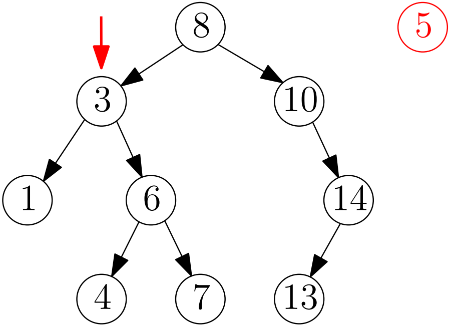
Binary Search Trees: Insertition



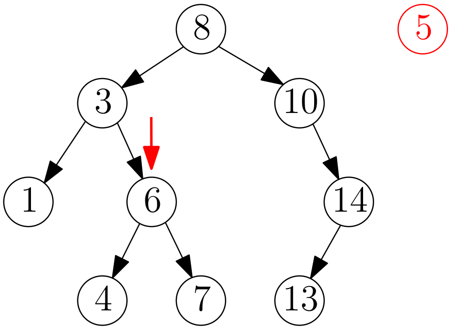
Binary Search Trees: Insertition



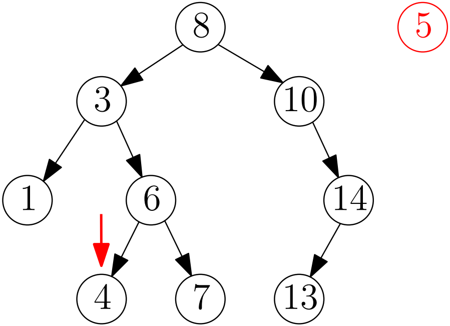
Binary Search Trees: Insertition



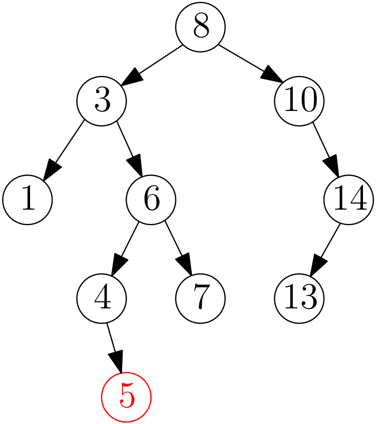
Binary Search Trees: Insertition



Binary Search Trees: Insertition



Binary Search Trees: Insertition

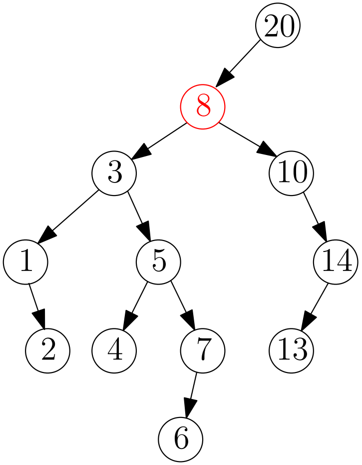


Binary Search Trees: Insertion

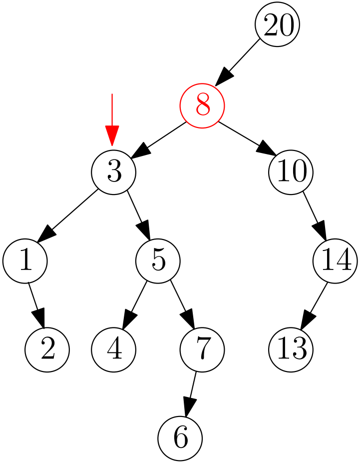
insert(v, key)

if $key < v.key$
    if $v.left =$ nil then
        create a new node u
        $u.key \leftarrow key, u.left \leftarrow$ nil$, u.right \leftarrow$ nil
        $v.left \leftarrow u$
    else insert$(v.left, key)$
else
    if $v.right =$ nil then
        create a new node u
        $u.key \leftarrow key, u.left \leftarrow$ nil$, u.right \leftarrow$ nil
        $v.right \leftarrow u$
    else insert$(v.right, key)$

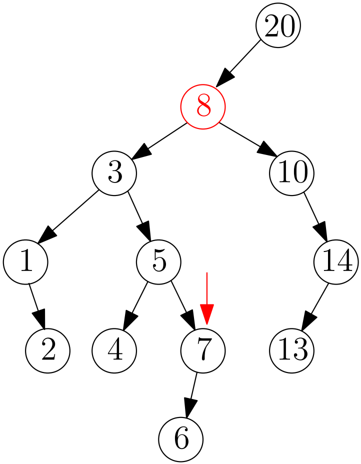
Binary Search Trees: Deletion



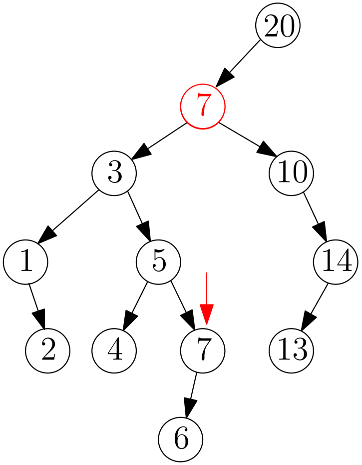
Binary Search Trees: Deletion



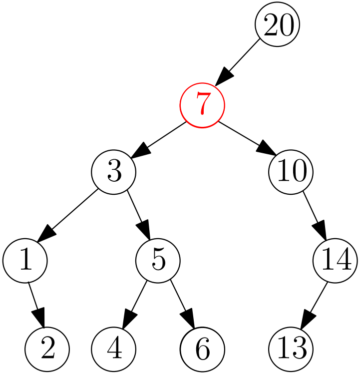
Binary Search Trees: Deletion



Binary Search Trees: Deletion



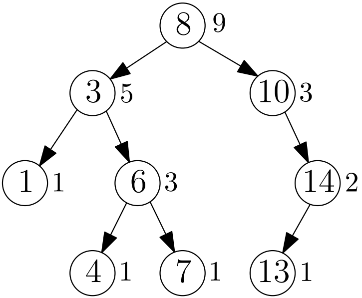
Binary Search Trees: Deletion



Binary Search Trees: Rank

Binary Search Trees: Rank

- Need to maintain a field "size"



Binary Search Trees: Rank

rank(v, key)

1. if $key \leq v.key$
2. if $v.left =$ nil then return 1
3. else return rank$(v.left, key)$
4. else
5. if $v.right =$ nil then return $v.size + 1$
6. else return $v.size - v.right.size +$ rank$(v.right, key)$

Running Time for Operations

- each operation takes time $O(d)$.
- $d =$ depth of tree
- best case:
- worst case:

Running Time for Operations

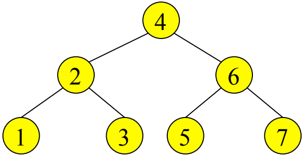
- each operation takes time $O(d)$.
- $d =$ depth of tree
- best case: $d = \Theta(\lg n)$
- worst case:

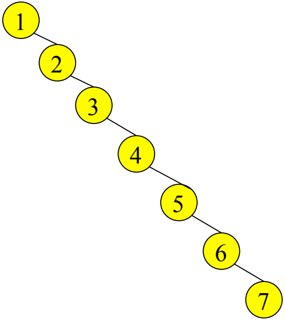
Running Time for Operations

- each operation takes time $O(d)$.
- $d =$ depth of tree
- best case: $d = \Theta(\lg n)$
- worst case: $d = \Theta(n)$

Running Time for Operations

- each operation takes time $O(d)$.
- $d =$ depth of tree
- best case: $d = \Theta(\lg n)$
- worst case: $d = \Theta(n)$





Self-Balancing BST: automatically keep the height of tree small

Self-Balancing BST: automatically keep the height of tree small

- AVL tree
- red-black tree
- Splay tree
- Treap
- ...

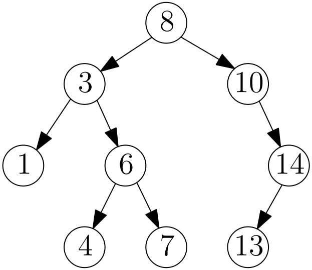
Self-Balancing BST: automatically keep the height of tree small

- AVL tree
- red-black tree
- Splay tree
- Treap
- ...

AVL Tree

Property of an AVL tree

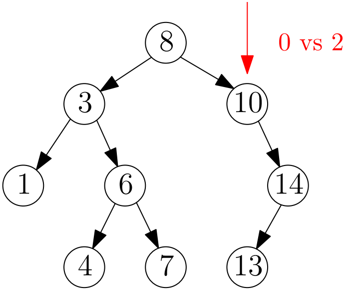
For every node v in the tree, the depths of the left-sub-tree of v and right-sub-tree of v differ by at most 1.



AVL Tree

Property of an AVL tree

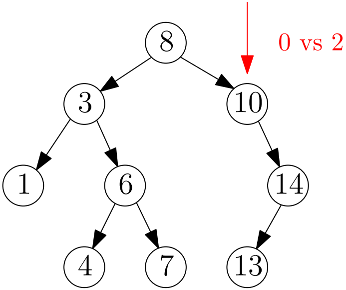
For every node v in the tree, the depths of the left-sub-tree of v and right-sub-tree of v differ by at most 1.



AVL Tree

Property of an AVL tree

For every node v in the tree, the depths of the left-sub-tree of v and right-sub-tree of v differ by at most 1.

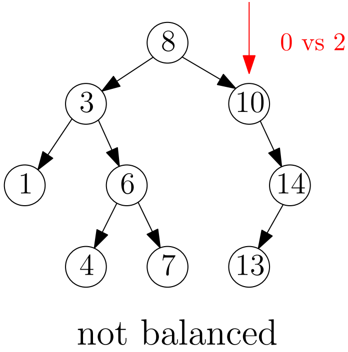


not balanced

AVL Tree

Property of an AVL tree

For every node v in the tree, the depths of the left-sub-tree of v and right-sub-tree of v differ by at most 1.



AVL Tree

Property of an AVL tree

For every node v in the tree, the depths of the left-sub-tree of v and right-sub-tree of v differ by at most 1.

- Why does the property guarantee that the height of a tree is $O(\log n)$?

AVL Tree

Property of an AVL tree

For every node v in the tree, the depths of the left-sub-tree of v and right-sub-tree of v differ by at most 1.

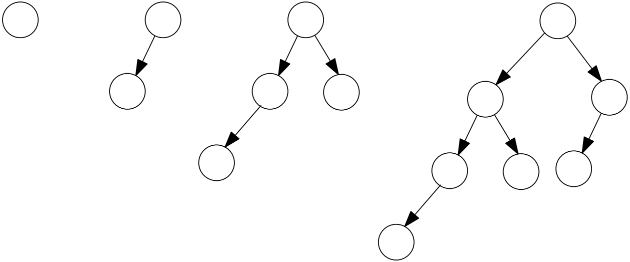
- Why does the property guarantee that the height of a tree is $O(\log n)$?
- $f(d)$: minimum number of nodes in an AVL tree of depth d

AVL Tree

Property of an AVL tree

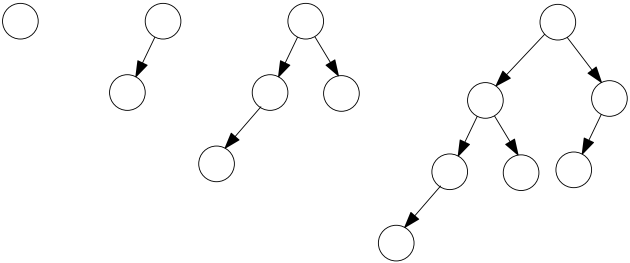
For every node v in the tree, the depths of the left-sub-tree of v and right-sub-tree of v differ by at most 1.

- Why does the property guarantee that the height of a tree is $O(\log n)$?
- $f(d)$: minimum number of nodes in an AVL tree of depth d



- $f(0) = 0, f(1) = 1, f(2) = 2, f(3) = 4, f(4) = 7 \cdots$

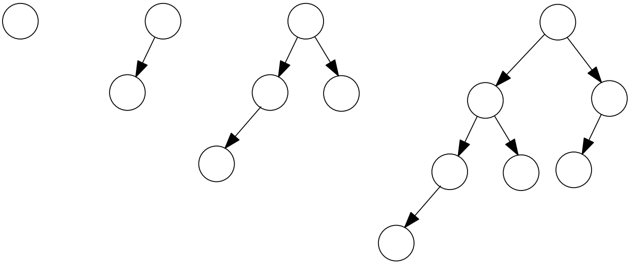
- $f(d)$: minimum number of nodes in an AVL tree of depth d



- Recursion:

$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 1 \\
f(d) &= f(d-1) + f(d-2) + 1 \qquad\qquad d \geq 2
\end{aligned}
$$

- $f(d)$: minimum number of nodes in an AVL tree of depth d



- Recursion:

$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 1 \\
f(d) &= f(d-1) + f(d-2) + 1 \qquad\qquad d \geq 2
\end{aligned}
$$

- $f(d) = 2^{\Theta(d)}$

Depth of AVL tree

- $f(d)$: minimum number of nodes in an AVL tree of depth d
- $f(d) = 2^{\Theta(d)}$

Depth of AVL tree

- $f(d)$: minimum number of nodes in an AVL tree of depth d
- $f(d) = 2^{\Theta(d)}$
- If a AVL tree has size n and depth d, then

$$n \geq f(d)$$

Depth of AVL tree

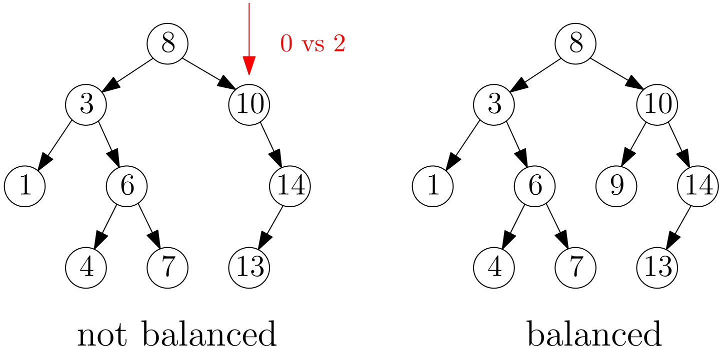
- $f(d)$: minimum number of nodes in an AVL tree of depth d
- $f(d) = 2^{\Theta(d)}$
- If a AVL tree has size n and depth d, then

$$n \geq f(d)$$

- Thus, $d = O(\log n)$

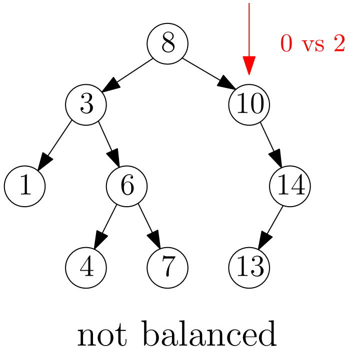
Property of an AVL tree

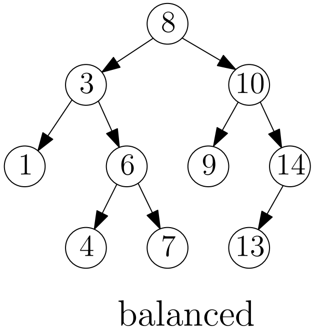
For every node v in the tree, the depths of the left-sub-tree of v and right-sub-tree of v differ by at most 1.



Property of an AVL tree

For every node v in the tree, the depths of the left-sub-tree of v and right-sub-tree of v differ by at most 1.

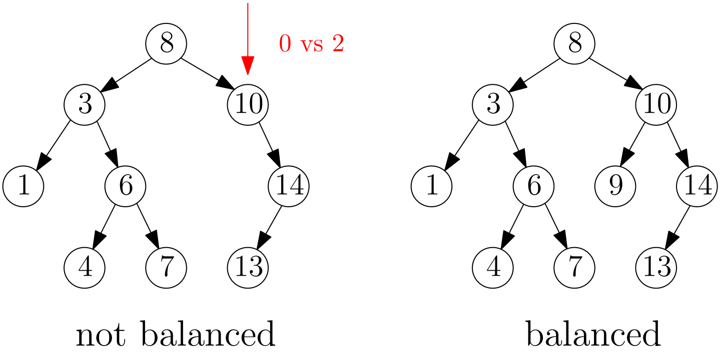




- How can we maintain the property?

Property of an AVL tree

For every node v in the tree, the depths of the left-sub-tree of v and right-sub-tree of v differ by at most 1.



- How can we maintain the property?
- Assume we only do insertions; there are no deletions.

Maintain Balance Property

Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion

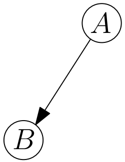
A

Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A

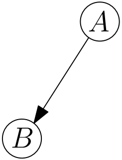
Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A



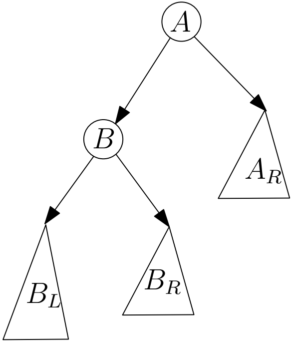
Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B



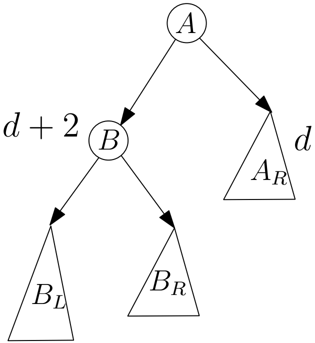
Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B



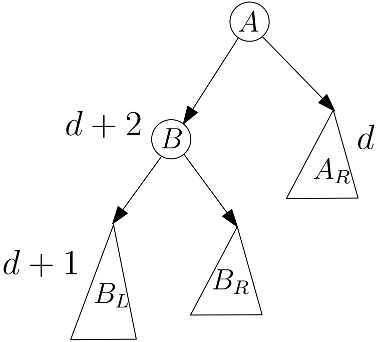
Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B



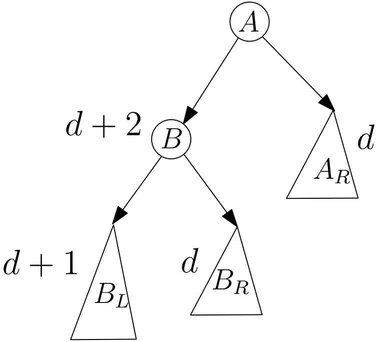
Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B



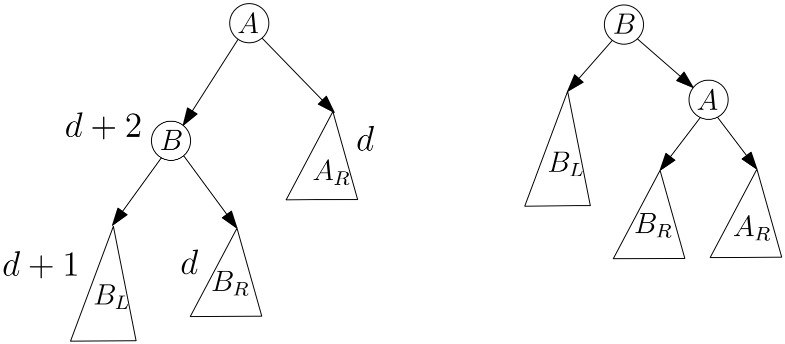
Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B



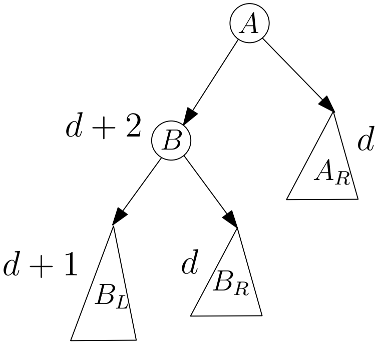
Maintain Balance Property

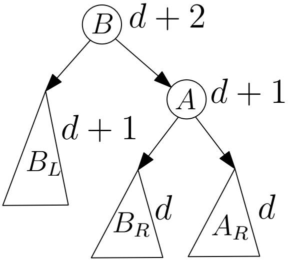
- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B



Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 1: we inserted an element to the left-sub-tree of B





Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A

Maintain Balance Property

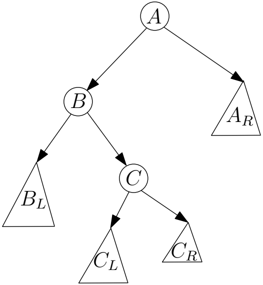
- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B

Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B

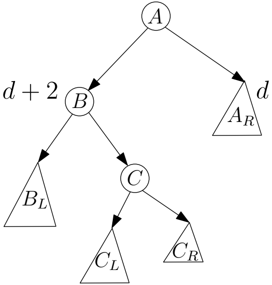
Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B



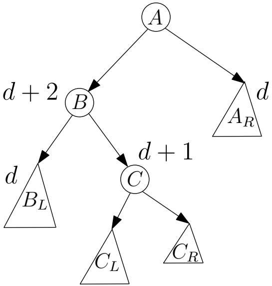
Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B



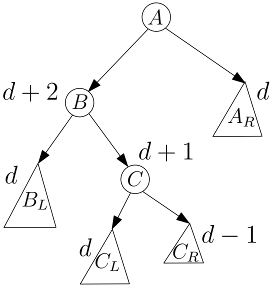
Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B



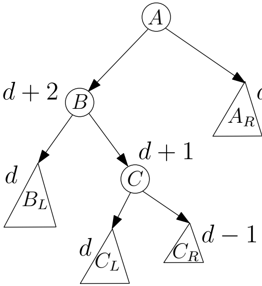
Maintain Balance Property

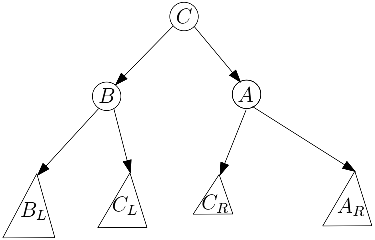
- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B



Maintain Balance Property

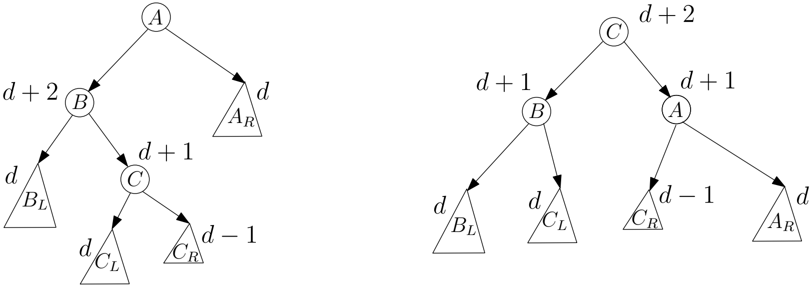
- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B





Maintain Balance Property

- A: the deepest node such that the balance property is not satisfied after insertion
- Wlog, we inserted an element to the left-sub-tree of A
- B: the root of left-sub-tree of A
- case 2: we inserted an element to the right-sub-tree of B
- C: the root of right-sub-tree of B



Outline

1. Heap: Concrete Data Structure for Priority Queue

2. Self-Balancing Binary-Search Tree
   - Counting inversions using Self-Balancing Binary-Search Tree
   - Binary Search Tree
   - Longest Increasing Subsequence using Self-Balancing BST

Recall: Longest Increasing Subsequence Problem

Def. Given a sequence $A = (a_1, a_2, \cdots, a_n)$ of n numbers, an increasing subsequence of A is a subsequence $(A_{i_1}, A_{i_2}, A_{i_3}, \cdots, A_{i,t})$ such that $1 \le i_1 < i_2 < i_3 < \cdots < i_t \le n$ and $a_{i_1} < a_{i_2} < a_{i_3} < \cdots < a_{i_t}$.

Exercise: Longest Increasing Subsequence

Input: $A = (a_1, a_2, \cdots, a_n)$ of n numbers

Output: The length of the longest increasing sub-sequence of A

Example:

- Input: $(10, 3, 9, 8, 2, 5, 7, 1, 12)$

Recall: Longest Increasing Subsequence Problem

Def. Given a sequence $A = (a_1, a_2, \cdots, a_n)$ of n numbers, an increasing subsequence of A is a subsequence $(A_{i_1}, A_{i_2}, A_{i_3}, \cdots, A_{i,t})$ such that $1 \le i_1 < i_2 < i_3 < \cdots < i_t \le n$ and $a_{i_1} < a_{i_2} < a_{i_3} < \cdots < a_{i_t}$.

Exercise: Longest Increasing Subsequence

Input: $A = (a_1, a_2, \cdots, a_n)$ of n numbers

Output: The length of the longest increasing sub-sequence of A

Example:

- Input: $(10, 3, 9, 8, 2, 5, 7, 1, 12)$
- Output: 4

Dynamic Programming for Longest Increasing Sub-sequence Problem

- $f[i]$: longest increasing sub-sequence ending at i.
- For every $i = 1, 2, 3, \cdots, n$,

$$f[i] = \max_{j<i:a_j<a_i} f(j) + 1,$$

assuming $\max_{j<i:a_j<a_i} f(j) = 0$ if no such j exists.

$O(n^2)$-Time Algorithm for LIS

LIS(A, n)

$ans \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  $f[i] \leftarrow 0$
  for $j \leftarrow 1$ to $i - 1$ **do**
    if $A[j] < A[i]$ and $f[j] + 1 > f[i]$ **then** $f[i] \leftarrow f[j] + 1$
  if $f[i] > ans$ **then** $ans \leftarrow f[i]$
return ans

Improving Running Time to $O(n \log n)$ Using Self-Balancing BST

LIS(A, n)

1. $T \leftarrow$ empty Self-Balancing BST, \\ each element in T is an integer and associated with a f value
2. $ans \leftarrow 1$
3. **for** $i \leftarrow 1$ to n **do**
4. $\quad f[i] \leftarrow T$.max-f-value-over-elements-less-than$(A[i])+1$ \\ the function returns the maximum f value over all elements in T that are less than $A[i]$
5. $\quad T$.insert$(A[i], f[i])$ \\ insert $A[i]$ with f value being $f[i]$ to T
6. $\quad$ **if** $f[i] > ans$ **then** $ans \leftarrow f[i]$
7. **return** ans

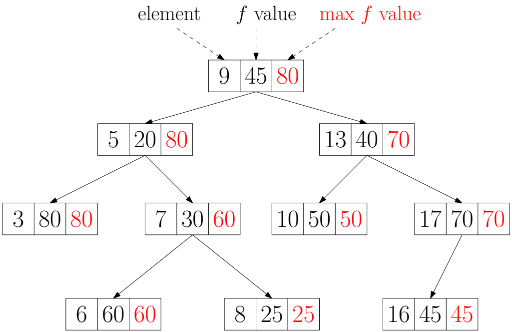
Q: How can we implement max-f-value-over-elements-less-than so that it runs in $O(\log n)$ time?

Q: How can we implement max-f-value-over-elements-less-than so that it runs in $O(\log n)$ time?

A: In each node of BST, we maintain the maximum f value over all nodes in the sub-tree rooted at the node.

Q: How can we implement max-f-value-over-elements-less-than so that it runs in $O(\log n)$ time?

A: In each node of BST, we maintain the maximum f value over all nodes in the sub-tree rooted at the node.



Q: How can we implement max-f-value-over-elements-less-than so that it runs in $O(\log n)$ time?

A: In each node of BST, we maintain the maximum f value over all nodes in the sub-tree rooted at the node.