CSE331 Introduction to Algorithms
Lecture 21: Review of
Graph Algorithms and Data Structures II

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

July 23, 2021

1. Introduction

2. Heaps
   - Insertion
   - Extracting the Minimum
   - Priority queues

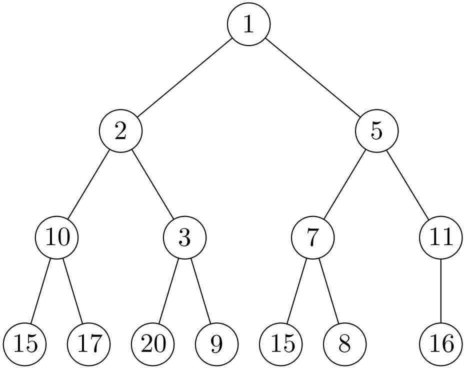
3. Binary search trees
   - Insertion
   - Traversal
   - Searching
   - Balanced binary search trees

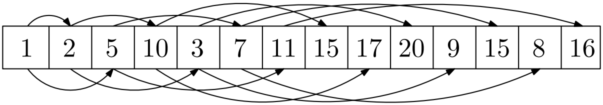
Introduction

- Topics:
  - Heaps and priority queues.
  - Binary search trees.
- **Reference**: Sections 6 and 12 of the textbook

  Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.
- I will not be following the textbook closely in this lecture.

Heaps





Heaps

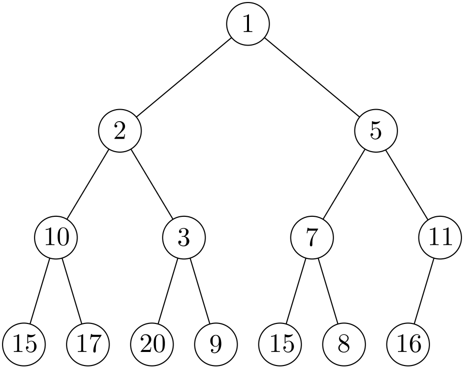
- A *heap* is a binary tree such that each node v contains a number $\text{key}(v)$ called a *key*, and possibly satellite data.
- The nodes of a heap have the *heap property*:

Property

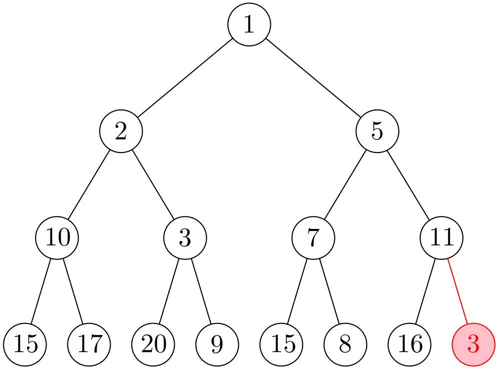
If v is the parent of w, then $\text{key}(v) \leqslant \text{key}(w)$.

- The heap is recorded in an array $H[1, \ldots, N]$.
- N is the maximum number of elements that the heap can store.
- The root is $H[1]$.
- The two children of $H[i]$ are $H[2i]$ and $H[2i+1]$.
- So the parent of $H[i]$ is $H[\lfloor i/2 \rfloor]$.
- If the heap records $n \leqslant N$ nodes, then they are recorded in $H[1 \ldots n]$.

Insertion

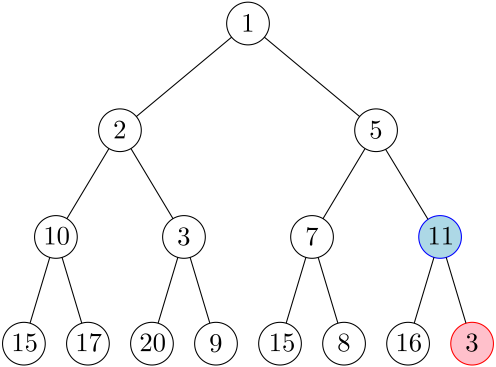


Insertion



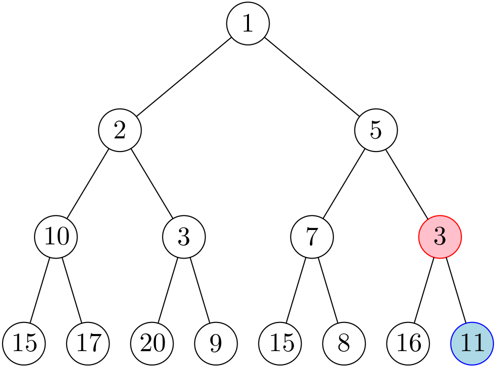
The new node is inserted at the last position

Insertion



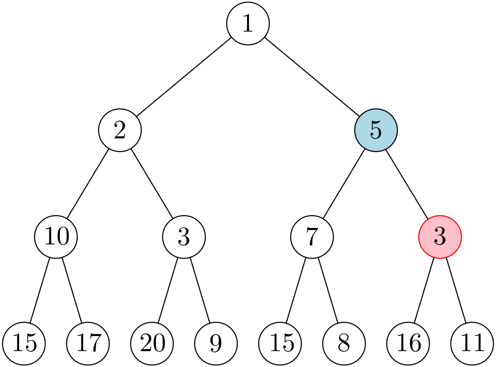
The heap property does not hold for the new node

Insertion



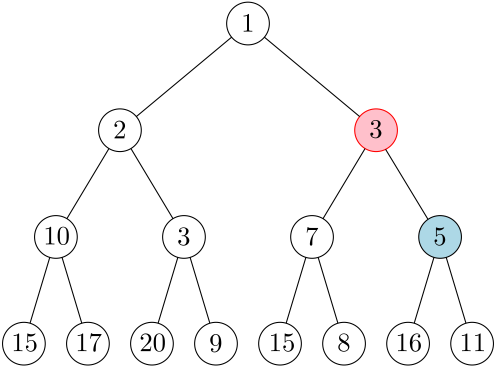
Fixing the heap

Insertion



The heap property does not hold

Insertion



Now the heap is fixed

Insertion

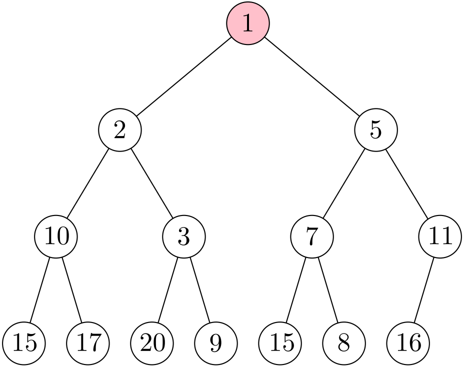
- If the heap contains n nodes, the new node is inserted at $H[n+1]$.
- Then we fix the heap by calling $\textsc{Heapify-up}(H, n+1)$

Pseudocode

procedure $\textsc{Heapify-up}(H, i)$
    if $i > 1$ **then**
        $p \leftarrow \lfloor i/2 \rfloor$ $\quad\triangleright$ p is the parent of i
        if $\text{key}(H[p]) > \text{key}(H[i])$ **then**
            exchange $H[i]$ with $H[p]$
            $\textsc{Heapify-up}(H, p)$

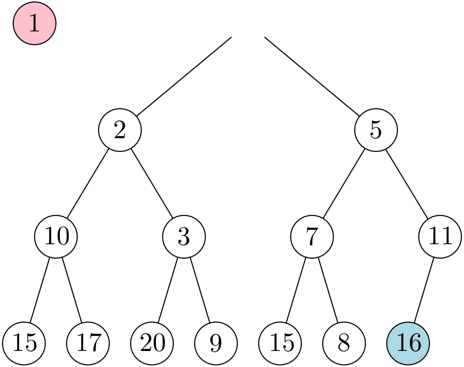
- It takes time $O(\log n)$ because i gets halved at each recursive call.

Extracting the Minimum



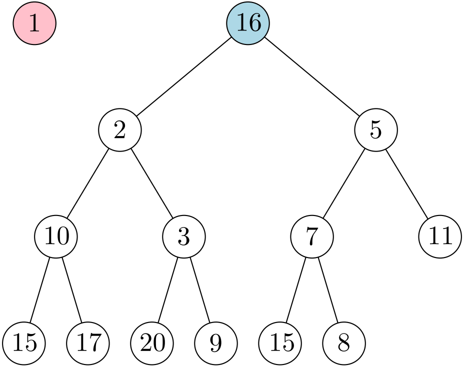
The minimum is at the root.

Extracting the Minimum



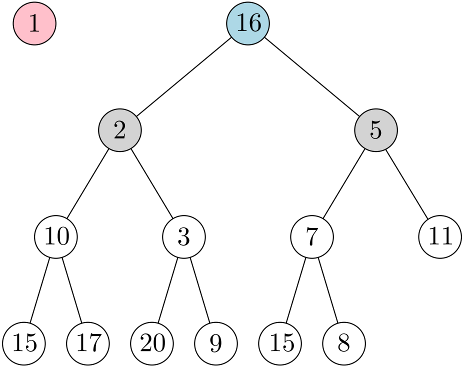
After we extract the minimum, a hole is left at the root.

Extracting the Minimum



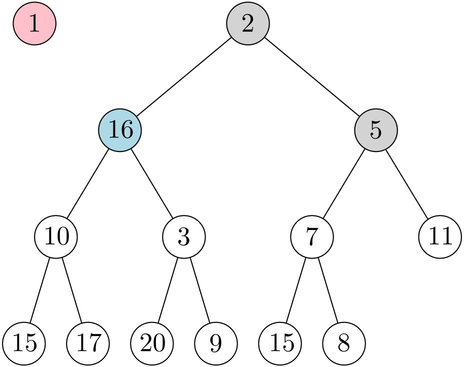
We move the last element to the root.

Extracting the Minimum



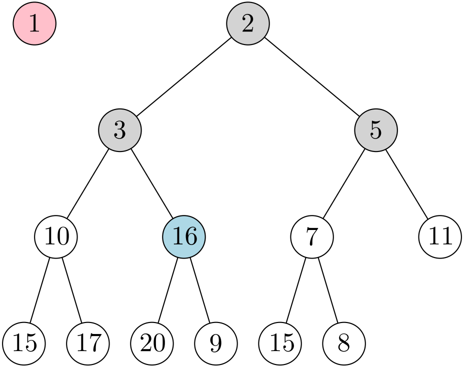
The heap property is violated.

Extracting the Minimum



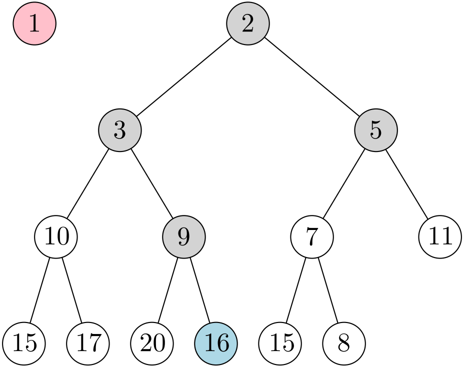
Fixing the heap.

Extracting the Minimum



Fixing the heap.

Extracting the Minimum



Now the heap is fixed.

Extracting the Minimum

- The minimum is at the root node.
- So we first extract the root node.
- We replace it with the last node.
- We fix the heap property by calling $\textsc{Heapify-down}(H)$.
  (See next slide.)

Extracting the Minimum

Pseudocode

```
procedure HEAPIFY-DOWN(H)
    n ← length(H)
    i ← 1
    while 2i ⩽ n do
        j ← the index of the child of i with smallest key.
        if key(H[i]) > key(H[j]) then
            exchange H[i] with H[j]
            i ← j
        else
            return
```

- This procedure runs in time $O(\log n)$ because i becomes $2i$ or $2i+1$ at the end of each iteration of the WHILE loop.

Heap Operations

Theorem

A heap records a set of n elements using $O(n)$ space. We can insert a new element in $O(\log n)$ time, and extract the element with minimum key in $O(\log n)$ time.

- We can also delete any element $H[i]$ in $O(\log n)$ time:
  - First $H[i] \leftarrow H[n]$.
  - Then, if the key of $H[i]$ is smaller than its parent, call $\textsc{Heapify-up}(H, i)$
  - Otherwise, if the key of $H[i]$ is larger than one of its child, call a modified version of $\textsc{Heapify-down}$ that starts at $H[i]$.

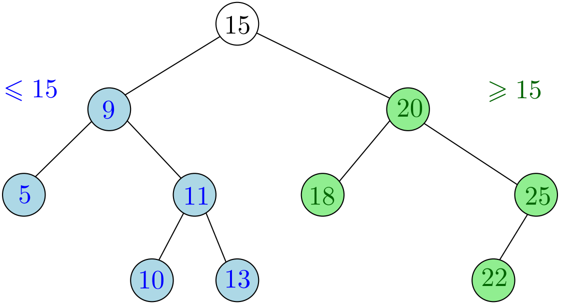
Priority Queues

- These two operations (INSERT and EXTRACTMIN) are the basic operations of an abstract data type called *priority queue*.
- Priority queues are often implemented using heaps, as they allow to perform each operation in $O(\log n)$ time.

Remarks

- What we described above is a *min heap*.
- In a *max heap*, the order is reversed: The key of a node is not larger than its parent, so the *largest* key is stored at the root of any subtree.
- A *max heap* allows to extract the *maximum*, to insert and to delete an element in $O(\log n)$ time.
- We can sort a set of n numbers by inserting them all into a heap, and then extracting the minimum repeatedly.
- It takes $O(n \log n)$ time.
- There is a slightly better algorithm for sorting using a heap, called HEAPSORT.
  - Use a *max heap*. (Why?)
  - All the elements can be inserted in $O(n)$ time, but we still need $\Theta(\log n)$ time for each extraction.
  - Not covered in CSE331.

Binary Search Trees

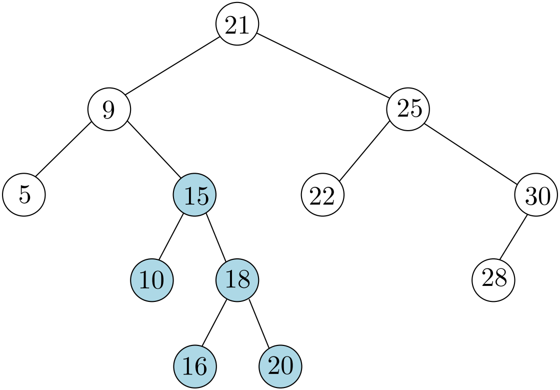


Definition (Binary search tree)

A *binary search tree (BST)* T is a binary tree that records a key at each node. Every node v of T has the following properties.

- For every node u in the left subtree of v, we have $\text{key}(u) \leqslant \text{key}(v)$.
- For every node w in the right subtree of v, we have $\text{key}(w) \geqslant \text{key}(v)$.

Subtrees of a BST



- BST with set of keys $\{5, 9, 10, 15, 16, 18, 20, 21, 22, 25, 28, 30\}$.

Subtrees of a BST

Proposition

The keys stored in a subtree T' of a binary search tree T are consecutive. So if the keys of T are $k_1 < k_2 < \cdots < k_n$, then T' stores $k_i < k_{i+1} < \cdots < k_j$ for $1 \leqslant i \leqslant j \leqslant n$.

Proof.

Done in class. $\square$

Binary Search Trees

Implementation

A node v of a BST records the following fields:

- key(v) — *the key of v*
- left(v) — *pointer to the left child of v*
- right(v) — *pointer to the right child of v*

The pointer left(v) or right(v) is set to NIL if the corresponding child does not exist.

- Node v may also record satellite data
- For instance, if T records points (x, y, z), the key could be x and (y, z) could be the satellite data.

Insertion into a BST

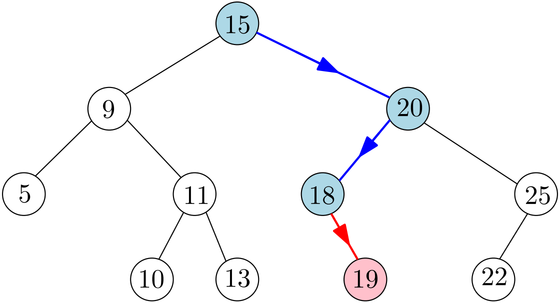
Inserting key k into a BST

procedure $\textsc{Insert}(r, k)$
    if $r = \text{NIL}$ **then**
        $r \leftarrow \textsc{NewNode}(k)$
    else if $k < \text{key}(r)$ **then**
        $\textsc{Insert}(\text{left}(r), k)$
    else
        $\textsc{Insert}(\text{right}(r), k)$

- The new key k is inserted from the *root* node r of the tree T.
- Insertion takes $O(h+1)$ time, where h is the height of the tree.

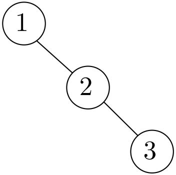
BST Insertion: Example

- Inserting 19 into the tree from Slide 25

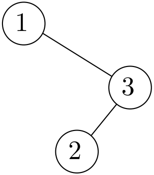


BST Insertion Orders

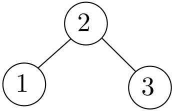
- The shape of a BST depends on the order of insertions.



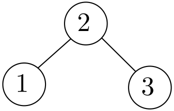
$1 \to 2 \to 3$



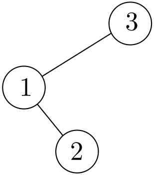
$1 \to 3 \to 2$



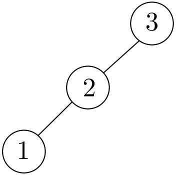
$2 \to 1 \to 3$



$2 \to 3 \to 1$



$3 \to 1 \to 2$



$3 \to 2 \to 1$

In-Order Traversal

- The keys of a binary search tree T can be printed in nondecreasing order by calling the following procedure, called *in-order traversal*, from the root of T.

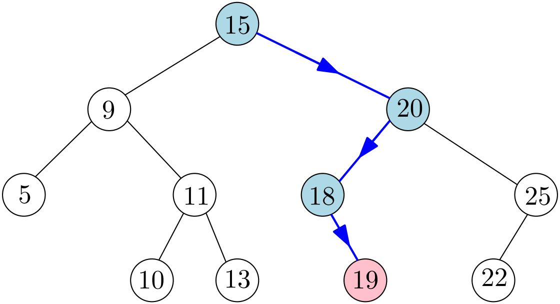
Pseudocode

```
procedure In-order(v)
    if v = NIL then
        return
    In-order(left(v))
    print key(v)
    In-order(right(v))
```

- On the BST from Slide 25, it prints:

5 9 10 11 13 15 18 20 22 25

Searching in a BST



Problem (Searching)

Given a binary search tree T and a key k, the searching problem is to decide whether k is the key of a node v of T, and if so, return v.

- The procedure on next slide allows to search in a BST in $O(h+1)$ time, where h is the height of the tree.

Searching in a BST

Pseudocode

procedure SEARCH(v, k)
    if $v = \text{NIL}$ **then**
        return NOTFOUND
    if $k < \text{key}(v)$ **then**
        return SEARCH(left(v), k)
    if $k > \text{key}(v)$ **then**
        return SEARCH(right(v), k)
    return v $\triangleright$ $k = \text{key}(v)$

Balanced Binary Search Trees

- A BST with n nodes has height at least $\lfloor \log n \rfloor$, so the (worst case) search time is $\Omega(\log n)$.
- There exist *balanced binary search trees* (BBST) whose height is $O(\log n)$, so the search time is $\Theta(\log n)$.
- It is possible to insert and delete nodes in $\Theta(\log n)$ time in a BBST.
  - It requires to rebalance (change the structure) of the BST while inserting/deleting.
- So BBSTs have the same asymptotic search time as a sorted array, and allow efficient insertion/deletion. Sorted arrays, on the other hand, do not allow efficient insertion/deletion.
- I do not cover BBSTs in details in this course, you should only know that it can be done.