CSE515 Advanced Algorithms
Lecture 5: Review of Elementary Data Structures and Graph Algorithms

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

March 17, 2021

1. Introduction
2. Arrays
3. Linked lists
    - Doubly linked lists
    - Stacks
    - Queues
4. Graphs
    - DFS
    - BFS
5. Heaps
    - Insertions
    - Extracting the Minimum
    - Priority queues
6. Binary search trees

Introduction

- In this lecture, I will review elementary data structures and graph algorithms.
  - Linked lists, stacks and queues.
  - Heaps and priority queues.
  - Graph traversals (BFS, DFS).
  - Binary search trees.
- I will not be following this textbook closely in this lecture.
- These algorithms and data structures are fundamental. They are typically covered in undergraduate data structure courses.
- **Reference**: Sections 6, 10, 12, 13, and 22 of the textbook

  Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.

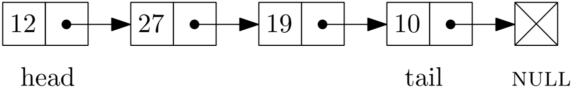
Arrays

- Array $A[1 \dots n]$ is created in $O(n)$ time.
- We can access element $A[i]$ at any index i in $O(1)$ time
  - This is called *random access*

Arrays

- Array $A[1 \ldots n]$ is created in $O(n)$ time.
- We can access element $A[i]$ at any index i in $O(1)$ time
    - This is called *random access*
- 2-dimensional array: $B[1 \ldots m, 1 \ldots n]$
- Idem: access $B[i,j]$ in $O(1)$ time, create array in $O(mn)$ time
- Generalizes to any dimension

Linked Lists

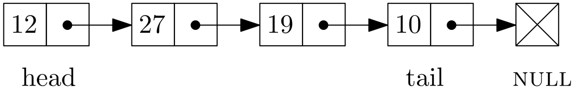


- Implementation: Each node in the list has two fields.

Node

- next — *reference to next node*
- data — *data stored at this node*

Linked Lists



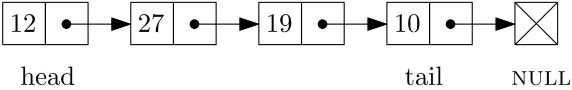
- Implementation: Each node in the list has two fields.

Node

- next — *reference to next node*
- data — *data stored at this node*

- Operations:
  - Insert/delete element at the head: $O(1)$ time.

Linked Lists



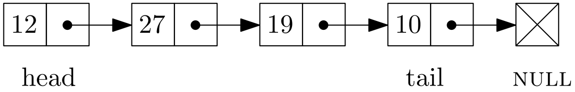
- Implementation: Each node in the list has two fields.

Node

- next — *reference to next node*
- data — *data stored at this node*

- Operations:
  - Insert/delete element at the head: $O(1)$ time.
  - Find an element in a list of size n in

Linked Lists



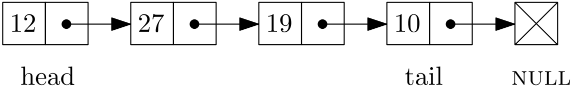
- Implementation: Each node in the list has two fields.

Node

- next — *reference to next node*
- data — *data stored at this node*

- Operations:
  - Insert/delete element at the head: $O(1)$ time.
  - Find an element in a list of size n in $O(n)$ time.

Linked Lists



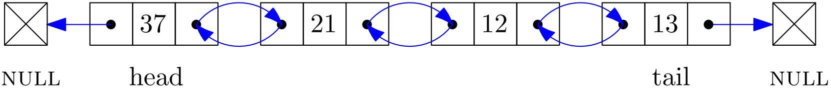
- Implementation: Each node in the list has two fields.

Node

- next — *reference to next node*
- data — *data stored at this node*

- Operations:
  - Insert/delete element at the head: $O(1)$ time.
  - Find an element in a list of size n in $O(n)$ time.
  - No random access: accessing/inserting/deleting an element in the middle of the list takes $O(n)$ time.

Doubly Linked Lists



Node

- next *reference to next node*
- prev *reference to previous node*
- data *data stored at this node*

List

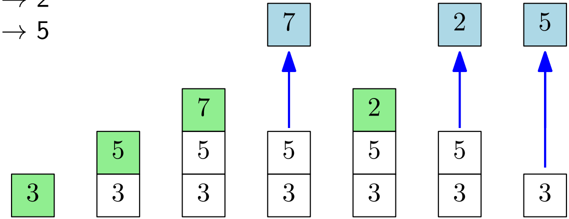
- head *reference to the head node*
- tail *reference to the tail node*

Doubly Linked Lists

- Operations:
  - Insert/delete element at the head or tail: $O(1)$ time.
  - Find an element in a list of size n in $O(n)$ time.
  - Delete/insert element at any location in $O(n)$ time.

Stacks

- A *stack* is an *abstract data type* with two operations:
  - push: insert an element
  - pop: remove from the stack the most recently inserted element
- Example:
  - Start with empty stack
  - push 3, push 5, push 7
  - pop → 7
  - push 2
  - pop → 2
  - pop → 5



Stacks

- This is called *LIFO*: last in, first out.
- A stack can be implemented with

Stacks

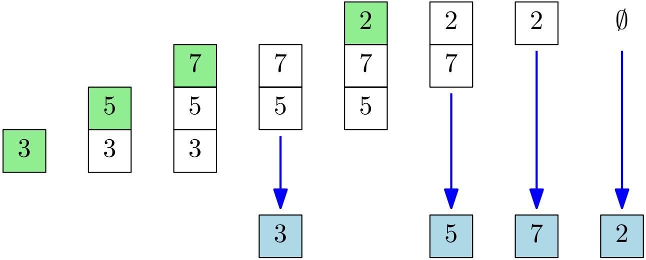
- This is called *LIFO*: last in, first out.
- A stack can be implemented with a linked list.
- Then each operation takes $O(1)$ time.

Stacks

- This is called *LIFO*: last in, first out.
- A stack can be implemented with a linked list.
- Then each operation takes $O(1)$ time.
- We can also use an array, where the last element is the top of the stack,and keep track of its index.

Queue

- A *queue* is an abstract data type with two operations:
  - enqueue: insert an element
  - dequeue: remove from the queue the earliest inserted element
- Example:
  - start with empty queue
  - enqueue 3, enqueue 5, enqueue 7
  - dequeue $\rightarrow$ 3
  - enqueue 2
  - dequeue $\rightarrow$ 5
  - dequeue $\rightarrow$ 7
  - dequeue $\rightarrow$ 2



Queue

- This is called *FIFO*: first in, first out.
- A queue can be implemented with

Queue

- This is called *FIFO*: first in, first out.
- A queue can be implemented with a doubly linked list.
- Then each operation takes $O(1)$ time.

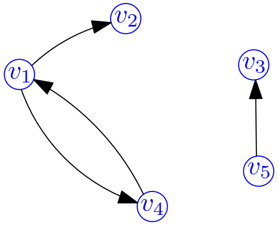
Queue

- This is called *FIFO*: first in, first out.
- A queue can be implemented with a doubly linked list.
- Then each operation takes $O(1)$ time.
- Can also be implemented with a singly linked list, and keep a pointer to the tail of the list.

Queue

- This is called *FIFO*: first in, first out.
- A queue can be implemented with a doubly linked list.
- Then each operation takes $O(1)$ time.
- Can also be implemented with a singly linked list, and keep a pointer to the tail of the list.
- We can also use an array, seen as a circular list, and keep track of the index of the head and tail.

Directed Graphs



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$n = 5$

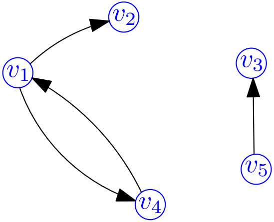
$E = \{(v_1, v_2), (v_1, v_4), (v_4, v_1), (v_5, v_3)\}$

$m = 4$

Directed graphs

A *directed graph* $G(V, E)$ consists of a set V of *vertices* and a set $E \subset V \times V$ of *edges*.

Directed Graphs



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$n = 5$

$E = \{(v_1, v_2), (v_1, v_4), (v_4, v_1), (v_5, v_3)\}$

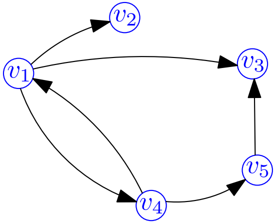
$m = 4$

Directed graphs

A *directed graph* $G(V, E)$ consists of a set V of *vertices* and a set $E \subset V \times V$ of *edges*.

- So an edge is an *ordered pair* of vertices.
- A vertex may also be called a *node*.
- Usually, the number of vertices is denoted $n = |V|$ and the number of edges is denoted $m = |E|$.

Adjacency Lists



$L(v_1) = \{v_2, v_3, v_4\}$

$L(v_2) = \emptyset$

$L(v_3) = \emptyset$

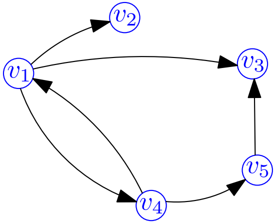
$L(v_4) = \{v_1, v_5\}$

$L(v_5) = \{v_3\}$

Adjacency lists

The *adjacency list* $L(v_i)$ of v_i is the set of vertices v_j such that $(v_i, v_j) \in E$.

Adjacency Lists



$L(v_1) = \{v_2, v_3, v_4\}$

$L(v_2) = \emptyset$

$L(v_3) = \emptyset$

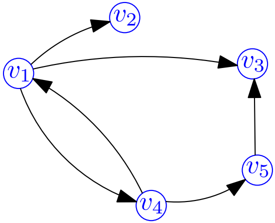
$L(v_4) = \{v_1, v_5\}$

$L(v_5) = \{v_3\}$

Adjacency lists

The *adjacency list* $L(v_i)$ of v_i is the set of vertices v_j such that $(v_i, v_j) \in E$. These vertices v_j are called the *neighbors* of v_i, and are said to be *adjacent* to v_i.

Adjacency Lists



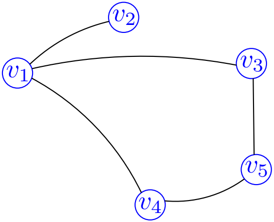
$$L(v_1) = \{v_2, v_3, v_4\}$$
$$L(v_2) = \emptyset$$
$$L(v_3) = \emptyset$$
$$L(v_4) = \{v_1, v_5\}$$
$$L(v_5) = \{v_3\}$$

Adjacency lists

The *adjacency list* $L(v_i)$ of v_i is the set of vertices v_j such that $(v_i, v_j) \in E$. These vertices v_j are called the *neighbors* of v_i, and are said to be *adjacent* to v_i.

- So a directed graph can be represented by a list of vertices, and an adjacency list for each vertex.

Undirected Graphs



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_5\}, \{v_3, v_5\}, \{v_4, v_5\}\}$

$L(v_1) = \{v_2, v_3, v_4\}$

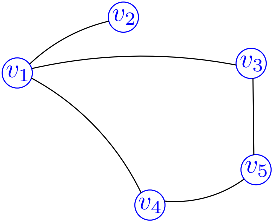
$L(v_2) = \{v_1\}$ $\qquad L(v_4) = \{v_1, v_5\}$

$L(v_3) = \{v_1, v_5\}$ $\qquad L(v_5) = \{v_3, v_4\}$

Directed graphs

An *undirected graph* $G(V, E)$ consists of a set V of *vertices* and a set E of *edges*. Each edge is an *unordered* pair of vertices.

Undirected Graphs



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_5\}, \{v_3, v_5\}, \{v_4, v_5\}\}$

$L(v_1) = \{v_2, v_3, v_4\}$

$L(v_2) = \{v_1\}$ $\qquad$ $L(v_4) = \{v_1, v_5\}$

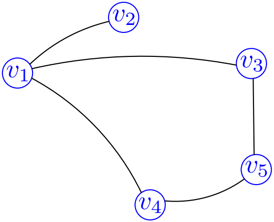
$L(v_3) = \{v_1, v_5\}$ $\qquad$ $L(v_5) = \{v_3, v_4\}$

Directed graphs

An *undirected graph* $G(V, E)$ consists of a set V of *vertices* and a set E of *edges*. Each edge is an *unordered* pair of vertices.

- Two vertices v_i, v_j are said to be adjacent, or neighbors, if $\{v_i, v_j\}$ is an edge.

Undirected Graphs



$$V = \{v_1, v_2, v_3, v_4, v_5\}$$

$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_5\}, \{v_3, v_5\}, \{v_4, v_5\}\}$$

$$L(v_1) = \{v_2, v_3, v_4\}$$

$$L(v_2) = \{v_1\} \qquad L(v_4) = \{v_1, v_5\}$$

$$L(v_3) = \{v_1, v_5\} \qquad L(v_5) = \{v_3, v_4\}$$

Directed graphs

An *undirected graph* $G(V, E)$ consists of a set V of *vertices* and a set E of *edges*. Each edge is an *unordered* pair of vertices.

- Two vertices v_i, v_j are said to be adjacent, or neighbors, if $\{v_i, v_j\}$ is an edge.
- We can also represent an undirected graph using adjacency lists.

Depth-First Search (DFS)

- *Depth-first search* (DFS) is an algorithm that, starting from a node s, finds all the nodes v such that there is a path from s to v in the graph.

Depth-First Search (DFS)

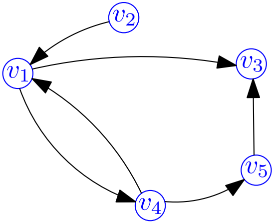
- *Depth-first search* (DFS) is an algorithm that, starting from a node s, finds all the nodes v such that there is a path from s to v in the graph.
- Initially, all nodes are *unmarked*.
- Then we call DFS(s).

Pseudocode

```
procedure DFS(node u)
    mark u
    for each v ∈ L(u) do
        if v is unmarked then
            DFS(v)
```

- It applies to directed and undirected graphs.

Example



$$L(v_1) = \{v_3, v_4\}$$
$$L(v_2) = \emptyset$$
$$L(v_3) = \emptyset$$
$$L(v_4) = \{v_1, v_5\}$$
$$L(v_5) = \{v_3\}$$

- Suppose we run DFS from v_4.
- Then nodes v_1, v_3, v_5 are visited in this order.
- v_2 remains unmarked.

Analysis

Proposition

DFS runs in

Analysis

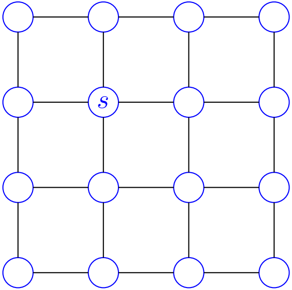
Proposition

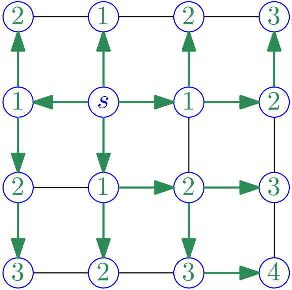
DFS runs in $O(n+m)$ time.

Proof.

We need $O(n)$ time to unmark all vertices. Then DFS is called once for each edge (twice for undirected graphs). □

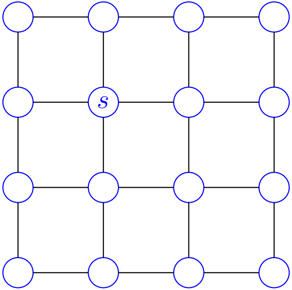
Breadth-First Search (BFS)

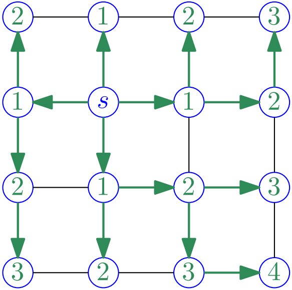




- *Breadth-first search* (BFS) visits the same set of nodes as DFS, but in a different order.

Breadth-First Search (BFS)





- *Breadth-first search* (BFS) visits the same set of nodes as DFS, but in a different order.
- In addition, it computes:
  - The distance from s to all visited nodes.
  - A tree T rooted at s, such that the shortest path from s to all nodes within T is also a shortest path in G.

Breadth-First Search (BFS)

Pseudocode

```
procedure BFS(G(V, E), s ∈ V)
    Q ← new queue containing only s
    T ← empty tree T(V, ∅)
    d ← array of n integers
    unmark all nodes
    mark s
    d(s) = 0
    while Q is nonempty do
        u ← Q.dequeue
        for each v ∈ L(u) do
            if v is unmarked then
                mark v
                enqueue v
                add edge (u, v) to T
                d(v) ← d(u) + 1                ▷ distance from s to u
```

Breadth-First Search (BFS)

- Proof of correctness (sketch): The queue ensures that nodes are visited by nondecreasing distance from s.
- Analysis:

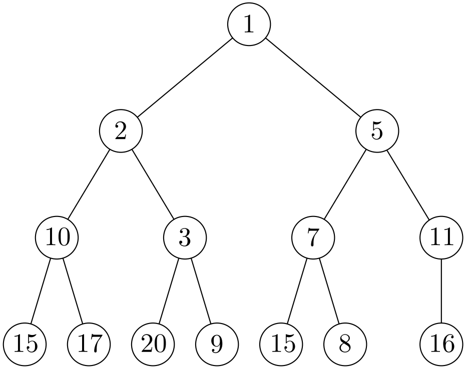
Breadth-First Search (BFS)

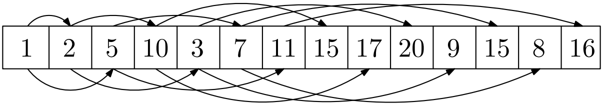
- Proof of correctness (sketch): The queue ensures that nodes are visited by nondecreasing distance from s.
- Analysis: Each node and edge is visited once, so

Proposition

BFS runs in $O(m+n)$ time.

Heaps





Heaps

- A *heap* is a binary tree such that each node v contains a number $\mathrm{key}(v)$ 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 $\mathrm{key}(v) \leqslant \mathrm{key}(w)$.

Heaps

- A *heap* is a binary tree such that each node v contains a number $\mathrm{key}(v)$ 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 $\mathrm{key}(v) \leqslant \mathrm{key}(w)$.

- 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 $H[\lfloor i/2 \rfloor]$.

Heaps

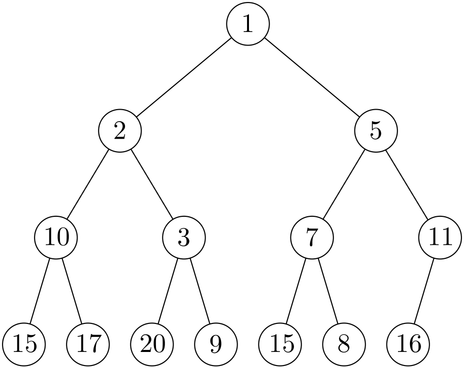
- A *heap* is a binary tree such that each node v contains a number $\text{key}(v)$ called a *key*, and possibly satellite data.
- The nodes of a heap have the *heap property*:

Property

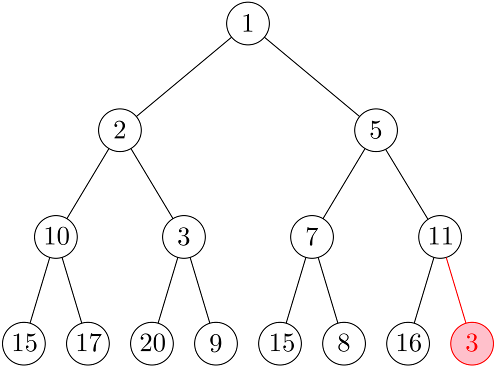
If v is the parent of w, then $\text{key}(v) \leqslant \text{key}(w)$.

- 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 $H[\lfloor i/2 \rfloor]$.
- When the heap records $n \leqslant N$ nodes, then they are recorded in $H[1 \ldots n]$.

Insertions

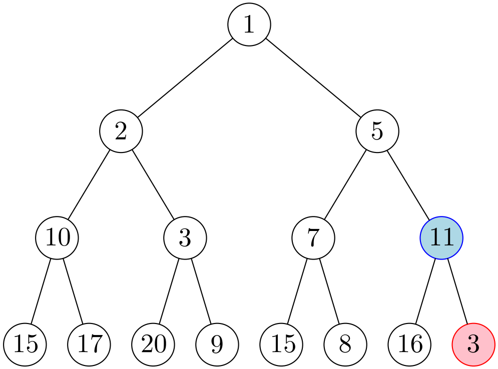


Insertions



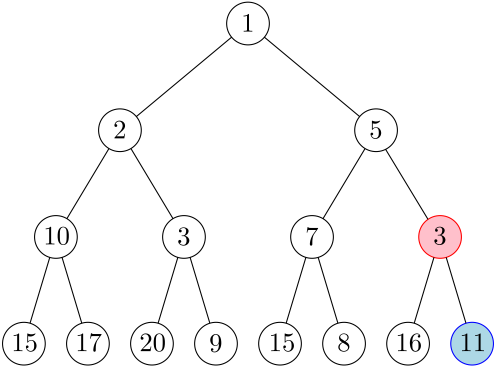
The new node is inserted at the last position

Insertions



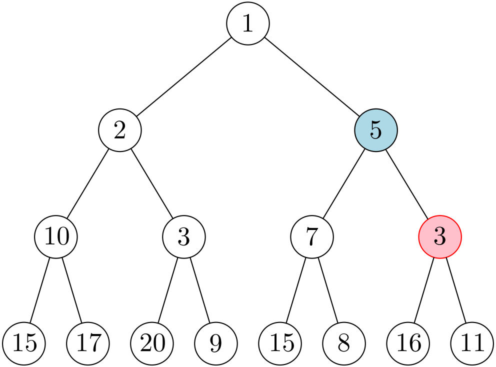
The heap property does not hold for the new node

Insertions



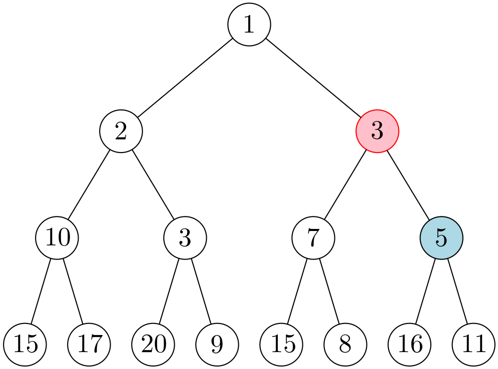
Fixing the heap

Insertions



The heap property does not hold

Insertions



Now the heap is fixed

Insertions

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

- It takes time

Insertions

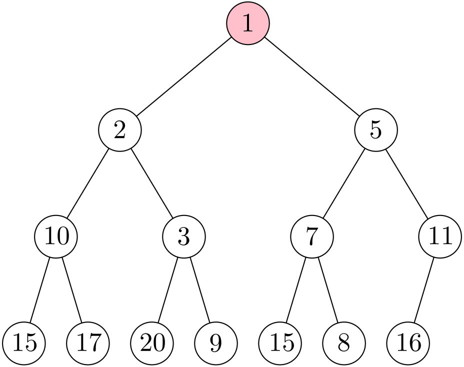
- If the heap contains n nodes, the new node is inserted at $H[n+1]$.
- Then we fix the heap by calling $\textsc{Heapify-up}(H, n+1)$

Pseudocode

```
procedure Heapify-up(H, i)
    if i > 1 then
        p ← ⌊i/2⌋                              ▷ p is the parent of i
        if key(H[p]) > key(H[i]) then
            Swap the contents of H[i] and H[p]
            Heapify-up(H, p)
```

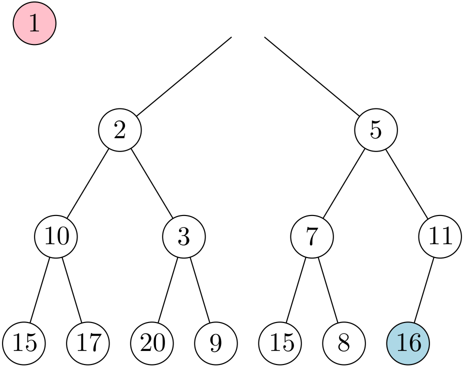
- It takes time $O(\log n)$ because i gets halved at each recursive call.

Extracting the Minimum



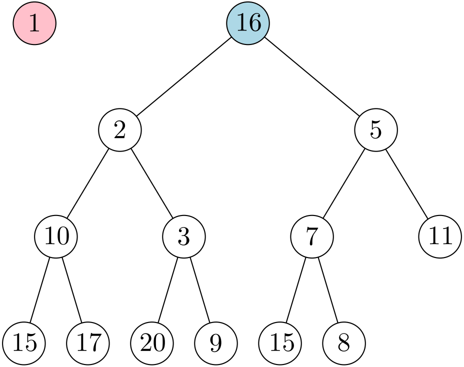
The minimum is at the root.

Extracting the Minimum



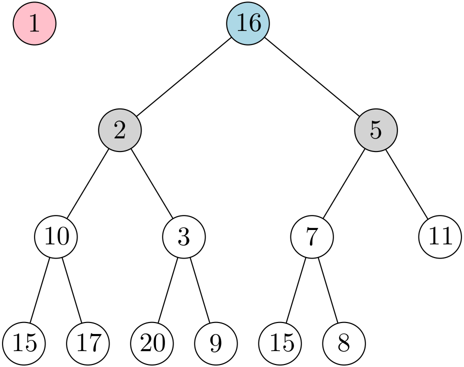
After we extract the minimum, a hole is left at the root.

Extracting the Minimum



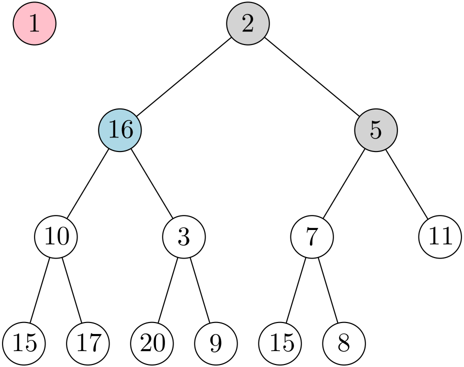
We move the last element to the root.

Extracting the Minimum



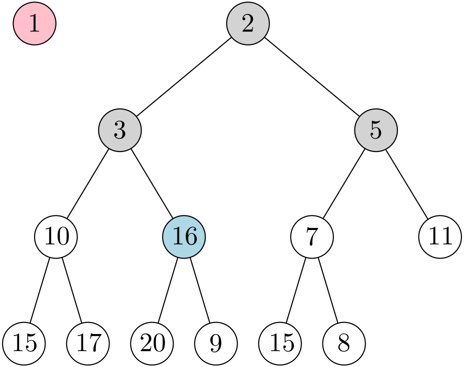
The heap property is violated.

Extracting the Minimum



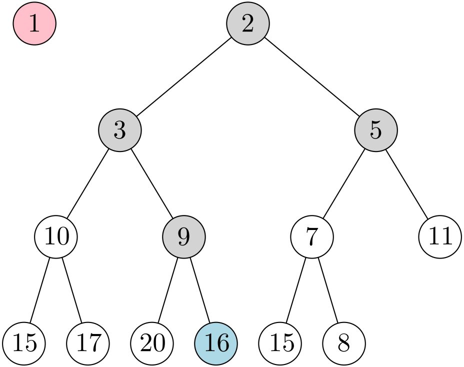
Fixing the heap.

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 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
            Swap the contents of H[i] and H[j]
            i ← j
        else
            return
```

- This procedure runs in time

Extracting the Minimum

Pseudocode

procedure HEAPIFY-DOWN(H)
  $n \leftarrow$ length(H)
  $i \leftarrow 1$
  while $2i \leqslant n$ **do**
    $j \leftarrow$ the index of the child of i with smallest key.
    if key($H[i]$) $>$ key($H[j]$) **then**
      Swap the contents of $H[i]$ and $H[j]$
      $i \leftarrow j$
    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.

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 HEAPIFY-UP(H, i)
  - Otherwise, if the key of $H[i]$ is larger than one of its child, call a modified version of 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

- We can sort a set of n numbers by inserting them all into a heap, and then extracting the minimum repeatedly.
- It takes

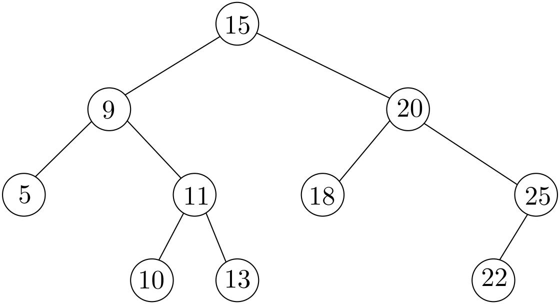
Remarks

- 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.

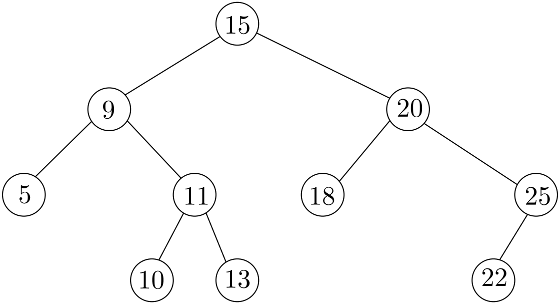
Remarks

- 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 way of sorting using a heap, called HEAPSORT, that inserts all the elements in $O(n)$ time, but still needs $\Theta(\log n)$ time for each extraction. (Not covered in CSE515.)

Binary Search Trees



Binary Search Trees

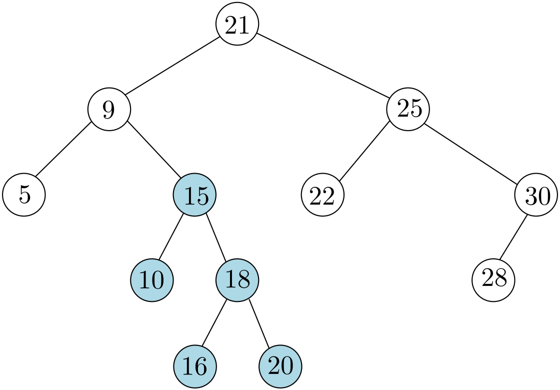


Definition (Binary search tree)

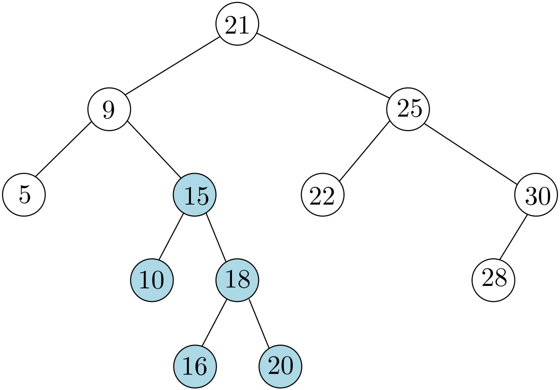
A *binary search tree (BST)* T is a binary tree that records a key at each node. Every node v of T has the following properties.

- 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



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 $1 \leqslant i \leqslant j \leqslant n$.

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

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.

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.
- In this lecture we do not use satellite data.

Insertion into a BST

Inserting key k into a BST

```
procedure INSERT(r, k)
    if r = NIL then
        r ← NEWNODE(k)
    else if k < key(r) then
        INSERT(left(r), k)
    else
        INSERT(right(r), k)
```

- The new key k is inserted from the *root* node r of the tree T.

Insertion into a BST

Inserting key k into a BST

```
procedure Insert(r, k)
    if r = NIL then
        r ← NewNode(k)
    else if k < key(r) then
        Insert(left(r), k)
    else
        Insert(right(r), k)
```

- The new key k is inserted from the *root* node r of the tree T.
- The root node is the only node without parent.

Insertion into a BST

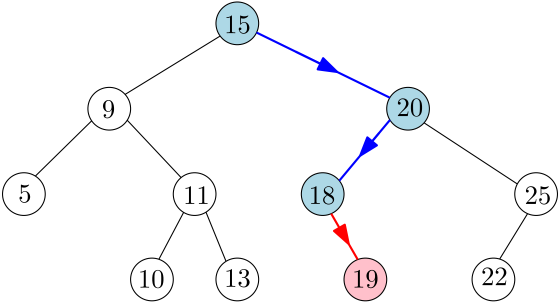
Inserting key k into a BST

```
procedure INSERT(r, k)
    if r = NIL then
        r ← NEWNODE(k)
    else if k < key(r) then
        INSERT(left(r), k)
    else
        INSERT(right(r), k)
```

- The new key k is inserted from the *root* node r of the tree T.
- The root node is the only node without parent.
- Insertion takes $O(h+1)$ time, where h is the height of the tree.

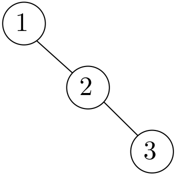
BST Insertion: Example

- Inserting 19 into the tree from Slide 42

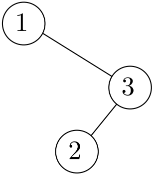


BST Insertion Orders

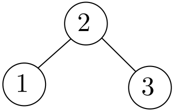
- The shape of a BST depends on the order of insertions.



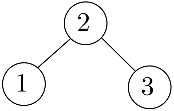
$1 \to 2 \to 3$



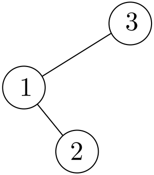
$1 \to 3 \to 2$



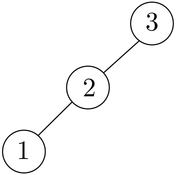
$2 \to 1 \to 3$



$2 \to 3 \to 1$



$3 \to 1 \to 2$



$3 \to 2 \to 1$

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))
```

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 42, 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 = NIL then
        return NOTFOUND
    if k < key(v) then
        return SEARCH(left(v), k)
    if k > key(v) then
        return SEARCH(right(v), k)
    return v                                  ▷ k = 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 $\Omega(\log n)$.

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* whose height is $O(\log n)$, so the search time is $\Theta(\log n)$.

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* whose height is $O(\log n)$, so the search time is $\Theta(\log n)$.
- It is also possible to insert and delete nodes in $\Theta(\log n)$ time in a balanced BST.

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* whose height is $O(\log n)$, so the search time is $\Theta(\log n)$.
- It is also possible to insert and delete nodes in $\Theta(\log n)$ time in a balanced BST.
  - It requires to rebalance (change the structure) of the BST while inserting/deleting.

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* whose height is $O(\log n)$, so the search time is $\Theta(\log n)$.
- It is also possible to insert and delete nodes in $\Theta(\log n)$ time in a balanced BST.
  - It requires to rebalance (change the structure) of the BST while inserting/deleting.
- So balanced BST 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.

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* whose height is $O(\log n)$, so the search time is $\Theta(\log n)$.
- It is also possible to insert and delete nodes in $\Theta(\log n)$ time in a balanced BST.
  - It requires to rebalance (change the structure) of the BST while inserting/deleting.
- So balanced BST 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.
- Balanced binary search trees are not covered in CSE515, but you should know that they exist. (Covered in CSE221 Data structures.)