

CSE331 Introduction to Algorithms

Lecture 14: Review of Elementary Data Structures and Graph Algorithms

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

July 11, 2017

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

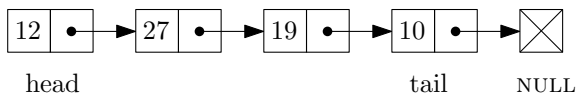
Introduction

- In this lecture, I will review elementary data structures and graph algorithms.
- They will be used in the next 3 lectures.
- Topics:
 - ▶ Linked lists, stacks and queues.
 - ▶ Heaps and priority queues.
 - ▶ Graph traversals (BFS, DFS).
- **Reference:** Sections 6, 10, and 22 of the textbook [Introduction to Algorithms](#) by Cormen, Leiserson, Rivest and Stein.
- I will not be following this textbook closely in this lecture.

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*
- 2-dimensional array: $B[1 \dots m, 1 \dots n]$
- Idem: access $B[i, j]$ in $O(1)$ time, create array in $O(mn)$ time
- Generalizes to any dimension

Linked Lists



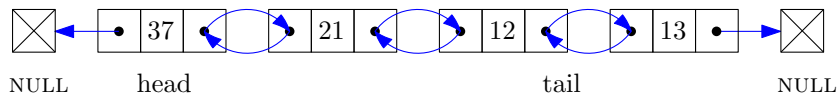
- Implementation: Each node in the list has two fields.

Node

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

- 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

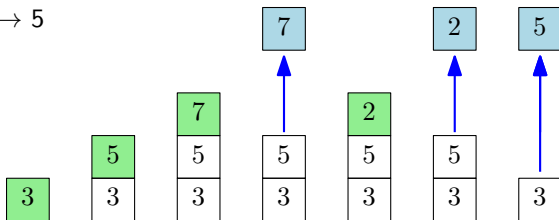
- head *reference to the head node*
- tail *reference to the tail node*

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

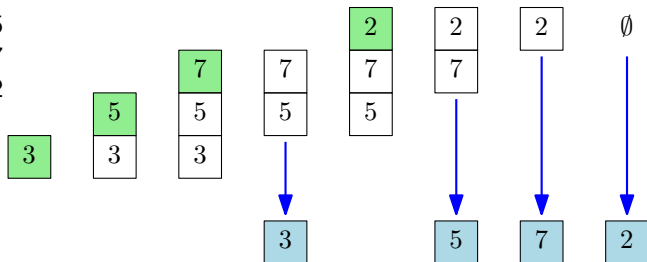


Stacks

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

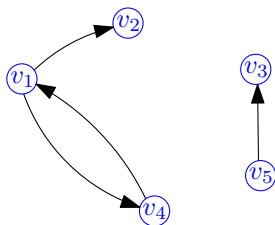
- A *queue* is an abstract data type with two operations:
 - ▶ enqueue: insert an element
 - ▶ dequeue: remove from the queue the earliest inserted element
- Example:
 - ▶ Start with empty stack
 - ▶ enqueue 3, enqueue 5, enqueue 7
 - ▶ dequeue \rightarrow 3
 - ▶ enqueue 2
 - ▶ dequeue \rightarrow 5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



Queue

- This is called *FIFO*: first in, first out.
- A stack 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 keeping track of the index of the head and tail.

Directed Graphs



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

$$n = 5$$

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

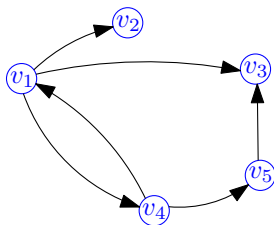
$$m = 4$$

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*.

- 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



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

$$L(v_2) = \emptyset$$

$$L(v_3) = \emptyset$$

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

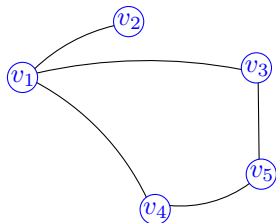
$$L(v_5) = \{v_3\}$$

Adjacency lists

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



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

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

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

$$L(v_2) = \{v_1\}$$

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

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

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

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)

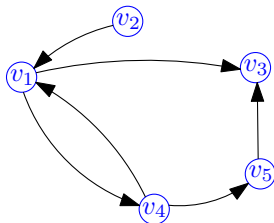
- *Depth-first search* (DFS) is an algorithm that, starting from a node s , finds all the nodes v such that there is a path from s to v in the graph.
- Initially, all nodes are *unmarked*.
- Then we call $\text{DFS}(s)$.

Pseudocode

```
1: procedure DFS(node  $u$ )
2:   mark  $u$ 
3:   for each  $v \in L(u)$  do
4:     if  $v$  is unmarked then
5:       DFS( $v$ )
```

- It applies to directed and undirected graphs.

Example



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

$$L(v_2) = \emptyset$$

$$L(v_3) = \emptyset$$

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

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

- 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

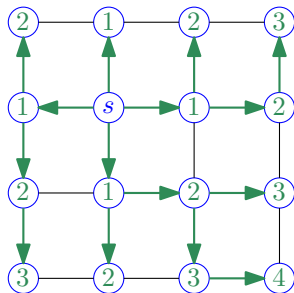
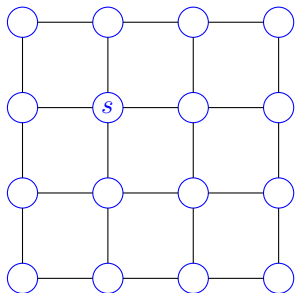
Proposition

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

```
1: procedure BFS( $G(V, E)$ ,  $s \in V$ )
2:    $Q \leftarrow$  new queue containing only  $s$ 
3:    $T \leftarrow$  empty tree  $T(V, \emptyset)$ 
4:    $d \leftarrow$  array of nodes
5:   unmark all nodes
6:   mark  $s$ 
7:    $d(s) = 0$ 
8:   while  $Q$  is nonempty do
9:      $u \leftarrow Q$ .dequeue
10:    for each  $v \in L(u)$  do
11:      if  $v$  is unmarked then
12:        mark  $v$ 
13:        enqueue  $v$ 
14:        add edge  $(u, v)$  to  $T$ 
15:         $d(v) \leftarrow d(u) + 1$  ▷ distance from  $s$  to  $u$ 
```

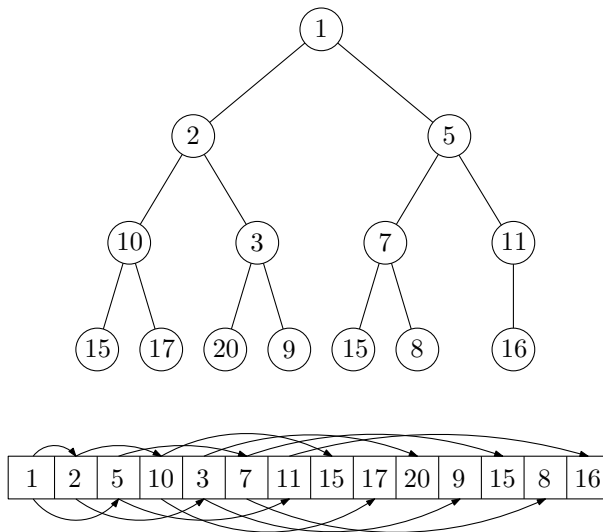
Breadth-First Search (BFS)

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

Proposition

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

Heaps



Heaps

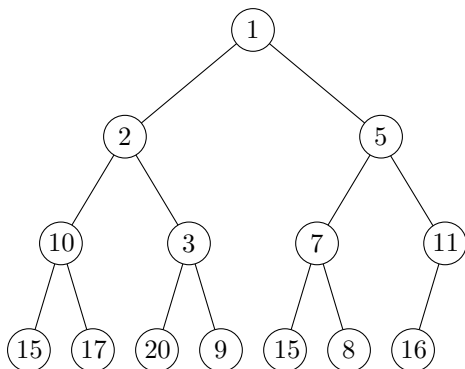
- A *heap* is a binary tree such that each node v contains a number $\text{key}(v)$ called a *key*, and possibly satellite data.
- The nodes of a heap have the *heap property*:

Property

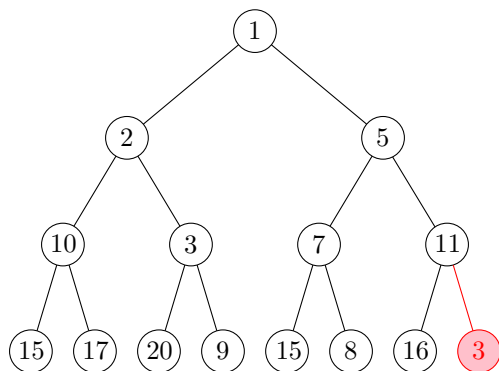
If v is the parent of w , then $\text{key}(v) \leq \text{key}(w)$.

- The heap is recorded in an array $H[1, \dots, N]$.
- 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 $n \leq N$ nodes, then they are recorded in $H[1 \dots n]$.

Insertions

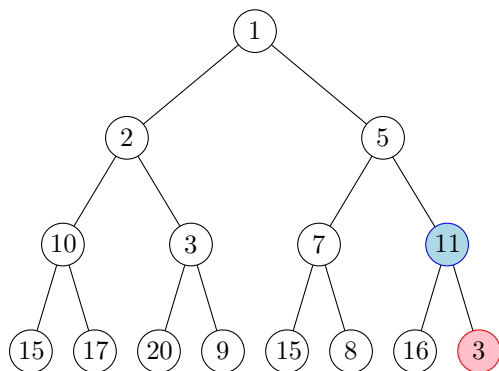


Insertions



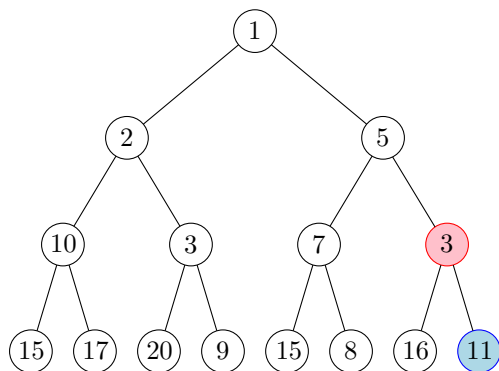
The new node is inserted at the last position

Insertions



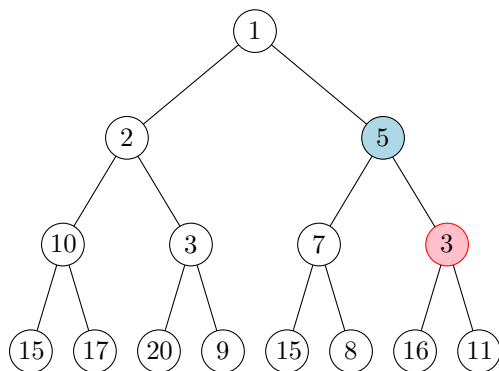
The heap property does not hold for the new node

Insertions



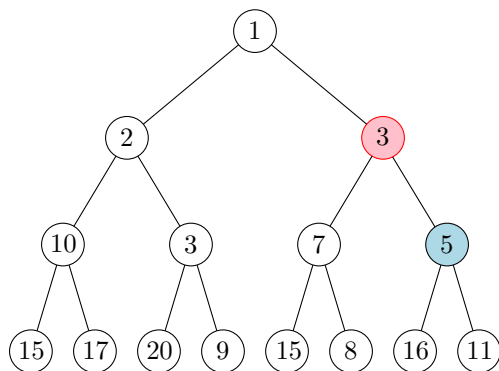
Fixing the heap

Insertions



The heap property does not hold

Insertions



Now the heap is fixed

Insertions

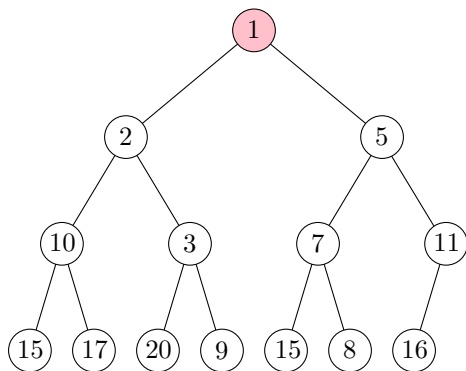
- If the heap contains n nodes, the new node is inserted at $H[n + 1]$.
- Then we fix the heap by calling $\text{HEAPIFY-UP}(H, n + 1)$

Pseudocode

```
1: procedure HEAPIFY-UP( $H, i$ )
2:   if  $i > 1$  then
3:      $p \leftarrow \lfloor i/2 \rfloor$  ▷  $p$  is the parent of  $i$ 
4:     if  $\text{key}(H[p]) > \text{key}(H[i])$  then
5:       Swap the contents of  $H[i]$  and  $H[p]$ 
6:       HEAPIFY-UP( $H, p$ )
```

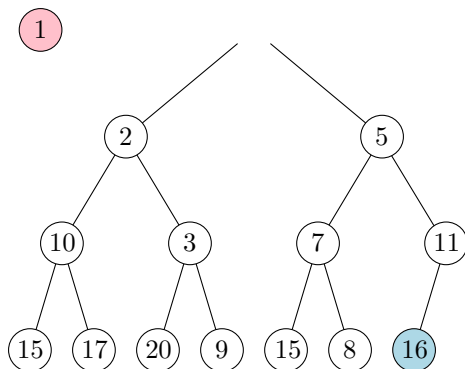
- It takes time $O(\log n)$ because i gets halved at each recursive call.

Extracting the Minimum



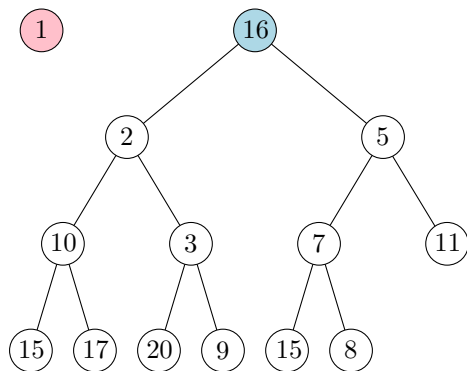
The minimum is at the root.

Extracting the Minimum



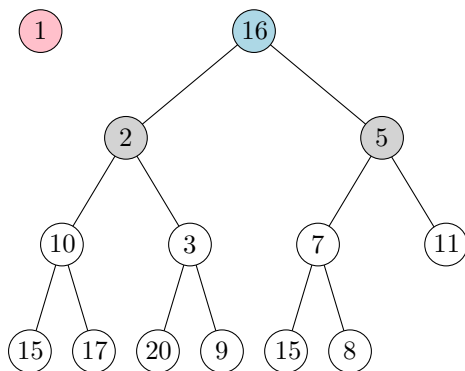
After we extract the minimum, a hole is left at the root.

Extracting the Minimum



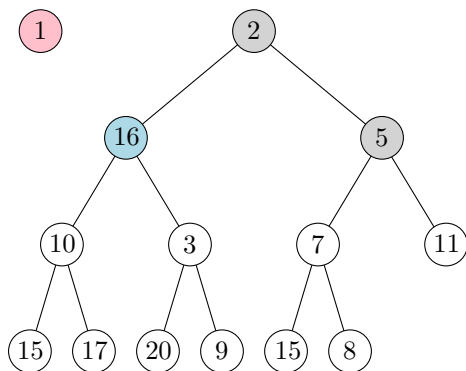
We move the last element to the root.

Extracting the Minimum



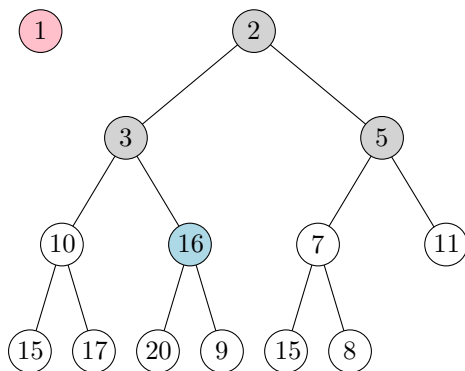
The heap property is violated.

Extracting the Minimum



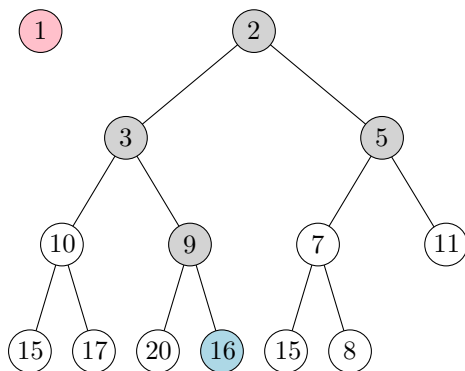
Fixing the heap.

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 $\text{HEAPIFY-DOWN}(H)$.
(See next slide.)

Extracting the Minimum

Pseudocode

```
1: procedure HEAPIFY-DOWN( $H$ )
2:    $n \leftarrow \text{length}(H)$ 
3:    $i \leftarrow 1$ 
4:   while  $2i \leq n$  do
5:      $j \leftarrow$  the index of the child of  $i$  with smallest key.
6:     if  $\text{key}(H[i]) > \text{key}(H[j])$  then
7:       Swap the contents of  $H[i]$  and  $H[j]$ 
8:        $i \leftarrow j$ 
9:     else
10:      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 $\text{HEAPIFY-UP}(H, i)$
 - ▶ 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

- What we described above is a *min heap*.
- In a *max heap*, the order is reversed: 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* so as to make the algorithm in-place.
 - ▶ 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.