Algorithm Complexity and Data Structure

AU4606: Network Optimization

AI4702: Network Intelligence and Optimization

Xiaoming Duan
Department of Automation
Shanghai Jiao Tong University

September 18, 2023

Last time

- Basics of graph theory
  - Graphs
  - Paths, cycles, walks
  - Degrees
  - Subgraphs
  - Connectivity
  - Components
  - Acyclic graphs
  - Trees
  - Bipartite graph
- Graph representations
  - Adjacency matrix
  - Incidence matrix
  - Adjacency list
- Network transformations

Today

1. Complexity analysis
   - Complexity measures
   - Asymptotic notation
2. Data structure
   - Why data structure?
   - Stacks and queues
   - d-heaps

Today

1. Complexity analysis
   - Complexity measures
   - Asymptotic notation

2. Data structure
   - Why data structure?
   - Stacks and queues
   - *d*-heaps

Solving a problem

Building blocks for solving a computational problem in computers

- A recipe, or algorithm: a step-by-step procedure
- Means for encoding this procedure in a computational device
- The application of the method to the data of a specific problem

Solving a problem

Building blocks for solving a computational problem in computers

- A recipe, or algorithm: a step-by-step procedure
- Means for encoding this procedure in a computational device
- The application of the method to the data of a specific problem

Key question: how do we measure algorithms' efficiency? (from 1970s)

- Computing resources needed for executing an algorithm
    1. Storage space (space complexity)
    2. Running time (time complexity)

Solving a problem

Building blocks for solving a computational problem in computers

- A recipe, or algorithm: a step-by-step procedure
- Means for encoding this procedure in a computational device
- The application of the method to the data of a specific problem

Key question: how do we measure algorithms' efficiency? (from 1970s)

- Computing resources needed for executing an algorithm
  1. Storage space (space complexity)
  2. Running time (time complexity)

Time complexity is usually measured in terms of "basic" operations

- Assignment steps
- Arithmetic steps (e.g., addition, subtraction, multiplication, division)
- Logical steps (e.g., conditional statement, comparisons)

# of steps performed by an algorithm = total # of basic operations

Adding two matrices

Algorithm Adding two matrices A and B

for $i = 1 : m$ **do**
  for $j = 1 : n$ **do**
    $C(i,j) = A(i,j) + B(i,j)$
  end for
end for

- $\#$ of additions: mn
- $\#$ of assignments: mn
- Total operations: $2mn$

Perhaps also $\#$ of accessing steps? $2mn$

Complexity measures

- Algorithms are applied to a class of problems
- One algorithm may take different time for different problem instances
- An algorithm may solve "good" instances quickly, but "bad" slowly

- Algorithms are applied to a class of problems
- One algorithm may take different time for different problem instances
- An algorithm may solve "good" instances quickly, but "bad" slowly

Complexity measures

Complexity measures

- Algorithms are applied to a class of problems
- One algorithm may take different time for different problem instances
- An algorithm may solve "good" instances quickly, but "bad" slowly

Complexity measures

- Empirical analysis: run the algorithm on many instances
  1. Pros: no analysis on algorithms required
  2. Cons: dependence on various factors; time consuming

Complexity measures

- Algorithms are applied to a class of problems
- One algorithm may take different time for different problem instances
- An algorithm may solve "good" instances quickly, but "bad" slowly

Complexity measures

- Empirical analysis: run the algorithm on many instances
  1. Pros: no analysis on algorithms required
  2. Cons: dependence on various factors; time consuming
- Average-case analysis: analyze alg. on instances and take average
  1. Pros: indicative when solving large number of different instances
  2. Cons: distributions of problem instances; difficult analysis

Complexity measures

- Algorithms are applied to a class of problems
- One algorithm may take different time for different problem instances
- An algorithm may solve "good" instances quickly, but "bad" slowly

Complexity measures

- Empirical analysis: run the algorithm on many instances
  1. Pros: no analysis on algorithms required
  2. Cons: dependence on various factors; time consuming
- Average-case analysis: analyze alg. on instances and take average
  1. Pros: indicative when solving large number of different instances
  2. Cons: distributions of problem instances; difficult analysis
- Worst-case analysis: analyze algorithm on "hardest" instance
  1. Pros: provides conclusive guarantees on how algorithms perform
  2. Cons: pathological cases

How to measure complexity

Algorithm Adding two matrices A and B

```
for i = 1 : m do
    for j = 1 : n do
        C(i, j) = A(i, j) + B(i, j)
    end for
end for
```

Takes roughly $2mn$ basic operations (time steps)

- Number of basic steps required depends on the problem instance
- Measure the complexity of algorithms in terms of "problem sizes"

Problem sizes

Algorithm Adding two matrices A and B

```
for i = 1 : m do
    for j = 1 : n do
        C(i, j) = A(i, j) + B(i, j)
    end for
end for
```

Problem sizes: # of bits to encode the problem data

- Adding matrices: $mn \log_2 M$, where M largest element in A and B
- Network flow problem
  1. Number of nodes n
  2. Number of arcs m
  3. Arc cost coefficient c_{ij}
  4. Arc capacity u_{ij}

  problem size approximately:

$$n \log n + m \log m + m \log C + m \log U$$

where $C = \max_{(i,j)\in A} c_{ij}$ and $U = \max_{(i,j)\in A} u_{ij}$

Polynomial time algorithms

- Polynomial-time algorithm: worst-case complexity is bounded by a polynomial function of the problem size, i.e., it is a polynomial function of n, m, $\log C$, and $\log U$
  - mn
  - n^2
  - $m + n \log C$
- Strongly polynomial-time algorithm if does not involve $\log C$ or $\log U$,
  - n
  - $n^2 m$
- Otherwise, a weakly polynomial-time algorithm
  - $m + n \log C$

Note: algorithms having complexity mnU is exponential!

Algorithm complexity with asymptotic notations

- We usually only care about the order of # of steps
- Ignore (distracting) constant factors

Algorithm Adding two matrices A and B

```
for i = 1 : m do
    for j = 1 : n do
        C(i,j) = A(i,j) + B(i,j)
    end for
end for
```

Takes roughly $2mn$ basic operations (time steps)
or $4mn$ steps

Algorithm complexity with asymptotic notations

- We usually only care about the order of # of steps
- Ignore (distracting) constant factors

Algorithm Adding two matrices A and B

for $i = 1 : m$ **do**
  for $j = 1 : n$ **do**
    $C(i,j) = A(i,j) + B(i,j)$
  end for
end for

Takes roughly $2mn$ basic operations (time steps)
or $4mn$ steps

Usually written as $O(mn)$

Asymptotic notation: big oh

Definition of Big Oh

Given two nonnegative functions $f, g : \mathbb{R} \to \mathbb{R}$, we say that

$$f = O(g)$$

if

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} < \infty$$

Asymptotic notation: big oh

Definition of Big Oh

Given two nonnegative functions $f, g : \mathbb{R} \to \mathbb{R}$, we say that

$$f = O(g)$$

if

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} < \infty$$

Definition of Big Oh

For $f, g : \mathbb{R} \to \mathbb{R}$, we say that $f = O(g)$ if there exists a constant $c > 0$ and an x_0 such that for all $x \geq x_0$, $f(x) \leq cg(x)$.

- $2x = O(x)$
- $x = O(x^2)$
- $10^8 x^2 + 3x + 2 = O(x^2)$
- $2^x + x^{10000} + 3 = O(2^x)$
- $c = O(1)$ for any $c > 0$

Suppose you want to make a statement of the form "the running time of the algorithm is a least...". Can you say it is "at least $O(n^2)$"?

Asymptotic notation: big omega

Suppose you want to make a statement of the form "the running time of the algorithm is a least...". Can you say it is "at least $O(n^2)$"? NO!

Definition of Big Omega

Given two nonnegative functions $f, g : \mathbb{R} \to \mathbb{R}$, we say that

$$f = \Omega(g)$$

if there exists a constant $c > 0$ and an x_0 such that for all $x \geq x_0$, $f(x) \geq cg(x)$.

Examples

- $x^2 = \Omega(x)$
- $2^x = \Omega(x^2)$
- $\frac{x}{100} = \Omega(100x + 25)$

Big Oh and Big Omega

$f(x) = O(g(x))$ if and only if $g(x) = \Omega(f(x))$.

Asymptotic notation: little oh

Definition of Big Oh

Given two nonnegative functions $f, g : \mathbb{R} \to \mathbb{R}$, we say that

$$f = O(g)$$

if

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} < \infty$$

What if we want to say some function is "strictly dominated" by another?

Asymptotic notation: little oh

Definition of Big Oh

Given two nonnegative functions $f, g : \mathbb{R} \to \mathbb{R}$, we say that

$$f = O(g)$$

if

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} < \infty$$

What if we want to say some function is "strictly dominated" by another?

Definition of Little Oh

Given two nonnegative functions $f, g : \mathbb{R} \to \mathbb{R}$, we say that

$$f = o(g)$$

if

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$$

Asymptotic notation: little oh

Definition of Little Oh

Given two nonnegative functions $f, g : \mathbb{R} \to \mathbb{R}$, we say that

$$f = o(g)$$

if

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$$

Examples

- $x^{0.99999} = o(x)$
- $\log x = o(x^{\epsilon})$ for any $\epsilon > 0$
- $\frac{1}{x} = o(1)$

Asymptotic notation: little oh

Maximum Flow and Minimum-Cost Flow in Almost-Linear Time

Publisher: IEEE Cite This PDF

Li Chen ; Rasmus Kyng ; Yang P. Liu ; Richard Peng ; Maximilian Probst Gutenberg ; Sushant Sachdeva **All Authors**

Abstract:
We give an algorithm that computes exact maximum flows and minimum-cost flows on directed graphs with m edges and polynomially bounded integral demands, costs, and capacities in $m^{1+o(1)}$ time. Our algorithm builds the flow through a sequence of $m^{1+o(1)}$ approximate undirected minimum-ratio cycles, each of which is computed and processed in amortized $m^{o(1)}$ time using a new dynamic graph data structure. Our framework extends to algorithms running in $m^{1+o(1)}$ time for computing flows that minimize general edge-separable convex functions to high accuracy. This gives almost-linear time algorithms for several problems including entropy-regularized optimal transport, matrix scaling, p-norm flows, and p-norm isotonic regression on arbitrary directed acyclic graphs.

***Abstract*—We give an algorithm that computes exact maximum flows and minimum-cost flows on directed graphs with m edges and polynomially bounded integral demands, costs, and capacities in $m^{1+o(1)}$ time. Our algorithm builds the flow through a sequence of $m^{1+o(1)}$ approximate undirected minimum-ratio cycles, each of which is computed and processed in amortized $m^{o(1)}$ time using a new dynamic graph data structure.**

Asymptotic notation: little omega

Definition of Little Omega

Given two nonnegative functions $f, g : \mathbb{R} \to \mathbb{R}$, we say that

$$f = \omega(g)$$

if

$$\lim_{x \to \infty} \frac{g(x)}{f(x)} = 0$$

Little Oh and Little Omega

$f(x) = o(g(x))$ if and only if $g(x) = \omega(f(x))$.

Examples

- $x^{1.5} = \omega(x)$
- $\sqrt{x} = \omega(\log^2 x)$

Asymptotic notation: theta

Definition of Theta

Given two nonnegative functions $f, g : \mathbb{R} \to \mathbb{R}$, then

$$f = \Theta(g) \qquad \text{if and only if} \qquad f = O(g) \text{ and } g = O(f)$$

Two functions grow equally fast

Examples

- $10x^3 - 20x^2 + 1 = \Theta(x^3)$
- $\pi^2 3^{x-7} + \frac{(2.7x^{133}+x^9-86)^4}{\sqrt{x}} - 1.08^{3x} = \Theta(3^x)$

Asymptotic notation: tilde

Definition of Tilde

Given two nonnegative functions $f, g : \mathbb{R} \to \mathbb{R}$, we say f is asymptotically equal to g, in symbols,

$$f \sim g$$

if

$$\lim_{x\to\infty} \frac{g(x)}{f(x)} = 1$$

Immediately

$$f \sim g \implies \begin{cases} f = O(g), \\ g = O(f), \\ f = \Theta(g). \end{cases}$$

- $\frac{1}{2}x^2 + 3x - 2 \sim \frac{1}{2}x^2$
- $e^x + 3x^2 \sim e^x$

Asymptotic notation: confusions

We know that

- $2x = O(x^2)$
- $x^2 = O(x^2)$

Therefore, we have $2x = x^2$?

Asymptotic notation: confusions

We know that

- $2x = O(x^2)$
- $x^2 = O(x^2)$

Therefore, we have $2x = x^2$?

More mathematically precise notation is

$$f \in O(g)$$

Asymptotic notation: confusions

We know that

- $2x = O(x^2)$
- $x^2 = O(x^2)$

Therefore, we have $2x = x^2$?

More mathematically precise notation is

$$f \in O(g)$$

In fact, people write

- $f = O(g)$
- $f \leq O(g)$
- f is $O(g)$
- $f \in O(g)$

to mean the same thing

Asymptotic notation: intuitions

O	"means"	$\leq$
Ω	"means"	$\geq$
o	"means"	$<$
ω	"means"	$>$
Θ	"means"	$=$

Asymptotic notation: exercises

$$O \qquad \Omega \qquad o \qquad \omega \qquad \Theta$$

$$2n + \log n = \boxed{}(n)$$

$$\log n = \boxed{}(n)$$

$$\sqrt{n} = \boxed{}(\log^{300} n)$$

$$n2^n = \boxed{}(n)$$

$$n^2 = \boxed{}(1.01^n)$$

Mental break: galactic algorithm

Galactic algorithm (hiding constant factors)

- A galactic algorithm is one that outperforms other algorithms for problems that are sufficiently large, but where "sufficiently large" is so big that the algorithm is never used in practice

Mental break: galactic algorithm

Galactic algorithm (hiding constant factors)

- A galactic algorithm is one that outperforms other algorithms for problems that are sufficiently large, but where "sufficiently large" is so big that the algorithm is never used in practice
- Matrix multiplication
  - Naive algorithm takes $O(n^3)$
  - practical Strassen algorithm takes $O(n^{2.807})$
  - Galactic Coppersmith–Winograd algorithm takes $O(n^{2.373})$

Mental break: galactic algorithm

Galactic algorithm (hiding constant factors)

- A galactic algorithm is one that outperforms other algorithms for problems that are sufficiently large, but where "sufficiently large" is so big that the algorithm is never used in practice
- Matrix multiplication
  - Naive algorithm takes $O(n^3)$
  - practical Strassen algorithm takes $O(n^{2.807})$
  - Galactic Coppersmith–Winograd algorithm takes $O(n^{2.373})$

Are exponential algorithms always useless?

Galactic algorithm (hiding constant factors)

- A galactic algorithm is one that outperforms other algorithms for problems that are sufficiently large, but where "sufficiently large" is so big that the algorithm is never used in practice
- Matrix multiplication
  - Naive algorithm takes $O(n^3)$
  - practical Strassen algorithm takes $O(n^{2.807})$
  - Galactic Coppersmith–Winograd algorithm takes $O(n^{2.373})$

Are exponential algorithms always useless?

- Simplex methods have exponential complexity, but used very often

Mental break: galactic algorithm

Galactic algorithm (hiding constant factors)

- A galactic algorithm is one that outperforms other algorithms for problems that are sufficiently large, but where "sufficiently large" is so big that the algorithm is never used in practice
- Matrix multiplication
  - Naive algorithm takes $O(n^3)$
  - practical Strassen algorithm takes $O(n^{2.807})$
  - Galactic Coppersmith–Winograd algorithm takes $O(n^{2.373})$

Are exponential algorithms always useless?

- Simplex methods have exponential complexity, but used very often

Problem complexity vs algorithm complexity

Mental break: galactic algorithm

Galactic algorithm (hiding constant factors)

- A galactic algorithm is one that outperforms other algorithms for problems that are sufficiently large, but where "sufficiently large" is so big that the algorithm is never used in practice
- Matrix multiplication
  - Naive algorithm takes $O(n^3)$
  - practical Strassen algorithm takes $O(n^{2.807})$
  - Galactic Coppersmith–Winograd algorithm takes $O(n^{2.373})$

Are exponential algorithms always useless?

- Simplex methods have exponential complexity, but used very often

Problem complexity vs algorithm complexity

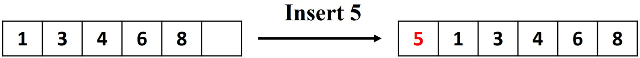
- Problem complexity: how much time does best algorithm take to solve
- Algorithm complexity: how much time does algorithm solve worst case

Today

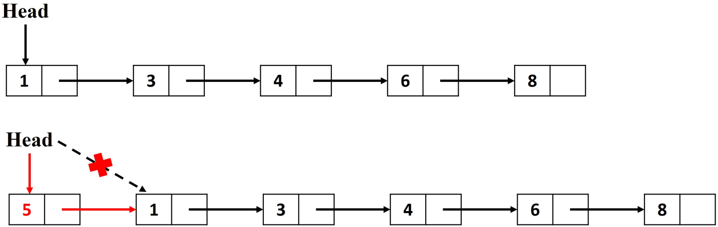
1. Complexity analysis
   - Complexity measures
   - Asymptotic notation
2. Data structure
   - Why data structure?
   - Stacks and queues
   - *d*-heaps

Why data structure?

- Operations can take different time on different data structure



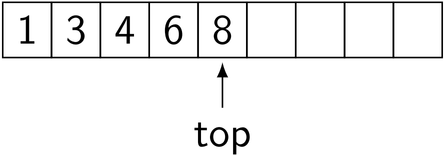
$O(n)$ time



$O(1)$ time

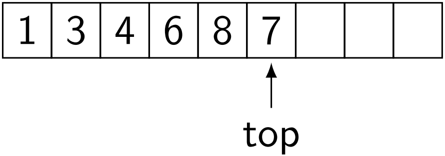
Two important data structures: stacks

- A stack is a special kind of ordered list (or set) in which all insertions and deletions take place at one end, called the top
  - Last-in-first-out



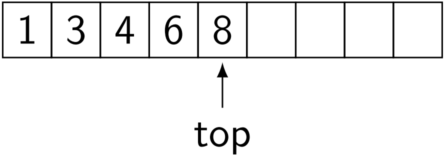
Two important data structures: stacks

- A stack is a special kind of ordered list (or set) in which all insertions and deletions take place at one end, called the top
  - Last-in-first-out



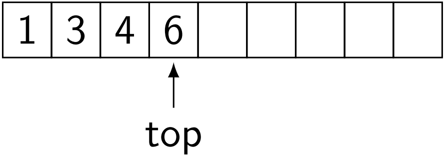
Add 7 to the stack

- A stack is a special kind of ordered list (or set) in which all insertions and deletions take place at one end, called the top
  - Last-in-first-out



Remove 7 from the stack

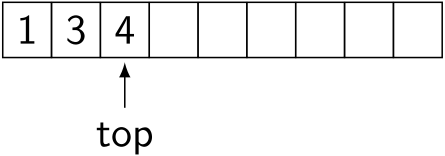
- A stack is a special kind of ordered list (or set) in which all insertions and deletions take place at one end, called the top
  - Last-in-first-out



Remove 8 from the stack

Two important data structures: stacks

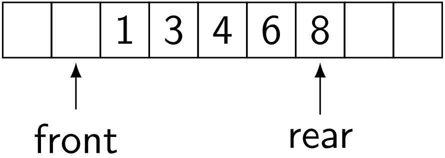
- A stack is a special kind of ordered list (or set) in which all insertions and deletions take place at one end, called the top
  - Last-in-first-out



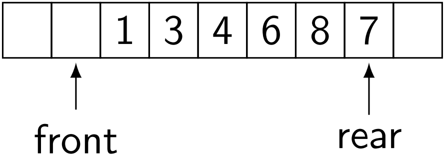
Remove 6 from the stack

Two important data structures: queues

- A queue is another special kind of list, with elements inserted at one end (the rear) and deleted from the other end (the front)
  - First-in-first-out



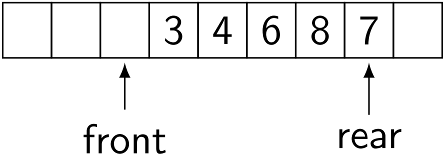
- A queue is another special kind of list, with elements inserted at one end (the rear) and deleted from the other end (the front)
  - First-in-first-out



7 enters the queue

Two important data structures: queues

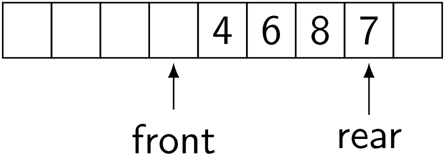
- A queue is another special kind of list, with elements inserted at one end (the rear) and deleted from the other end (the front)
  - First-in-first-out



1 leaves the queue

Two important data structures: queues

- A queue is another special kind of list, with elements inserted at one end (the rear) and deleted from the other end (the front)
  - First-in-first-out

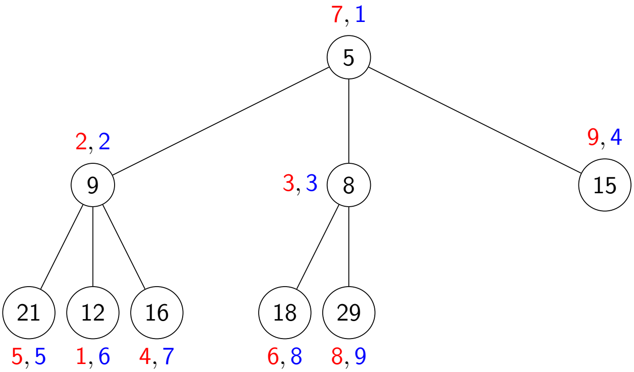


3 leaves the queue

d-heaps: operations

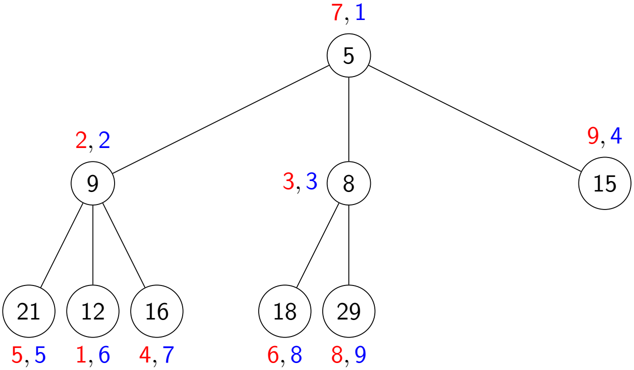
- Store and manipulate a collection H of elements when each element $i \in H$ has an associated real number $key(i)$
  - In shortest path problems, H is graph nodes, $key(i)$ is path length
- Basic operations
  1. create(H): create an empty heap H
  2. insert(i, H): insert an element i in the heap.
  3. find-min(i, H): find an element i with the minimum key in the heap.
  4. delete-min(i, H): delete the element i with the minimum key
  5. delete(i, H): delete an arbitrary element i from the heap.
  6. decrease-key(i, $value$, H): decrease the $key(i)$ to a smaller $value$
  7. increase-key(i, $value$, H): increase the $key(i)$ to a larger $value$
- The elements are stored as a rooted tree

d-heaps: properties

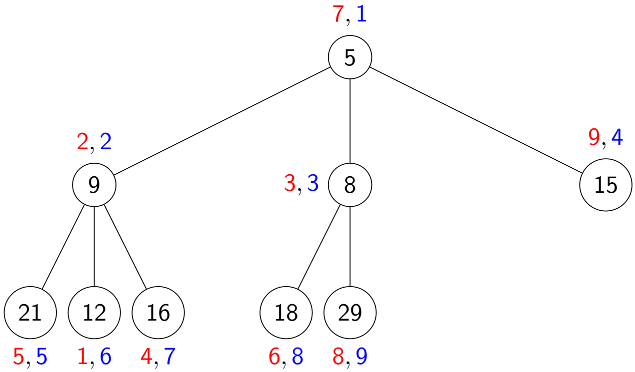


- Keys of elements are shown in the rooted tree
  1. Red indices are indices of elements (e.g., graph nodes)
  2. Blue indices are indices of elements in the tree
- Each node has at most d successors

d-heaps: properties

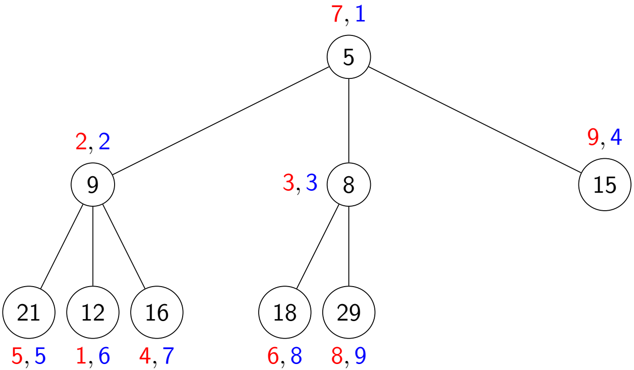


- Depth of a node: the number of arcs in the unique path to the root
  - node 8 has depth 2
- Nodes added in increasing order of depth values, and for the same depth, from left to right
  1. At most d^k nodes in depth k
  2. At most $(d^{k+1} - 1)/(d - 1)$ nodes between depth 0 and k
  3. The depth of an n-node d-heap is at most $\lfloor \log_d n \rfloor$



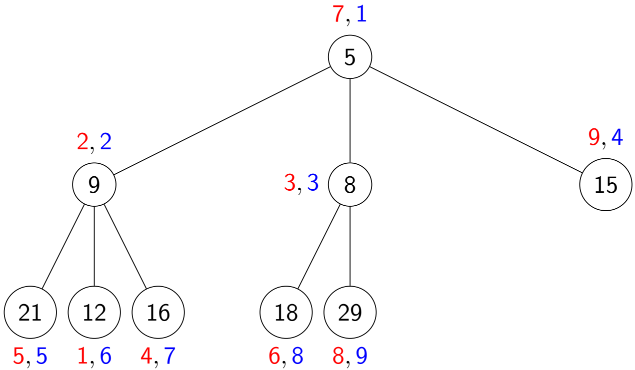
- Using an array with *last* being the number of nodes
  DHEAP=
  $\left[7:5,\ 2:9,\ 3:8,\ 9:15,\ 5:21,\ 1:12,\ 4:16,\ 6:18,\ 8:29\right]$
  last $= 9$
- Position array: position(i) $= j$, e.g., position(3) $= 3$, position(6) $= 8$

d-heaps: accessing predecessors and successors



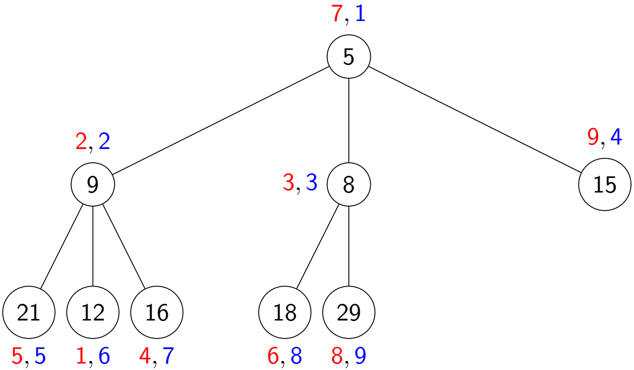
- Predecessor of node in position i is in position $\lceil (i-1)/d \rceil$
  e.g., Pred(8)=$\lceil (8-1)/3 \rceil$=3; Pred(6)=$\lceil (6-1)/3 \rceil$=2
- Successors of node in position i are in positions $id - d + 2, \ldots, id + 1$
  e.g., Succ(2)=$\{5, 6, 7\}$

d-heaps: accessing predecessors and successors



- Predecessor of node in position i is in position $\lceil (i-1)/d \rceil$
  e.g., Pred(6)= 3; Pred(1)=2
- Successors of node in position i are in positions $id - d + 2, \ldots, id + 1$
  e.g., Succ(2)={1, 4, 5}

d-heaps: order property



- Key of node i is less than or equal to each of its successors, i.e., $key(i) \leq key(j)$ for $j \in \text{Succ}(i)$
- The root node of the d-heap has the smallest key

d-heaps: swapping

- Heap operations are reduced to swaps that take $O(1)$ time
- swap(i, j): swap the positions of i and j

  before swap$(2, 7)$:

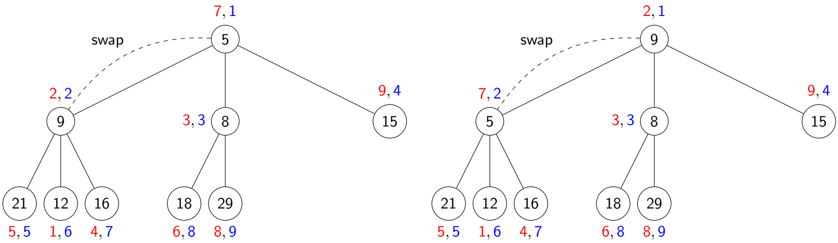
  $[7 : 5,\ 2 : 9,\ 3 : 8,\ 9 : 15,\ 5 : 21,\ 1 : 12,\ 4 : 16,\ 6 : 18,\ 8 : 29]$

  position$(2) = 2$, position$(7) = 1$

  after swap$(2, 7)$:

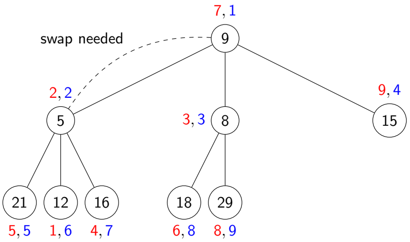
  $[2 : 9,\ 7 : 5,\ 3 : 8,\ 9 : 15,\ 5 : 21,\ 1 : 12,\ 4 : 16,\ 6 : 18,\ 8 : 29]$

  position$(2) = 1$, position$(7) = 2$

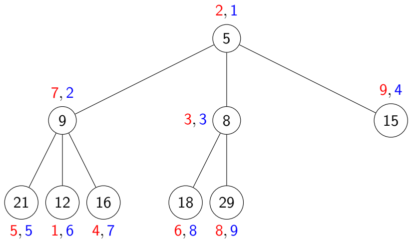


(a) Before swap (b) After swap

- Recall order property: $key(i) \leq key(j)$ for $j \in \text{Succ}(i)$
- Suppose $key(j)$ decreases and $key(j) < key(i)$ for some $j \in \text{Succ}(i)$
  - sift up



(a) Before swap

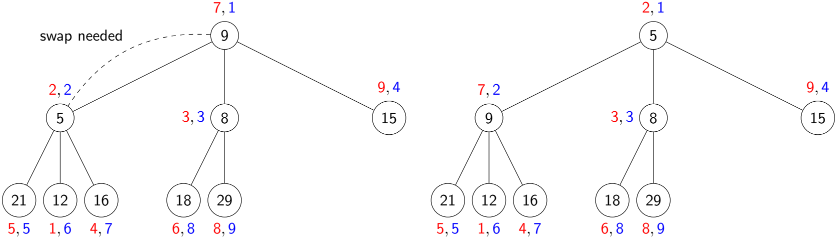


(b) After swap

$key(2)$ decreases to 5

- If node's key decreases, takes at most $O(\log_d n)$ to restore order

- Recall order property: $key(i) \leq key(j)$ for $j \in \text{Succ}(i)$
- Suppose $key(i)$ increases and $key(i) > key(j)$ for some $j \in \text{Succ}(i)$
  - sift down



(a) Before swap (b) After swap

$key(7)$ increases to 9

- If node's key increases, takes at most $O(d \cdot log_d n)$ to restore order

d-heaps: performing heap operations

1. find-min(i, H): root node, $O(1)$
2. insert(i, H): inset to the end, and swap up, $O(\log_d n)$
3. decrease-key(i, *value*, H): swap up $O(\log_d n)$
4. delete-min(i, H): make last node root, swap down $O(d \cdot log_d n)$
5. delete(i, H): fill with last node, swap down $O(d \cdot log_d n)$
6. increase-key(i, *value*, H): swap down $O(d \cdot log_d n)$

Sorting n elements?

d-heaps: performing heap operations

1. find-min(i, H): root node, $O(1)$
2. insert(i, H): inset to the end, and swap up, $O(\log_d n)$
3. decrease-key(i, $value$, H): swap up $O(\log_d n)$
4. delete-min(i, H): make last node root, swap down $O(d \cdot log_d n)$
5. delete(i, H): fill with last node, swap down $O(d \cdot log_d n)$
6. increase-key(i, $value$, H): swap down $O(d \cdot log_d n)$

Sorting n elements?

1. Create a d-heap: add one at a time and swap up $O(n \log_d n)$

d-heaps: performing heap operations

1. find-min(i, H): root node, $O(1)$
2. insert(i, H): inset to the end, and swap up, $O(\log_d n)$
3. decrease-key(i, $value$, H): swap up $O(\log_d n)$
4. delete-min(i, H): make last node root, swap down $O(d \cdot log_d n)$
5. delete(i, H): fill with last node, swap down $O(d \cdot log_d n)$
6. increase-key(i, $value$, H): swap down $O(d \cdot log_d n)$

Sorting n elements?

1. Create a d-heap: add one at a time and swap up $O(n \log_d n)$
2. Find minimum element and delete it n times, $O(n) + O(nd \cdot \log_d n)$

d-heaps: performing heap operations

1. find-min(i, H): root node, $O(1)$
2. insert(i, H): inset to the end, and swap up, $O(\log_d n)$
3. decrease-key(i, $value$, H): swap up $O(\log_d n)$
4. delete-min(i, H): make last node root, swap down $O(d \cdot log_d n)$
5. delete(i, H): fill with last node, swap down $O(d \cdot log_d n)$
6. increase-key(i, $value$, H): swap down $O(d \cdot log_d n)$

Sorting n elements?

1. Create a d-heap: add one at a time and swap up $O(n \log_d n)$
2. Find minimum element and delete it n times, $O(n) + O(nd \cdot \log_d n)$
3. Total: $O(nd \cdot \log_d n)$

Upcoming

Week 1-8 (AU4606 & AI4702):

- Introduction (1 lecture)
- Preparations (3 lectures)
  - basics of graph theory
  - algorithm complexity and data structure (this lecture)
  - graph search algorithm (next lecture)
- Shortest path problems (3 lectures)
- Maximum flow problems (5 lectures)
- Minimum cost flow problems (3 lectures)
- Introduction to multi-agent systems (1 lecture)
- Introduction to cloud networks (1 lecture)

Week 9-16 (AU4606):

- Simplex and network simplex methods (2 lectures)
- Global minimum cut problems (3 lectures)
- Minimum spanning tree problems (3 lectures)