Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Philip Bille

Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Algorithms and Data Structures

- Algorithmic problem. Precisely defined relation between input and output.
- Algorithm. Method to solve an algorithmic problem.
  - Discrete and unambiguous steps.
  - Mathematical abstraction of a program.
- Data structure. Method for organizing data to enable queries and updates.

Example: Find max

- Find max. Given a array A[0..n-1], find an index i, such that A[i] is maximal.
  - Input. Array A[0..n-1].
  - Output. An index i such that $A[i] \geq A[j]$ for all indices $j \neq i$.
- Algorithm.
  - Process A from left-to-right and maintain value and index of maximal value seen so far.
  - Return index.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

Description of Algorithms

- Natural language.
  - Process A from left-to-right and maintain value and index of maximal value seen so far.
  - Return index.
- Program.
- Pseudocode.

```
public static int findMax(int[] A) {
   int max = 0;
   for(i = 0; i < A.length; i++)
        if (A[i] > A[max]) max = i;
   return max;
}
```

```
FindMax(A, n)
   max = 0
   for i = 0 to n-1
      if (A[i] > A[max]) max = i
   return max
```

Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Peaks

- Peak. A[i] is a peak if A[i] is as least as large as it's neighbors:
  - A[i] is a peak if $A[i-1] \le A[i] \ge A[i+1]$ for $i \in \{1, .., n-2\}$
  - A[0] is a peak if $A[0] \ge A[1]$.
  - A[n-1] is a peak if $A[n-2] \le A[n-1]$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

- Peak finding. Given a array A[0..n-1], find an index i such that A[i] is a peak.
  - Input. A array A[0..n-1].
  - Output. An index i such that A[i] is a peak.

Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Algorithm 1

- Algorithm 1. For each entry check if it is a peak. Return the index of the first peak.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

- Pseudocode.

```
PEAK1(A, n)
   if A[0] ≥ A[1] return 0
   for i = 1 to n-2
      if A[i-1] ≤ A[i] ≥ A[i+1] return i
   if A[n-2] ≤ A[n-1] return n-1
```

- Challenge. How do we analyze the algorithm?

Theoretical Analysis

- Running time/time complexity.
  - T(n) = number of steps that the algorithm performs on input of size n.
- Steps.
  - Read/write to memory (x := y, A[i], i = i + 1, ...)
  - Arithmetic/boolean operations (+, -, *, /, %, &&, ||, &, |, ^, ~)
  - Comparisons (<, >, =<, =>, =, ≠)
  - Program flow (if-then-else, while, for, goto, function call, ..)
- Worst-case time complexity. Maximal running time over all inputs of size n.

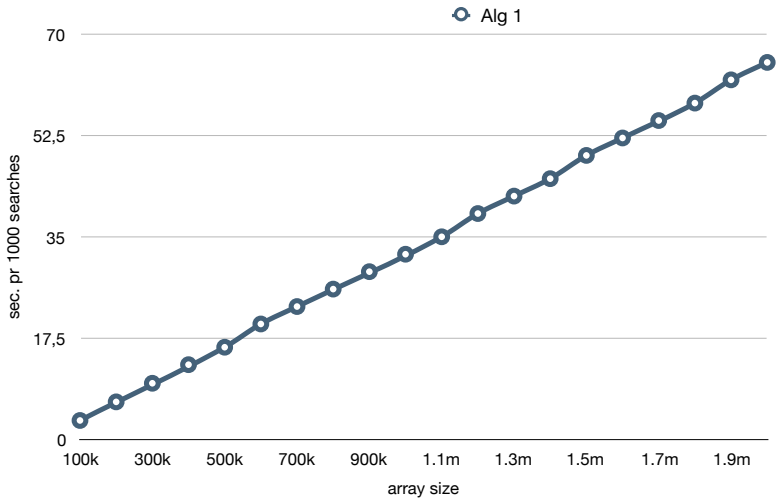
Theoretical Analysis

- Running time. What is the running time T(n) for algorithm 1?

```
PEAK1(A, n)
   if A[0] ≥ A[1] return 0                    c1
   for i = 1 to n-2                           (n-2)·c2
      if A[i-1] ≤ A[i] ≥ A[i+1] return i
   if A[n-2] ≤ A[n-1] return n-1              c3
```

$T(n) = c_1 + (n-2)\cdot c_2 + c_3$

- T(n) is a linear function of n: T(n) = an + b
- Asymptotic notation. T(n) = Θ(n)
- Experimental analysis.
  - What is the experimental running time of algorithm 1?
  - How does the experimental analysis compare to the theoretical analysis?



Peaks

- Algorithm 1 finds a peak in Θ(n) time.
- Theoretical and experimental analysis agrees.
- Challenge. Can we do better?

Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Algorithm 2

- Observation. A maximal entry A[i] is a peak.
- Algorithm 2. Find a maximal entry in A with FindMax(A, n).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

```
FindMax(A, n)
   max = 0
   for i = 0 to n-1
      if (A[i] > A[max]) max = i
   return max
```

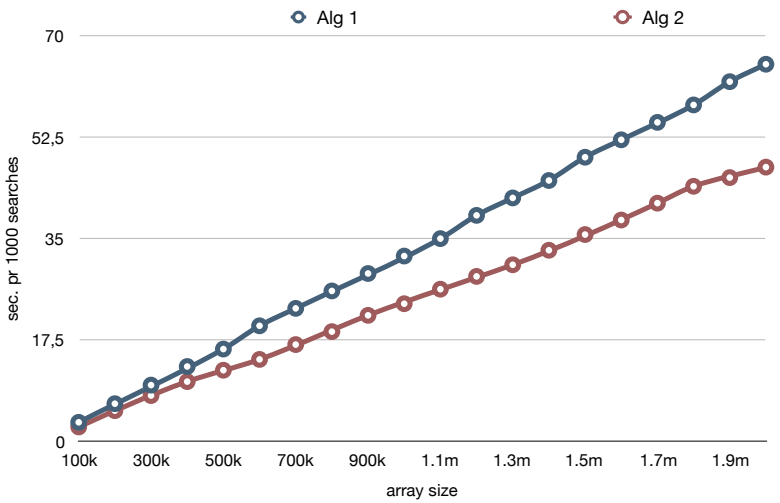
Theoretical Analysis

- Running time. What is the running time T(n) for algorithm 2?

```
FindMax(A, n)
   max = 0                                  c4
   for i = 0 to n-1                         n·c5
      if (A[i] > A[max]) max = i
   return max                               c6
```

$T(n) = c_4 + n \cdot c_5 + c_6 = \Theta(n)$

- Experimental analysis. Better constants?



Peaks

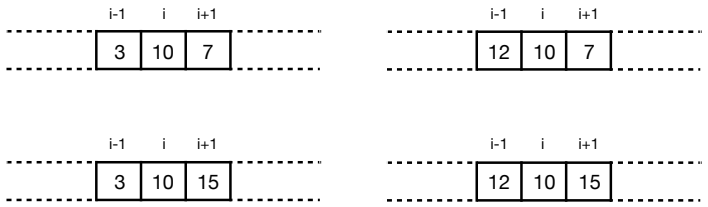
- Theoretical analysis.
  - Algorithm 1 and 2 find a peak in Θ(n) time.
- Experimental analysis.
  - Algorithm 1 and 2 run in Θ(n) time in practice.
  - Algorithm 2 is a constant factor faster than algorithm 1.
- Challenge. Can we do significantly better?

Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Algorithm 3

- Clever idea.
  - Consider any entry A[i] and it's neighbors A[i-1] and A[i+1].
  - Where can a peak be relative to A[i]?
    - Neighbor are ≤ A[i] ⟹ A[i] is a peak.
    - Otherwise A is increasing in at least one direction ⟹ peak must exist in that direction.



- Challenge. How can we turn this into a fast algorithm?

Algorithm 3

- Algorithm 3.
  - Consider the middle entry A[m] and neighbors A[m-1] and A[m+1].
  - If A[m] is a peak, return m.
  - Otherwise, continue search recursively in half with the increasing neighbor.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

Algorithm 3

- Algorithm 3.
  - Consider the middle entry A[m] and neighbors A[m-1] and A[m+1].
  - If A[m] is a peak, return m.
  - Otherwise, continue search recursively in half with the increasing neighbor.

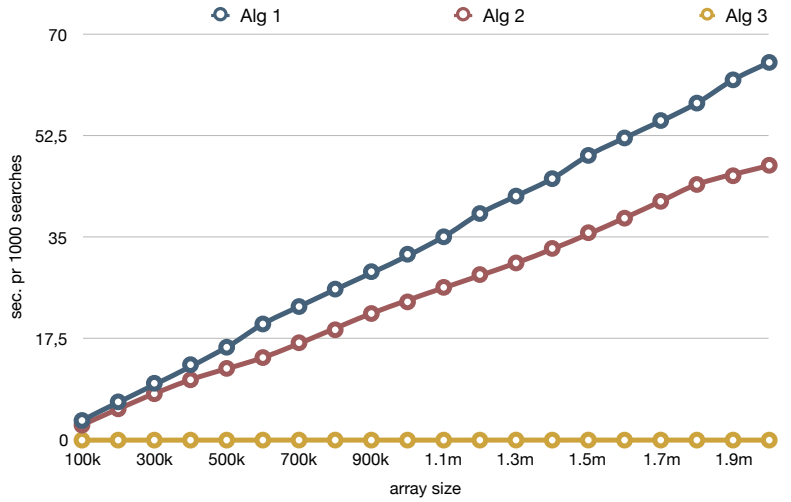
```
PEAK3(A,i,j)
   m = ⌊(i+j)/2)⌋

   if A[m] ≥ neighbors return m
   elseif A[m-1] > A[m]
      return PEAK3(A,i,m-1)
   elseif A[m] < A[m+1]
      return PEAK3(A,m+1,j)
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

Algorithm 3

```
PEAK3(A,i,j)
   m = ⌊(i+j)/2)⌋

   if A[m] ≥ neighbors return m
   elseif A[m-1] > A[m]
      return PEAK3(A,i,m-1)
   elseif A[m] < A[m+1]
      return PEAK3(A,m+1,j)
```

- Running time.
- A recursive call takes constant time.
- How many recursive calls?
- A recursive call halves size of interval. We stop when array has size 1.
  - 1st recursive call: n/2
  - 2nd recursive call: n/4
  - ….
  - k^{th} recursive call: $n/2^k$
  - ….
- $\Longrightarrow$ After ~$\log_2 n$ recursive call array has size ≤ 1.
- $\Longrightarrow$ Running time is $\Theta(\log n)$
- Experimental analysis. Significantly better?



Peaks

- Theoretical analysis.
  - Algorithm 1 and 2 finds a peak in Θ(n) time.
  - Algorithm 3 finds a peak in Θ(log n) time.
- Experimental analysis.
  - Algorithm 1 and 2 run in Θ(n) time in practice.
  - Algorithm 2 is a constant factor faster than algorithm 1.
  - Algorithm 3 is much, much faster than algorithm 1 and 3.

Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Searching and Sorting

- Searching
  - Linear search
  - Binary search
- Sorting
  - Insertion sort
  - Merge sort

Philip Bille

Searching and Sorting

- Searching
  - Linear search
  - Binary search
- Sorting
  - Insertion sort
  - Merge sort

Searching

- Searching. Given a sorted array A and number x, determine if x appears in the array.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	4	7	12	16	18	25	28	31	33	36	42	45	47	50

Linear Search

- Linear search. Check if each entry matches x.
- Time?
- Challenge. Can we take advantage of the sorted order of the array?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	4	7	12	16	18	25	28	31	33	36	42	45	47	50

Binary Search

- Binary search. Compare x to middle entry m in A.
  - if A[m] = x return true and stop.
  - if A[m] < x continue recursively on the right half.
  - if A[m] > x continue recursively on the left half.
- If array size ≤ 0 return false and stop.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	4	7	12	16	18	25	28	31	33	36	42	45	47	50

Binary Search

```
BINARYSEARCH(A,i,j,x)
   if j < i return false
   m = ⌊(i+j)/2⌋
   if A[m] = x return true
   elseif A[m] < x return BINARYSEARCH(A,m+1,j,x)
   else return BINARYSEARCH(A,i,m-1,x)     // A[m] > x
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	4	7	12	16	18	25	28	31	33	36	42	45	47	50

- Time?
- Analysis 1. Analogue of recursive peak algorithm.
  - A recursive call takes constant time.
  - Each recursive call halves the size of the array. We stop when the size is ≤ 0.
  - ⇒ Running time is O(log n)

Binary Search

- Analysis 2. Let T(n) be the running time for binary search.
  - Solve the recurrence relation for T(n).

$$T(n) = \begin{cases} T(n/2) + c & \text{if } n > 1 \\ d & \text{otherwise} \end{cases}$$

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c \\ &= T\left(\frac{n}{4}\right) + c + c \\ &= T\left(\frac{n}{8}\right) + c + c + c \\ &\vdots \\ &= T\left(\frac{n}{2^k}\right) + ck \\ &\vdots \\ &= T\left(\frac{n}{2^{\log_2 n}}\right) + c \log_2 n \\ &= T(1) + c \log_2 n \\ &= d + c \log_2 n \\ &= O(\log n) \end{aligned}$$

Searching

- We can search in
  - O(n) time with linear search.
  - O(log n) time with binary search.

Searching and Sorting

- Searching
  - Linear search
  - Binary search
- Sorting
  - Insertion sort
  - Merge sort

Sorting

- Sorting. Given array A[0..n-1] return array B[0..n-1] with same values as A but in sorted order.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
33	4	25	28	45	18	7	12	36	1	47	42	50	16	31

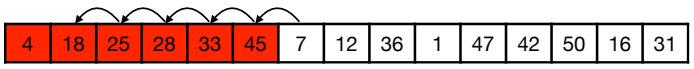
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	4	7	12	16	18	25	28	31	33	36	42	45	47	50

Applications

- Obvious.
  - Sort list of names, show Google PageRank results, show social media feed in chronological order.
- Non obvious.
  - Data compression, computer graphics, bioinformatics, recommendations systems.
- Easy problem for sorted data.
  - Search, find median, find duplicates, find closest pair, find outliers.

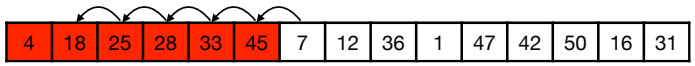
Insertion Sort

- Insertion sort. Start with unsorted array A.
- Proceed left-to-right in n rounds.
- Round i:
  - Subarray A[0..i-1] is sorted.
  - Insert A[i] into A[0..i-1] to make A[0..i] sorted.



Insertion Sort

```
INSERTIONSORT(A, n)
   for i = 1 to n-1
      j = i
      while j > 0 and A[j-1] > A[j]
         swap A[j] og A[j-1]
         j = j - 1
```



- Time?
  - To insert A[i] we use c·i time for constant c.
  - ⟹ total time T(n):

$$T(n) = \sum_{i=1}^{n-1} ci = c \sum_{i=1}^{n-1} i = \frac{cn(n-1)}{2} = O(n^2)$$

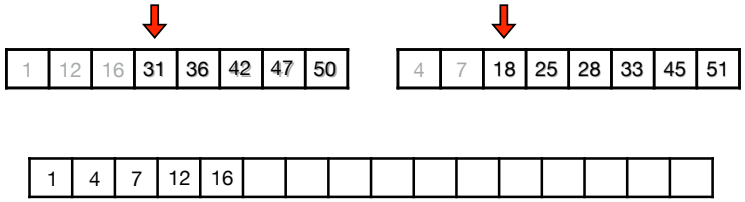
- Challenge. Can we sort faster?

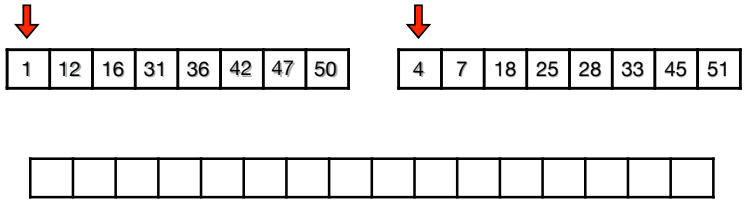
Merge sort

- Merge sort.
  - Idea. Recursive sorting via merging sorted subarrays.

Merge

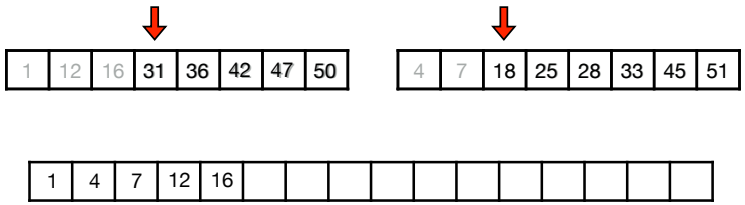
- Goal. Combine two sorted arrays into a single sorted array.
- Idea.
  - Scan both arrays left-to-right. In each step:
    - Insert smallest of the two entries in new array.
    - Move forward in array with smallest entry.
    - Repeat until input arrays are exhausted.





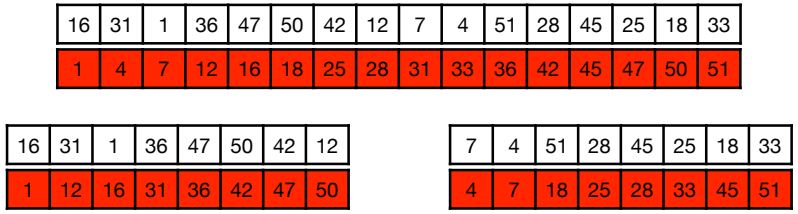
Merge

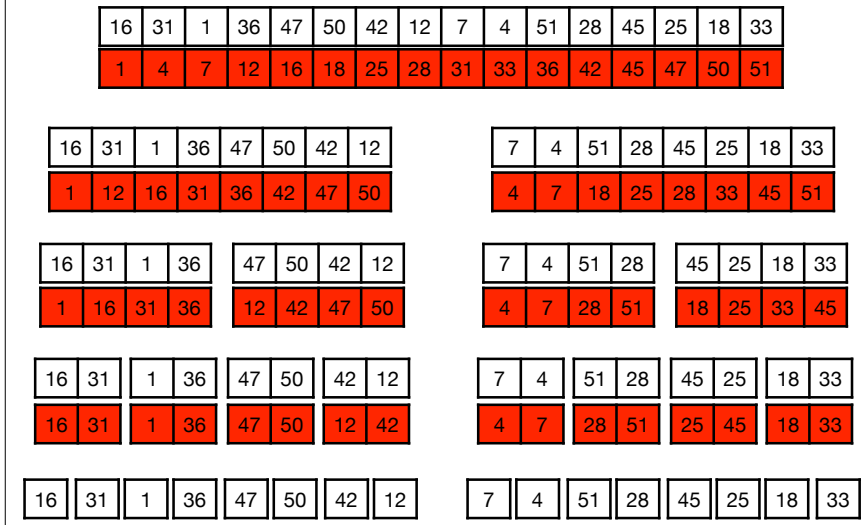
- Time. Merging two arrays A_1 og A_2?
  - Each step take O(1) time.
  - Each step we move forward in one array.
  - $\Rightarrow O(|A_1| + |A_2|)$ time.



Merge Sort

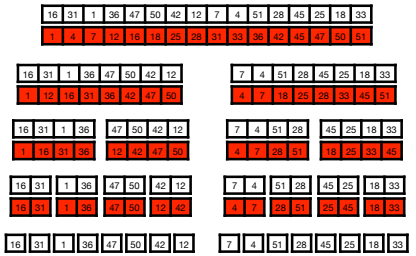
- Merge sort.
- If $|A| \leq 1$, return A.
- Otherwise:
  - Split A into halves.
  - Sort each half recursively.
  - Merge the two halves.





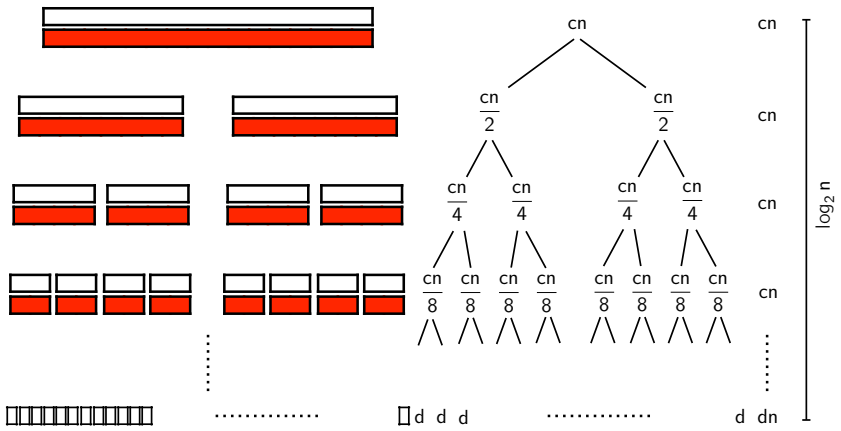
Merge Sort

```
MERGESORT(A,i,j)
    if i < j
        m =⌊(i+j)/2⌋
        MERGESORT(A,i,m)
        MERGESORT(A,m+1,j)
        MERGE(A, i, m, j)
```



- Time?
- Construct recursion tree.

Merge Sort



$$T(n) = cn\log_2 n + dn = O(n \log n)$$

Sorting

- We can sort in
  - $O(n^2)$ time with insertion sort.
  - O(nlog n) time with merge sort.

Divide and Conquer

- Merge sort is example of a divide and conquer algorithm.
- Algorithmic design paradigm.
  - Divide. Split problem into subproblems.
  - Conquer. Solve subproblems recursively.
  - Combine. Combine solution for subproblem to a solution for problem.
- Merge sort.
  - Divide. Split array into halves.
  - Conquer. Sort each half.
  - Combine. Merge halves.

Searching and Sorting

- Searching
  - Linear search
  - Binary search
- Sorting
  - Insertion sort
  - Merge sort

Analysis of Algorithms

- Analysis of algorithms
  - Running time
  - Space
- Asymptotic notation
  - O, Θ og Ω-notation.
- Experimental analysis of algorithms

Philip Bille

Analysis of Algorithms

- Analysis of algorithms
  - Running time
  - Space
- Asymptotic notation
  - O, Θ og Ω-notation.
- Experimental analysis of algorithms

Analysis of Algorithms

- Goal. Determine and predict computational resources and correct of algorithms.

- Ex.
  - Does my route finding algorithm work?
  - How quickly can I answer a query for a route?
  - Can it scale to 10k queries per second?
  - Will it run out of memory with large maps?
  - How many cache-misses does the algorithm generate per query? And how does this affect performance?

- Primary focus
  - Correctness, running time, space usage.
  - Theoretical and experimental analysis.

Running time

- Running time. Number of steps an algorithm performs on an input of size n.
- Steps.
  - Read/write to memory (x := y, A[i], i = i + 1, ...)
  - Arithmetic/boolean operations (+, -, *, /, %, &&, ||, &, |, ^, ~)
  - Comparisons (<, >, =<, =>, =, ≠)
  - Program flow (if-then-else, while, for, goto, function call, ..)
- Terminologi. Running time, time, time complexity.

Running time

- Worst-case running time. Maximal running time over all input of size n.
- Best-case running time. Minimal running time over all input of size n.
- Average-case running time. Average running time over all input of size n.

- Terminologi. Time = worst-case running time (unless otherwise stated).
- Other variants. Amortized, randomized, determinstic, non-deterministic, etc.

Space

- Space. Number of memory cells used by the algorithm
- Memory cells.
  - Variables and pointers/references = 1 memory cells.
  - Array of length k = k memory cells.
- Terminologi. Space, memory, storage, space complexity.

Analysis of Algorithms

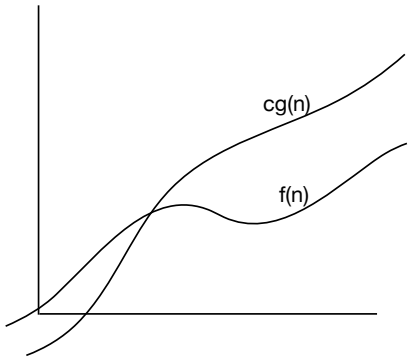
- Analysis of algorithms
  - Running time
  - Space
- Asymptotic notation
  - O, Θ og Ω-notation.
- Experimental analysis of algorithms

Asymptotic Notation

- Asymptotic notation.
  - O, Θ og Ω-notation.
  - Notation to bound the asymptotic growth of functions.
  - Fundamental tool for talking about time and space of algorithms.

O-notation

- Def. f(n) = O(g(n)) hvis f(n) ≤ cg(n) for large n.



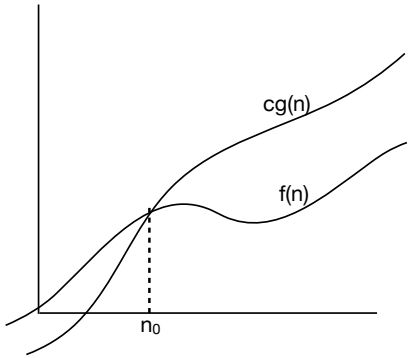
O-notation

- Ex. f(n) = O(n^2) if f(n) ≤ cn^2 for large n.

- 5n^2 = O(n^2)?
  - 5n^2 ≤ 5n^2 for large n.
- 5n^2 + 3 = O(n^2)?
  - 5n^2 + 3 ≤ 6n^2 for large n.
- 5n^2 + 3n = O(n^2)?
  - 5n^2 + 3n ≤ 6n^2 for large n.
- 5n^2 + 3n^2 = O(n^2)?
  - 5n^2 + 3n^2 = 8n^2 ≤ 8n^2 for large n.
- 5n^3 = O(n^2)?
  - 5n^3 ≥ cn^2 for all constants c for large n.

O-notation

- Def. f(n) = O(g(n)) if f(n) ≤ cg(n) for large n.
- Def. f(n) = O(g(n)) if exists constants c, $n_0 > 0$, such that for all $n \geq n_0$, f(n) ≤ cg(n).



O-notation

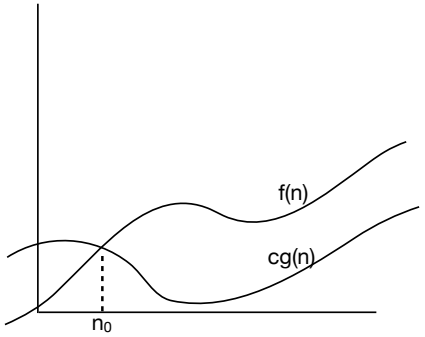
- Notation.
  - O(g(n)) is a er set of functions.
  - Think of = as ∈ or ⊆.
  - f(n) = O(n^2) is ok. O(n^2) = f(n) is not!

O-notation

- $f(n) = O(g(n))$ if $f(n) \le cg(n)$ for large n.
- Exercise.
  - Let $f(n) = 3n + 2n^3 - n^2$ and $h(n) = 4n^2 + \log n$
  - Which are true?
  - $f(n) = O(n)$
  - $f(n) = O(n^3)$
  - $f(n) = O(n^4)$
  - $h(n) = O(n^2 \log n)$
  - $h(n) = O(n^2)$
  - $h(n) = O(f(n))$
  - $f(n) = O(h(n))$

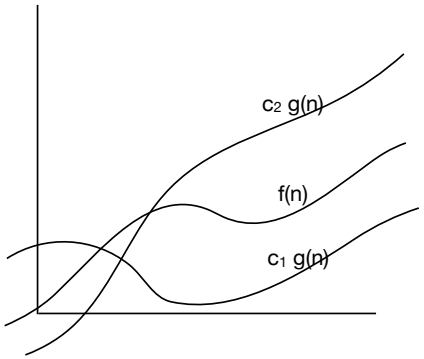
Ω-notation

- Def. $f(n) = \Omega(g(n))$ if $f(n) \ge cg(n)$ for large n.
- Def. $f(n) = \Omega(g(n))$ if exists constants c, $n_0 > 0$, such that for all $n \ge n_0$, $f(n) \ge cg(n)$



Θ-notation

- Def. $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$



Asymptotic Notation

- $f(n) = O(g(n))$ if $f(n) \le cg(n)$ for large n.
- $f(n) = \Omega(g(n))$ if $f(n) \ge cg(n)$ for large n.
- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$.

- Exercise. Which are true? ($\log^k n$ is $(\log n)^k$)
  - $n \log^3 n = O(n^2)$
  - $2^n + 5n^7 = \Omega(n^3)$
  - $n^2(n - 5)/5 = \Theta(n^2)$
  - $4\, n^{1/100} = \Omega(n)$
  - $n^3/300 + 15 \log n = \Theta(n^3)$
  - $2^{\log n} = O(n)$
  - $\log^2 n + n + 7 = \Omega(\log n)$

Asymptotic Notation

- Basic properties.
  - Any polynomial grows proportional to it's leading term.

$$a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d = \Theta(n^d)$$

  - All logarithms are asymptotically the same.

$$\log_a(n) = \frac{\log_b n}{\log_b a} = \Theta(\log_c(n)) \qquad \text{for all constants } a, b > 0$$

  - All logarithms grows slower than all polynomials.

$$\log(n) = O(n^d) \qquad \text{for all } d > 0$$

  - All polynomials grow slower than all exponentials.

$$n^d = O(r^n) \qquad \text{for all } d > 0 \text{ and } r > 1$$

Typical Running Times

```
for i = 1 to n
  < θ(1) time operation >
```

```
for i = 1 to n
  for j = 1 to n
    < θ(1) time operation >
```

```
for i = 1 to n
  for j = i to n
    < θ(1) time operation >
```

Typical Running Times

$$T(n) = \begin{cases} T(n/2) + \Theta(1) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

$$T(n) = \begin{cases} 2T(n/2) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

$$T(n) = \begin{cases} 2T(n/2) + \Theta(1) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

$$T(n) = \begin{cases} T(n/2) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

Analysis of Algorithms

- Analysis of algorithms
  - Running time
  - Space
- Asymptotic notation
  - O, Θ og Ω-notation.
- Experimental analysis of algorithms

Experimental Analysis

- Challenge. Can we experimentally estimate the theoretical running time?
- Doubling technique.
  - Run program and measure time for different input sizes.
  - Examine the time increase when we double the size of the input.
- Ex.
- Input size x 2 and time x 4.
- $\Longrightarrow$ Algorithm probably runs in quadratic time.
  - $T(n) = cn^2$
  - $T(2n) = c(2n)^2 = c2^2n^2 = c4n^2$
  - $T(2n)/T(n) = 4$

n	time	ratio
5000	0	-
10000	0,2	-
20000	0,6	3
40000	2,3	3,8
80000	9,4	4
160000	37,8	4

Analysis of Algorithms

- Analysis of algorithms
  - Running time
  - Space
- Asymptotic notation
  - O, Θ og Ω-notation.
- Experimental analysis of algorithms

Introduction to Data Structures

- Data Structures
- Stacks and Queues
- Linked Lists
- Dynamic Arrays

Philip Bille

Introduction to Data Structures

- Data Structures
- Stacks and Queues
- Linked Lists
- Dynamic Arrays

Data Structures

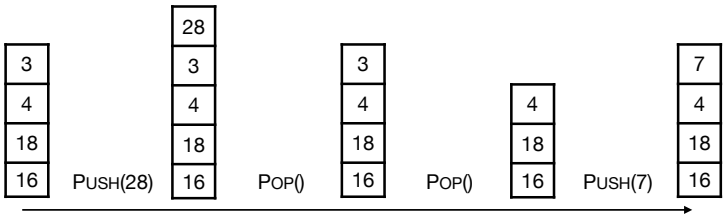
- Data structure. Method for organizing data for efficient access, searching, manipulation, etc.
- Goal.
  - Fast.
  - Compact
- Terminology.
  - Abstract vs. concrete data structure.
  - Dynamic vs. static data structure.

Introduction to Data Structures

- Data Structures
- Stacks and Queues
- Linked Lists
- Dynamic Arrays

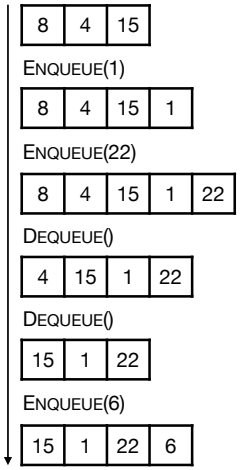
Stack

- Stack. Maintain dynamic sequence (stack) S supporting the following operations:
  - PUSH(x): add x to S.
  - POP(): remove and return the most recently added element in S.
  - ISEMPTY(): return true if S is empty.



Queue

- Queue. Maintain dynamic sequence (queue) Q supporting the following operations:
  - ENQUEUE(x): add x to Q.
  - DEQUEUE(): remove and return the earliest added element in Q.
  - ISEMPTY(): return true if Q is empty.

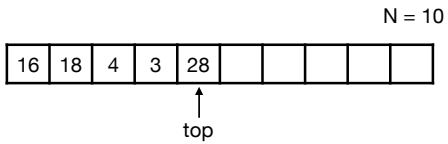


Applications

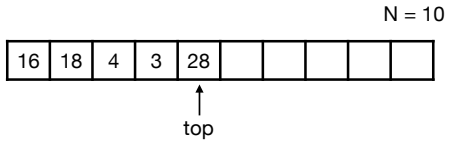
- Stacks.
  - Virtual machines
  - Parsing
  - Function calls
  - Backtracking
- Queues.
  - Scheduling processes
  - Buffering
  - Breadth-first searching

Stack Implementation

- Stack. Stack with capacity N
- Data structure.
  - Array S[0..N-1]
  - Index top. Initially top = -1
- Operations.
  - PUSH(x): Add x at S[top+1], top = top + 1
  - POP(): return S[top], top = top - 1
  - ISEMPTY(): return true if top = -1.
  - Check for overflow and underflow in PUSH and POP.



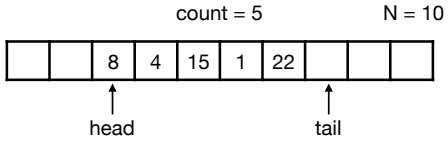
Stack Implementation



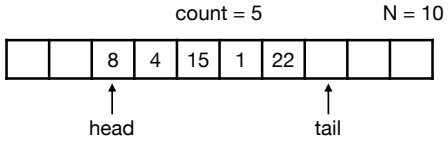
- Time
  - PUSH in O(1) time.
  - POP in O(1) time.
  - ISEMPTY in O(1) time.
- Space.
  - O(N) space.
- Limitations.
  - Capacity must be known.
  - Wasting space.

Queue Implementation

- Queue. Queue with capacity N.
- Data structure.
  - Array Q[0..N-1]
  - Indices head and tail and a counter.
- Operations.
  - ENQUEUE(x): add x at Q[tail], update count and tail cyclically.
  - DEQUEUE(): return Q[head], update count and head cyclically.
  - ISEMPTY(): return true if count = 0.
  - Check for overflow and underflow in DEQUEUE and ENQUEUE.



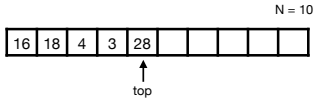
Queue Implementation



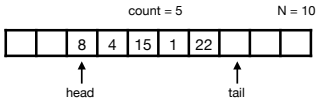
- Time.
  - ENQUEUE in O(1) time.
  - DEQUEUE in O(1) time.
  - ISEMPTY in O(1) time.
- Space.
  - O(N) space.
- Limitations.
  - Capacity must be known.
  - Wasting space.

Stacks and Queues

- Stack.
  - Time. PUSH, POP, ISEMPTY in O(1) time.
  - Space. O(N)



- Queue.
  - Time. ENQUEUE, Dequeue, ISEMPTY in O(1) time.
  - Space. O(N)



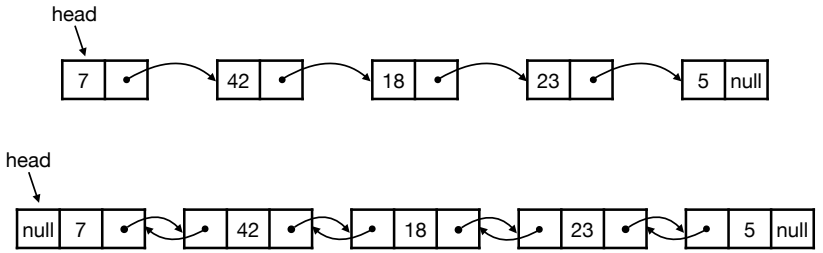
- Challenge. Can we get linear space and constant time?

Introduction to Data Structures

- Data Structures
- Stacks and Queues
- Linked Lists
- Dynamic Arrays

Linked Lists

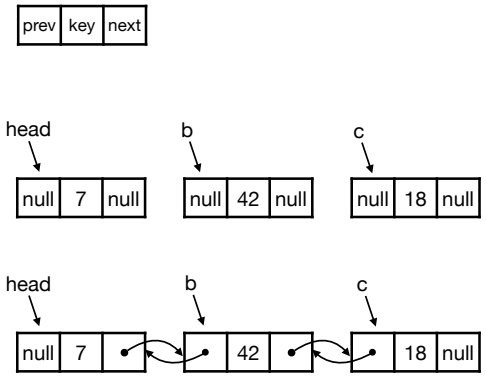
- Linked lists.
  - Data structure to maintain dynamic sequence of elements in linear space.
  - Sequence order determined by pointers/references called links.
  - Fast insertion and deletion of elements and contiguous sublists.
  - Singly-linked vs doubly-linked.



Linked Lists

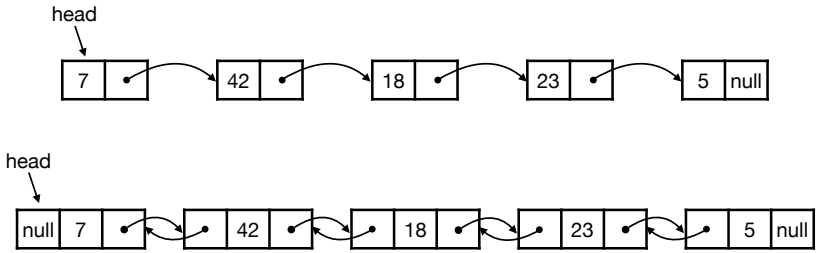
- Doubly-linked lists in Java.

```
class Node {
    int key;
    Node next;
    Node prev;
}

Node head = new Node();
Node b = new Node();
Node c = new Node();
head.key = 7;
b.key = 42;
c.key = 18;

head.prev = null;
head.next = b;
b.prev = head;
b.next = c;
c.prev = b;
c.next = null;
```



Linked Lists

- Simple operations.
  - SEARCH(head, k): return node with key k. Return null if it does not exist.
  - INSERT(head, x): insert node x in front of list. Return new head.
  - DELETE(head, x): delete node x in list.



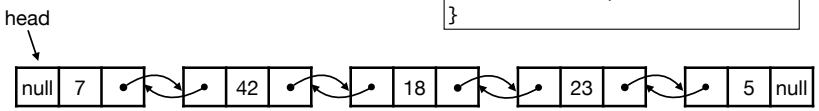
Linked Lists

- Operations in Java.

```
Node Search(Node head, int value) {
    Node x = head;
    while (x != null) {
        if (x.key == value) return x;
        x = x.next;
    }
    return null;
}
```

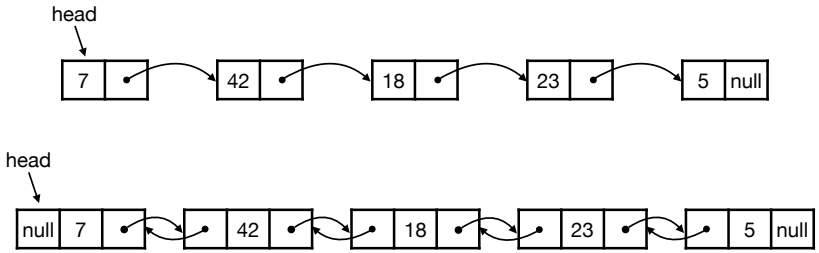
```
Node Insert(Node head, Node x) {
    x.prev = null;
    x.next = head;
    head.prev = x;
    return x;
}
```

```
Node Delete(Node head, Node x) {
    if (x.prev != null)
        x.prev.next = x.next;
    else head = x.next;
    if (x.next != null)
        x.next.prev = x.prev;
    return head;
}
```



- Ex. Let p be a new with key 10 and let q be node with key 23 in list. Trace execution of `Search(head,18)`, `Insert(head,p)` og `Delete(head,q)`.

Linked Lists



- Time.
  - SEARCH in O(n) time.
  - INSERT and DELETE in O(1) time.
- Space.
  - O(n)

Stack and Queue Implementation

- Ex. Consider how to implement stack and queue with linked lists efficiently.

- Stack. Maintain dynamic sequence (stack) S supporting the following operations:
  - PUSH(x): add x to S.
  - POP(): remove and return the most recently added element in S.
  - ISEMPTY(): return true if S is empty.

- Queue. Maintain dynamic sequence (queue) Q supporting the following operations:
  - ENQUEUE(x): add x to Q.
  - DEQUEUE(): remove and return the earliest added element in Q.
  - ISEMPTY(): return true if S is empty.

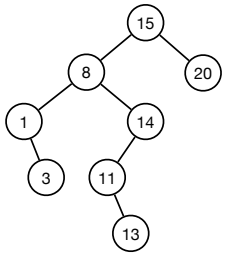
Stack and Queue Implementation

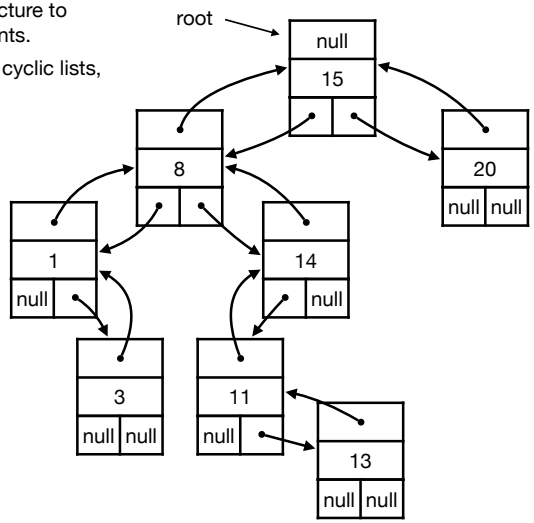
- Stacks and queues using linked lists
- Stack.
  - Time. PUSH, POP, ISEMPTY in O(1) time.
  - Space. O(n)

- Queue.
  - Time. ENQUEUE, DEQUEUE, ISEMPTY in O(1) time.
  - Space. O(n)

Linked Lists

- Linked list. Flexible data structure to maintiain sequence of elements.
- Other linked data structures: cyclic lists, trees, graphs, …





Introduction to Data Structures

- Data Structures
- Stacks and Queues
- Linked Lists
- Dynamic Arrays

Stack Implementation with Array

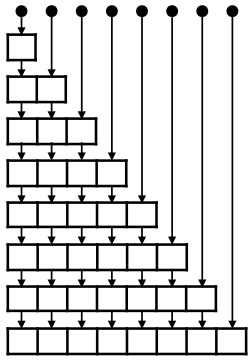
- Challenge. Can we implement a stack efficiently with arrays?
  - Do we need a fixed capacity?
  - Can we get linear space and constant time?

Dynamic Arrays

- Goal.
  - Implement a stack using arrays in O(n) space for n elements.
  - As fast as possible.
  - Focus on PUSH. Ignore POP and ISEMPTY for now.
- Solution 1
  - Start with array of size 1.
- PUSH(x):
  - Allocate new array of size + 1.
  - Move all elements to new array.
  - Delete old array.

Dynamic Arrays

- PUSH(x):
  - Allocate new array of size + 1.
  - Move all elements to new array.
  - Delete old array.



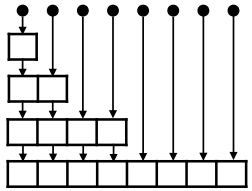
- Time. Time for n PUSH operations?
  - ith PUSH takes O(i) time.
  - ⇒ total time is $1 + 2 + 3 + 4 + \ldots + n = O(n^2)$
- Space. O(n)
- Challenge. Can we do better?

Dynamic Arrays

- Idea. Only copy elements some times
- Solution 2.
  - Start with array of size 1.
- PUSH(x):
  - If array is full:
    - Allocate new array of twice the size.
    - Move all elements to new array.
    - Delete old array.

Dynamic Arrays

- PUSH(x):
  - If array is full:
    - Allocate new array of twice the size.
    - Move all elements to new array.
    - Delete old array.



- Time. Time for n PUSH operations?
  - PUSH 2^k takes $O(2^k)$ time.
  - All other PUSH operations take O(1) time.
  - ⇒ total time $< 1 + 2 + 4 + 8 + 16 + \ldots + 2^{\log n} + n = O(n)$
- Space. O(n)

Dynamic Arrays

- Stack with dynamic array.
  - n PUSH operations in O(n) time and space.
  - Extends to n PUSH, POP og ISEMPTY operations in O(n) time.
- Time is amortized O(1) per operation.
- With more clever tricks we can deamortize to get O(1) worst-case time per operation.
- Queue with dynamic array.
  - Similar results as stack.
- Global rebuilding.
  - Dynamic array is an example of global rebuilding.
  - Technique to make static data structures dynamic.

Stack and Queues

Data structure	PUSH	POP	ISEMPTY	Space
Array with capacity N	O(1)	O(1)	O(1)	O(N)
Linked List	O(1)	O(1)	O(1)	O(n)
Dynamic Array 1	O(n)	O(1)†	O(1)	O(n)
Dynamic Array 2	O(1)†	O(1)†	O(1)	O(n)
Dynamic Array 3	O(1)	O(1)	O(1)	O(n)

† = amortized

Introduction to Data Structures

- Data Structures
- Stacks and Queues
- Linked Lists
- Dynamic Arrays

Introduction to Graphs

- Undirected Graphs
- Representation
- Depth-First Search
  - Connected Components
- Breadth-First Search
  - Bipartite Graphs

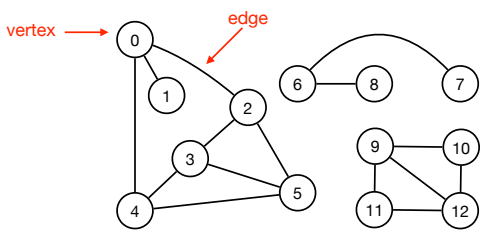
Philip Bille

Introduction to Graphs

- Undirected Graphs
- Representation
- Depth-First Search
  - Connected Components
- Breadth-First Search
  - Bipartite Graphs

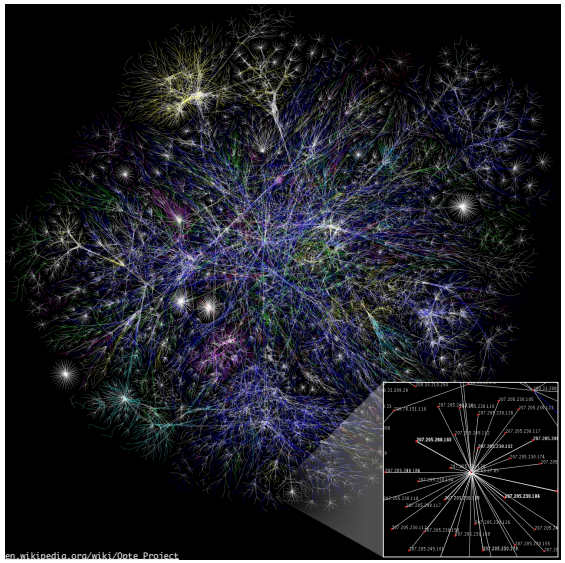
Undirected graphs

- Undirected graph. Set of vertices pairwise joined by edges.



- Why graphs?
  - Models many natural problems from many different areas.
  - Thousands of practical applications.
  - Hundreds of well-known graph algorithms.

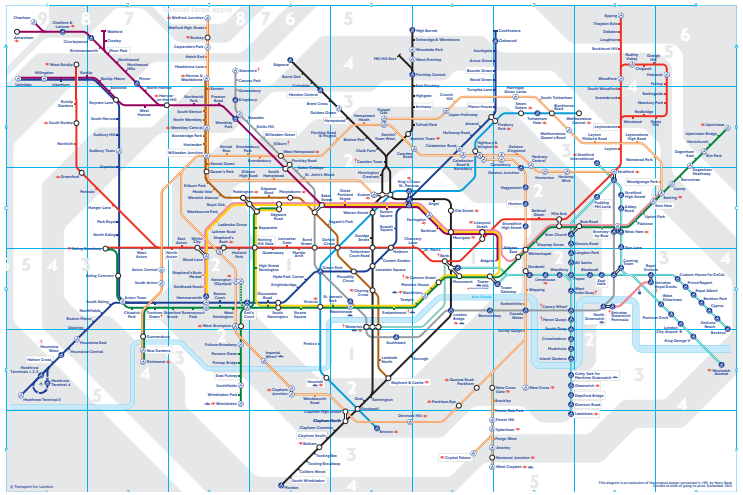
Visualizing the Internet



Visualizing Friendships on Facebook

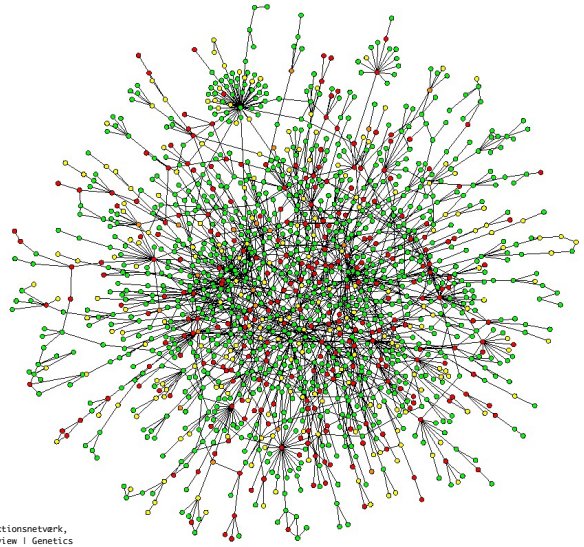
"Visualizing friendships", Paul Butler

London Metro



London metro, London Transport

Protein Interaction Networks



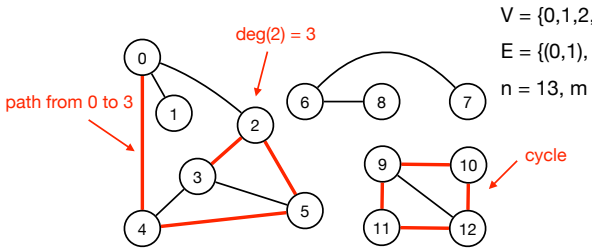
Protein-protein interaktionsnetværk,
Jeong et al, Nature Review | Genetics

Applications of Graphs

Graph	Vertices	Edges
communication	computers	cables
transport	intersections	roads
transport	airports	flight routes
games	position	valid move
neural network	neuron	synapses
financial network	stocks	transactions
circuit	logical gates	connections
food chain	species	predator-prey
molecule	atom	bindings

Terminology

- Undirected graph. G = (V, E)
  - V = set of vertices
  - E = set of edges (each edge is a pair of vertices)
  - n = |V|, m = |E|
- Path. Sequence of vertices connected by edges.
- Cycle. Path starting and ending at the same vertex.
- Degree. deg(v) = the number of neighbors of v, or edges incident to v.
- Connectivity. A pair of vertices are connected if there is a path between them



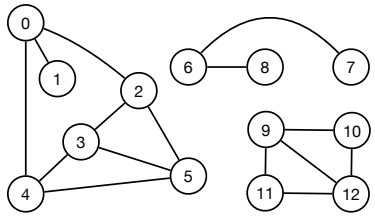
V = {0,1,2,…,12}

E = {(0,1), (0,2) (0,4),(2,3),…, (11,12)}

n = 13, m = 15

Undirected Graphs

- Lemma. $\sum_{v \in V} \deg(v) = 2m$.
- Proof. How many times is each edge counted in the sum?



Algoritmic Problems on Graphs

- Path. Is there a path connecting s and t?
- Shortest path. What is the shortest path connecting s and t?
- Longest path. What is the longest path connecting s and t?

- Cycle. Is there a cycle in the graph?
- Euler tour. Is there a cycle that uses each edge exactly once?
- Hamilton cycle. Is there a cycle that uses each vertex exactly once?

- Connectivity. Are all pairs of vertices connected?
- Minimum spanning tree. What is the best way of connecting all vertices?
- Biconnectivity. Is there a vertex whose removal would cause the graph to be disconnected?

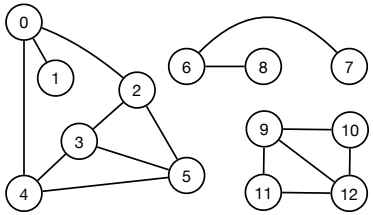
- Planarity. Is it possible to draw the graph in the plane without edges crossing?
- Graph isomorphism. Do these sets of vertices and edges represent the same graph?

Introduction to Graphs

- Undirected Graphs
- Representation
- Depth-First Search
  - Connected Components
- Breadth-First Search
  - Bipartite Graphs

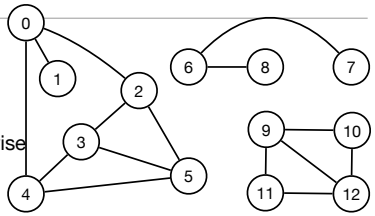
Representation

- Graph G with n vertices and m edges.
- Representation. We need the following operations on graphs.
  - ADJACENT(v, u): determine if u and v are neighbors.
  - NEIGHBORS(v): return all neighbors of v.
  - INSERT(v, u): add the edge (v, u) to G (unless it is already there).



Adjacency Matrix

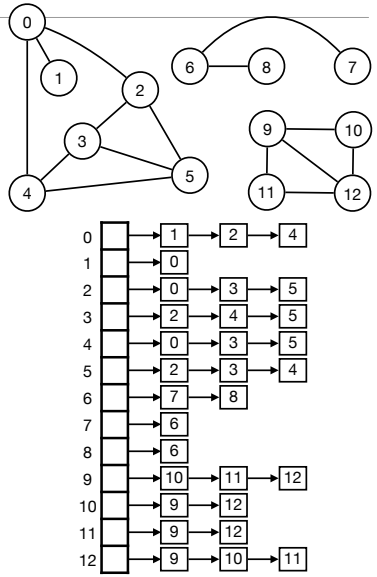
- Graph G with n vertices and m edges.
- Adjacency matrix.
  - 2D n × n array A.
  - A[i,j] = 1 if i and j are neighbors, 0 otherwise
- Complexity?
- Space. $O(n^2)$
- Time.
  - ADJACENT and INSERT in O(1) time.
  - NEIGHBOURS in O(n) time.



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	1	0	1	0	0	0	0	0	0	0
3	0	0	1	0	1	1	0	0	0	0	0	0	0
4	1	0	0	1	0	1	0	0	0	0	0	0	0
5	0	0	1	1	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	1	1	0	0	0	0
7	0	0	0	0	0	0	1	0	0	0	0	0	0
8	0	0	0	0	0	0	1	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	1
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	1	1	0

Adjacency List

- Graph G with n vertices and m edges.
- Adjacency list.
  - Array A[0..n-1].
  - A[i] is a linked list of all neighbors of i.
- Complexity?
- Space. $O(n + \sum_{v \in V} \deg(v)) = O(n + m)$
- Time.
  - ADJACENT, NEIGHBOURS, INSERT O(deg(v)) time.



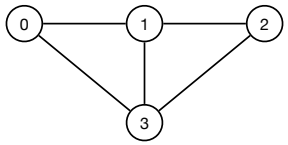
Representation

Data structure	ADJACENT	NEIGHBOURS	INSERT	space
adjacency matrix	O(1)	O(n)	O(1)	$O(n^2)$
adjacency list	O(deg(v))	O(deg(v))	O(deg(v))	O(n+m)

- Real world graphs are often sparse.

Representation

```
n = 4
adj = [[] for i in range(n)]
adj[0].append(1)
adj[1].append(0)
adj[0].append(3)
adj[3].append(0)
adj[1].append(2)
adj[2].append(1)
adj[1].append(3)
adj[3].append(1)
adj[2].append(3)
adj[3].append(2)
```



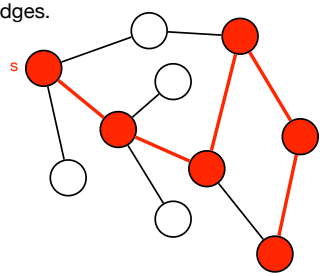
[[1, 3], [0, 2, 3], [1, 3], [0, 1, 2]]

Introduction to Graphs

- Undirected Graphs
- Representation
- Depth-First Search
  - Connected Components
- Breadth-First Search
  - Bipartite Graphs

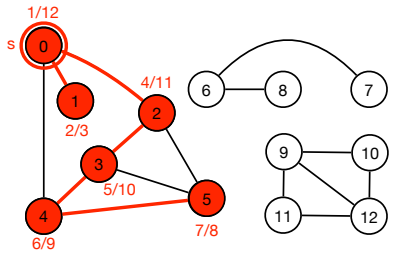
Depth-First Search

- Algorithm for systematically visiting all vertices and edges.
- Depth first search from vertex s.
  - Unmark all vertices and visit s.
  - Visit vertex v:
    - Mark v.
    - Visit all unmarked neighbours of v recursively.



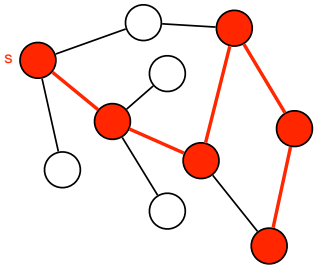
- Intuition.
  - Explore from s in some direction, until we read dead end.
  - Backtrack to the last position with unexplored edges.
  - Repeat.

- Discovery time. First time a vertex is visited.
- Finish time. Last time a vertex is visited.



Depth-First Search

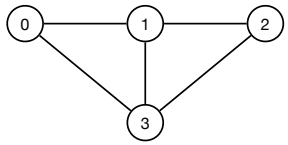
```
DFS(s)
   time = 0
   DFS-VISIT(s)

DFS-VISIT(v)
   v.d = time++
   mark v
   for each unmarked neighbor u
      u.π = v
      DFS-VISIT(u)
   v.f = time++
```



- Time. (on adjacency list representation)
  - Recursion? once per vertex.
  - O(deg(v)) time spent on vertex v.
  - $\Longrightarrow$ total $O(n + \sum_{v \in V} \deg(v)) = O(n + m)$ time.
  - Only visits vertices connected to s.

Depth-First Search

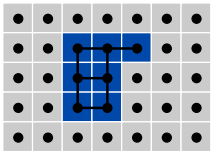
```
visited = [False for i in range(n)]

def dfs(s):
   if (visited[s]):
      return
   visited[s] = True
#   print(s)
   for u in adj[s]:
      dfs(u)

dfs(0)
```



0
1
2
3

Flood Fill

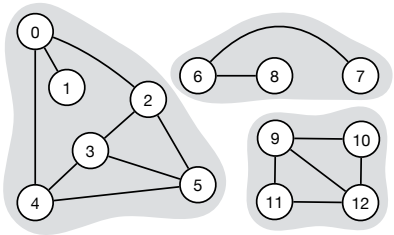
- Flood fill. Chance the color of a connected area of green pixels.



- Algorithm.
  - Build a grid graph and run DFS.
  - Vertex: pixel.
  - Edge: between neighboring pixels of same color.
  - Area: connected component

Connected Components

- Definition. A connected component is a maximal subset of connected vertices.



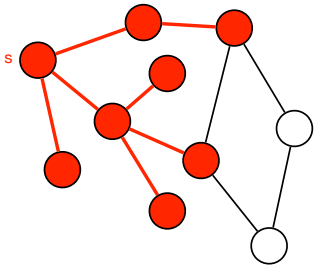
- How to find all connected components?
- Algorithm.
  - Unmark all vertices.
  - While there is an unmarked vertex:
    - Chose an unmarked vertex v, run DFS from v.
- Time. O(n + m).

Introduction to Graphs

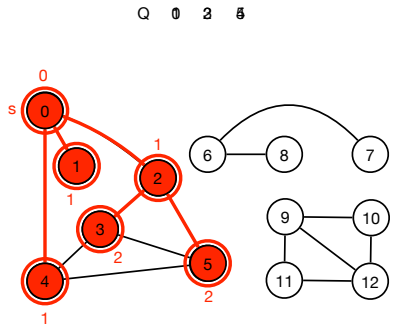
- Undirected Graphs
- Representation
- Depth-First Search
  - Connected Components
- Breadth-First Search
  - Bipartite Graphs

Breadth-First Search

- Breadth first search from s.
  - Unmark all vertices and initialize queue Q.
  - Mark s and Q.ENQUEUE(s).
  - While Q is not empty:
    - v = Q.DEQUEUE().
    - For each unmarked neighbor u of v
      - Mark u.
      - Q.ENQUEUE(u).

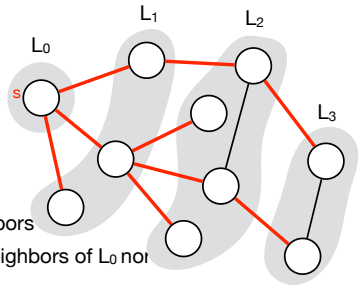


- Intuition.
  - Explore, starting from s, in all directions - in increasing distance from s.
- Shortest paths from s.
  - Distance to s in BFS tree = shortest distance to s in the original graph.



Shortest Paths

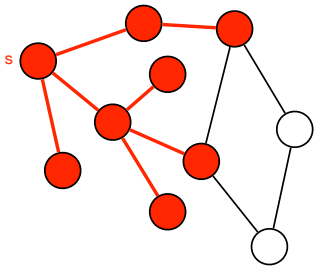
- Lemma. BFS finds the length of the shortest path from s to all other vertices.
- Intuition.
  - BFS assigns vertices to layers. Layer i contains all vertices of distance i to s.



  - What does each layer contain?
  - L_0 : {s}
  - L_1 : all neighbours of L_0.
  - L_2 : all neighbours of L_1 that are not neighbors
  - L_3 : all neighbours of L_2 that neither are neighbors of L_0 nor
  - ...
  - L_i : all neighbours of L_{i-1} that are not neighbors of any L_j for j < i-1
    - = all vertices of distance i from s.

Breadth-First Search

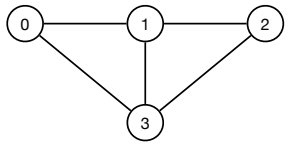
```
BFS(s)
   mark s
   s.d = 0
   Q.ENQUEUE(s)
   repeat until Q.ISEMPTY()
      v = Q.DEQUEUE()
      for each unmarked neighbor u
         mark u
         u.d = v.d + 1
         u.π = v
         Q.ENQUEUE(u)
```



- Time. (on adjacency list representation)
  - Each vertex is visited at most once.
  - O(deg(v)) time spent on vertex v.
  - $\Longrightarrow$ total $O(n + \sum_{v \in V} \deg(v)) = O(n + m)$ time.
  - Only vertices connected to s are visited.

Breadth-First Search

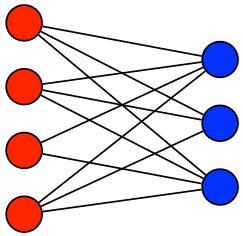
```
from collections import deque
q = deque()
visited = [False for i in range(n)]
distance = [-1 for i in range(n)]


visited[0] = True
distance[0] = 0
q.append(0)
while q:
   s = q.popleft()
#    print(s)
   for u in adj[s]:
      if (visited[u]):
         continue
      visited[u] = True
      distance[u] = distance[s]+1
      q.append(u)
```




```
0
1
3
2
```

Bipartite Graphs

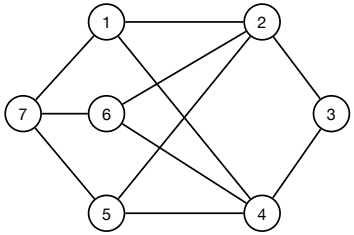
- Definition. A graph is bipartite if and only if all vertices can be colored red and blue such that every edge has exactly one red endpoint and one blue endpoint.
- Equivalent definition. A graph is bipartite if and only if its vertices can be partitioned into two sets V_1 and V_2 such that all edges go between V_1 and V_2.

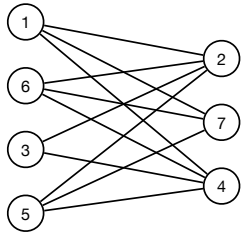


- Application.
  - Scheduling, matching, assigning clients to servers, assigning jobs to machines, assigning students to advisors/labs, …
  - Many graph problems are *easier* on bipartite graphs.

Bipartite Graphs

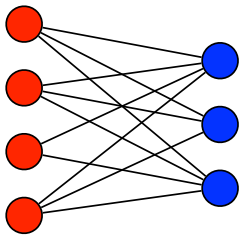
- Challenge. Given a graph G, determine whether G is bipartite.





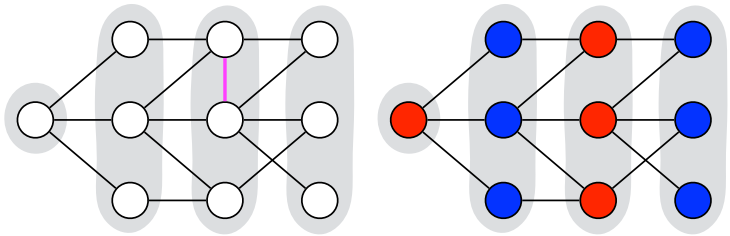
Bipartite Graphs

- Lemma. A graph G is bipartite if and only if all cycles in G have even length.
- Proof. ⟹
  - If G is bipartite, all cycles start and end on the same side.



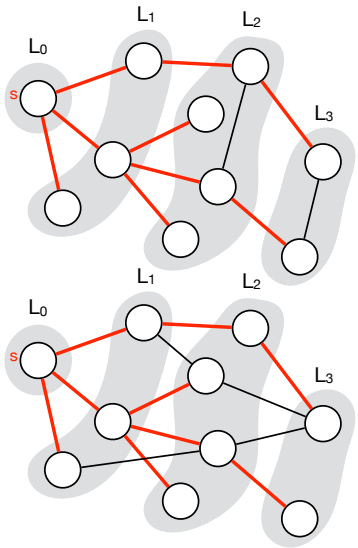
Bipartite Graphs

- Lemma. A graph G is bipartite if and only if all cycles in G have even length.
- Proof. ⟸
  - Choose a vertex v and consider BFS layers L_0, L_1, …, L_k.
  - All cycles have even length
  - ⟹ There is no edge between vertices of the same layer
  - ⟹ We can colors layers with alternating red and blue colors.
  - ⟹ G is bipartite.



Bipartite Graphs

- Algorithm.
  - Run BFS on G.
  - For each edge in G, check if it's endpoints are in the same layer.
- Time.
  - O(n + m)



Graph Algorithms

Algorithm	Time	Space
Depth first search	O(n + m)	O(n + m)
Breadth first search	O(n + m)	O(n + m)
Connected components	O(n + m)	O(n + m)
Bipartite	O(n + m)	O(n + m)

- All on the adjacency list representation.

Introduction to Graphs

- Undirected Graphs
- Representation
- Depth-First Search
  - Connected Components
- Breadth-First Search
  - Bipartite Graphs

Directed Graphs

- Directed Graphs
- Representation
- Search
- Topological Sorting
- Directed Acyclic Graphs
- Strongly Connected Components
- Implicit Graphs

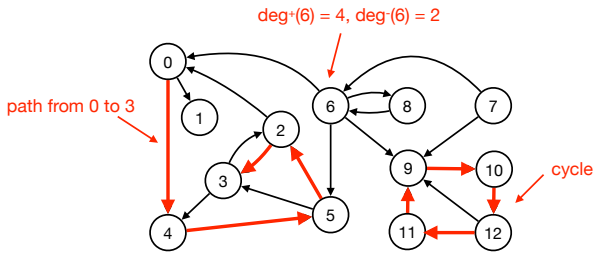
Philip Bille

Directed Graphs

- Directed Graphs
- Representation
- Search
- Topological Sorting
- Directed Acyclic Graphs
- Strongly Connected Components
- Implicit Graphs

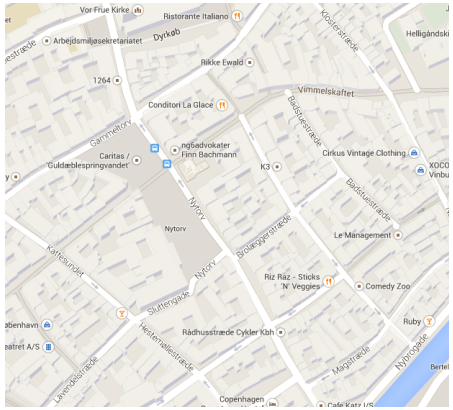
Directed Graphs

- Directed graph. Set of vertices pairwise joined by directed edges.



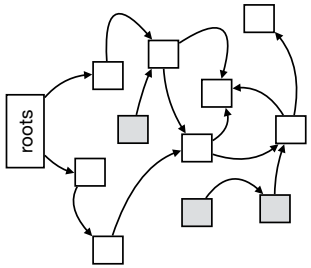
Road Networks

- Vertex = intersection, edge = (one-way) road.



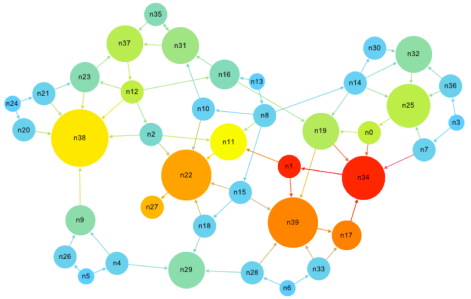
Garbage Collection

- Vertex = object, edge = pointer/reference.
- Which objects are reachable from a root?



WWW

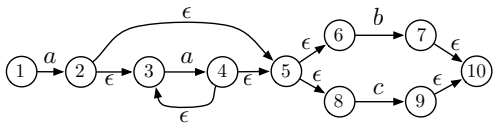
- Vertex = homepage, edge = hyperlink.
- Web Crawling
- PageRank



http://computationalculture.net/article/what_is_in_pagerank

Automata and Regular Expressions

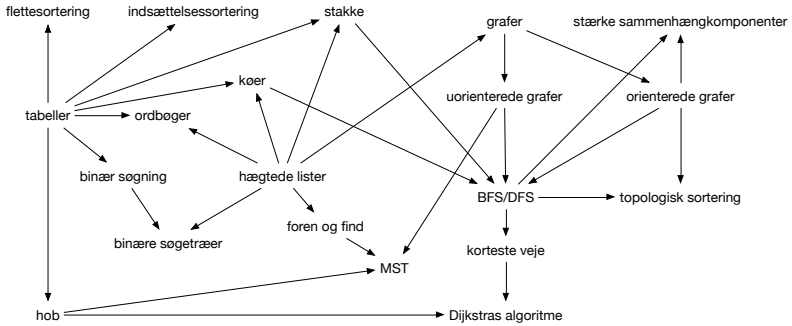
- Vertex = state, edge = state transition.
- Does the automaton accept “aab” = is there a path from 1 to 10 that matches "aab"?
- Regular expressions can be represented as automata.



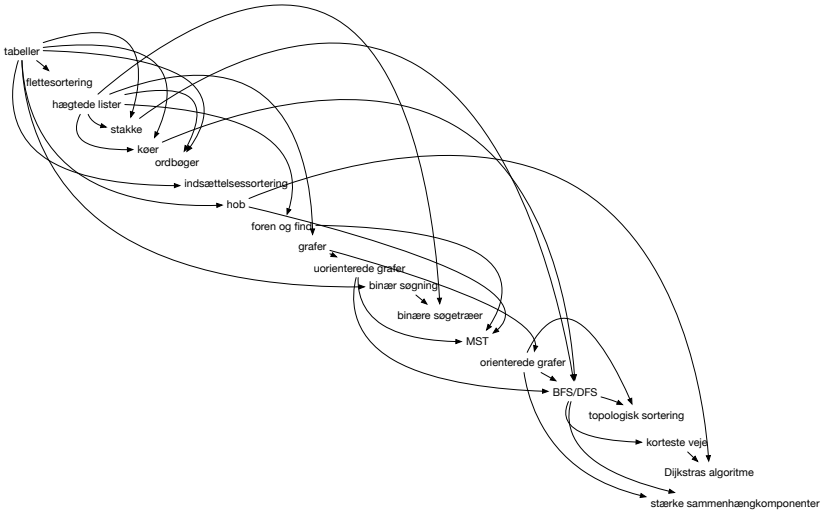
R = a·(a*)·(b|c)

Dependencies

- Vertices = topics, edge = dependency.
- Are there any cyclic dependencies? Can we find an ordering of vertices that avoids cyclic dependencies?



Dependencies

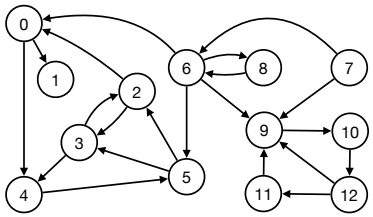


Applications

Graph	Vertices	Edges
internet	homepage	hyperlink
transport	intersection	one-way road
scheduling	job	precedence relation
disease outbreak	person	infects relation
citation	paper	citation
object graph	objects	pointers/references
object hierarchy	class	inheritance
control-flow	code	jump

Directed Graphs

- Lemma. $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = m$.
- Proof. Every edge has exactly one start and end vertex.



Algorithmic Problems on Directed Graphs

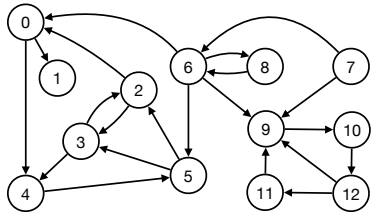
- Path. Is there a path from s to t?
- Shortest path. What is the shortest path from s to t.
- Directed acyclic graph. Is there a cycle in the graph?
- Topological sorting. Can we order the vertices such that all edges are directed in same direction?
- Strongly connected component. Is there a path between all pairs of vertices?
- Transitive closure. For which vertices is there a path from v to w?

Directed Graphs

- Directed Graphs
- Representation
- Search
- Topological Sorting
- Directed Acyclic Graphs
- Strongly Connected Components
- Implicit Graphs

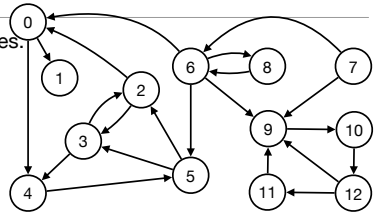
Representation

- G directed graph with n vertices and m edges.
- Representation. We need the following operations on directed graphs.
  - PointsTo(v, u): determine if v points to u.
  - Neighbors(v): return all vertices that v points to.
  - Insert(v, u): add edge (v, u) to G (unless it is already there).



Adjacency Matrix

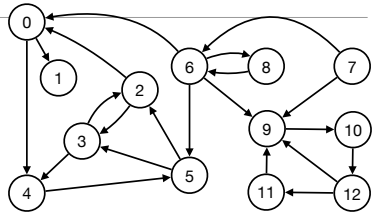
- Directed graph G with n vertices and m edges.
- Adjacency matrix.
  - 2D n × n array A.
  - A[i,j] = 1 if i points to j, 0 otherwise.
- Space. $O(n^2)$
- Time.
  - PointsTo in O(1) time.
  - Neighbors(v) in O(n) time.
  - Insert(v, u) in O(1) time.

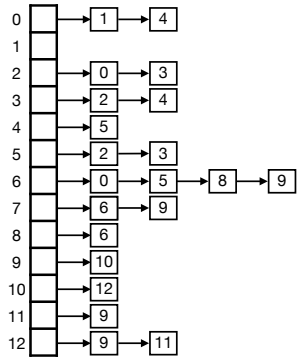


	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	1	0	0	0	0	0	0	0	0	0
3	0	0	1	0	1	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0	0	0	0	0
5	0	0	1	1	0	0	0	0	0	0	0	0	0
6	1	0	0	0	0	1	0	0	1	1	0	0	0
7	0	0	0	0	0	0	1	0	0	1	0	0	0
8	0	0	0	0	0	0	1	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	1
11	0	0	0	0	0	0	0	0	0	1	0	0	0
12	0	0	0	0	0	0	0	0	0	1	0	1	0

Adjacency List

- Directed graph G with n vertices and m edges.
- Adjacency list.
  - Array A[0..n-1].
  - A[i] is a linked list of all nodes that i points to.
- Space. $O(n + \sum_{v \in V} \deg^+(v)) = O(n + m)$
- Time.
  - PointsTo, Neighbors and Insert in O(deg(v)) time.





Representation

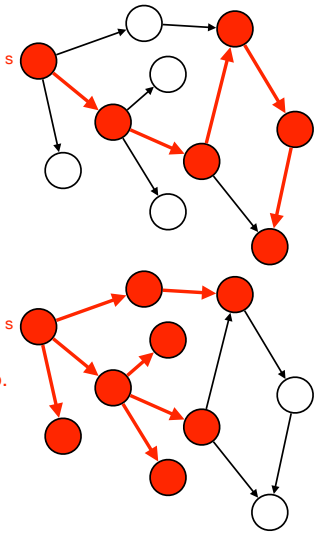
Data structure	PointsTo	Neighbors	Insert	Space
adjacency matrix	O(1)	O(n)	O(1)	$O(n^2)$
adjacency list	$O(\deg^+(v))$	$O(\deg^+(v))$	$O(\deg^+(v))$	O(n+m)

Directed Graphs

- Directed Graphs
- Representation
- **Search**
- Topological Sorting
- Directed Acyclic Graphs
- Strongly Connected Components
- Implicit Graphs

Search

- Depth first search from s.
  - Unmark all vertices and visit s.
  - Visit vertex s:
    - Mark v.
    - Visit all unmarked neighbors that v points to recursively.
- Breadth first search from s.
  - Unmark all vertices and initialize queue Q.
  - Mark s and Q.Enqueue(s).
  - While Q is not empty:
    - v = Q.Dequeue().
    - For each unmarked neighbor u that v points to.
      - Mark u.
      - Q.Enqueue(u).
- Time. O(n + m)

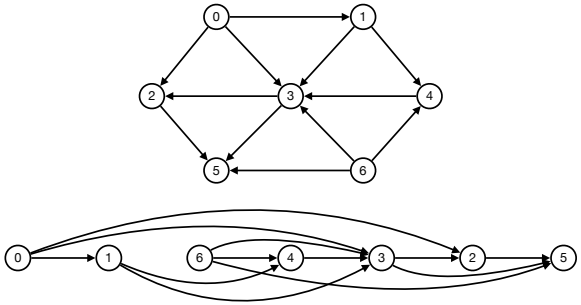


Directed Graphs

- Directed Graphs
- Representation
- Search
- **Topological Sorting**
- Directed Acyclic Graphs
- Strongly Connected Components
- Implicit Graphs

Topological Sorting

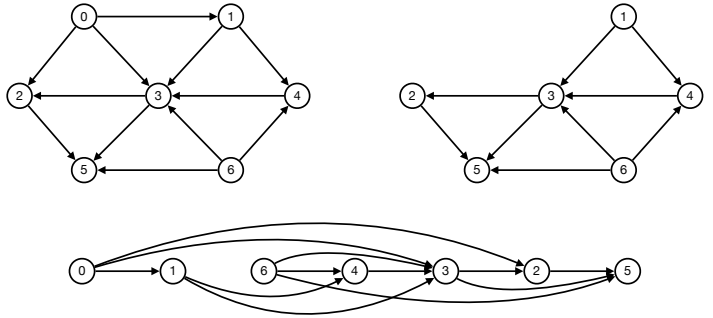
- Topological sorting. Ordering of vertices $v_0, v_1, \ldots, v_{n-1}$ from left to right such that all edges are directed to the right.

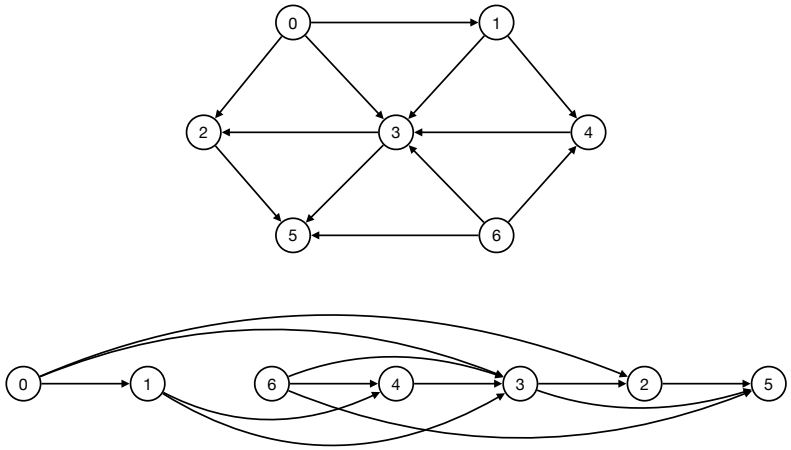


- Challenge. Compute a topological sorting or determine that none exists.

Topological Sorting

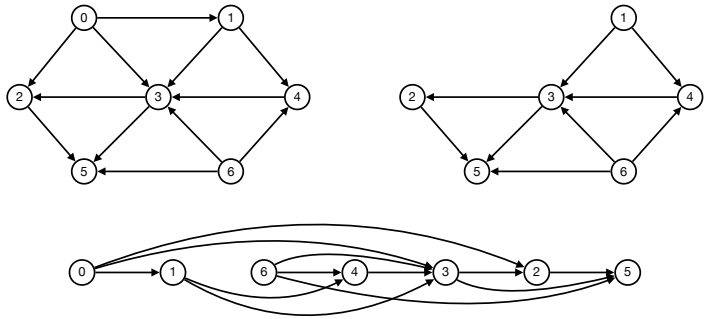
- Algorithm.
  - Find v with in-degree 0.
  - Output v and recurse on G - {v}.





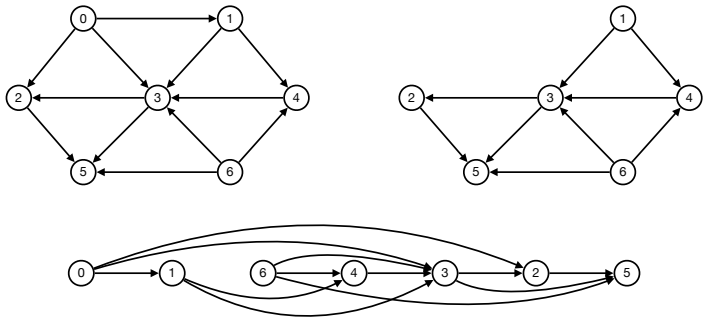
Topological Sorting

- Correctness?
- Lemma. G has topological sorting $\Longleftrightarrow$ G has vertex v with in-degree 0 and G - {v} has topological sorting.



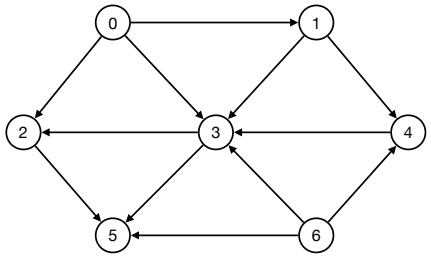
Topological Sorting

- Challenge. How do we implement algorithm efficiently on adjacency list representation?



Topological Sorting

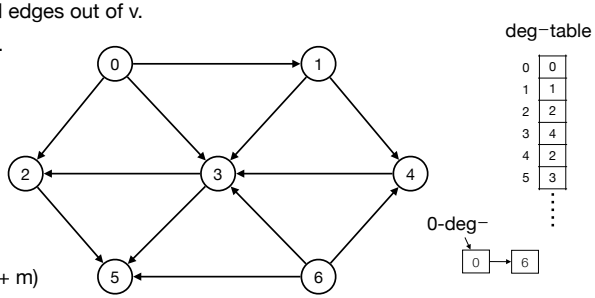
- Solution 1. Construct reverse graph G^R.
  - Search in adjacency list representation of G^R to find vertex v with in-degree 0.
  - Remove v and edges out of v.
  - Put v leftmost.
  - Repeat.



- Time per vertex.
  - Find vertex v with in-degree 0: O(n).
  - Remove edges out of v: $O(\deg^+(v))$
- Total time. $O(n^2 + \sum_{v \in V} \deg^+(v)) = O(n^2 + m) = O(n^2)$.

Topological Sorting

- Solution 2. Maintain in-degree of every vertex + linked list of all vertices with in-degree 0.
  - Find vertex v with in-degree 0.
  - Remove v and edges out of v.
  - Put v leftmost.
  - Repeat



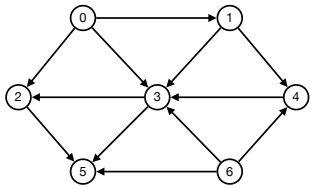
- Initialization. O(n + m)
- Time per vertex.
  - Remove vertex v with in-degree 0: O(1).
  - Remove edges out of v: $O(\deg^+(v))$
- Total time. $O(n + \sum_{v \in V} \deg^+(v)) = O(n + m)$.

Directed Graphs

- Directed Graphs
- Representation
- Search
- Topological Sorting
- Directed Acyclic Graphs
- Strongly Connected Components
- Implicit Graphs

Directed Acyclic Graphs

- Directed acyclic graph (DAG). G is a DAG if it contains no (directed) cycles.



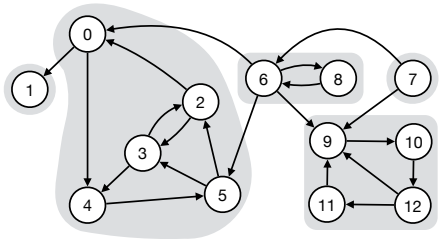
- Challenge. Determine whether or not G is a DAG.
- Equivalence of DAGs and topological sorting. G is a DAG ⟺ G has a topological sorting (see exercises).
- Algorithm.
  - Compute a topological sorting.
  - If success output yes, otherwise no.
- Time. O(n+ m)

Directed Graphs

- Directed Graphs
- Representation
- Search
- Topological Sorting
- Directed Acyclic Graphs
- Strongly Connected Components
- Implicit Graphs

Strongly Connected Components

- Def. v and u are strongly connected if there is a path from v to u and u to v.
- Def. A strongly connected component is a maximal subset of strongly connected vertices.



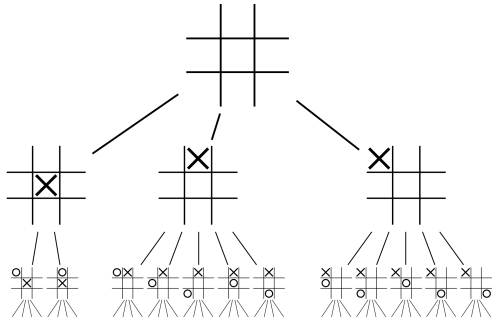
- Theorem. We can compute the strongly connected components in a graph in O(n + m) time.
- See CLRS 22.5.

Directed Graphs

- Directed Graphs
- Representation
- Search
- Topological Sorting
- Directed Acyclic Graphs
- Strongly Connected Components
- Implicit Graphs

Implicit Graphs

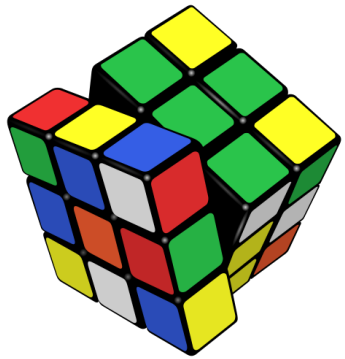
- Implicit graph. Undirected/directed graph with implicit representation.
- Implicit representation.
  - Start vertex s + algorithm to generate neighbors of a vertex.
- Applications. Games, AI, etc.



Implicit Graphs

- Rubiks cube
  - n+m = 43.252.003.274.489.856.000 ~ 43 trillions.
- What is the smallest number of moves needed to solve a cube from any starting configuration?

year	lower bound	upper bound
1981	18	52
1990	18	42
1992	18	39
1992	18	37
1995	18	29
1995	20	29
2005	20	28
2006	20	27
2007	20	26
2008	20	25
2008	20	23
2008	20	22
2010	20	20



Directed Graphs

- Directed Graphs
- Representation
- Search
- Topological Sorting
- Directed Acyclic Graphs
- Strongly Connected Components
- Implicit Graphs

Priority Queues

- Priority Queues
- Trees and Heaps
- Representations of Heaps
- Algorithms on Heaps
- Building a Heap
- Heapsort

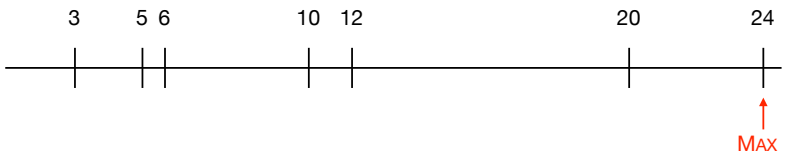
Philip Bille

Priority Queues

- Priority Queues
- Trees and Heaps
- Representations of Heaps
- Algorithms on Heaps
- Building a Heap
- Heapsort

Priority Queues

- Priority queues. Maintain dynamic set S supporting the following operations. Each element has key x.key and satellite data x.data.
  - MAX(): return element with largest key.
  - EXTRACTMAX(): return and remove element with largest key.
  - INCREASEKEY(x, k): set x.key = k. (assume k ≥ x.key)
  - INSERT(x): set S = S ∪ {x}

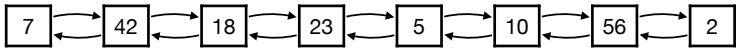


Priority Queues

- Applications.
  - Scheduling
  - Shortest paths in graphs (Dijkstra's algorithm)
  - Minimum spanning trees in graphs (Prim's algorithm)
  - Compression (Huffman's algorithm)
  - …
- Challenge. How can we solve problem with current techniques?

Priority Queues

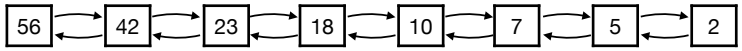
- Solution 1: Linked list. Maintain S in a doubly-linked list.



- MAX(): linear search for largest key.
- EXTRACTMAX(): linear search for largest key. Remove and return element.
- INCREASEKEY(x, k): set x.key = k.
- INSERT(x): add element to front of list (assume element does not exist in S beforehand).
- Time.
  - MAX and EXTRACTMAX in O(n) time (n = |S|).
  - INCREASEKEY and INSERT in O(1) time.
- Space.
  - O(n).

Priority Queues

- Solution 2: Sorted linked list. Maintain S in a sorted doubly-linked list.



- MAX(): return first element.
- EXTRACTMAX(): return and remove first element.
- INCREASEKEY(x, k): set x.key = k. Linear search to move x to correct position.
- INSERT(x): linear search to insert x at correct position.
- Time.
  - MAX and EXTRACTMAX in O(1) time.
  - INCREASEKEY and INSERT in O(n) time.
- Space.
  - O(n).

Priority Queues

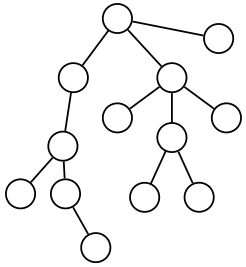
Data structure	MAX	EXTRACTMAX	INCREASEKEY	INSERT	Space
linked list	O(n)	O(n)	O(1)	O(1)	O(n)
sorted linked list	O(1)	O(1)	O(n)	O(n)	O(n)

- Challenge. Can we do significantly better?

Priority Queues

Trees

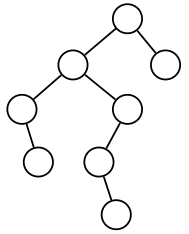
- Rooted trees.
  - Nodes (or vertices) connected with edges.
  - Connected and acyclic.
  - Designated root node.
  - Special type of graph.



- Terminology.
  - Children, parent, descendant, ancestor, leaves, internal nodes, path,..
- Depth and height.
  - Depth of v = length of path from v to root.
  - Height of v = length of path from v to descendant leaf.
  - Depth of T = height of T = length of longest path from root to a leaf.

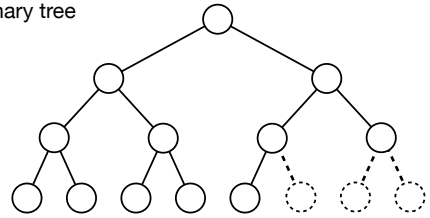
Trees

- Binary tree.
  - Rooted tree.
  - Each node has at most two children called the left child and right child



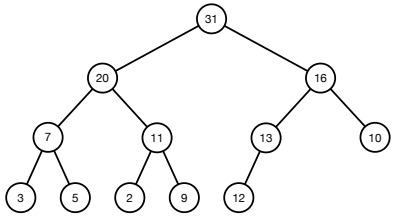
- Complete binary tree. Binary tree where all levels of tree are full.
- Almost complete binary tree. Complete binary tree with 0 or more rightmost leaves deleted.

- Lemma. Height af an (almost) complete binary tree with n nodes is Θ(log n).
- Pf. See exercises.



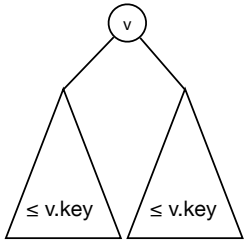
Heaps

- Heaps. Almost complete binary tree. All nodes store one element and the tree satisfies heap-order.



- Heap-order.
  - For all nodes v:
    - all keys in left subtree and right subtree are ≤ v.key.

- Max-heap vs min-heap.



Priority Queues

- Priority Queues
- Trees and Heaps
- Representations of Heaps
- Algorithms on Heaps
- Building a Heap
- Heapsort

Heap

- Data structure. We need the following navigation operations on a heap.
  - PARENT(x): return parent of x.
  - LEFT(x) : return left child of x.
  - RIGHT(x): return right child of x.

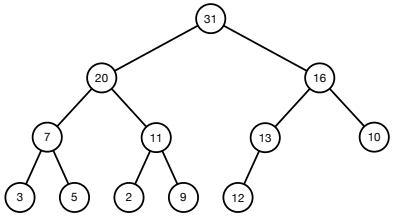
- Challenge. How can we represent a heap compactly to support fast navigation?

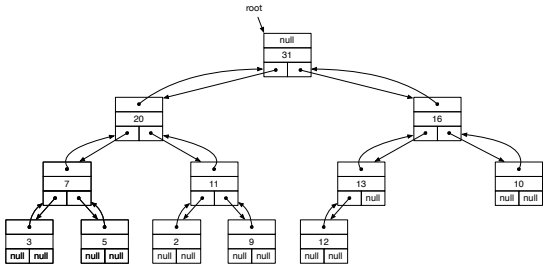
Heap

- Linked representation. Each node stores
  - v.key
  - v.parent
  - v.left
  - v.right

- PARENT, LEFT, RIGHT by following pointer.

- Time. O(1)
- Space. O(n)



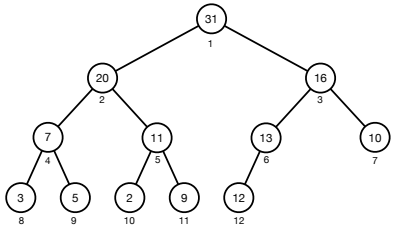


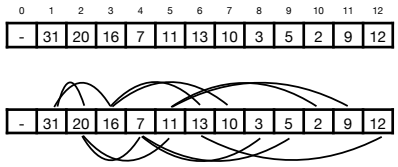
Heap

- Array representation.
  - Array H[0..n]
  - H[0] unused
  - H[1..n] stores nodes in level order.

- PARENT(x): return ⌊x/2⌋
- LEFT(x) : return 2x.
- RIGHT(x): return 2x + 1

- Time. O(1)
- Space. O(n)



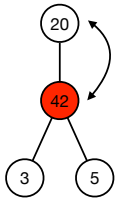


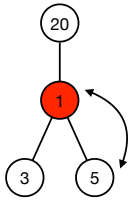
Priority Queues

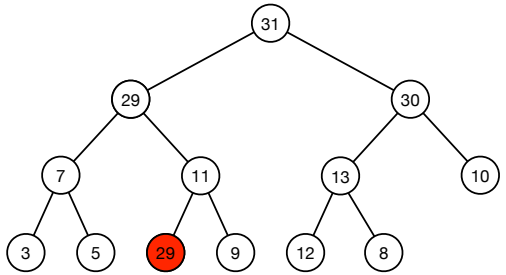
- Priority Queues
- Trees and Heaps
- Representations of Heaps
- Algorithms on Heaps
- Building a Heap
- Heapsort

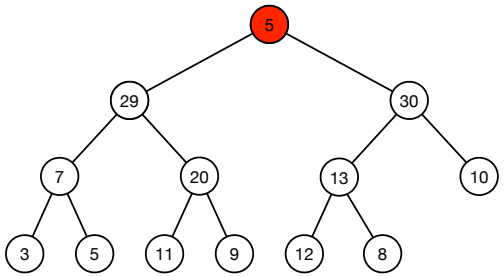
Algorithms on Heaps

- BubbleUp(x):
  - If heap order is violated at node x because key is larger than key at parent:
  - Swap x and parent
  - Repeat with parent until heap order is satisfied.
- BubbleDown(x):
  - If heap order is violated at node x because key is smaller than key at left or right child:
  - Swap x and child c with largest key.
  - Repeat with child until heap order is satisfied.



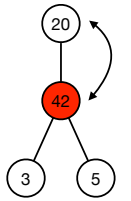


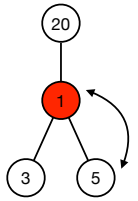




Algorithms on Heaps

- BubbleUp(x):
  - If heap order is violated at node x because key is larger than key at parent:
  - Swap x and parent
  - Repeat with parent until heap order is satisfied.
- BubbleDown(x):
  - If heap order is violated at node x because key is smaller than key at left or right child:
  - Swap x and child c with largest key.
  - Repeat with child until heap order is satisfied.
- Time.
  - BubbleUp and BubbleDown in O(log n) time.
- How can we use them to implement a priority queue?





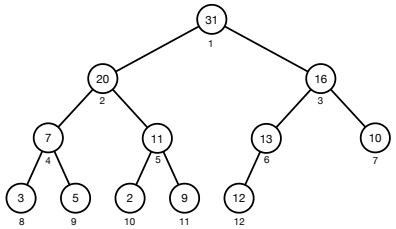
Priority Queues

```
MAX()
    return H[1]
```

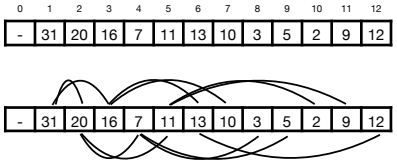
```
INSERT(x)
    n = n + 1
    H[n] = x
    BUBBLEUP(n)
```

```
EXTRACTMAX()
    r = H[1]
    H[1] = H[n]
    n = n - 1
    BUBBLEDOWN(1)
    return r
```

```
INCREASEKEY(x,k)
    H[x] = k
    BUBBLEUP(x)
```



- Ex. Trace execution of following sequence in initially empty heap: 2, 5, 7, 6, 4, E, E
- Numbers mean INSERT og E is EXTRACTMAX. Draw heap after each operation.



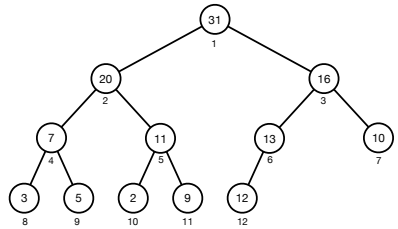
Priority Queues

```
MAX()
    return H[1]
```

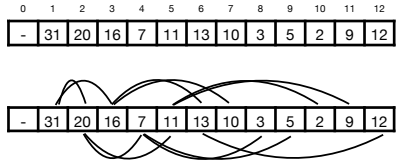
```
INSERT(x)
    n = n + 1
    H[n] = x
    BUBBLEUP(n)
```

```
EXTRACTMAX()
    r = H[1]
    H[1] = H[n]
    n = n - 1
    BUBBLEDOWN(1)
    return r
```

```
INCREASEKEY(x,k)
    H[x] = k
    BUBBLEUP(x)
```



- Time.
  - MAX in O(1) time.
  - EXTRACTMAX, INCREASEKEY, and INSERT in O(log n) time.



Priority Queues

Data structure	MAX	EXTRACTMAX	INCREASEKEY	INSERT	Space
linked list	O(n)	O(n)	O(1)	O(1)	O(n)
sorted linked list	O(1)	O(1)	O(n)	O(n)	O(n)
heap	O(1)	O(log n)	O(log n)	O(log n)	O(n)

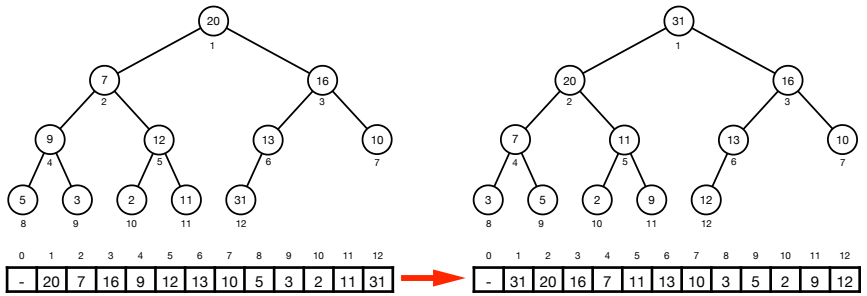
- Heaps with array data structure is an example of an implicit data structure.

Priority Queues

- Priority Queues
- Trees and Heaps
- Representations of Heaps
- Algorithms on Heaps
- Building a Heap
- Heapsort

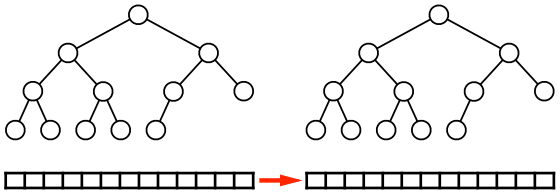
Building a Heap

- Building a heap. Given n integers in a array H[1..n], convert array to a heap.



Building a Heap

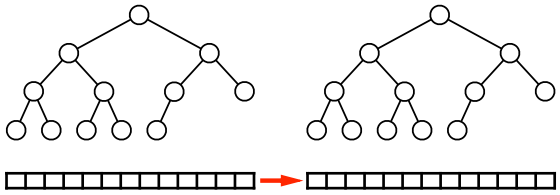
- Solution 1: top-down construction
  - For all nodes in increasing level order apply BUBBLEUP.



- Time.
  - For each node of depth d, we use O(d) time.
  - 1 node of depth 0, 2 nodes of depth 1, 4 nodes of depth 2, ..., ~n/2 nodes of depth log n.
  - ⇒ total time is O(n log n)
- Challenge. Can we do better?

Building a Heap

- Solution 2: bottom-up construction
  - For all nodes in decreasing level order apply BUBBLEDOWN.



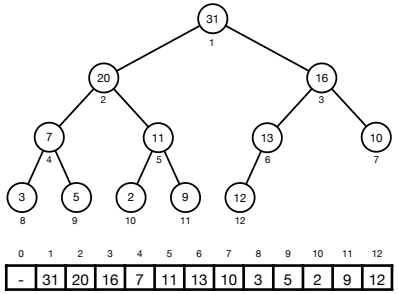
- Time.
  - For each node of height h we use O(h) time.
  - 1 node of height log n, 2 nodes of height log n - 1, ..., n/4 nodes of height 1, n/2 nodes of height 0.
  - ⇒ total time is O(n) (see exercise)

Priority Queues

- Priority Queues
- Trees and Heaps
- Representations of Heaps
- Algorithms on Heaps
- Building a Heap
- Heapsort

Heapsort

- Sorting. How can we sort an array H[1..n] using a heap?
- Solution.
  - Build a heap for H.
  - Apply n EXTRACTMAX.
    - Insert results in the end of array.
  - Return H.

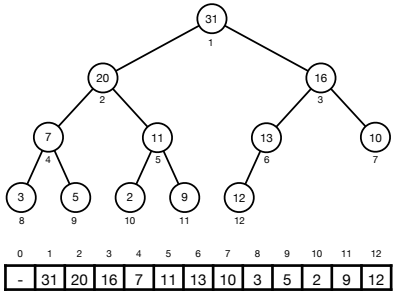


- Time.
  - Heap construction in O(n) time.
  - n EXTRACTMAX in O(nlog n) time.
  - ⇒ total time is O(nlog n).

Heapsort

- Theorem. We can sort an array in O(n log n) time.

- Uses only O(1) extra space.
- In-place sorting algorithm.



Priority Queues

- Priority Queues
- Trees and Heaps
- Representations of Heaps
- Algorithms on Heaps
- Building a Heap
- Heapsort

Union Find

- Union Find
- Quick Find
- Quick Union
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

Philip Bille

Union Find

- Union Find
- Quick Find
- Quick Union
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

Philip Bille

Union Find

- Union find. Maintain a dynamic family of sets supporting the following operations:
  - INIT(n): construct sets {0}, {1},…, {n-1}
  - UNION(i,j): forms the union of the two sets that contain i and j. If i and j are in the same set nothing happens.
  - FIND(i): return a representative for the set that contains i.

INIT(9)

{0} {1} {2} {3} {4} {5} {6} {7} {8}

UNION(5,0)

{1, 0, 6} {8, 3, 2, 7} {4, 5} → {1, 0, 6, 4, 5} {8, 3, 2, 7}

Union Find

- Applications.
  - Dynamic connectivity.
  - Minimum spanning tree.
  - Unification in logic and compilers.
  - Nearest common ancestors in trees.
  - Hoshen-Kopelman algorithm in physics.
  - Games (Hex and Go)
  - Illustration of clever techniques in data structure design.

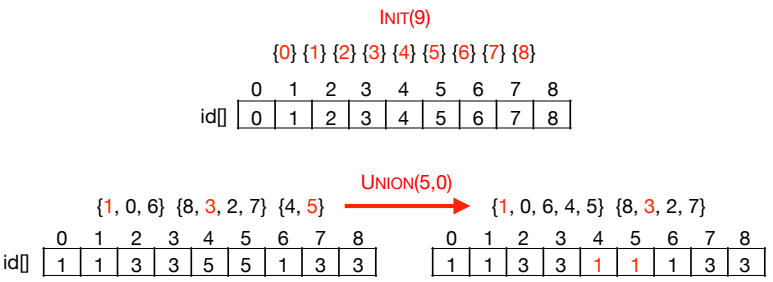
Union Find

- Union Find
- **Quick Find**
- Quick Union
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

Philip Bille

Quick Find

- Quick find. Maintain array id[0..n-1] such that id[i] = representative for i.
  - INIT(n): set elements to be their own representative.
  - UNION(i,j): if FIND(i) ≠ FIND(j), update representative for all elements in one of the sets.
  - FIND(i): return representative.

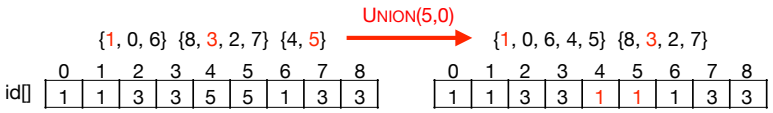


Quick Find

```
INIT(n):
   for k = 0 to n-1
      id[k] = k
```

```
FIND(i):
   return id[i]
```

```
UNION(i,j):
   iID = FIND(i)
   jID = FIND(j)
   if (iID ≠ jID)
      for k = 0 to n-1
         if (id[k] == iID)
            id[k] = jID
```



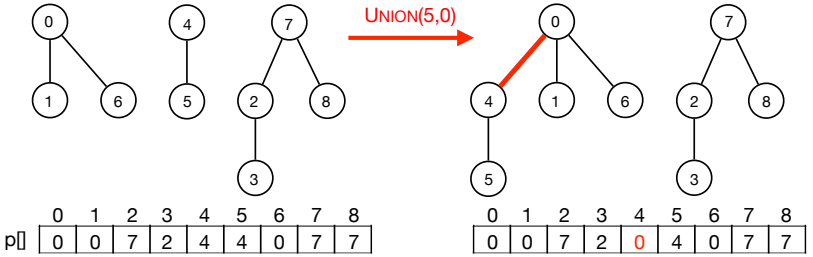
- Time.
  - O(n) time for INIT, O(n) time for UNION, and O(1) tid for FIND.

Union Find

- Union Find
- Quick Find
- **Quick Union**
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

Quick Union

- Quick union. Maintain each sets as a rooted tree.
- Store trees as array p[0..n-1] such that p[i] is the parent of i and p[root] = root. Representative is the root of tree.
  - INIT(n): create n trees with one element each.
  - UNION(i,j): if FIND(i) ≠ FIND(j), make the root of one tree the child of the root of the other tree.
  - FIND(i): follow path to root and return root.



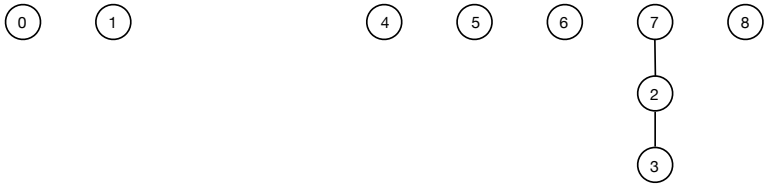
INIT(9)

UNION(3,2)

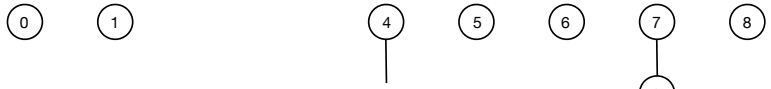
UNION(3,2)

UNION(2,7)

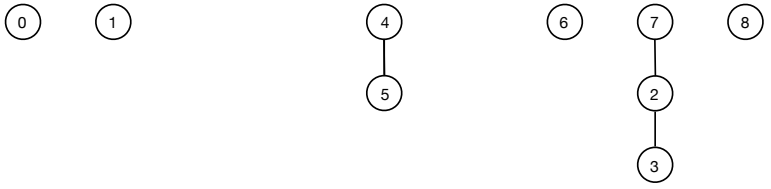
UNION(2,7)



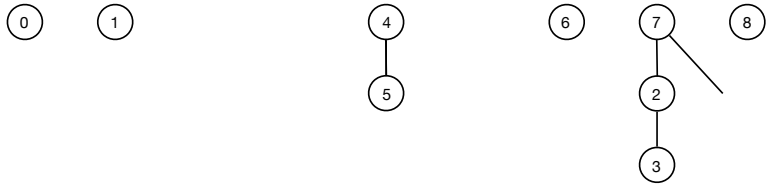
UNION(5,4)



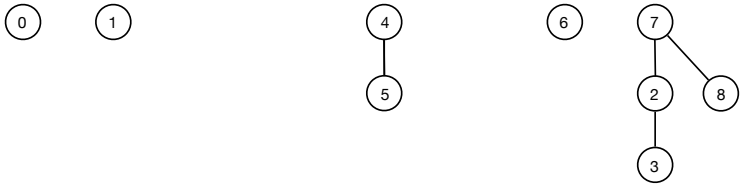
UNION(5,4)



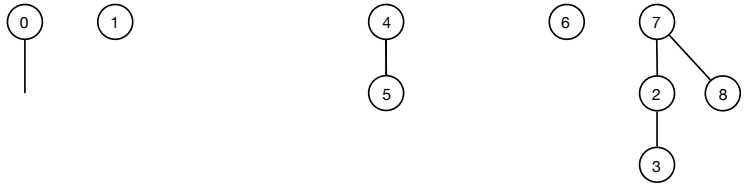
UNION(8,3)



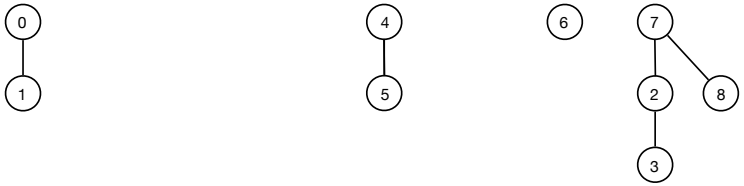
UNION(8,3)



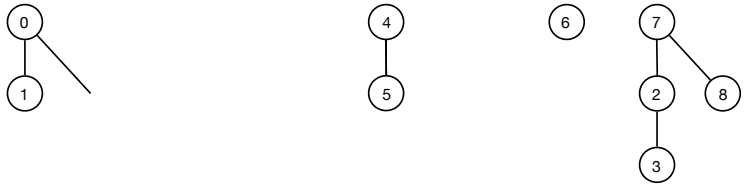
UNION(1,0)



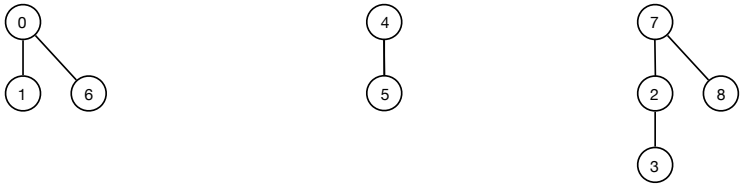
UNION(1,0)



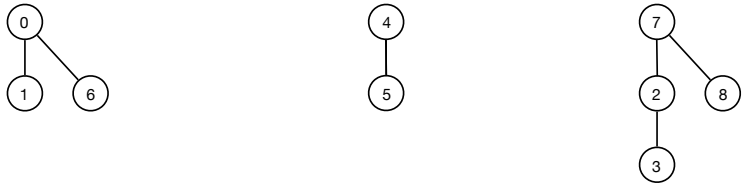
UNION(6,1)



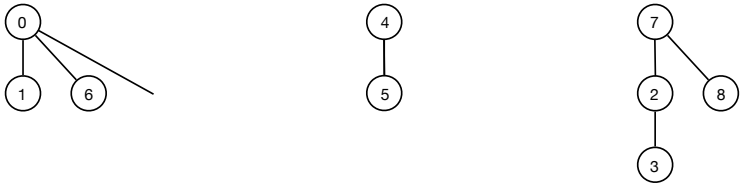
UNION(6,1)



UNION(7,3)



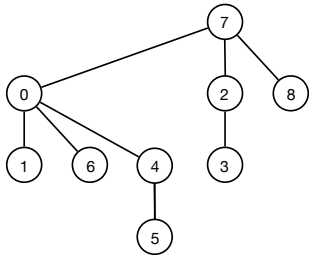
UNION(5,0)



UNION(5,0)

UNION(6,2)

UNION(6,2)



Quick Union

- INIT(n): create n trees with one element each.
- UNION(i,j): if FIND(i) ≠ FIND(j), make the root of one tree the child of the root of the other tree.
- FIND(i): follow path to root and return root.

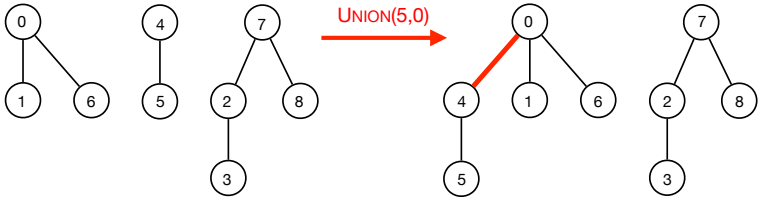
- Exercise. Show data structure after each operation in the following sequence.
  - INIT(7), UNION(0,1), UNION(2,3), UNION(5,1), UNION(5,0), UNION(0,3), UNION(5,2), UNION(4,3), UNION(4,6).

Quick Union

```
INIT(n):
   for k = 0 to n-1
      p[k] = k
```

```
FIND(i):
   while (i != p[i])
      i = p[i]
   return i
```

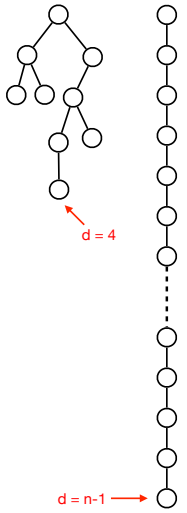
```
UNION(i,j):
   r_i = FIND(i)
   r_j = FIND(j)
   if (r_i ≠ r_j)
      p[r_i] = r_j
```



- Time.
  - O(n) time for INIT, O(d) tid for UNION and FIND, where d is the depth of the tree.

Quick Union

- UNION and FIND depend on the depth of the tree.
- Bad news. Depth can be n-1.
- Challenge. Can combine trees to limit the depth?

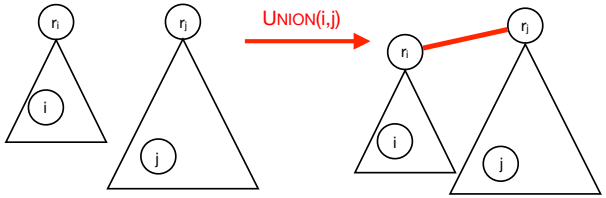


Union Find

- Union Find
- Quick Find
- Quick Union
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

Weighted Quick Union

- Weighted quick union. Extension of quick union.
- Maintain extra array sz[0..n-1] such sz[i] = the size of the subtree rooted at i.
  - INIT: as before + initialize sz[0..n-1].
  - FIND: as before.
  - UNION(i,j): if FIND(i) ≠ FIND(j), make the root of the smaller tree the child of the root of the larger tree.
- Intuition. UNION balances the trees.



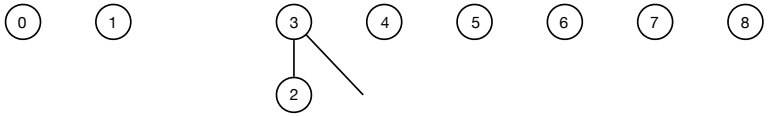
$sz[r_j] = sz[r_i] + sz[r_j]$

INIT(9)

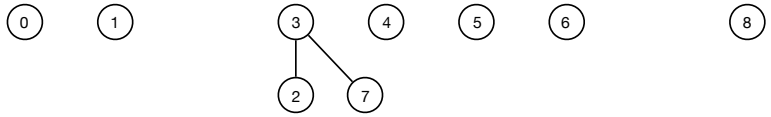
UNION(3,2)

UNION(3,2)

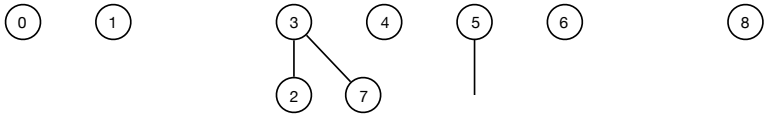
UNION(2,7)



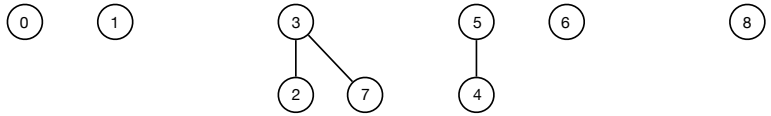
UNION(2,7)



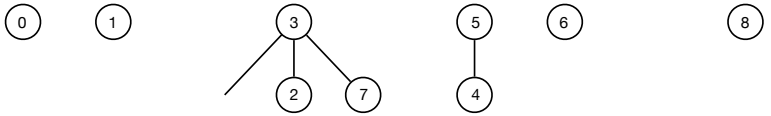
UNION(5,4)



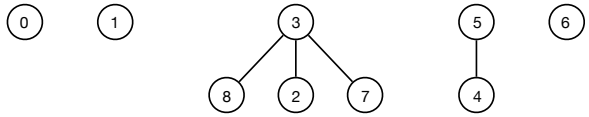
UNION(5,4)



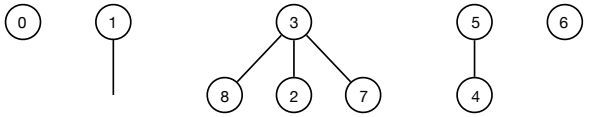
UNION(8,3)



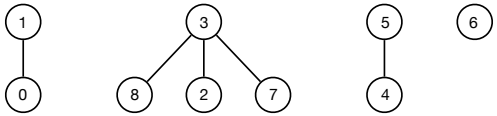
UNION(8,3)



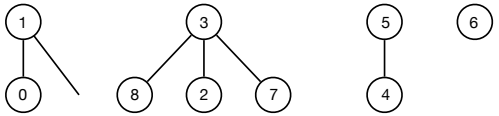
UNION(1,0)



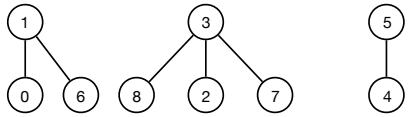
UNION(1,0)



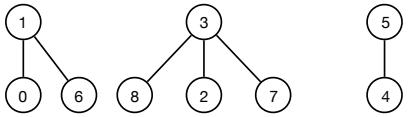
UNION(6,1)



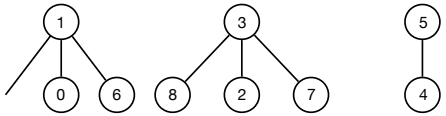
UNION(6,1)



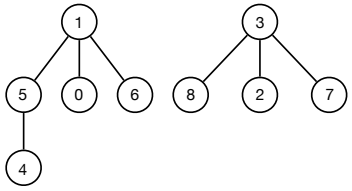
UNION(7,3)



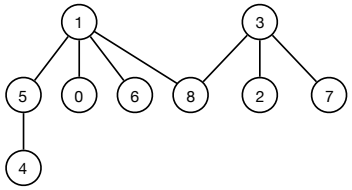
UNION(5,1)



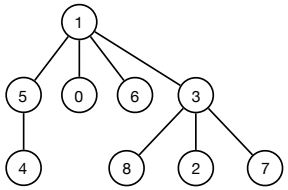
UNION(5,1)



UNION(6,3)

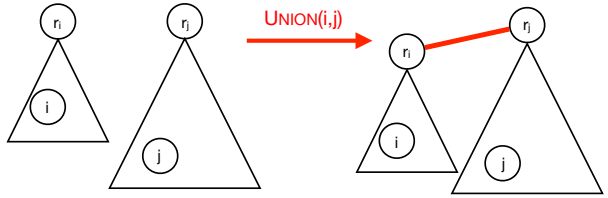


UNION(6,3)



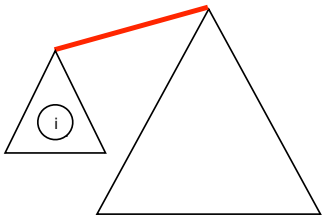
Weighted Quick Union

```
UNION(i,j):
   ri = FIND(i)
   rj = FIND(j)
   if (ri ≠ rj)
      if (sz[ri] < sz[rj])
         p[ri] = rj
         sz[rj] = sz[ri] + sz[rj]
      else
         p[rj] = ri
         sz[ri] = sz[ri] + sz[rj]
```



Weighted Quick Union

- Lemma. With weighted quick union the depth of a node is at most $\log_2 n$.
- Proof.
  - Consider node i with depth d_i.
  - Initially $d_i = 0$.
  - d_i increases with 1 when the tree is combined with a larger tree.
  - The combined tree is at least twice the size.
  - We can double the size of trees at most $\log_2 n$ times.
  - $\Longrightarrow d_i \leq \log_2 n$.



Union Find

Data structure	UNION	FIND
quick find	O(n)	O(1)
quick union	O(n)	O(n)
weighted quick union	O(log n)	O(log n)

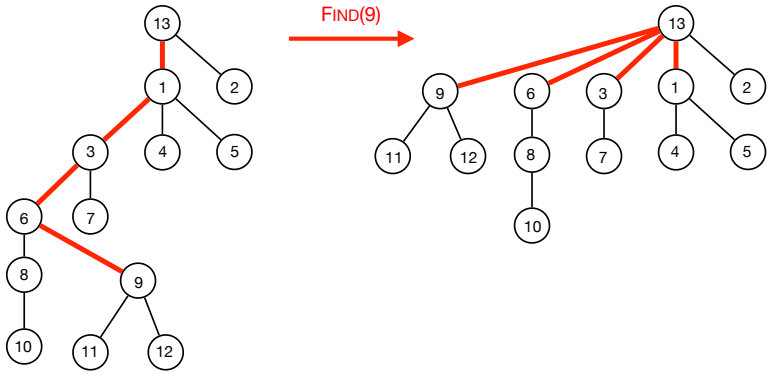
- Challenge. Can we do even better?

Union Find

- Union Find
- Quick Find
- Quick Union
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

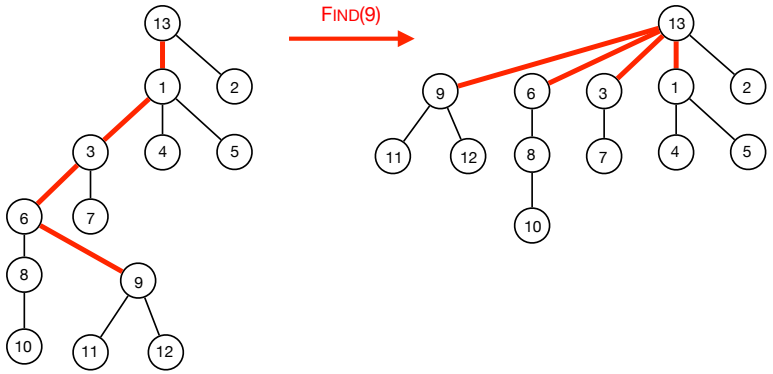
Path Compression

- Path compression. Compress path on FIND. Make all nodes on the path children of the root.
- No change in running time for a single FIND. Subsequent FIND become faster.
- Works with both quick union and weighted quick union.



Path Compression

- Theorem [Tarjan 1975]. With path compression any sequence of m FIND and UNION operations on n elements take O(n + m α(m,n)) time.
- α(m,n) is the inverse of Ackermanns function. α(m,n) ≤ 5 for any practical input.
- Theorem [Fredman-Saks 1985]. It is not possible to support m FIND and UNION operations O(n + m) time.



Union Find

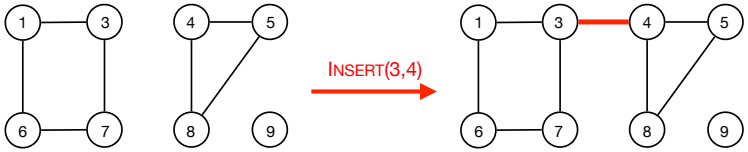
- Union Find
- Quick Find
- Quick Union
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

Dynamic Connectivity

- Dynamic connectivity. Maintain a dynamic graph supporting the following operations:
  - INIT(n): create a graph G with n vertices and no edges.
  - CONNECTED(u,v): determine if u og v are connected.
  - INSERT(u, v): add edge (u,v). We assume (u,v) does not already exists.

Dynamic Connectivity

- Implementation with union find.
  - INIT(n): initialize a union find data structure with n elements.
  - CONNECTED(u,v): FIND(u) == FIND(v).
  - INSERT(u, v): UNION(u,v)



- Time
  - O(n) time for INIT, O(log n) time for CONNECTED, and O(log n) time for INSERT

Union Find

- Union Find
- Quick Find
- Quick Union
- Weighted Quick Union
- Path Compression
- Dynamic Connectivity

Minimum Spanning Trees

- Minimum Spanning Trees
- Representation of Weighted Graphs
- Properties of Minimum Spanning Trees
- Prim's Algorithm
- Kruskal's Algorithm

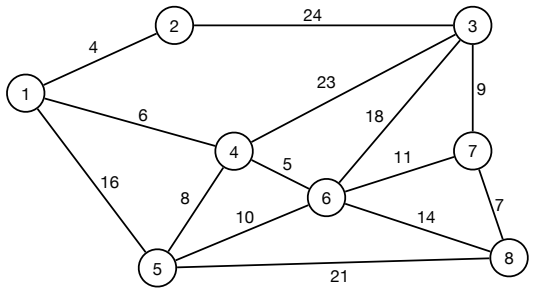
Philip Bille

Minimum Spanning Trees

- Minimum Spanning Trees
- Representation of Weighted Graphs
- Properties of Minimum Spanning Trees
- Prim's Algorithm
- Kruskal's Algorithm

Minimum Spanning Trees

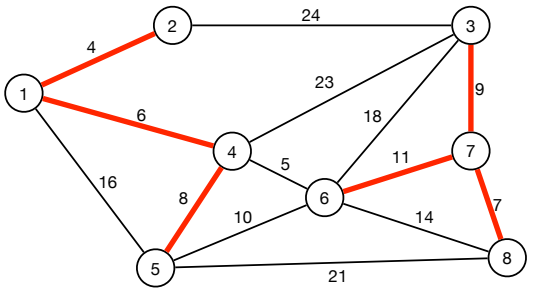
- Weighted graphs. Weight w(e) on each edge e in G.
- Spanning tree. Subgraph T of G over all vertices that is connected and acyclic.
- Minimum spanning tree (MST). Spanning tree of minimum total weight.



Graph G

Minimum Spanning Trees

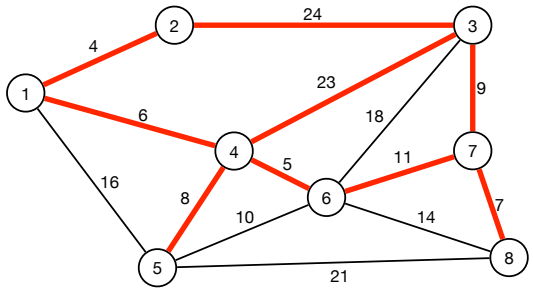
- Weighted graphs. Weight w(e) on each edge e in G.
- Spanning tree. Subgraph T of G over all vertices that is connected and acyclic.
- Minimum spanning tree (MST). Spanning tree of minimum total weight.



Not connected

Minimum Spanning Trees

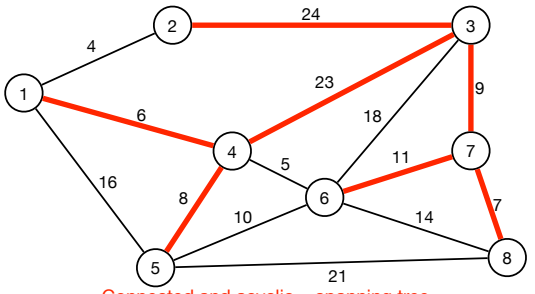
- Weighted graphs. Weight w(e) on each edge e in G.
- Spanning tree. Subgraph T of G over all vertices that is connected and acyclic.
- Minimum spanning tree (MST). Spanning tree of minimum total weight.



Connected and cyclic

Minimum Spanning Trees

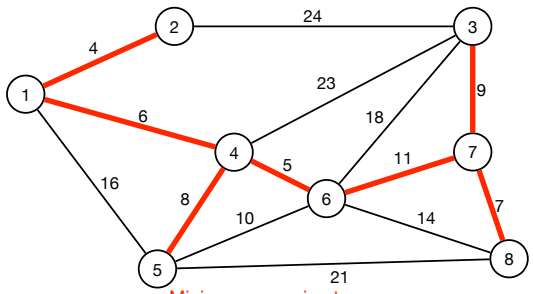
- Weighted graphs. Weight w(e) on each e in G.
- Spanning tree. Subgraph T of G over all vertices that is connected and acyclic.
- Minimum spanning tree (MST). Spanning tree of minimum total weight.



Connected and acyclic = spanning tree
Total weight = 6 + 8 + 23 + 24 + 9 + 11 + 7 = 88

Minimum Spanning Trees

- Weighted graphs. Weight w(e) on each e in G.
- Spanning tree. Subgraph T of G over all vertices that is connected and acyclic.
- Minimum spanning tree (MST). Spanning tree of minimum total weight.



Minimum spanning tree
Total weight = 4 + 6 + 5 + 8 + 11 + 9 + 7 = 50

Applications

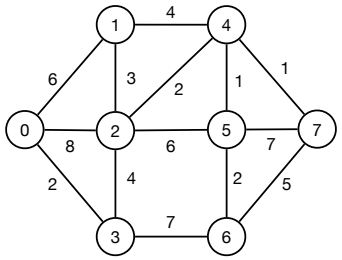
- Network design.
  - Computer, road, telephone, electrical, circuit, cable tv, hydralic, …
- Approximation algorithms.
  - Travelling salesperson problem, steiner trees.
- Other applications.
  - Meteorology, cosmology, biomedical analysis, encoding, image analysis, ...

Minimum Spanning Trees

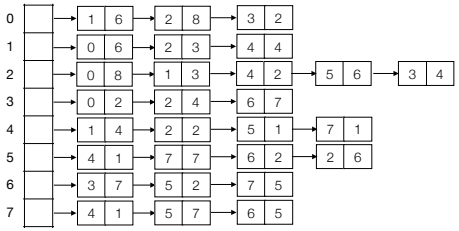
- Minimum Spanning Trees
- Representation of Weighted Graphs
- Properties of Minimum Spanning Trees
- Prim's Algorithm
- Kruskal's Algorithm

Representation of Weighted Graphs

- Adjacency matrix and adjacency list.
- Similar for directed graphs.



	0	1	2	3	4	5	6	7
0	0	6	8	2	0	0	0	0
1	6	0	3	0	4	0	0	0
2	8	3	0	4	2	6	0	0
3	2	0	4	0	0	0	7	0
4	0	4	2	0	0	1	0	1
5	0	0	6	0	1	0	2	7
6	0	0	0	7	0	2	0	5
7	0	0	0	0	1	7	5	0



Minimum Spanning Trees

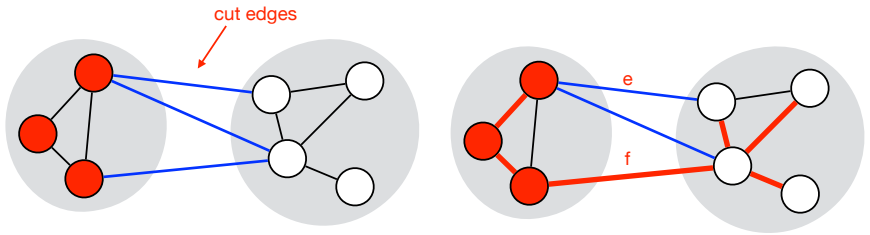
- Minimum Spanning Trees
- Representation of Weighted Graphs
- Properties of Minimum Spanning Trees
- Prim's Algorithm
- Kruskal's Algorithm

Properties of Minimum Spanning Trees

- Assume for simplicity:
  - All edge weights are distinct.
  - G is connected.
- $\Longrightarrow$ MST exists and is unique.

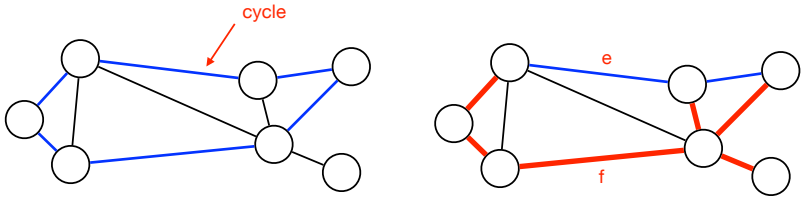
Cut Property

- Def. A cut is a partition of the vertices into two non-empty sets.
- Def. A cut edge is an edge crossing the cut.
- Cut property. For any cut, the lightest cut edge is in the MST.
- Proof.
  - Assume the lightest cut edge e is not in the MST.
  - Replace other cut edge f with e.
  - Produces a new spanning with smaller weight.



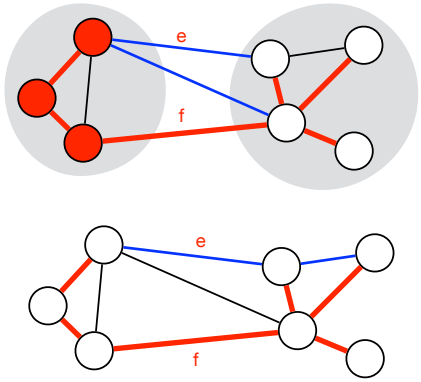
Cycle Property

- Cycle property. For any cycle, the heaviest edge is not in the MST.
- Proof.
  - Assume heaviest edge f in cycle is in MST.
  - Replace f with lighter edge e in cycle.
  - Produces a new spanning tree with smaller weight.



Properties of Minimum Spanning Trees

- Cut property. For any cut, the lightest cut edge is in the MST.
- Cycle property. For any cycle, the heaviest edge is not in the MST.

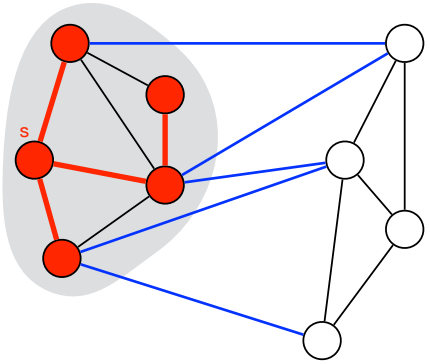


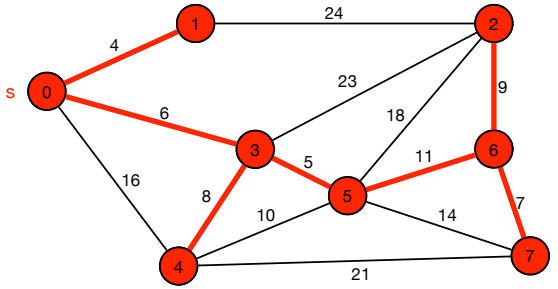
Minimum Spanning Trees

- Minimum Spanning Trees
- Representation of Weighted Graphs
- Properties of Minimum Spanning Trees
- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm

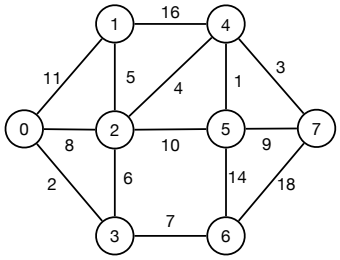
- Grow a tree T from some vertex s.
- In each step, add lightest edge with one endpoint i T.
- Stop when T has n-1 edges.





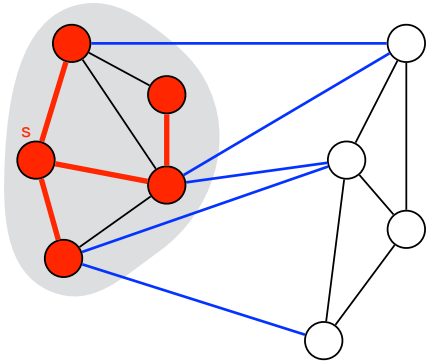
Prim's Algorithm

- Grow a tree T from some vertex s.
- In each step, add lightest edge with one endpoint i T.
- Stop when T has n-1 edges.
- Exercise. Show execution of Prim's algorithm from vertex 0 on the following graph.



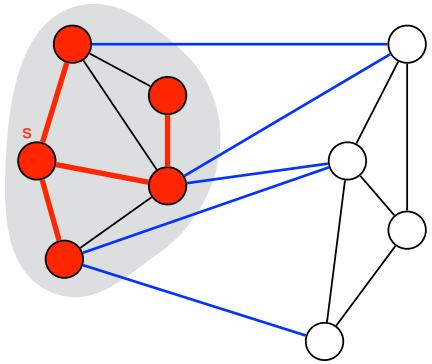
Prim's Algorithm

- Lemma. Prim's algorithm computes the MST.
- Proof.
  - Consider cut between T and other vertices.
  - We add lightest cut edge to T.
  - Cut property $\Longrightarrow$ edge is in MST $\Longrightarrow$ T is MST after n-1 steps.



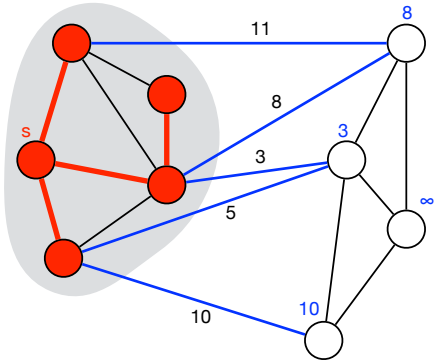
Prim's Algorithm

- Implementation. How do we implement Prim's algorithm?
- Challenge. Find the lightest cut edge.



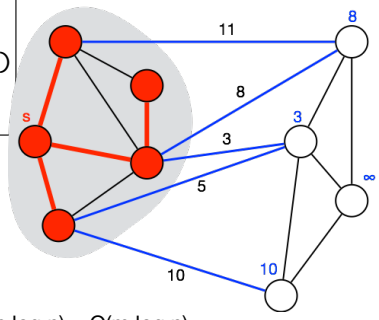
Prim's Algorithm

- Implementation. Maintain vertices outside T in priority queue.
  - Key of vertex v = weight of lightest cut edge (∞ if no cut edge).
  - In each step:
    - Find lightest edge = EXTRACT-MIN
    - Update weight of neighbors of new vertex with DECREASE-KEY.



Prim's Algorithm

```
PRIM(G, s)
   for all vertices v∈V
      v.key = ∞
      v.π = null
      INSERT(P,v)
   DECREASE-KEY(P,s,0)
   while (P ≠ ∅)
      u = EXTRACT-MIN(P)
      for all neighbors v of u
         if (v ∈ P and w(u,v)<key[v])
            DECREASE-KEY(P,v,w(u,v))
            v.π = u
```



- Time.
  - n EXTRACT-MIN
  - n INSERT
  - O(m) DECREASE-KEY
- Total time with min-heap. O(n log n + n log n + m log n) = O(m log n)

Prim's Algorithm

- Priority queues and Prim's algorithm. Complexity of Prim's algorithm depend on priority queue.
  - n INSERT
  - n EXTRACT-MIN
  - O(m) DECREASE-KEY

Priority queue	INSERT	EXTRACT-MIN	DECREASE-KEY	Total
array	O(1)	O(n)	O(1)	$O(n^2)$
binary heap	O(log n)	O(log n)	O(log n)	O(m log n)
Fibonacci heap	O(1)†	O(log n)†	O(1)†	O(m + n log n)

† = amortized

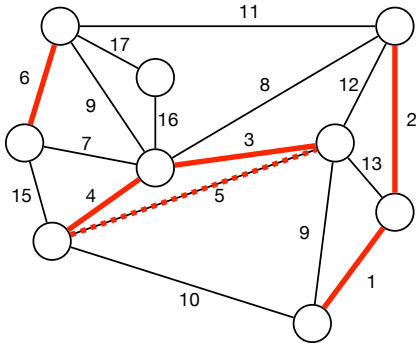
- Greed. Prim's algorithm is a greedy algorithm.
  - Makes local optimal choices in each step that lead to global optimal solution.

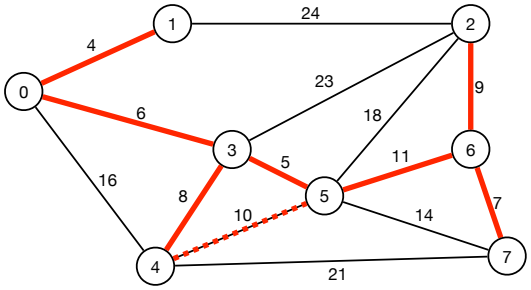
Minimum Spanning Trees

- Minimum Spanning Trees
- Representation of Weighted Graphs
- Properties of Minimum Spanning Trees
- Prim's Algorithm
- Kruskal's Algorithm

Kruskal's Algorithm

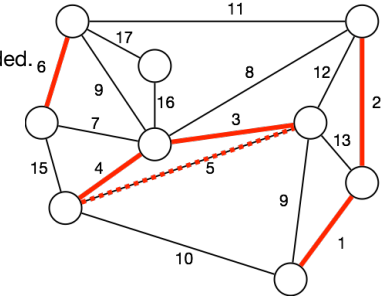
- Consider edges from lightest to heaviest.
- In each step, add edge to T if it does not create a cycle.
- Stop when T has n-1 edges.





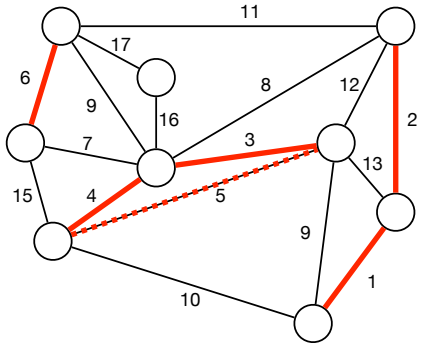
Kruskal's Algorithm

- Lemma. Kruskal's algorithm computes the MST.
- Proof.
  - Algorithms considers edges from light to heavy. At edge e = (u,v):
  - Case 1. e creates a cycle and is not added to T.
    - e must be heaviest edge on cycle.
    - Cycle property $\Longrightarrow$ e is not in MST.
  - Case 2. e does not create a cycle and is added to T.
    - e must be lightest edge in cut.
    - Cut property $\Longrightarrow$ e is in MST.
  - $\Longrightarrow$ T is MST when n-1 edges are added.



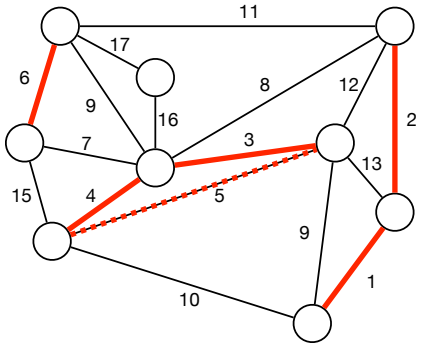
Kruskal's Algorithm

- Implementation. How do we implement Kruskal's algorithm?
- Challenge. Check if an edge form a cycle.



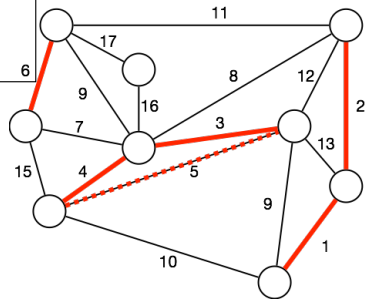
Kruskal's Algorithm

- Implementation. Maintain edges in a data structure for dynamic connectivity.
- In each step:
  - Check if an edge creates a cycle = CONNECTED.
  - Add new edge = INSERT.



Kruskal's Algorithm

```
KRUSKAL(G)
   Sort edges
   INIT(n)
   for all edges (u,v) i sorted order
      if (!CONNECTED(u,v))
         INSERT(u,v)
   return all inserted edges
```



- Time.
  - Sorting m edges.
  - 1 INIT
  - m CONNECTED
  - n INSERT
- Total time. O(m log m + n + m log n + n log n) = O(m log n).
- Greed. Kruskal's algorithm is also a greedy algorithm.

Minimum Spanning Trees

- What is the best algorithm for computing MSTs?

Year	Time	Authors
???	O(m log n)	Jarnik, Prim, Dijkstra, Kruskal, Boruvka, ?
1975	O(m log log n)	Yao
1986	O(m log* n)	Fredman, Tarjan
1995	O(m)‡	Karger, Klein, Tarjan
2000	O(nα(m,n))	Chazelle
2002	optimal	Pettie, Ramachandran

‡ = randomized

Minimum Spanning Trees

- Minimum Spanning Trees
- Representation of Weighted Graphs
- Properties of Minimum Spanning Trees
- Prim's Algorithm
- Kruskal's Algorithm

Shortest Paths

- Shortest Paths
- Properties of Shortest Paths
- Dijkstra's Algorithm
- Shortest Paths on DAGs

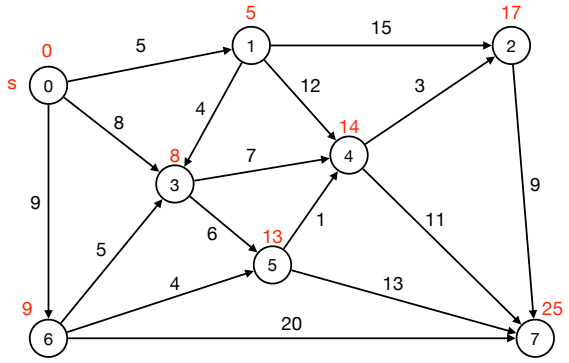
Philip Bille

Shortest Paths

- Shortest Paths
- Properties of Shortest Paths
- Dijkstra's Algorithm
- Shortest Paths on DAGs

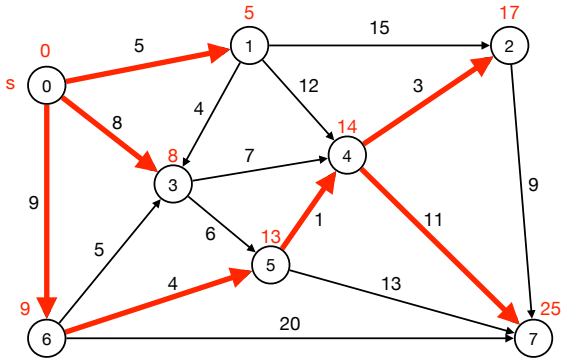
Shortest Paths

- Shortest paths. Given a directed, weighted graph G and vertex s, find shortest path from s to all vertices in G.



Shortest Paths

- Shortest paths. Given a directed, weighted graph G and vertex s, find shortest path from s to all vertices in G.
- Shortest path tree. Represent shortest paths in a tree from s.



Applications

- Routing, scheduling, pipelining, ...

Shortest Paths

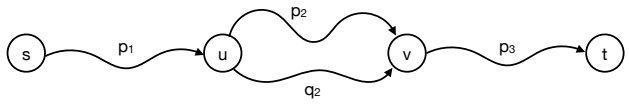
- Shortest Paths
- Properties of Shortest Paths
- Dijkstra's Algorithm
- Shortest Paths on DAGs

Properties of Shortest Paths

- Assume for simplicity:
  - All vertices are reachable from s.
- $\Longrightarrow$ a (shortest) path to each vertex always exists.

Properties of Shortest Paths

- Subpath property. Any subpath of a shortest path is a shortest path.
- Proof.
  - Consider shortest path from s to t consisting of p_1, p_2 and p_3.



  - Assume q_2 is shorter than p_2.
  - $\Longrightarrow$ Then p_1, q_2 and p_3 is shorter than p.

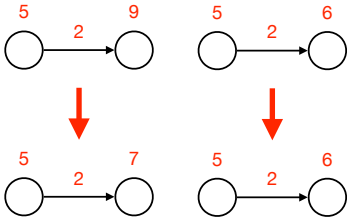
Shortest Paths

- Shortest Paths
- Properties of Shortest Paths
- Dijkstra's Algorithm
- Shortest Paths on DAGs

Dijkstra's Algorithm

- Goal. Given a directed, weighted graph with non-negative weights and a vertex s, compute shortest paths from s to all vertices.

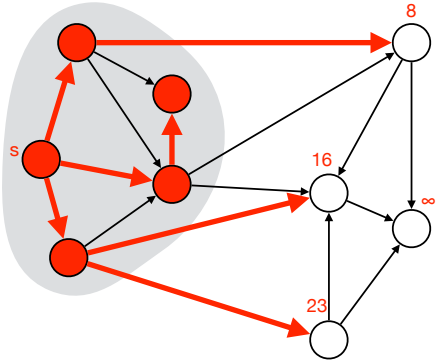
- Dijkstra's algorithm.
  - Maintains distance estimate v.d for each vertex v = length of shortest known path from s to v.
  - Updates distance estimates by relaxing edges.

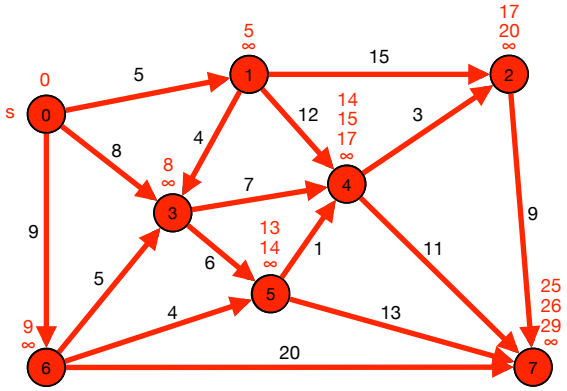


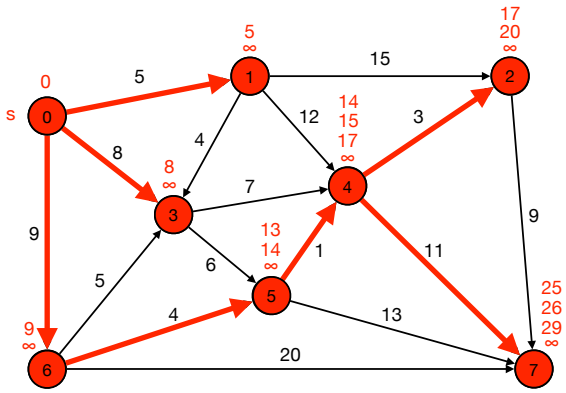

```
RELAX(u,v)
   if (v.d > u.d + w(u,v))
      v.d = u.d + w(u,v)
```

Dijkstra's Algorithm

- Initialize s.d = 0 and v.d = ∞ for all vertices v ∈ V\{s}.
- Grow tree T from s.
- In each step, add vertex with smallest distance estimate to T.
- Relax all outgoing edges of v.

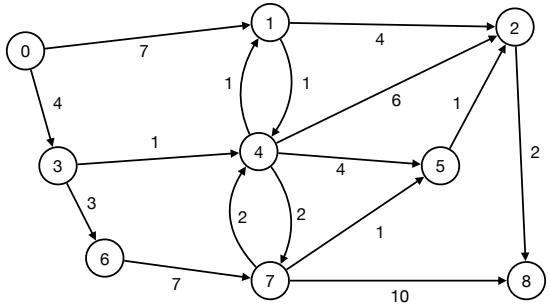






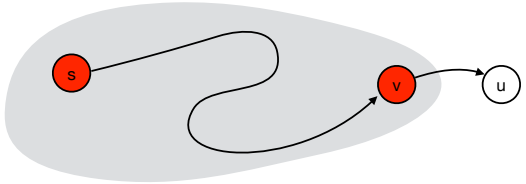
Dijkstra's Algorithm

- Initialize s.d = 0 and v.d = ∞ for all vertices v ∈ V\{s}.
- Grow tree T from s.
- In each step, add vertex with smallest distance estimate to T.
- Relax all outgoing edges of v.
- Exercise. Show execution of Dijkstra's algorithm from vertex 0.



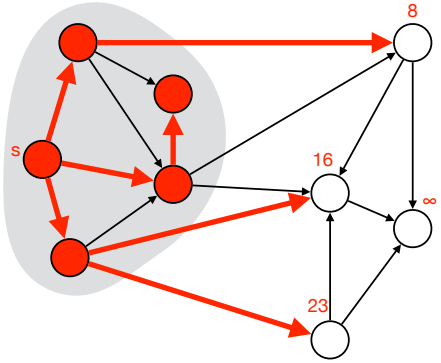
Dijkstra's Algorithm

- Lemma. Dijkstra's algorithms computes shortest paths.
- Proof.
  - Consider some step after growing tree T and assume distances in T are correct.
  - Consider closest vertex u of s not in T.
  - Shortest path from s to u ends with an edge e = (v,u).
  - v is closer than u to s $\Longrightarrow$ v is in T. (u was closest not in T)
  - $\Longrightarrow$ shortest path to u is in T except e.
  - e is relaxed $\Longrightarrow$ distance estimate to v is correct shortest distance.
  - Dijkstra adds e to T $\Longrightarrow$ T is shortest path tree after n-1 steps.



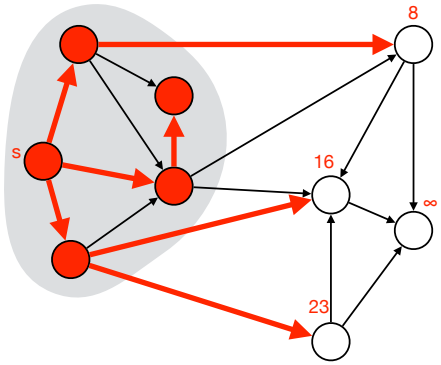
Dijkstra's Algorithm

- Implementation. How do we implement Dijkstra's algorithm?
- Challenge. Find vertex with smallest distance estimate.



Dijkstra's Algorithm

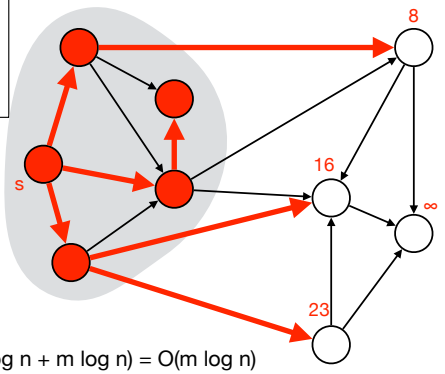
- Implementation. Maintain vertices outside T in priority queue.
  - Key of vertex v = v.d.
  - In each step:
    - Find vertex u with smallest distance estimate = EXTRACT-MIN
    - Relax edges that u point to with DECREASE-KEY.



Dijkstra's Algorithm

```
DIJKSTRA(G, s)
   for all vertices v∈V
      v.d = ∞
      v.π = null
      INSERT(P,v)
   DECREASE-KEY(P,s,0)
   while (P ≠ ∅)
      u = EXTRACT-MIN(P)
      for all v that u point to
         RELAX(u,v)
```

```
RELAX(u,v)
   if (v.d > u.d + w(u,v))
      v.d = u.d + w(u,v)
      DECREASE-KEY(P,v,v.d)
      v.π = u
```



- Time.
  - n EXTRACT-MIN
  - n INSERT
  - < m DECREASE-KEY
- Total time with min-heap. O(n log n + n log n + m log n) = O(m log n)

Dijkstra's Algorithm

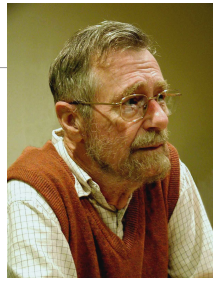
- Priority queues and Dijkstra's algorithm. Complexity of Dijkstra's algorithm depend on priority queue.
  - n INSERT
  - n EXTRACT-MIN
  - < m DECREASE-KEY

Priority queue	INSERT	EXTRACT-MIN	DECREASE-KEY	Total
array	O(1)	O(n)	O(1)	$O(n^2)$
binary heap	O(log n)	O(log n)	O(log n)	O(m log n)
Fibonacci heap	O(1)†	O(log n)†	O(1)†	O(m + n log n)

† = amortized

- Greed. Dijkstra's algorithm is a greedy algorithm.

Edsger W. Dijkstra



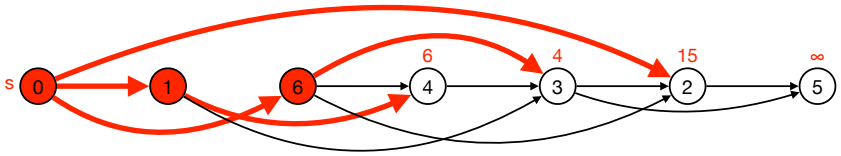
- Edsger Wybe Dijkstra (1930-2002)
- Dijkstra algorithm. "A note on two problems in connexion with graphs". Numerische Mathematik 1, 1959.
- Contributions. Foundations for programming, distributed computation, program verifications, etc.
- Quotes. "*Object-oriented programming is an exceptionally bad idea which could only have originated in California.*"
- "*The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.*"
- "*APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.*"

Shortest Paths

- Shortest Paths
- Properties of Shortest Paths
- Dijkstra's Algorithm
- Shortest Paths on DAGs

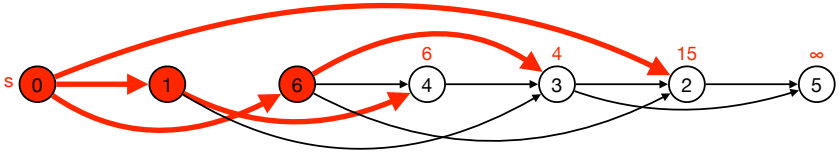
Shortest Paths on DAGs

- Challenge. Is it computationally easier to find shortest paths on DAGs?
- DAG shortest path algoritme.
  - Process vertices in topological order.
  - For each vertex v, relax all edges from v.
- Also works for negative edge weights.



Shortest Paths on DAGs

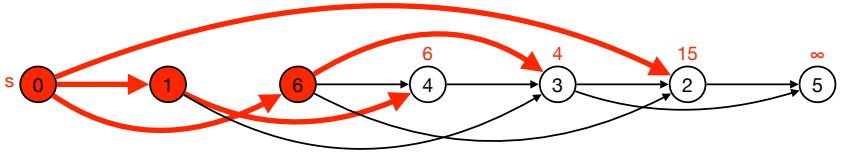
- Lemma. Algorithm computes shortest paths in DAGs.



- Proof.
  - Consider some step after growing tree T and assume distances in T are correct.
  - Consider next vertex u of s not in T.
  - Any path to u consists vertices in T + edge e to u.
  - Edge e is relaxed $\Longrightarrow$ distance to u is shortest.

Shortest Paths on DAGs

- Implementation.
  - Sort vertices in topological order.
  - Relax outgoing edges from each vertex.
- Total time. O(m + n).



Shortest Paths Variants

- Vertices
  - Single source.
  - Single source, single target.
  - All-pairs.
- Edge weights.
  - Non-negative.
  - Arbitrary.
  - Euclidian distances.
- Cycles.
  - No cycles
  - No negative cycles.

Shortest Paths

- Shortest Paths
- Properties of Shortest Paths
- Dijkstra's Algorithm
- Shortest Paths on DAGs

Search Trees

- Dynamic Ordered Sets
- Binary Search Trees
- Balanced Search Trees

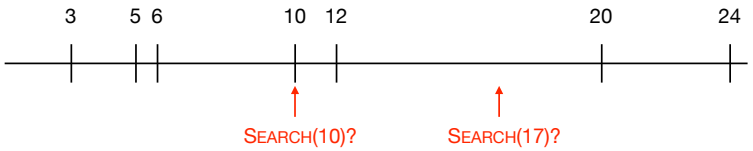
Philip Bille

Search Trees

- Dynamic Ordered Sets
- Binary Search Trees
- Balanced Search Trees

Dynamic Ordered Sets

- Dynamic Ordered Sets. Maintain dynamic set S supporting the following operations. Each element x has key x.key and satellite data x.data.
  - SEARCH(k): return element x such that x.key = k if it exists. Otherwise return null.
  - INSERT(x): add x to S (assume x.key is not already in S).
  - DELETE(x): remove x from S.
- We want to maintain elements ordered by the keys. Allows efficient support for many other important operations and other features.



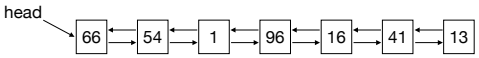
Dynamic Ordered Sets

- Applications.
  - Dictionaries.
  - Indexes.
  - Filesystem.
  - Databases.
  - ....

- Challenge. How can we solve problem with current techniques?

Dynamic Ordered Sets

- Solution 1: linked list. Maintain S in a doubly-linked list.



- SEARCH(k): linear search for largest key ≤ k.
- INSERT(x): insert x in the front of list.
- DELETE(x): remove x from list.

- Time.
  - SEARCH in O(n) time (n = |S|).
  - INSERT and DELETE in O(1) time.
- Space.
  - O(n).

Dynamic Ordered Sets

- Solution 2: sorted array. Maintain S in an sorted array according to keys.

1	2	3	4	5	6	7
1	13	16	41	54	66	96

- SEARCH(k): binary search for k.
- INSERT(x): find index using SEARCH(x.key). Build new array of size +1 with x inserted.
- DELETE(x): build new array of size -1 with element with key k removed.

- Time.
  - SEARCH in O(log n) time.
  - INSERT and DELETE in O(n) time.
- Space.
  - O(n).

Nearest Neighbor

Data structure	SEARCH	INSERT	DELETE	Space
linked list	O(n)	O(1)	O(1)	O(n)
sorted array	O(log n)	O(n)	O(n)	O(n)

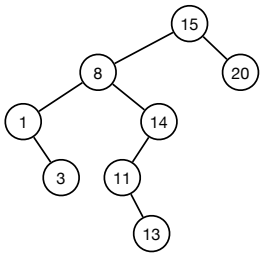
- Challenge. Can we do significantly better?

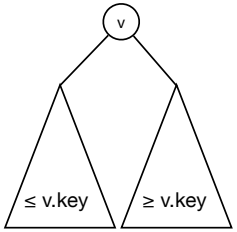
Search Trees

- Dynamic Ordered Sets
- Binary Search Trees
- Balanced Search Trees

Binary Search Trees

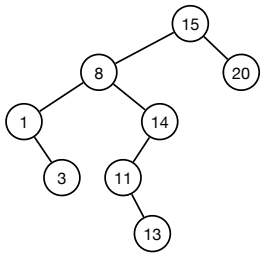
- Binary tree.
  - Rooted tree
  - Each internal node has a left child and/or a right child.
- Binary search tree.
  - Binary tree in symmetric order.
- Symmetric order. For each vertex v:
  - all vertices in left subtree are < v.key.
  - all vertices in right subtree are > v.key.





Binary Search Trees

- Symmetric order ~ inorder traversal outputs the keys in sorted order.



- Inorder traversal.
  - Visit left subtree recursively.
  - Visit vertex.
  - Visit right subtree recursively.
- Preorder traversal.
  - Visit vertex.
  - Visit left subtree recursively.
  - Visit right subtree recursively.
- Postorder traversal.
  - Visit left subtree recursively.
  - Visit right subtree recursively.
  - Visit vertex.

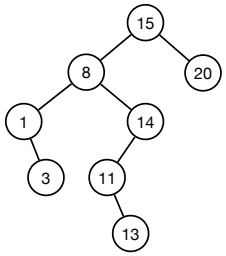
Inorder: 1, 3, 8, 11, 13, 14, 15, 20

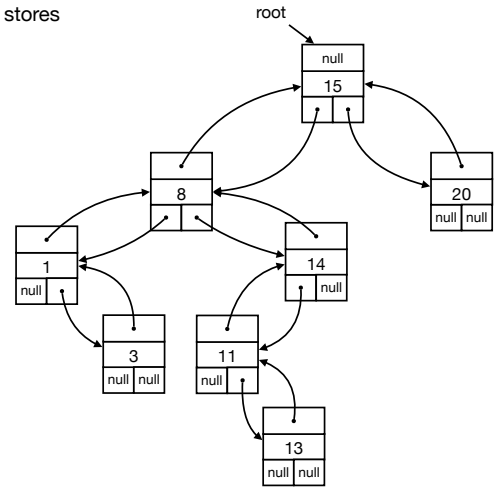
Preorder: 15, 8, 1, 3, 14, 11, 13, 20

Postorder: 3, 1, 13, 11, 14, 8, 20, 15

Binary Search Trees

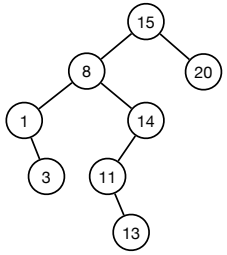
- Representation. Each node x stores
  - x.key
  - x.left
  - x.right
  - x.parent
  - (x.data)
- Space. O(n)





Binary Search Trees

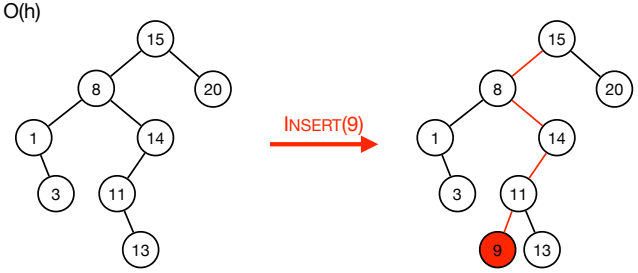
- SEARCH(k): traverse tree top-down.
  - Compare key k against key in node.
  - If equal return element. If less go left. If greater go right.
  - If we reach bottom, return null.
- Time. O(h)



Binary Search Trees

- INSERT(x): traverse tree top-down and compare keys.
  - search for x.
  - add x at leaf.

- Time. O(h)



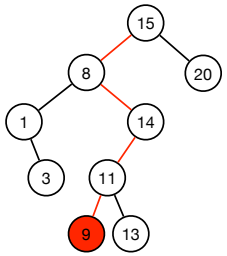
Binary Search Trees

- INSERT(x): traverse tree top-down and compare keys.
  - if less go left; if greater go right; if equal, return node.
  - if null, insert x.

- Exercise. Insert following sequence in binary search tree: 6, 14, 3, 8, 12, 9, 34, 1, 7

Binary Search Trees

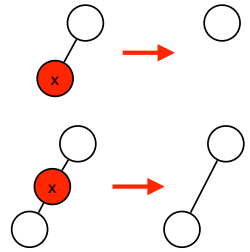
```
INSERT(x,v)
   if (v == null) return x
   if (x.key ≤ v.key)
      v.left = INSERT(x, v.left)
   if (x.key > v.key)
      v.right = INSERT(x, v.right)
```

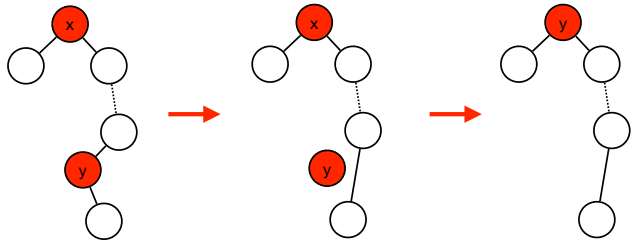


- Time. O(h)

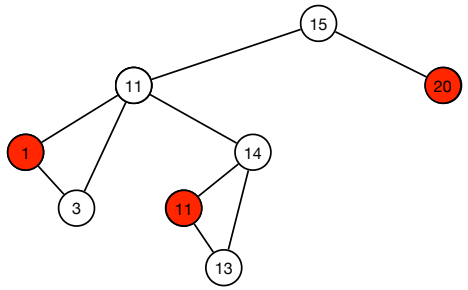
Binary Search Trees

- DELETE(x):
  - 0 children: remove x.
  - 1 child: splice x.
  - 2 children: find y = node with smallest key > x.key. Splice y and replace x by y.





DELETE 20 1 8



Binary Search Trees

- DELETE(x):
  - 0 children: remove x.
  - 1 child: splice x.
  - 2 children: find y = node with smallest key > x.key. Splice y and replace x by y.
- Time. O(h)

Dynamic Ordered Sets

Data structure	SEARCH	INSERT	DELETE	Space
linked list	O(n)	O(1)	O(1)	O(n)
sorted array	O(log n)	O(n)	O(n)	O(n)
binary search tree	O(h)	O(h)	O(h)	O(n)

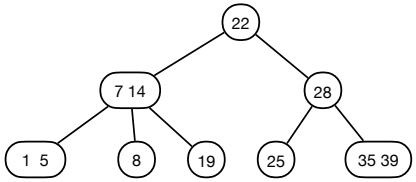
- Height. Depends on sequence of operations.
  - h = Ω(n) worst-case and h = Θ(log n) on average.
- Challenge. Can we maintain height at O(log n) worst-case?

Search Trees

- Dynamic Ordered Sets
- Binary Search Trees
- Balanced Search Trees

Balanced Search Trees

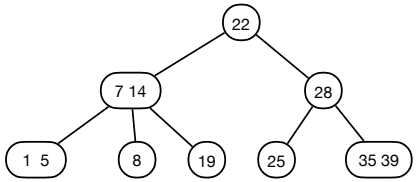
- 2-3 Tree.
  - Rooted tree.
  - Each internal node has 2 or 3 children.
    - 2-node: 2 children and 1 key
    - 3-node: 3 children and 2 keys.
  - Symmetric order.
    - Inorder traversal outputs the keys in sorted order.
  - Perfect balance.
    - Every path from root to a leaf has the same length
  - ⇒ height of tree is Θ(log n)



Balanced Search Trees

- SEARCH(k): traverse tree top-down.
  - Compare key k against keys in node.
  - If equal return element. Otherwise, recurse in child with interval containing k and recurse.
  - If we reach bottom, return null.

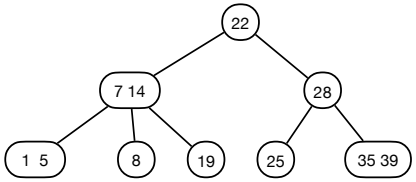
- Time. O(log n)



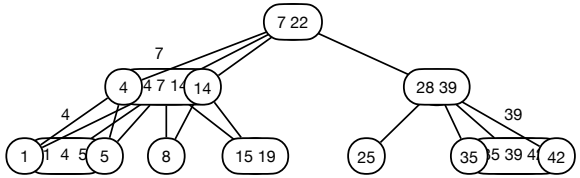
Balanced Search Trees

- INSERT(x):
  - Search for x.
  - Add x at leaf.
  - If too large, move middle key to parent. Repeat if necessary.

- Time. O(log n)



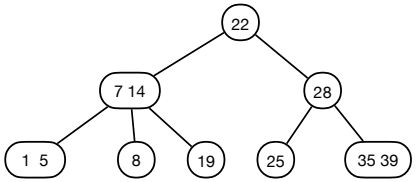
INSERT 15 42 4



Balanced Search Trees

- DELETE(x):
  - Search for x.
  - If x is a not a leaf, find node with smallest key > x.key, swap with x, and delete it.
  - If too small, take from parent. Repeat if necessary.

- Time. O(log n)



Dynamic Ordered Sets

Data structure	SEARCH	INSERT	DELETE	Space
linked list	O(n)	O(1)	O(1)	O(n)
sorted array	O(log n)	O(n)	O(n)	O(n)
binary search tree	O(h)	O(h)	O(h)	O(n)
2-3 tree	O(log n)	O(log n)	O(log n)	O(n)

- Can we do better?
  - Many variants of balanced search trees supporting many different operations.
  - Many efficient practical solutions.
  - Optimal time bounds for comparison-based data structures.
  - Even better bounds possible with more advanced techniques.

Search Trees

- Dynamic Ordered Sets
- Binary Search Trees
- Balanced Search Trees