CIS2520 Data Structures
Fall 2011, Lab 10

Mohammad Naeem
mnaeem@uoguelph.ca

topics

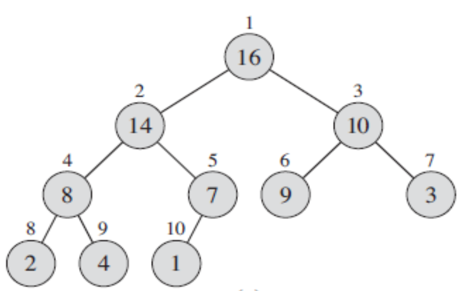
- heap sort
- quick sort
- Dijkstra's shortest path algorithm
- insertion sort

heap sort

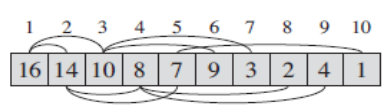
- $O(n \log n)$
- Sorts in place i.e., a constant number of array elements need to be stored outside the array
- introduces an algorithm design technique using a data structure
- based on **heap**

heap

- an array $A[\mathbf{1..}A.\boldsymbol{legth}]$
  - heap length $A.\boldsymbol{heap-length}$ ***(number of elements in array)***
  - heap size $A.\boldsymbol{heap-size}$ ***(number of valid elements)***
- can be viewed as nearly complete binary tree i.e., filled at all levels except possibly leaves
- **A[1]** is the root

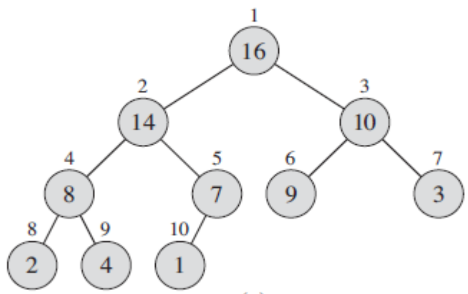


(a)



(b)

given the index i of a node, its left and right children can be easily found

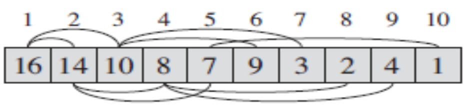


(a)

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i) ← **left child**
1 **return** $2i$

RIGHT(i) ← **right child**
1 **return** $2i + 1$



(b)

$2i$, $2i + 1$, and $i/2$ often built-in hardware so processing heap is faster

binary heap types

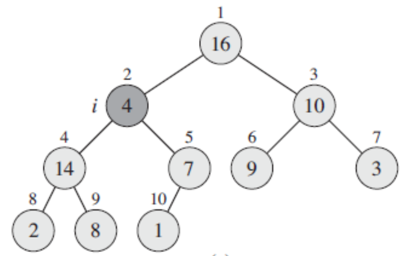
- max-heap
  -value of a node is at most the value of its parent
  -at root, the largest value
  $A[\text{PARENT}(i)] \geq A[i]$
- min heap
  at root, the smallest value
  $A[\text{PARENT}(i)] \leq A[i]$.

heap-property

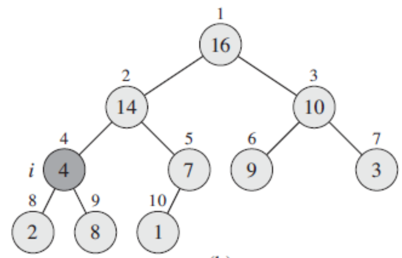
**for sorting,
max-heap used**

maintaining the heap property

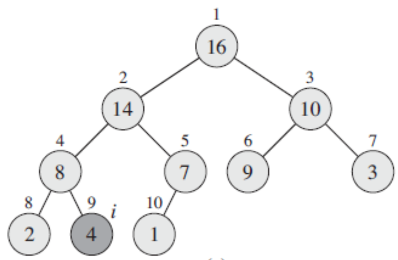
function Max_Heapify()
l = LEFT(i)
r=RIGHT(i)
if l $\leq$ A. heap $-$ size and $A[l] > A[i]$
largest = l
else largest = I
if $r \leq$ A. heap $-$ size and $A[r] > A[large]$
largest = r
if largest $\neq$ i
exchange $A[i]$ with $A[largest]$
Max-Heapify(A, largest)
return



(a)



(b)



(c)

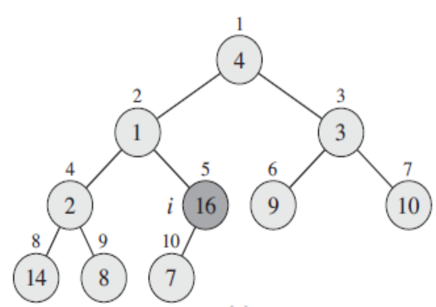
building a heap

- applies MAX-HEAPIFY in a bottom-up fashion
- converts an array $A[1..n]$ an into a heap, $n = A.length$

function Build-Max-Heapify(A)
A.heap-size = A.length
for $i = \lfloor A.length/2 \rfloor$ **downto** 1
Max-Heapify(A,i)
return

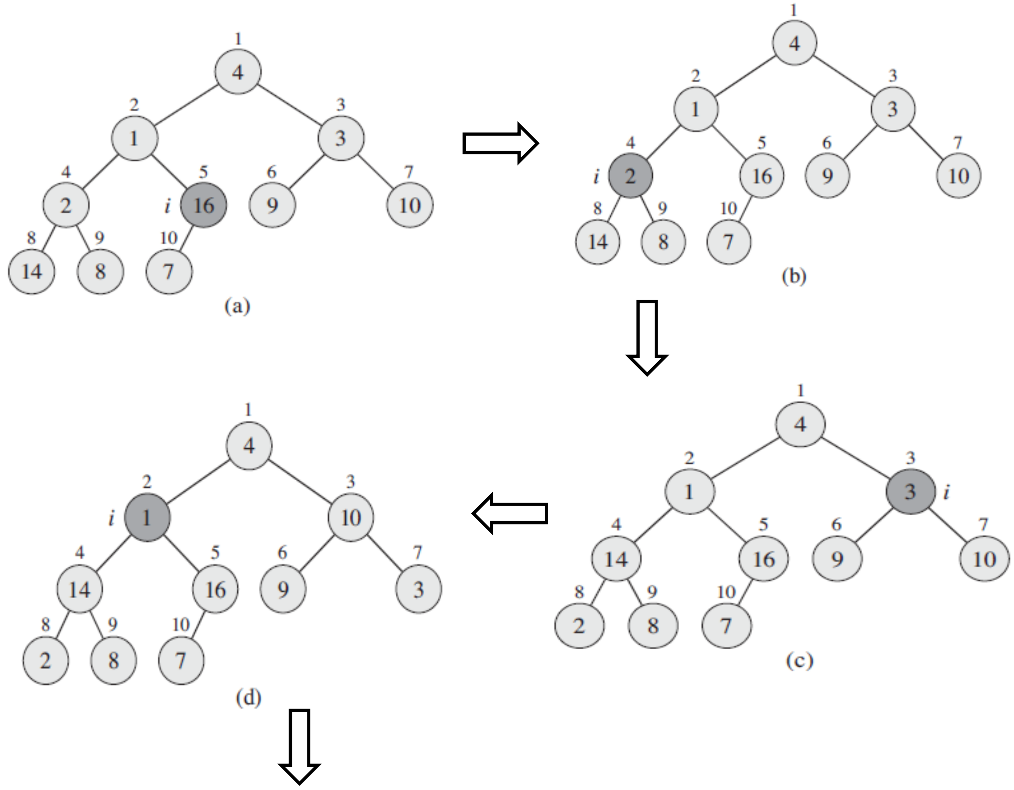
- example:

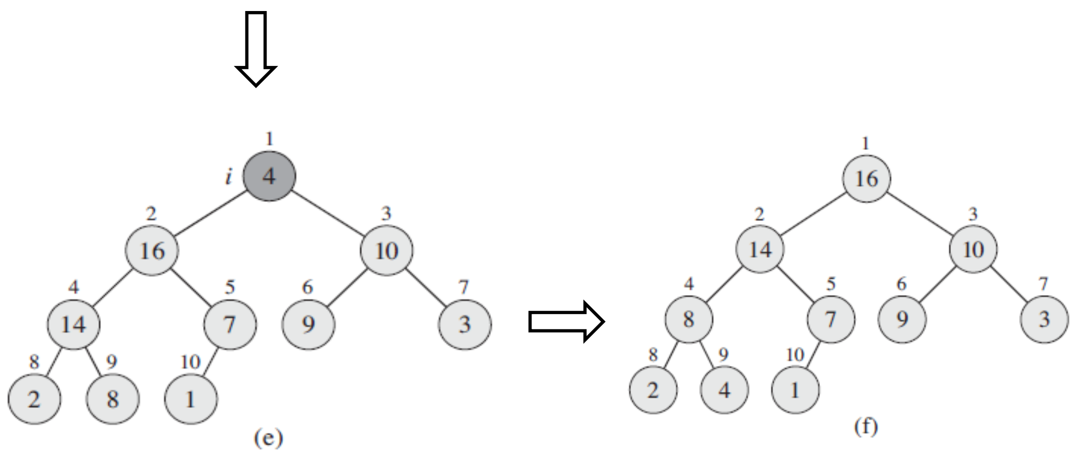
A	4	1	3	2	16	9	10	14	8	7



(a)

is $A[\text{PARENT}(i)] \geq A[i]$
satisfied for the heap ?


(a)
(b)
(c)
(d)


(e)
(f)

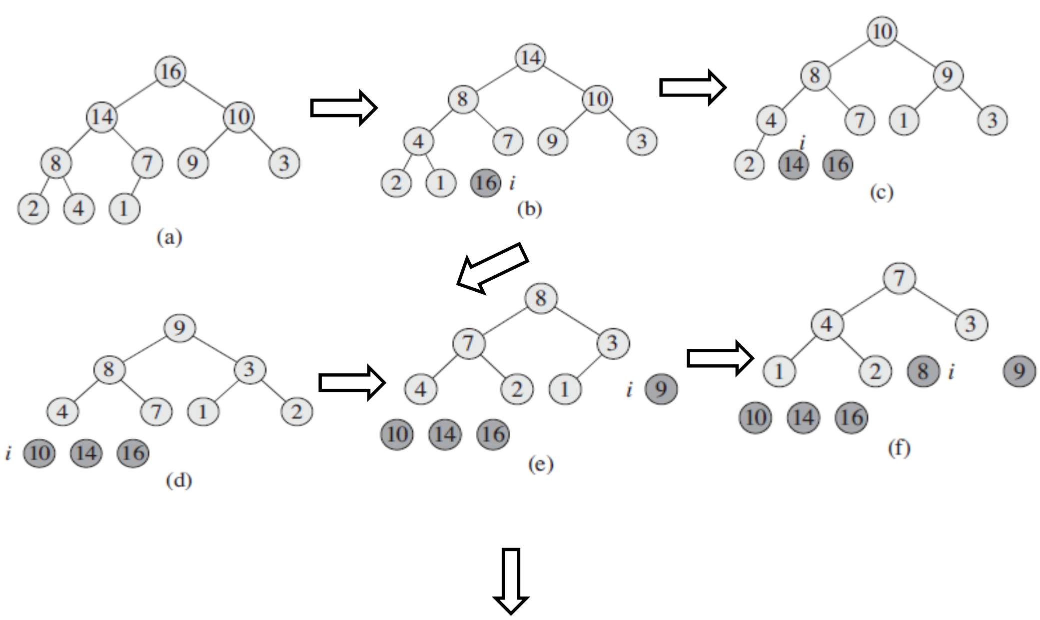
heap sort

converts the given array i.e., *A* into a heap

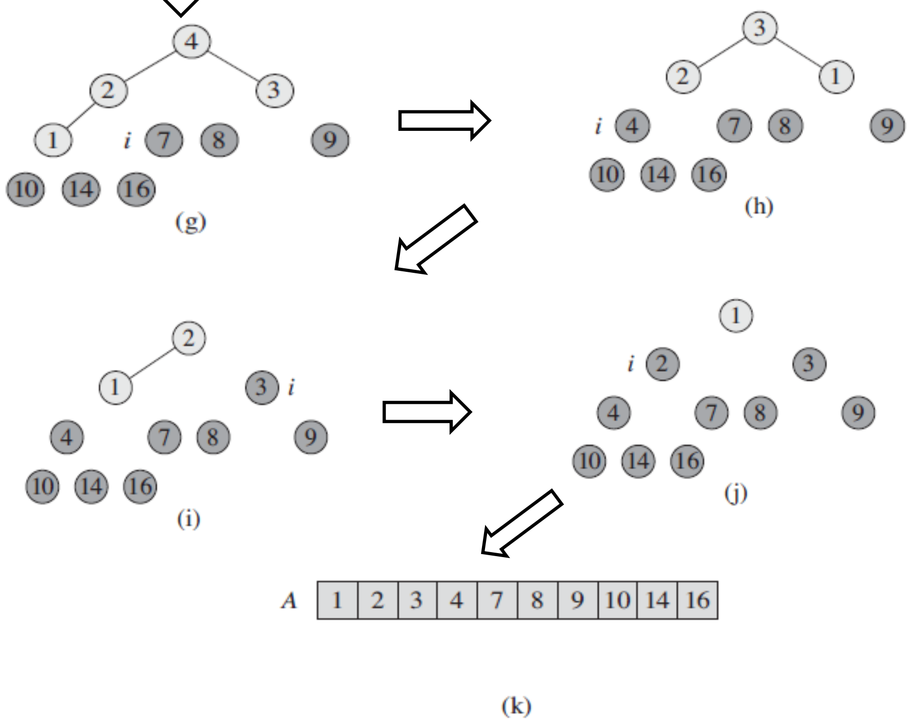
function Heapsort(A)
 Build-Max-Heapify(A)
 for i = A.length **downto** 2
  exchange A[1] with A[i]
  A.heap-size = A.heaps-ize -1
  Max-Heapify(A,1)
return

restore the max-heap property of the resulting heap

the heap sort



the heap sort



the heap sort algorithm

function Heapsort(A)

Build-Max-Heapify(A) } $O(n)$

for i = A.length **downto** 2
  exchange A[1] with A[i]
  A.heap-size = A.heaps-ize -1
  Max-Heapify(A,1)

} $O(\log n)$ } $O(n \log n)$

return

quick sort

- **divide-and-conquer** algorithm
- three step process:
  - **divide**
  - **conquer**
  - **combine**

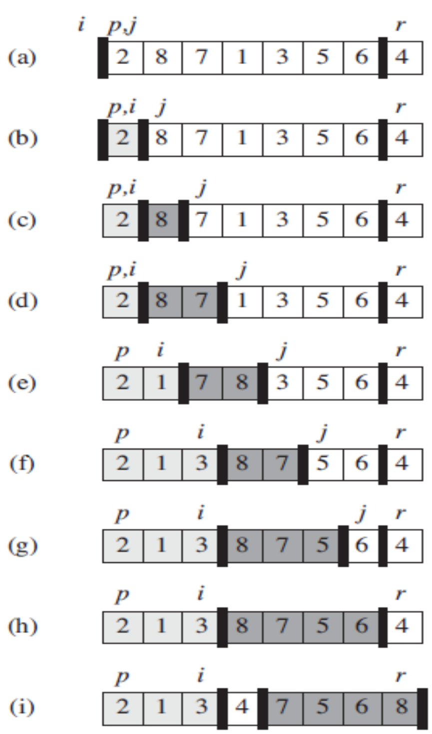
function Quicksort(A, p r)
 if p < r
  q = Partition (A, p, r)
  Quicksort(A, p, q-1)
  Quicksort(A, q+1, r)
 return

function Partition(A, p, r)

 x = A[r]
 l = p – 1

 for j = p **to** r – 1
  if $A[j] \leq x$
   $i = i + 1$
   exchange $A[i] with\ A[j]$

 exchange $A[i+1] with\ A[r]$
 return $i + 1$



initial array $A[p..r]$

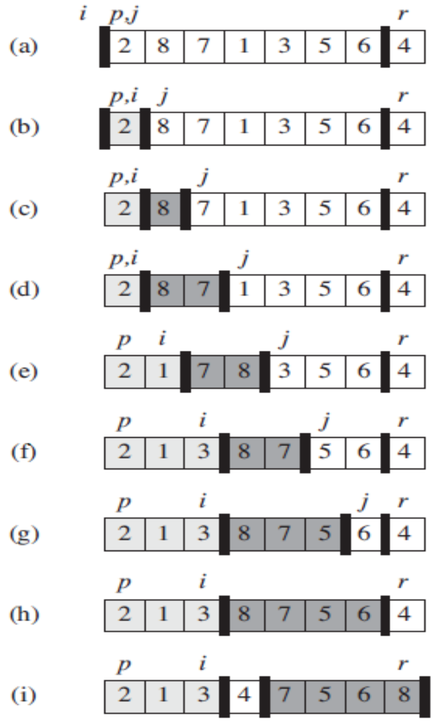
function Partition(A, p, r)

 x = A[r] ← **pivot element (4)**
 l = p – 1

 for j = p **to** r – 1
  if $A[j] \leq x$
   $i = i + 1$
   exchange $A[i] with\ A[j]$

 exchange $A[i+1] with\ A[r]$
 return $i + 1$

function Quicksort(A, p r)
 if p < r
  q = Partition (A, p, r)
  Quicksort(A, p, q-1)
  Quicksort(A, q+1, r)
 return

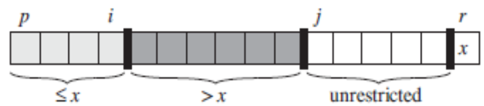


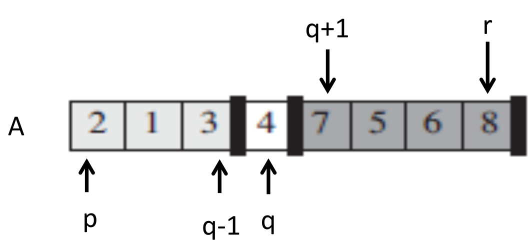
function Partition(A, p, r)

x = A[r]
I = p – 1

for j = p **to** r – 1
if $A[j] \leq x$
$i = i + 1$
exchange $A[i] with\ A[j]$

exchange $A[i+1] with\ A[r]$
return $i + 1$





function Partition(A, p, r)

x = A[r]
I = p – 1

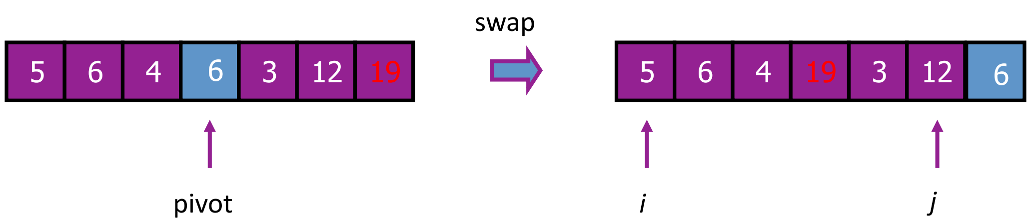
for j = p **to** r – 1
if $A[j] \leq x$
$i = i + 1$
exchange $A[i] with\ A[j]$

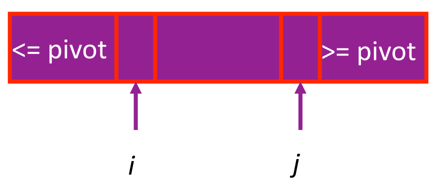
exchange $A[i+1] with\ A[r]$
return $i + 1$

function Quicksort(A, p r)
if p < r
q = Partition (A, p, r)
Quicksort(A, p, q-1)
Quicksort(A, q+1, r)
return

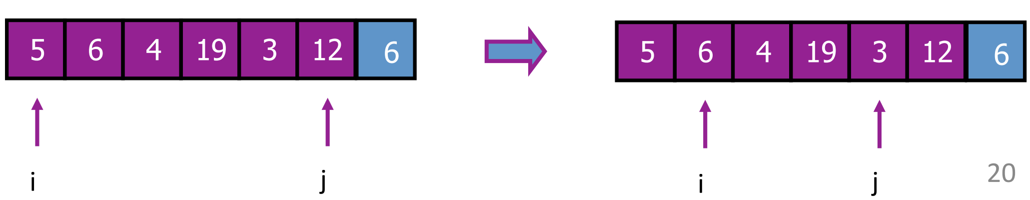
A better partition

- want to partition an array A[left .. right]
- first, get the pivot element out of the way by swapping it with the last element. (Swap pivot and A[right])
- let i start at the first element and j start at the next-to-last element (i = left, j = right – 1)

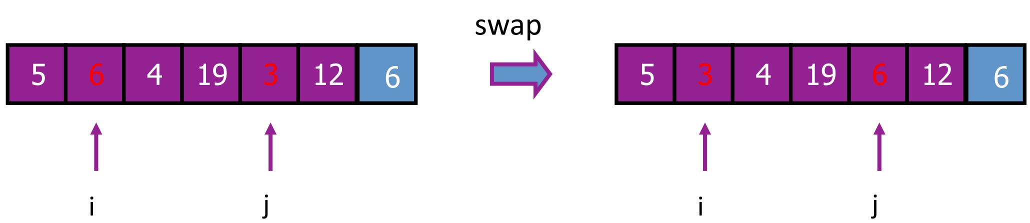




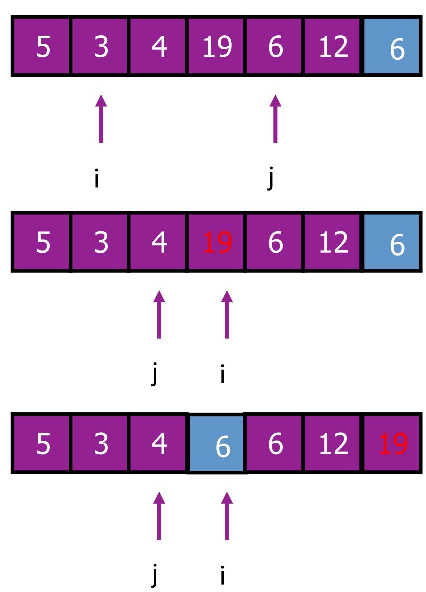
- want to have
  - A[x] <= pivot, for x < i
  - A[x] >= pivot, for x > j
- when i < j
  - Move i right, skipping over elements smaller than the pivot
  - Move j left, skipping over elements greater than the pivot
  - When both i and j have stopped
    - A[i] >= pivot
    - A[j] <= pivot



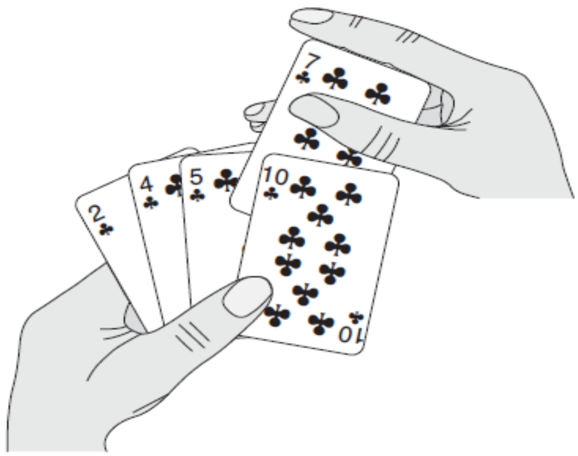
- when i and j have stopped and i is to the left of j
  - swap A[i] and A[j]
    - The large element is pushed to the right and the small element is pushed to the left
  - after swapping
    - A[i] <= pivot
    - A[j] >= pivot
  - repeat the process until i and j cross



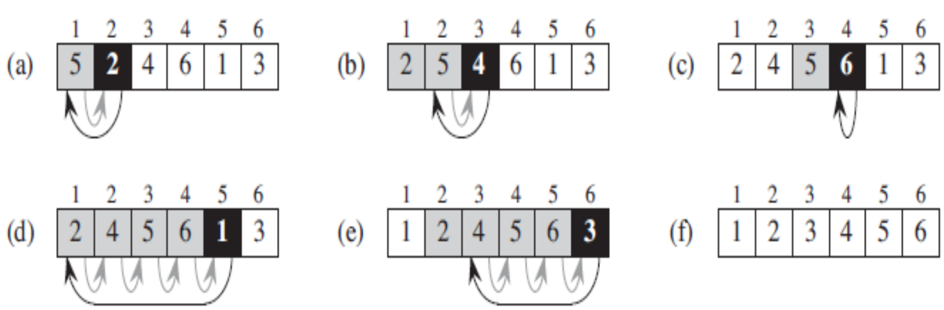
- when i and j have crossed
  - swap A[i] and pivot
- result:
  - A[x] <= pivot, for x < i
  - A[x] >= pivot, for x > i



insertion sort



insertion sort



insertion sort

```
function Insertion-Sort(A)

  for j = 2 to A.length
   key = A[j]
   //Insert A[j] into the sorted sequence A[1..j – 1]
   i = j − 1

   while i > 0 and A[i] > key
     A[i + 1] = A[i]
     i = i − 1

   A[i + 1] = key

return
```

move back through the list until an appropriate position is found for key

then insert key at its right position

shortest path problem in graphs

- given a weighted directed graph

  $G(V, E)$, $w: E \rightarrow \mathbb{R}$ mapping edges into real-valued weights

- length $w(p)$ of path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

weight $\delta(u, v)$ of the shortest path between u and v

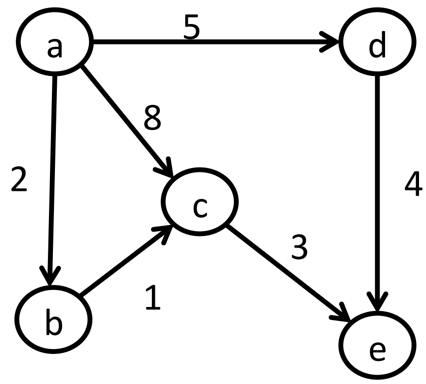
$$\delta(u, v) = \begin{cases} \min\{w(p): u \rightarrow^{p} v\} & \text{if a path exists between u and v} \\ \infty & \text{otherwise} \end{cases}$$

shortest path problem

$\delta(a, e) = w(p): p = \langle a, b, c, e \rangle$

$= w(a, b) + w(b, c) + w(c, e)$

$= 2 + 1 + 3 = 6$



Dijkstra’s shortest path algorithm

○ single destination shortest path
○ single-pair shortest path
○ all-pairs shortest path

Dijkstra shortest-path algorithm

function Initialize-Single-Source(G,s)
 for each vertex v ∈ $G.V$
  v.d = ∞
  v.$\pi = NIL$
 s.d = 0
return

source

graph

Initially, the path lengths from the source node to other nodes are set to ∞

function Dijkstra (G, w, s)
 Initialize-Single-Source(G, s)
 S = ∅
 Q = G.v
 while Q ≠ ∅
  u = Extract-Min(Q)
  S = S U { u}
  for each vertex v ∈ $G.Adj[u]$
   Relax (u, v, w)
return

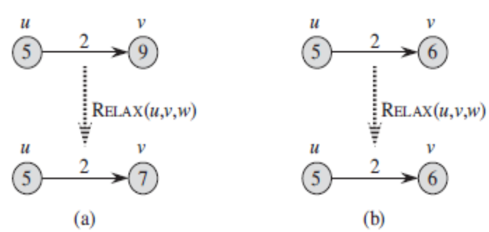
preceding node during the search

nodes maintained in a priority queue to optimize performance

function Relax (u, v, w)

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

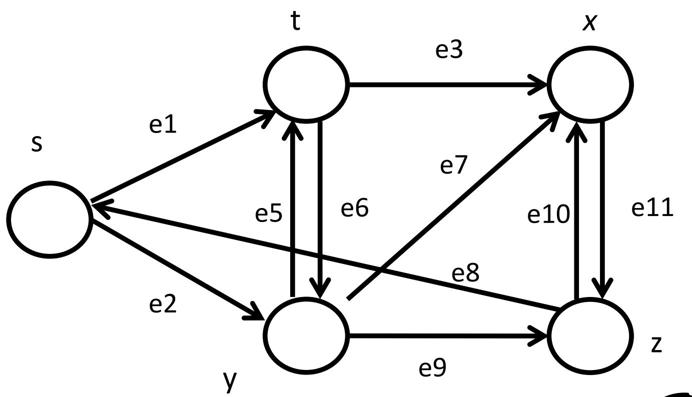
return

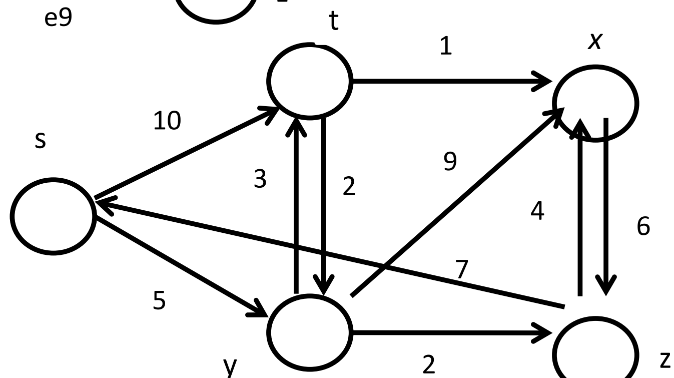


replaces the length of the path to *v* via some other node with length of the path to it via *u* if greater

set *u* as the preceding node of *v*

G= (V, E) = ({s, t, x, y, z}, {e1, e2, e3, e4, e5, e6, e7, e8, e9, e10})



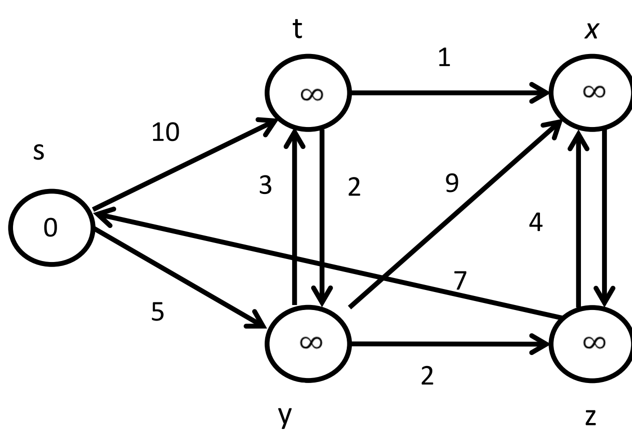


step 1: distance assignment

nodes assigned distance values:

-0 to initial node, s

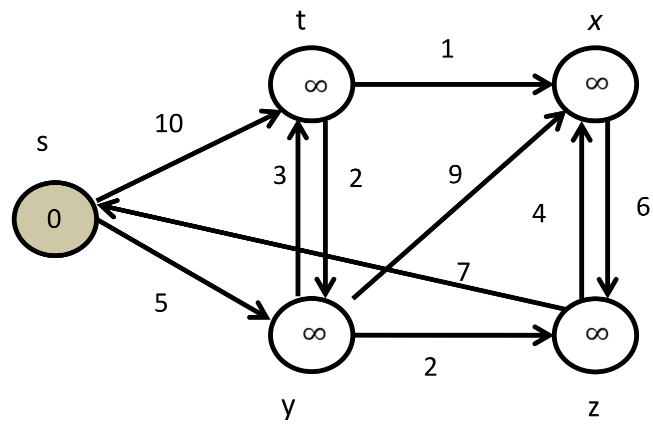
-∞ to all other nodes



Dijkstra's Algorithm

step 2: partitioning into visited-unvisited lists

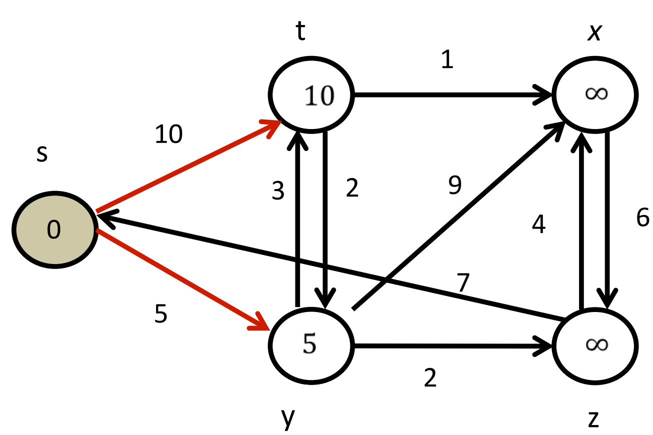
-set of visited nodes S = {}
- current node s
-set un-visited nodes = *Q* = {s, t, x, y, z}



Dijkstra's Algorithm

step 3: assign tentative distances to neighbours (of current)

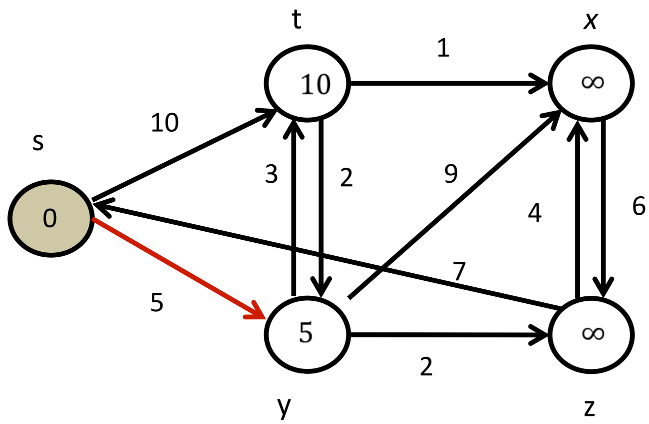
-set of visited nodes S = {}
- current node s
-set un-visited nodes = Q = {s, t, x, y, z}



Dijkstra's Algorithm

step 4: mark the current node as visited &
remove it from un-visited, put it into visited

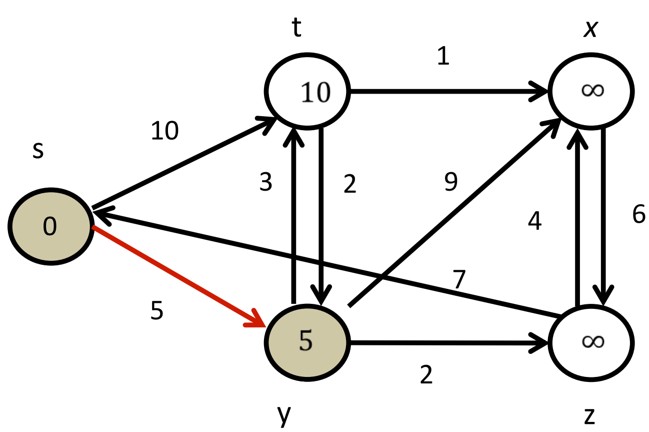
S = {s}
Q = { t, x, y, z}



Dijkstra's Algorithm

step 5: next current node with the lowest tentative
distance from the present current node

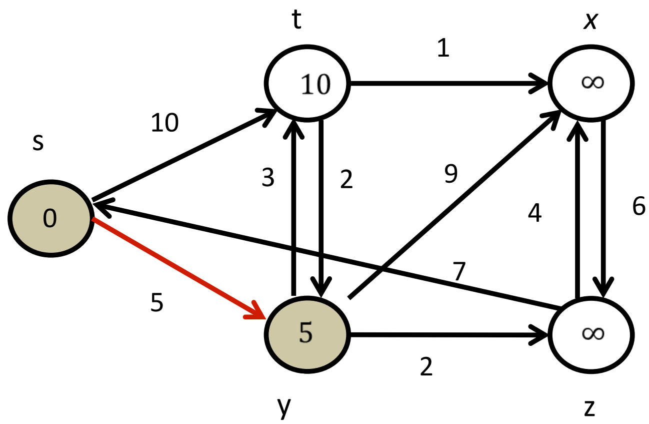
S = {s}
current : **y**
Q= { t, x, **y**, z}



Dijkstra’s Algorithm

step 6: stop if the current node is the destination node (stop if Q empty for source to all node case), otherwise go to step 3

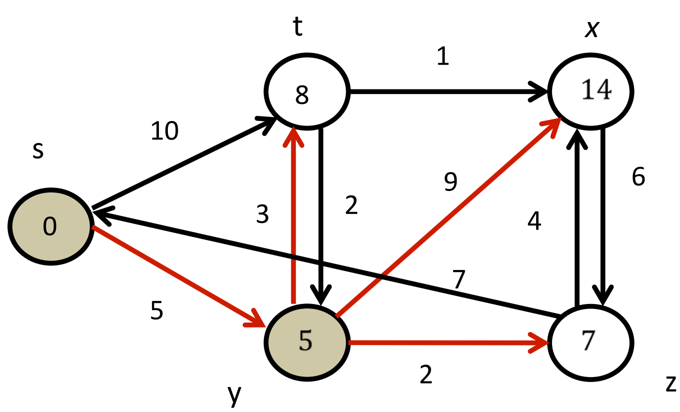
S = {s}
current : **y**
Q = { t, x, **y**, z}



Dijkstra’s Algorithm

step 3: assign tentative distances to neighbours (of current)

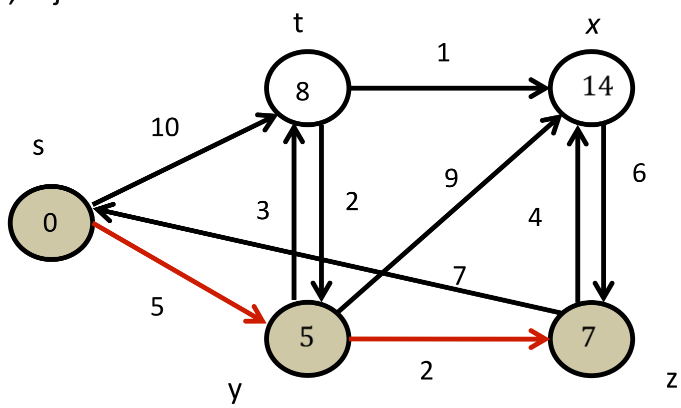
-set of visited nodes S = {s}
- current node **y**
-set un-visited nodes = *Q* = {t, x, **y**, z}



Dijkstra's Algorithm

step 4: next current node: node with lowest distance
from present current node

S = {s, y}
current : **z**
Q = { t, x, **z**}



Dijkstra shortest-path algo for given source-destination pair
(pseudo-code notation slightly different)

function Dijkstra(Graph, source, target)

for each vertex v in Graph
    dist[v] = infinity
    previous[v] = undefined

initialization

dist[source] = 0
Q = the set of all nodes in Graph
while Q is not empty
    u = vertex in Q with smallest distance in dist[]
    if u = target return target
     if dist[u] = infinity
       break

return if target found

```
remove u from Q
for each neighbor v of u
alt  =  dist[u] + dist_between(u, v)

if alt < dist[v]
  dist[v] = alt
  previous[v] = u
  decrease-key v in Q
```

Relax

```
return dist[]
```


End