Breadth First Search/Dijkstra.
CS 170: Algorithms
CS 170: Algorithms

Breadth First Search/Dijkstra.
CS 170: Algorithms

Breadth First Search/Dijkstra.
CS 170: Algorithms

Breadth First Search/Dijkstra.
Breadth First Search/Dijkstra.
Breadth First Search/Dijkstra.
Breadth First Search/Dijkstra.
Lecture in a minute.

Breadth First Search of graph:
Search with queue instead of stack.
Get "distances" from source.
Proof idea: queue has distance 0, then level 1, ...

Dijkstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
Idea: ignores "new" nodes.

Runtime:
\[ V \text{ extracts,} \quad E \text{ reduce-key for p-queue.} \]

Priority Queue:
Implementation: degree \( d \) tree.

Heap Property: children larger than parent.
Minimum at top.
Remove min:
\[ O \left( d \log_2 d \right) \text{ time: Replace min/percolate down.} \]
Reduce Key:
\[ O \left( \log_2 d \right) \text{ : percolate up.} \]
Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
   Proof idea: queue has distance 0, then level 1, …
Lecture in a minute.

Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
    Proof idea: queue has distance 0, then level 1, …

Dijkstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
    Idea: ignores “new” nodes.
Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.
Lecture in a minute.

Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
Proof idea: queue has distance 0, then level 1, …

Dijkstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
Idea: ignores “new” nodes.
Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
Implementation: degree $d$ tree.
Lecture in a minute.

Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
Proof idea: queue has distance 0, then level 1, …

Dijkstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
Idea: ignores “new” nodes.
Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
Implementation: degree $d$ tree.
Heap Property: children larger than parent.
Breadth First Search of graph:
    Search with queue instead of stack.
    Get “distances” from source.
    Proof idea: queue has distance 0, then level 1, …

Dijkstra: shortest paths in weighted graph:
    Replace weights by paths + BFS
    Implement using priority queue.
    Idea: ignores “new” nodes.
    Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
    Implementation: degree $d$ tree.
    Heap Property: children larger than parent.
    Minimum at top.
Lecture in a minute.

Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
Proof idea: queue has distance 0, then level 1, …

Dijkstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
Idea: ignores “new” nodes.
Runtime: \(|V|\) extracts, \(|E|\) reduce-key for p-queue.

Priority Queue:
Implementation: degree \(d\) tree.
Heap Property: children larger than parent.
Minimum at top.
Remove min: \(O(d \log_d n)\) time: Replace min/percolate down.
Lecture in a minute.

Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
Proof idea: queue has distance 0, then level 1, ...

Dijkstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
Idea: ignores “new” nodes.
Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
Implementation: degree $d$ tree.
Heap Property: children larger than parent.
Minimum at top.
Remove min: $O(d \log_d n)$ time: Replace min/percolate down.
Reduce Key: $O(\log_d n)$: percolate up.
Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
Proof idea: queue has distance 0, then level 1, …

Dijkstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
Idea: ignores “new” nodes.
Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
Implementation: degree $d$ tree.
Heap Property: children larger than parent.
Minimum at top.
Remove min: $O(d \log_d n)$ time: Replace min/percolate down.
Reduce Key: $O(\log_d n)$: percolate up.
Binary Heap.

Heap$^1$: bigger children.

$\text{values only}$
Binary Heap.

Heap\(^1\): bigger children.  
\[\rightarrow\] smallest at root.

---

\(^1\)values only
Binary Heap.

Heap\(^1\): bigger children.  
\[\Rightarrow\] smallest at root.

Insert(7):

\(^1\)values only
Binary Heap.

Heap\(^1\): bigger children. ⇒ smallest at root.

![Binary Heap Diagram]

Insert(7): Bubble up: check parent.

\(^1\)values only
Binary Heap.

Heap\(^1\): bigger children.  
⇒ smallest at root.

Insert(7): Bubble up: check parent.  depth comp.

\(^1\)values only
Binary Heap.

Heap\(^1\): bigger children.  
\[\implies\] smallest at root.

Insert(7): Bubble up: check parent.  .  depth comp.
DeleteMin:

\(^1\)values only
Binary Heap.

Heap\(^1\): bigger children.
⇒ smallest at root.

Insert(7): Bubble up: check parent. depth \(\text{comp.}\) comp.
DeleteMin:
Binary Heap.

Heap\(^1\): bigger children.  
⇒ smallest at root.

Insert(7): Bubble up: check parent.  .  depth comp.  
DeleteMin: Replace.

\(^1\)values only
Binary Heap.

Heap\(^1\): bigger children. 
\[\rightarrow\] smallest at root.

Insert(7): Bubble up: check parent. \[\text{depth}\] comp.
DeleteMin: Replace. Bubble down: check both children.

\(^1\)values only
Binary Heap.

Heap\(^1\): bigger children.
\[\implies\] smallest at root.

Insert(7): Bubble up: check parent. \(\cdot\) depth comp.
DeleteMin: Replace. Bubble down: check both children.. \(2 \times \text{depth}\) – comparisons.

\(^1\)values only
$d$-ary heap

Degree – $d$, Depth – $\log_d n$. 
**d-ary heap**

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$. 
$d$-ary heap

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)
$d$-ary heap

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
**d-ary heap**

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
**d-ary heap**

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.
\textit{d-ary heap}

Degree – \(d\), Depth – \(\log_d n\).
Insert/DecreaseKey – \(\log n / \log d\).
DeleteMin – \(d \log n / \log d\). (Check all children.)

Dijkstra:
\(O(|V|)\) deletemins. \(O(d \log n / \log d)\) each.
\(O(|E|)\) insert/decrease-keys. \(O(\log n / \log d)\) each.

\(O(|V|d \log n / \log d + |E| \log n / \log d)\).
\[d\text{-ary heap}\]

Degree – \(d\), Depth – \(\log_d n\).
Insert/DecreaseKey – \(\log n / \log d\).
DeleteMin – \(d \log n / \log d\). (Check all children.)

Dijkstra:
\(O(|V|)\) deletemins. \(O(d \log n / \log d)\) each.
\(O(|E|)\) insert/decrease-keys. \(O(\log n / \log d)\) each.

\(O(|V|d \log n / \log d + |E| \log n / \log d)\).

Optimal Choice:
**d-ary heap**

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V|d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E| / |V|$ (average degree/2)
**d-ary heap**

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V|d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E| / |V|$ (average degree/2)
$O(|E| \log n / \log d)$
$d$-ary heap

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V|d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.
**d-ary heap**

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E| / |V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$d$-ary heap

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V|d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E| / |V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$O(\log n)$ per delete.
$d$-ary heap

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.
$O(|V|d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$O(\log n)$ per delete.
$O(1)$ average decrease-key.
$d$-ary heap

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V|d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E| / |V| \text{ (average degree/2)}$
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$O(\log n)$ per delete.
$O(1)$ average decrease-key.

$O(|V| \log |V| + |E|)$. 
**d-ary heap**

Degree – $d$, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V|\log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/\|V\|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$O(\log n)$ per delete.
$O(1)$ average decrease-key.

$O(|V| \log |V| + |E|)$.
Linear for moderately dense graphs!
Lecture in a minute.

Breadth First Search of graph:
Search with queue instead of stack.
Get "distances" from source.
Proof idea: queue has distance 0, then level 1, ...

Dijkstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
Idea: ignores "new" nodes.

Runtime:
\[ V \] extracts, \[ E \] reduce-key for p-queue.

Priority Queue:
Implementation: degree \( d \) tree.

Heap Property: children larger than parent.
Minimum at top.
Remove min:
\[ O(d \log d \cdot n) \] time: Replace min/percolate down.
Reduce Key:
\[ O(\log d \cdot n) \]: percolate up.
Lecture in a minute.

Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
   Proof idea: queue has distance 0, then level 1, …
Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
   Proof idea: queue has distance 0, then level 1, . . .

Djikstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
   Idea: ignores “new” nodes.
Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.
Lecture in a minute.

Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
Proof idea: queue has distance 0, then level 1, …

Dijkstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
Idea: ignores “new” nodes.
Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
Implementation: degree $d$ tree.
Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
   Proof idea: queue has distance 0, then level 1, …

Djikstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
   Idea: ignores “new” nodes.
Runtime: \(|V|\) extracts, \(|E|\) reduce-key for p-queue.

Priority Queue:
Implementation: degree \(d\) tree.
   Heap Property: children larger than parent.
Lecture in a minute.

Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
  Proof idea: queue has distance 0, then level 1, …

Dijkstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
  Idea: ignores “new” nodes.
Runtime: \(|V|\) extracts, \(|E|\) reduce-key for p-queue.

Priority Queue:
Implementation: degree \(d\) tree.
  Heap Property: children larger than parent.
    Minimum at top.
Lecture in a minute.

Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
Proof idea: queue has distance 0, then level 1, ...

Dijkstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
Idea: ignores “new” nodes.
Runtime: \(|V|\) extracts, \(|E|\) reduce-key for p-queue.

Priority Queue:
Implementation: degree \(d\) tree.
Heap Property: children larger than parent.
Minimum at top.
Remove min: \(O(d \log_d n)\) time: Replace min/percolate down.
Breadth First Search of graph:
  Search with queue instead of stack.
  Get “distances” from source.
  Proof idea: queue has distance 0, then level 1, ...

Dijkstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
  Idea: ignores “new” nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree $d$ tree.
  Heap Property: children larger than parent.
    Minimum at top.
  Remove min: $O(d \log_d n)$ time: Replace min/percolate down.
  Reduce Key: $O(\log_d n)$: percolate up.
Breadth First Search of graph:
Search with queue instead of stack.
Get "distances" from source.
Proof idea: queue has distance 0, then level 1, ...

Dijkstra: shortest paths in weighted graph:
Replace weights by paths + BFS
Implement using priority queue.
Idea: ignores "new" nodes.
Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
Implementation: degree $d$ tree.
Heap Property: children larger than parent.
   Minimum at top.
Remove min: $O(d \log_d n)$ time: Replace min/percolate down.
Reduce Key: $O(\log_d n)$: percolate up.