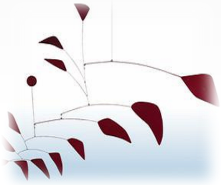
EECS 4101-5101
Advanced Data Structures



Shahin Kamali

Topic 4c Priority Queue Applications

York University

Picture is from the cover of the textbook CLRS.

Today's Class

- We review applications of priority queues (in particular, the Fibonacci Heap implementation).
  - Discrete event simulation
  - Dijkstra's shortest-path algorithm.
  - Huffman Encoding

Discrete-event Simulation

- Discrete-event Simulation is used to understand the behavior of **systems**, i.e., a collection of entities (e.g., people and machines) that interact over time.

Discrete-event Simulation

- Discrete-event Simulation is used to understand the behavior of **systems**, i.e., a collection of entities (e.g., people and machines) that interact over time.
- **Example 1:** A natural scientist asks whether the number of wolves and the number of sheep in a terrain (system) stabilize in the long run, and if so to what values.

Discrete-event Simulation

- Discrete-event Simulation is used to understand the behavior of **systems**, i.e., a collection of entities (e.g., people and machines) that interact over time.
- **Example 1:** A natural scientist asks whether the number of wolves and the number of sheep in a terrain (system) stabilize in the long run, and if so to what values.
    - The number of wolves changes with a constant birth rate and a death rate that is inversely proportional to the number of sheep.

Discrete-event Simulation

- Discrete-event Simulation is used to understand the behavior of **systems**, i.e., a collection of entities (e.g., people and machines) that interact over time.
- **Example 1:** A natural scientist asks whether the number of wolves and the number of sheep in a terrain (system) stabilize in the long run, and if so to what values.
  - The number of wolves changes with a constant birth rate and a death rate that is inversely proportional to the number of sheep.
  - The number of sheep changes with a constant birth rate and a death rate that is directly proportional to the number of wolves.

Discrete-event Simulation

- Discrete-event Simulation is used to understand the behavior of **systems**, i.e., a collection of entities (e.g., people and machines) that interact over time.
- **Example 1:** A natural scientist asks whether the number of wolves and the number of sheep in a terrain (system) stabilize in the long run, and if so to what values.
  - The number of wolves changes with a constant birth rate and a death rate that is inversely proportional to the number of sheep.
  - The number of sheep changes with a constant birth rate and a death rate that is directly proportional to the number of wolves.
  - Each **event** is birth/death of a wolf/sheep.

Discrete-event Simulation

- Discrete-event Simulation is used to understand the behavior of **systems**, i.e., a collection of entities (e.g., people and machines) that interact over time.

Discrete-event Simulation

- Discrete-event Simulation is used to understand the behavior of **systems**, i.e., a collection of entities (e.g., people and machines) that interact over time.
- **Example 2:** A computer scientist would like to know whether a particular server is a "bottleneck" in a system of jobs that circulate in a network of servers (e.g., CPU's and I/O devices).

Discrete-event Simulation

- Discrete-event Simulation is used to understand the behavior of **systems**, i.e., a collection of entities (e.g., people and machines) that interact over time.
- **Example 2:** A computer scientist would like to know whether a particular server is a "bottleneck" in a system of jobs that circulate in a network of servers (e.g., CPU's and I/O devices).
  - Whether a server always is busy while the other servers are mostly idle.

Discrete-event Simulation

- Discrete-event Simulation is used to understand the behavior of **systems**, i.e., a collection of entities (e.g., people and machines) that interact over time.
- **Example 2:** A computer scientist would like to know whether a particular server is a "bottleneck" in a system of jobs that circulate in a network of servers (e.g., CPU's and I/O devices).
  - Whether a server always is busy while the other servers are mostly idle.
  - Each **event** is start/end of a job.

Discrete-event Simulation

- To simulate discrete event systems, we note each event has a discrete start time, known as **simulation time**.
- The events are added to a priority queue with their simulation time used as the priority.
- The execution of the simulation proceeds by repeatedly pulling the top of the queue and executing the event thereon.

Discrete-event Simulation

- To simulate discrete event systems, we note each event has a discrete start time, known as **simulation time**.
- The events are added to a priority queue with their simulation time used as the priority.
- The execution of the simulation proceeds by repeatedly pulling the top of the queue and executing the event thereon.
- What is the time complexity of the entire simulation with n events?
  - **Binary/binomial heaps:** n events are added for a total cost of $O(n \log n)$ and extracted, again for a total cost of $O(n \log n)$, summing to $O(n \log n)$.

Discrete-event Simulation

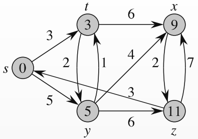
- To simulate discrete event systems, we note each event has a discrete start time, known as **simulation time**.
- The events are added to a priority queue with their simulation time used as the priority.
- The execution of the simulation proceeds by repeatedly pulling the top of the queue and executing the event thereon.
- What is the time complexity of the entire simulation with n events?
  - **Binary/binomial heaps:** n events are added for a total cost of $O(n \log n)$ and extracted, again for a total cost of $O(n \log n)$, summing to $O(n \log n)$.
  - **Fibonacci heaps:** n events are added for a total cost of n and extracted for a total cost of $O(n \log n)$, summing to $O(n \log n)$ for the entire simulation

Discrete-event Simulation

- To simulate discrete event systems, we note each event has a discrete start time, known as **simulation time**.
- The events are added to a priority queue with their simulation time used as the priority.
- The execution of the simulation proceeds by repeatedly pulling the top of the queue and executing the event thereon.
- What is the time complexity of the entire simulation with n events?
  - **Binary/binomial heaps:** n events are added for a total cost of $O(n \log n)$ and extracted, again for a total cost of $O(n \log n)$, summing to $O(n \log n)$.
  - **Fibonacci heaps:** n events are added for a total cost of n and extracted for a total cost of $O(n \log n)$, summing to $O(n \log n)$ for the entire simulation
- **The selection of the heap type does not change the time complexity for discrete-time simulation.**

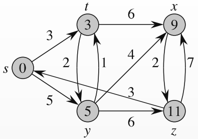
Single-source Shortest Path

- In a **shortest-paths problem**, we are given a weighted, directed graph $G = (V, E)$, with n vertices and m edges, and with weight function w mapping edges to real-valued weights.
- The weight $w(p)$ of a path is the sum of the weights of its edges.
- In the single-source shortest path problem, we want to find a shortest path from a **given source** vertex $s \in V$ to each vertex $u \in V$.
  - The output is stored in a shortest path tree.



Single-source Shortest Path

- In a **shortest-paths problem**, we are given a weighted, directed graph $G = (V, E)$, with n vertices and m edges, and with weight function w mapping edges to real-valued weights.
- The weight $w(p)$ of a path is the sum of the weights of its edges.
- In the single-source shortest path problem, we want to find a shortest path from a **given source** vertex $s \in V$ to each vertex $u \in V$.
    - The output is stored in a shortest path tree.
    - If negative weights are allowed, we use slower Bellman-Ford algorithm, which runs in $\Theta(mn)$; otherwise, we use the faster Dijkstra's algorithm.

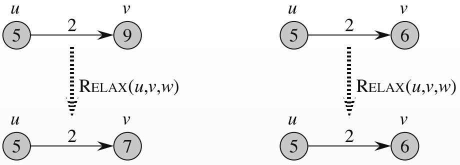


Relaxation

- We use a **Relax** procedure which takes and edge (u, v) and tests whether we can improve the shortest path to v found so far by going through u and, if so, updating $v.d$ (the estimated distance) and $v.\pi$ (the parent of v).

RELAX(u, v, w)

$$
\begin{array}{l}
\textbf{if } v.d > u.d + w(u,v) \\
\qquad v.d = u.d + w(u,v) \\
\qquad v.\pi = u
\end{array}
$$



(a) (b)

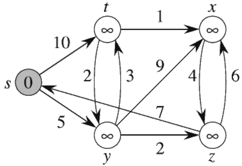
Dijkstra's Algorithm

- Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined.
  - The algorithm repeatedly I) selects the vertex $u \in V - S$ with the minimum shortest-path estimate, II) adds u to S, and III) relaxes all edges leaving u.
  - We use a min-priority queue Q of vertices, keyed by their estimate d values.

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Dijkstra's Exmaple

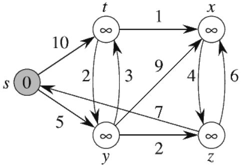
- Initially, $Q = G.V$, $S = \phi$, and $s.d = 0$ and $v.d = \infty$ for any $v \neq s$.
- Repeatedly take the vertex u with smallest estimate, add it to S, and relax edges leaving u.



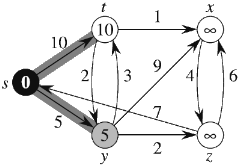
(a)

Dijkstra's Exmaple

- Initially, $Q = G.V$, $S = \phi$, and $s.d = 0$ and $v.d = \infty$ for any $v \neq s$.
- Repeatedly take the vertex u with smallest estimate, add it to S, and relax edges leaving u.



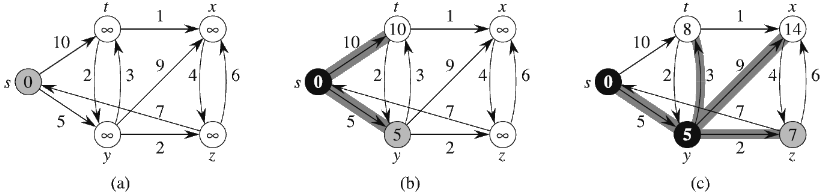
(a)



(b)

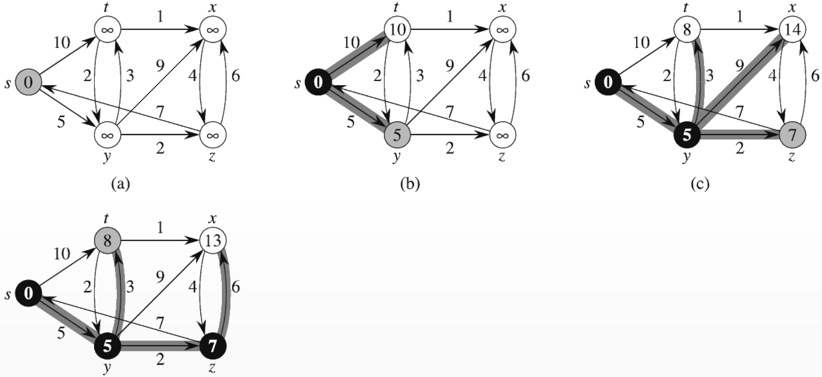
Dijkstra's Exmaple

- Initially, $Q = G.V$, $S = \phi$, and $s.d = 0$ and $v.d = \infty$ for any $v \neq s$.
- Repeatedly take the vertex u with smallest estimate, add it to S, and relax edges leaving u.



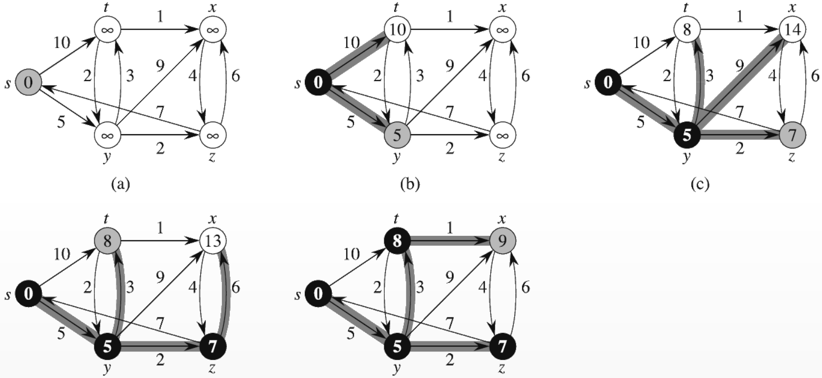
Dijkstra's Exmaple

- Initially, $Q = G.V$, $S = \phi$, and $s.d = 0$ and $v.d = \infty$ for any $v \neq s$.
- Repeatedly take the vertex u with smallest estimate, add it to S, and relax edges leaving u.



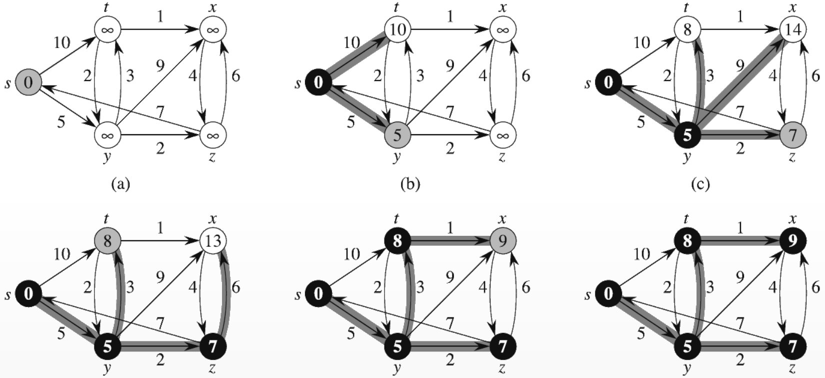
Dijkstra's Exmaple

- Initially, $Q = G.V$, $S = \phi$, and $s.d = 0$ and $v.d = \infty$ for any $v \neq s$.
- Repeatedly take the vertex u with smallest estimate, add it to S, and relax edges leaving u.



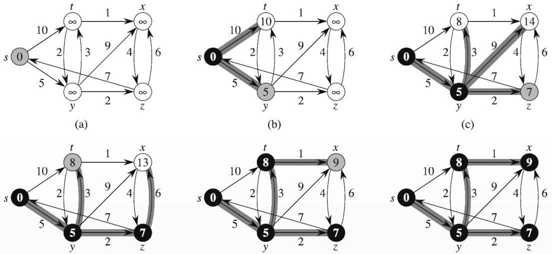
Dijkstra's Exmaple

- Initially, $Q = G.V$, $S = \phi$, and $s.d = 0$ and $v.d = \infty$ for any $v \neq s$.
- Repeatedly take the vertex u with smallest estimate, add it to S, and relax edges leaving u.



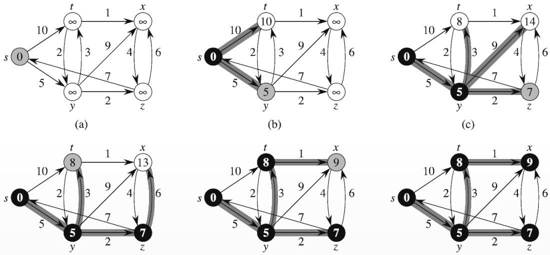
Dijkstra's Analysis

- Dijkstra's algorithm calculates the shortest path from s to every vertex.
  - Anytime we put a new vertex u in S (the vertices already added to the tree), we can say that we already know the shortest path from s to u.



Dijkstra's Analysis

- Dijkstra's algorithm calculates the shortest path from s to every vertex.
    - Anytime we put a new vertex u in S (the vertices already added to the tree), we can say that we already know the shortest path from s to u.
    - Vertices are added to S in the sorted order of their distance from s.



Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?
- **Binary/binomial heaps:**

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?
- **Binary/binomial heaps:**
  - Creating initial heap takes $O(n)$ (why?)
  - Each vertex is extracted once from a priority queue of size $O(n)$; summing to $\Theta(n \log n)$ for all vertices.
  - Each edge $e = (u, v)$ is visited exactly once (in Line 7, when we visit its starting point and relax e).

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?
- **Binary/binomial heaps:**
  - Creating initial heap takes $O(n)$ (why?)
  - Each vertex is extracted once from a priority queue of size $O(n)$; summing to $\Theta(n \log n)$ for all vertices.
  - Each edge $e = (u, v)$ is visited exactly once (in Line 7, when we visit its starting point and relax e).
  - After relax, we reduce the key of the endpoint v in Q; this takes a $O(\log n)$ decrease-key operation $\rightarrow O(m \log n)$ over all edges.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?
- **Binary/binomial heaps:**
  - Creating initial heap takes $O(n)$ (why?)
  - Each vertex is extracted once from a priority queue of size $O(n)$; summing to $\Theta(n \log n)$ for all vertices.
  - Each edge $e = (u, v)$ is visited exactly once (in Line 7, when we visit its starting point and relax e).
  - After relax, we reduce the key of the endpoint v in Q; this takes a $O(\log n)$ decrease-key operation $\rightarrow O(m \log n)$ over all edges.
  - In total, the running time is $\Theta((m + n) \log n)$.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?
- **Binary/binomial heaps:**

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?
- **Binary/binomial heaps:**
  - Creating initial heap takes $O(n)$ (why?)
  - Each vertex is extracted once from a priority queue of size $O(n)$; summing to $\Theta(n \log n)$ for all vertices.
  - Each edge $e = (u, v)$ is visited exactly once (in Line 7, when we visit its starting point and relax e).

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?
- **Binary/binomial heaps:**
  - Creating initial heap takes $O(n)$ (why?)
  - Each vertex is extracted once from a priority queue of size $O(n)$; summing to $\Theta(n \log n)$ for all vertices.
  - Each edge $e = (u, v)$ is visited exactly once (in Line 7, when we visit its starting point and relax e).
  - After relax, we reduce the key of the endpoint v in Q; this takes **a** $O(1)$ decrease-key operation $\rightarrow O(m)$ over all edges.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Dijkstra's Algorithm

- What is the time complexity of the Dijkstra's algorithm?
- **Binary/binomial heaps:**
  - Creating initial heap takes $O(n)$ (why?)
  - Each vertex is extracted once from a priority queue of size $O(n)$; summing to $\Theta(n \log n)$ for all vertices.
  - Each edge $e = (u, v)$ is visited exactly once (in Line 7, when we visit its starting point and relax e).
  - After relax, we reduce the key of the endpoint v in Q; this takes **a** $O(1)$ decrease-key operation $\rightarrow O(m)$ over all edges.
  - In total, the running time is $\Theta(m + n \log n)$.

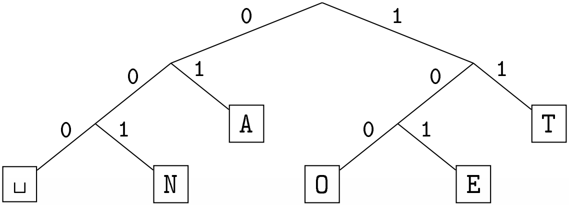
```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Dijkstra's Algorithm

- The running time of Dijkstra's algorithm is $O(n \log n + m)$ when a Fibonacci heap is used, which is better than $O((n + m) \log n)$ of a Binary/binomial heap implementation

Prefix-Free Encoding/Decoding

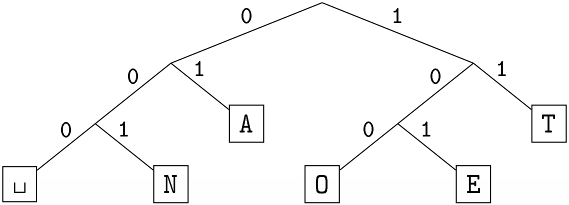
- Binary trees that represent codes are **prefix-free** in the sense that the code for a character c is not the prefix of a code for a character c'.
  - There is always an optimal encoding which is prefix-free.
  - Prefix-free codes are easy to decode!



- Encode `AN␣ANT`
- Decode `111000001010111`

Prefix-Free Encoding/Decoding

- Binary trees that represent codes are **prefix-free** in the sense that the code for a character c is not the prefix of a code for a character c'.
  - There is always an optimal encoding which is prefix-free.
  - Prefix-free codes are easy to decode!



- Encode `AN␣ANT` → `010010000100111`
- Decode `111000001010111` → `TO␣EAT`

Building the Huffman Tree

- For a given source text S, how to determine the "best" tree which minimizes the length of C?
  1. Determine the frequency of each character $c \in \Sigma$ in S
  2. Make $|\Sigma|$ height-0 trees holding each character $c \in \Sigma$. Assign a "frequency" to each tree: sum of frequencies of all letters in tree (initially, these are just the character frequencies.)
  3. Merge two trees with the least frequencies, new frequency is their sum (corresponds to adding one bit to the encoding of each character)
  4. Repeat Step 3 until there is only 1 tree left; this is D.

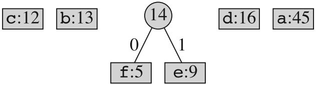
Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

f:5 e:9 c:12 b:13 d:16 a:45

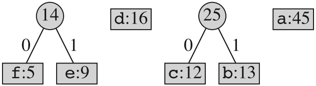
Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



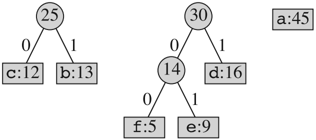
Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



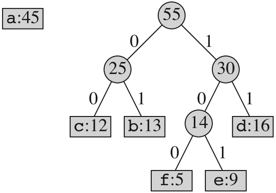
Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



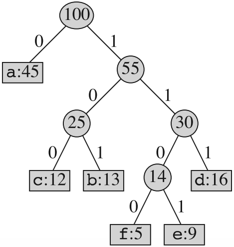
Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



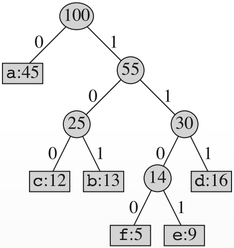
Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5



Building Huffman Example

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100



Building the Huffman Tree

- For a given source text S, how to determine the "best" tree which minimizes the length of C?
  1. Determine the frequency of each character $c \in \Sigma$ in S
  2. Make $|\Sigma|$ height-0 trees holding each character $c \in \Sigma$.
  3. Merge two trees with the least frequencies, new frequency is their sum
     (corresponds to adding one bit to the encoding of each character)
  4. Repeat Step 3 until there is only 1 tree left; this is D.
- What data structure should we store the trees in to make this efficient?

Building the Huffman Tree

- For a given source text S, how to determine the "best" tree which minimizes the length of C?
  1. Determine the frequency of each character $c \in \Sigma$ in S
  2. Make $|\Sigma|$ height-0 trees holding each character $c \in \Sigma$.
  3. Merge two trees with the least frequencies, new frequency is their sum
     (corresponds to adding one bit to the encoding of each character)
  4. Repeat Step 3 until there is only 1 tree left; this is D.
- What data structure should we store the trees in to make this efficient?
  A min-ordered heap! Step 3 is two *extract-min*s and one *insert*

Building the Huffman Tree

- For a given source text S, how to determine the "best" tree which minimizes the length of C?
  1. Determine the frequency of each character $c \in \Sigma$ in S
  2. Make $|\Sigma|$ height-0 trees holding each character $c \in \Sigma$.
  3. Merge two trees with the least frequencies, new frequency is their sum
     (corresponds to adding one bit to the encoding of each character)
  4. Repeat Step 3 until there is only 1 tree left; this is D.
- What data structure should we store the trees in to make this efficient?
  A min-ordered heap! Step 3 is two *extract-min*s and one *insert*
- Does Fibonacci heap have an advantage over binary/binomial heap here?

Summary

- Priority queues are used as “black-boxes" in many classic algorithms.

Summary

- Priority queues are used as “black-boxes" in many classic algorithms.
- Sometimes Fibonacci heaps provide better running time for the algorithm.
  - When there are insert queries are more frequent than delete queries.
  - When we use many decrease-key queries (e.g., Dijkstra’s algorithm)

Summary

- Priority queues are used as "black-boxes" in many classic algorithms.
- Sometimes Fibonacci heaps provide better running time for the algorithm.
  - When there are insert queries are more frequent than delete queries.
  - When we use many decrease-key queries (e.g., Dijkstra's algorithm)
- On the negative side, Fibonacci heaps are harder to implement!