Quadtrees

Outline:

I. Meshing

II. Definition of a quadtree

III. Tree height and size

IV. Search for a neighbor

V. Balancing a quadtree
Modeling with the finite element methods (FEMs)

- mechanics-based
- a finer mesh causes
  - higher accuracy
  - more expensive computation
I. Meshing

Modeling with the finite element methods (FEMs)

- mechanics-based
- a finer mesh causes
  - higher accuracy
  - more expensive computation
I. Meshing

Modeling with the finite element methods (FEMs)

- mechanics-based
- a finer mesh causes
  - higher accuracy
  - more expensive computation

Applications:
- structural analysis & product design
- heat transfer
- fluid flow
- electromagnetism
- graphics & gaming, etc.
Mesh Generation

Domain: $U \times U$ grid
where $U = 2^j$ for some integer $j$

Assumptions:

- Vertices of components have integer coordinates between 0 and $U$.
- Edges of components have four different orientations: $0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}$.
Triangulation Task

Compute a triangular mesh of the square that is...
Triangulation Task

Compute a triangular mesh of the square that is

- **conforming** (no edge of a triangle is allowed to contain a vertex of another triangle in its interior)
Triangulation Task

Compute a triangular mesh of the square that is

- **conforming** (no edge of a triangle is allowed to contain a vertex of another triangle in its interior)
Triangulation Task

Compute a triangular mesh of the square that is

- **conforming** (no edge of a triangle is allowed to contain a vertex of another triangle in its interior)

violated by $\Delta_1$
Compute a triangular mesh of the square that is

- **conforming** (no edge of a triangle is allowed to contain a vertex of another triangle in its interior)

violated by $\Delta_1$

- **Input respecting** (component edges must be contained in the union of the edges of the mesh triangles)
Triangulation Task

Compute a triangular mesh of the square that is

- **conforming** (no edge of a triangle is allowed to contain a vertex of another triangle in its interior)
  - violated by $\Delta_1$

- **Input respecting** (component edges must be contained in the union of the edges of the mesh triangles)
  - violated by $\Delta_2$
Triangulation Task

Compute a triangular mesh of the square that is

- **conforming** (no edge of a triangle is allowed to contain a vertex of another triangle in its interior)

  violated by $\Delta_1$

- **Input respecting** (component edges must be contained in the union of the edges of the mesh triangles)

  violated by $\Delta_2$

- **well-shaped** (angles of any mesh triangle to be in the range between $\frac{\pi}{4}$ and $\frac{\pi}{2}$)
Triangulation Task

Compute a triangular mesh of the square that is

- **conforming** (no edge of a triangle is allowed to contain a vertex of another triangle in its interior)
  
  violated by $\Delta_1$

- **Input respecting** (component edges must be contained in the union of the edges of the mesh triangles)
  
  violated by $\Delta_2$

- **well-shaped** (angles of any mesh triangle to be in the range between $\frac{\pi}{4}$ and $\frac{\pi}{2}$)
  
  violated by $\Delta_3$
Compute a triangular mesh of the square that is

- **conforming** (no edge of a triangle is allowed to contain a vertex of another triangle in its interior)

  violated by $\Delta_1$

- **Input respecting** (component edges must be contained in the union of the edges of the mesh triangles)

  violated by $\Delta_2$

- **well-shaped** (angles of any mesh triangle to be in the range between $\frac{\pi}{4}$ and $\frac{\pi}{2}$)

  violated by $\Delta_3$

- **non-uniform** (mesh should be fine near the edges of components and coarse far away from them)
Steiner Triangulation

$1 \times 1$ component located at $(1,1)$ in a $16 \times 16$ grid.
Steiner Triangulation

1 × 1 component located at (1,1) in a 16 × 16 grid.

Delaunay triangulation of the eight vertices of the square component and grid’s bounding box.
Steiner Triangulation

1 × 1 component located at (1,1) in a 16 × 16 grid.

Delaunay triangulation of the eight vertices of the square component and grid’s bounding box.
Steiner Triangulation

1 × 1 component located at (1,1) in a 16 × 16 grid.

Delaunay triangulation of the eight vertices of the square component and grid’s bounding box.

Angles too small!
Steiner Triangulation

1 $\times$ 1 component located at (1,1) in a 16 $\times$ 16 grid.

Solution:
- Include some mesh vertices as extra points, which are called *Steiner points*.

Delaunay triangulation of the eight vertices of the square component and grid’s bounding box.

Angles too small!
Steiner Triangulation

1 × 1 component located at (1,1) in a 16 × 16 grid.

Solution:

- Include some mesh vertices as extra points, which are called Steiner points.
- Choose these points to form a non-uniform grid.

Delaunay triangulation of the eight vertices of the square component and grid’s bounding box.

Angles too small!
Steiner Triangulation

1 × 1 component located at (1,1) in a 16 × 16 grid.

Delaunay triangulation of the eight vertices of the square component and grid’s bounding box.

Angles too small!

Solution:

- Include some mesh vertices as extra points, which are called Steiner points.
- Choose these points to form a non-uniform grid.

52 triangles (down from 512 uniform triangles)
II. Quadtree

A rooted tree in which

- every node corresponds to a square;
- every internal node $v$ has four children which represents the four quadrants of the node’s corresponding square.
More Terminology
More Terminology

*Face*: a square in the quadtree subdivision (although it may have $\geq 4$ vertices).
More Terminology

Face: a square in the quadtree subdivision (although it may have ≥ 4 vertices).
More Terminology

**Face**: a square in the quadtree subdivision (although it may have $\geq 4$ vertices).

**Corner**: a vertex at the corner of the square.
More Terminology

**Face**: a square in the quadtree subdivision (although it may have $\geq 4$ vertices).

**Corner**: a vertex at the corner of the square.
More Terminology

**Face**: a square in the quadtree subdivision (although it may have $\geq 4$ vertices).

**Corner**: a vertex at the corner of the square.

**Side**: a line segment connecting two consecutive corners of the square.
More Terminology

*Face*: a square in the quadtree subdivision (although it may have \( \geq 4 \) vertices).

*Corner*: a vertex at the corner of the square.

*Side*: a line segment connecting two consecutive corners of the square.
More Terminology

- **Face**: a square in the quadtree subdivision (although it may have $\geq 4$ vertices).

- **Corner**: a vertex at the corner of the square.

- **Side**: a line segment connecting two consecutive corners of the square.

- **Edges of the square**: all the edges in the subdivision that are contained in the square’s boundary.
More Terminology

**Face**: a square in the quadtree subdivision (although it may have $\geq 4$ vertices).

**Corner**: a vertex at the corner of the square.

**Side**: a line segment connecting two consecutive corners of the square.

**Edges of the square**: all the edges in the subdivision that are contained in the square’s boundary.
More Terminology

**Face**: a square in the quadtree subdivision (although it may have $\geq 4$ vertices).

**Corner**: a vertex at the corner of the square.

**Side**: a line segment connecting two consecutive corners of the square.

**Edges of the square**: all the edges in the subdivision that are contained in the square’s boundary.

A side consists of $\geq 1$ edges.
More Terminology

**Face**: a square in the quadtree subdivision (although it may have $\geq 4$ vertices).

**Corner**: a vertex at the corner of the square.

**Side**: a line segment connecting two consecutive corners of the square.

**Edges of the square**: all the edges in the subdivision that are contained in the square’s boundary.

A side consists of $\geq 1$ edges.

Two squares are *neighbors* if they share an edge.
Point Storage

$P$: a set of points

$\sigma = [x_\sigma, x'_\sigma] \times [y_\sigma, y'_\sigma]$: a square to store $P$. 
Point Storage

$P$: a set of points

$\sigma = [x_\sigma, x'_\sigma] \times [y_\sigma, y'_\sigma]$: a square to store $P$.

**Strategy**: Recursively split every square that contains $> 1$ point.
Point Storage

\( P \): a set of points

\( \sigma = [x_\sigma, x'_\sigma] \times [y_\sigma, y'_\sigma] \): a square to store \( P \).

**Strategy**: Recursively split every square that contains \( > 1 \) point.

- If \( |P| \leq 1 \) then the quadtree is a single leaf node that stores \( P \) and \( \sigma \).
Point Storage

$P$: a set of points

$\sigma = [x_\sigma, x'_\sigma] \times [y_\sigma, y'_\sigma]$: a square to store $P$.

**Strategy**: Recursively split every square that contains $> 1$ point.

- If $|P| \leq 1$ then the quadtree is a single leaf node that stores $P$ and $\sigma$.
- Otherwise, let

\[
\begin{align*}
x_{\text{mid}} &= \frac{x_\sigma + x'_\sigma}{2} \\
y_{\text{mid}} &= \frac{y_\sigma + y'_\sigma}{2}
\end{align*}
\]
Point Storage

$P$: a set of points

$\sigma = [x_\sigma, x'_\sigma] \times [y_\sigma, y'_\sigma]$: a square to store $P$.

**Strategy:** Recursively split every square that contains $> 1$ point.

- If $|P| \leq 1$ then the quadtree is a single leaf node that stores $P$ and $\sigma$.

- Otherwise, let

\[
x_{\text{mid}} = \frac{x_\sigma + x'_\sigma}{2} \quad y_{\text{mid}} = \frac{y_\sigma + y'_\sigma}{2}
\]

and define

\[
P_{NE} = \{ p \in P \mid p_x > x_{\text{mid}} \text{ and } p_y > y_{\text{mid}} \} \\
P_{NW} = \{ p \in P \mid p_x \leq x_{\text{mid}} \text{ and } p_y > y_{\text{mid}} \} \\
P_{SW} = \{ p \in P \mid p_x \leq x_{\text{mid}} \text{ and } p_y \leq y_{\text{mid}} \} \\
P_{SE} = \{ p \in P \mid p_x > x_{\text{mid}} \text{ and } p_y \leq y_{\text{mid}} \}
\]
Point Storage

\( P \): a set of points

\( \sigma = [x_\sigma, x'_\sigma] \times [y_\sigma, y'_\sigma] \): a square to store \( P \).

**Strategy**: Recursively split every square that contains \( > 1 \) point.

- If \( |P| \leq 1 \) then the quadtree is a single leaf node that stores \( P \) and \( \sigma \).

- Otherwise, let

\[
x_{\text{mid}} = \frac{x_\sigma + x'_\sigma}{2} \quad y_{\text{mid}} = \frac{y_\sigma + y'_\sigma}{2}
\]

and define

\[
P_{\text{NE}} = \{ p \in P \mid x > x_{\text{mid}} \text{ and } p_y > y_{\text{mid}} \}
\]

\[
P_{\text{NW}} = \{ p \in P \mid x \leq x_{\text{mid}} \text{ and } p_y > y_{\text{mid}} \}
\]

\[
P_{\text{SW}} = \{ p \in P \mid x \leq x_{\text{mid}} \text{ and } p_y \leq y_{\text{mid}} \}
\]

\[
P_{\text{SE}} = \{ p \in P \mid x > x_{\text{mid}} \text{ and } p_y \leq y_{\text{mid}} \}
\]

Store \( P_{\text{NE}}, P_{\text{NW}}, P_{\text{SW}}, P_{\text{SE}} \) in the four quadrants \( \sigma_{\text{NE}}, \sigma_{\text{NW}}, \sigma_{\text{SW}}, \sigma_{\text{SE}} \) of \( \sigma \), respectively.
Recursive Construction

- Compute the smallest enclosing square for the point set $P$. 
Recursive Construction

- Compute the smallest enclosing square for the point set $P$.
- Split the square into four quadrants: NE, NW, SW, and SE.
Recursive Construction

- Compute the smallest enclosing square for the point set $P$.
- Split the square into four quadrants: NE, NW, SW, and SE.
- Partition $P$ accordingly into four subsets, one for each quadrant.
Recursive Construction

- Compute the smallest enclosing square for the point set $P$.
- Split the square into four quadrants: NE, NW, SW, and SE.
- Partition $P$ accordingly into four subsets, one for each quadrant.
- Recursively construct quadtrees for each quadrant.
Recursive Construction

- Compute the smallest enclosing square for the point set $P$.
- Split the square into four quadrants: NE, NW, SW, and SE.
- Partition $P$ accordingly into four subsets, one for each quadrant.
- Recursively construct quadtrees for each quadrant.
- Stop at each quadrant containing $\leq 1$ point.
III. Tree Height

$s$: side length of the initial square containing $P$.

c: minimum distance between any two points in $P$.

**Lemma** The height of a quadtree for $P$ is at most $\log \left( \frac{s}{c} \right) + \frac{3}{2}$. 
III. Tree Height

\( s \): side length of the initial square containing \( P \).

\( c \): minimum distance between any two points in \( P \).

**Lemma** The height of a quadtree for \( P \) is at most \( \log \left( \frac{s}{c} \right) + \frac{3}{2} \).

**Proof** Every internal node contains \( \geq 2 \) points.
III. Tree Height

$s$: side length of the initial square containing $P$.

c: *minimum distance* between any two points in $P$.

**Lemma** The height of a quadtree for $P$ is at most $\log \left( \frac{s}{c} \right) + \frac{3}{2}$.

**Proof** Every internal node contains $\geq 2$ points.

\[\downarrow\]

Its corresponding square must have a diagonal of length $\geq c$. 
s: side length of the initial square containing P.
c: minimum distance between any two points in P.

Lemma The height of a quadtree for P is at most $\log \left( \frac{s}{c} \right) + \frac{3}{2}$.

Proof Every internal node contains $\geq 2$ points.

\[ \downarrow \]

Its corresponding square must have a diagonal of length $\geq c$.

\[ \downarrow \]

The square’s side length must be $\geq \frac{c}{\sqrt{2}}$. 
III. Tree Height

$s$: side length of the initial square containing $P$.
$c$: minimum distance between any two points in $P$.

**Lemma** The height of a quadtree for $P$ is at most $\log \left( \frac{s}{c} \right) + \frac{3}{2}$.

**Proof** Every internal node contains $\geq 2$ points.

\[ \downarrow \]
Its corresponding square must have a diagonal of length $\geq c$.

\[ \downarrow \]
The square’s side length must be $\geq \frac{c}{\sqrt{2}}$.

Meanwhile, when the depth increases by one, the side length of the corresponding square halves.
III. Tree Height

$s$: side length of the initial square containing $P$.
$c$: minimum distance between any two points in $P$.

**Lemma** The height of a quadtree for $P$ is at most $\log \left( \frac{s}{c} \right) + \frac{3}{2}$.

**Proof** Every internal node contains $\geq 2$ points.

\[ \downarrow \]

Its corresponding square must have a diagonal of length $\geq c$.

\[ \downarrow \]

The square’s side length must be $\geq c/\sqrt{2}$.

Meanwhile, when the depth increases by one, the side length of the corresponding square halves.

\[ \downarrow \]

For an internal node at depth $i$, the side length becomes $s/2^i$. 
III. Tree Height

\( s \): side length of the initial square containing \( P \).

\( c \): minimum distance between any two points in \( P \).

**Lemma** The height of a quadtree for \( P \) is at most \( \log \left( \frac{s}{c} \right) + \frac{3}{2} \).

**Proof** Every internal node contains \( \geq 2 \) points.

\[ \downarrow \]

Its corresponding square must have a diagonal of length \( \geq c \).

\[ \downarrow \]

The square’s side length must be \( \geq \frac{c}{\sqrt{2}} \).

Meanwhile, when the depth increases by one, the side length of the corresponding square halves.

\[ \downarrow \]

For an internal node at depth \( i \), the side length becomes \( s/2^i \).
III. Tree Height

$s$: side length of the initial square containing $P$.
$c$: minimum distance between any two points in $P$.

**Lemma** The height of a quadtree for $P$ is at most $\log \left( \frac{s}{c} \right) + \frac{3}{2}$.

**Proof** Every internal node contains $\geq 2$ points.

\[ \downarrow \]

Its corresponding square must have a diagonal of length $\geq c$.

\[ \downarrow \]

The square’s side length must be $\geq \frac{c}{\sqrt{2}}$.

Meanwhile, when the depth increases by one, the side length of the corresponding square halves.

\[ \downarrow \]

For an internal node at depth $i$, the side length becomes $s/2^i$.

\[ i \leq \log \frac{\sqrt{s}}{c} = \log \frac{s}{c} + \frac{1}{2} \]
III. Tree Height

$s$: side length of the initial square containing $P$.
$c$: minimum distance between any two points in $P$.

**Lemma** The height of a quadtree for $P$ is at most $\log \left( \frac{s}{c} \right) + \frac{3}{2}$.

**Proof** Every internal node contains $\geq 2$ points.

- Its corresponding square must have a diagonal of length $\geq c$.

- The square’s side length must be $\geq c/\sqrt{2}$.

Meanwhile, when the depth increases by one, the side length of the corresponding square halves.

- For an internal node at depth $i$, the side length becomes $s/2^i$.

The lemma then follows from that the tree height is one greater than the maximum depth of any internal node.
III. Tree Height

$s$: side length of the initial square containing $P$.
$c$: minimum distance between any two points in $P$.

**Lemma** The height of a quadtree for $P$ is at most $\log \left( \frac{s}{c} \right) + \frac{3}{2}$.

**Proof** Every internal node contains $\geq 2$ points.

\[ \downarrow \]
Its corresponding square must have a diagonal of length $\geq c$.

\[ \downarrow \]
The square’s side length must be $\geq c/\sqrt{2}$.

Meanwhile, when the depth increases by one, the side length of the corresponding square halves.

\[ \downarrow \]
For an internal node at depth $i$, the side length becomes $s/2^i$.

\[ i \leq \log \frac{s\sqrt{2}}{c} = \log \frac{s}{c} + \frac{1}{2} \]

The lemma then follows from that the tree height is one greater than the maximum depth of any internal node.
Number of Nodes

**Theorem 1**  A quadtree of depth $d$ storing $P$ has $O((d + 1)n)$ nodes and can be constructed in $O((d + 1)n)$ time.
Number of Nodes

**Theorem 1** A quadtree of depth \(d\) storing \(P\) has \(O((d + 1)n)\) nodes and can be constructed in \(O((d + 1)n)\) time.

**Proof** Consider the subtree rooted at an internal node \(v\). Let \(\eta(v)\) and \(\mu(v)\) be the numbers of leaves and internal nodes in the subtree, respectively.
Theorem 1  A quadtree of depth $d$ storing $P$ has $O((d + 1)n)$ nodes and can be constructed in $O((d + 1)n)$ time.

Proof  Consider the subtree rooted at an internal node $v$. Let $\eta(v)$ and $\mu(v)$ be the numbers of leaves and internal nodes in the subtree, respectively.

First, we prove by mathematical induction that $\eta(v) = 1 + 3\mu(v)$.
Number of Nodes

Theorem 1  A quadtree of depth $d$ storing $P$ has $O((d + 1)n)$ nodes and can be constructed in $O((d + 1)n)$ time.

Proof  Consider the subtree rooted at an internal node $v$. Let $\eta(v)$ and $\mu(v)$ be the numbers of leaves and internal nodes in the subtree, respectively.

First, we prove by mathematical induction that $\eta(v) = 1 + 3\mu(v)$.

- This is true in the base case where the four children $v_1, v_2, v_3, v_4$ of $v$ are leaves, i.e., $\mu(v) = 1$. 
**Theorem 1** A quadtree of depth $d$ storing $P$ has $O((d + 1)n)$ nodes and can be constructed in $O((d + 1)n)$ time.

**Proof** Consider the subtree rooted at an internal node $v$. Let $\eta(v)$ and $\mu(v)$ be the numbers of leaves and internal nodes in the subtree, respectively.

First, we prove by mathematical induction that $\eta(v) = 1 + 3\mu(v)$.

- This is true in the base case where the four children $v_1, v_2, v_3, v_4$ of $v$ are leaves, i.e., $\mu(v) = 1$.
- In the general case, the number of leaves in the subtree is

$$\eta(v) = \eta(v_1) + \eta(v_2) + \eta(v_3) + \eta(v_4)$$
**Theorem 1** A quadtree of depth \(d\) storing \(P\) has \(O((d+1)n)\) nodes and can be constructed in \(O((d+1)n)\) time.

**Proof** Consider the subtree rooted at an internal node \(v\). Let \(\eta(v)\) and \(\mu(v)\) be the numbers of leaves and internal nodes in the subtree, respectively.

First, we prove by mathematical induction that \(\eta(v) = 1 + 3\mu(v)\).

- This is true in the base case where the four children \(v_1, v_2, v_3, v_4\) of \(v\) are leaves, i.e., \(\mu(v) = 1\).
- In the general case, the number of leaves in the subtree is

\[
\eta(v) = \eta(v_1) + \eta(v_2) + \eta(v_3) + \eta(v_4) \\
= 3(\mu(v_1) + \mu(v_2) + \mu(v_3) + \mu(v_4)) + 4
\]
Number of Nodes

Theorem 1 A quadtree of depth $d$ storing $P$ has $O((d + 1)n)$ nodes and can be constructed in $O((d + 1)n)$ time.

Proof Consider the subtree rooted at an internal node $v$. Let $\eta(v)$ and $\mu(v)$ be the numbers of leaves and internal nodes in the subtree, respectively.

First, we prove by mathematical induction that $\eta(v) = 1 + 3\mu(v)$.

- This is true in the base case where the four children $v_1, v_2, v_3, v_4$ of $v$ are leaves, i.e., $\mu(v) = 1$.

- In the general case, the number of leaves in the subtree is

$$\eta(v) = \eta(v_1) + \eta(v_2) + \eta(v_3) + \eta(v_4)$$

$$= 3(\mu(v_1) + \mu(v_2) + \mu(v_3) + \mu(v_4)) + 4$$

$$= 3(\mu(v) - 1) + 4$$
Number of Nodes

**Theorem 1** A quadtree of depth $d$ storing $P$ has $O((d + 1)n)$ nodes and can be constructed in $O((d + 1)n)$ time.

**Proof** Consider the subtree rooted at an internal node $v$. Let $\eta(v)$ and $\mu(v)$ be the numbers of leaves and internal nodes in the subtree, respectively.

First, we prove by mathematical induction that $\eta(v) = 1 + 3\mu(v)$.

- This is true in the base case where the four children $v_1, v_2, v_3, v_4$ of $v$ are leaves, i.e., $\mu(v) = 1$.
- In the general case, the number of leaves in the subtree is

$$
\eta(v) = \eta(v_1) + \eta(v_2) + \eta(v_3) + \eta(v_4)
= 3(\mu(v_1) + \mu(v_2) + \mu(v_3) + \mu(v_4)) + 4
= 3(\mu(v) - 1) + 4
= 3\mu(v) + 1
$$
Number of Nodes

**Theorem 1** A quadtree of depth $d$ storing $P$ has $O((d + 1)n)$ nodes and can be constructed in $O((d + 1)n)$ time.

**Proof** Consider the subtree rooted at an internal node $v$. Let $\eta(v)$ and $\mu(v)$ be the numbers of leaves and internal nodes in the subtree, respectively.

First, we prove by mathematical induction that $\eta(v) = 1 + 3\mu(v)$.

- This is true in the base case where the four children $v_1, v_2, v_3, v_4$ of $v$ are leaves, i.e., $\mu(v) = 1$.
- In the general case, the number of leaves in the subtree is
  \[
  \eta(v) = \eta(v_1) + \eta(v_2) + \eta(v_3) + \eta(v_4)
  = 3(\mu(v_1) + \mu(v_2) + \mu(v_3) + \mu(v_4)) + 4
  = 3(\mu(v) - 1) + 4
  = 3\mu(v) + 1
  \]

Thus, it suffices to bound $\mu(v)$ for the size of the quadtree.
Proof (cont’d)

The associated square of any internal node contains \( \geq 2 \) points (otherwise it would be a leaf not an internal node).
Proof (cont’d)

The associated square of any internal node contains $\geq 2$ points (otherwise it would be a leaf not an internal node). The squares of all internal nodes at the same depth are disjoint.
Proof (cont’d)

The associated square of any internal node contains $\geq 2$ points (otherwise it would be a leaf not an internal node). The squares of all internal nodes at the same depth are disjoint. There are $n$ points in total distributed among these squares.
Proof (cont’d)

The associated square of any internal node contains $\geq 2$ points (otherwise it would be a leaf not an internal node).
The squares of all internal nodes at the same depth are disjoint.
There are $n$ points in total distributed among these squares.

There are $\leq n/2$ internal nodes at the depth.
Proof (cont’d)

The associated square of any internal node contains $\geq 2$ points (otherwise it would be a leaf not an internal node).
The squares of all internal nodes at the same depth are disjoint.
There are $n$ points in total distributed among these squares.

There are $\leq n/2$ internal nodes at the depth.

$\Rightarrow$

There are $O((d + 1)n)$ internal nodes in the tree.
(We include 1 in the Big-O in case $d = 0$.)
Proof (cont’d)

The associated square of any internal node contains \( \geq 2 \) points (otherwise it would be a leaf not an internal node).

The squares of all internal nodes at the same depth are disjoint.

There are \( n \) points in total distributed among these squares.

There are \( \leq n/2 \) internal nodes at the depth.

\[ \text{There are } O((d + 1)n) \text{ internal nodes in the tree.} \]

(We include 1 in the Big-\( O \) in case \( d = 0 \).)

The quadtree has \( O((d + 1)n) \) nodes.
Proof (cont’d)

The associated square of any internal node contains \( \geq 2 \) points (otherwise it would be a leaf not an internal node). The squares of all internal nodes at the same depth are disjoint. There are \( n \) points in total distributed among these squares.

There are \( \leq n/2 \) internal nodes at the depth.

\[ \text{There are } O((d + 1)n) \text{ internal nodes in the tree.} \]
(We include 1 in the Big-\( O \) in case \( d = 0 \).)

\[ \text{The quadtree has } O((d + 1)n) \text{ nodes.} \]

The amount of time spent at an internal node is linear in the number of points in its associated square.
Proof (cont’d)

The associated square of any internal node contains $\geq 2$ points (otherwise it would be a leaf not an internal node). The squares of all internal nodes at the same depth are disjoint. There are $n$ points in total distributed among these squares.

There are $\leq n/2$ internal nodes at the depth.

$\Downarrow$

There are $O((d + 1)n)$ internal nodes in the tree. (We include 1 in the Big-$O$ in case $d = 0$.)

$\Downarrow$

The quadtree has $O((d + 1)n)$ nodes.

The amount of time spent at an internal node is linear in the number of points in its associated square.

$n$ points are distributed among the squares at the same depth.
The associated square of any internal node contains $\geq 2$ points (otherwise it would be a leaf not an internal node).

The squares of all internal nodes at the same depth are disjoint.

There are $n$ points in total distributed among these squares. There are $\leq n/2$ internal nodes at the depth.

There are $O((d + 1)n)$ internal nodes in the tree. (We include 1 in the Big-O in case $d = 0$.)

The quadtree has $O((d + 1)n)$ nodes.

The amount of time spent at an internal node is linear in the number of points in its associated square.

$n$ points are distributed among the squares at the same depth. $O(n)$ work at each depth.
Proof (cont’d)

The associated square of any internal node contains ≥ 2 points (otherwise it would be a leaf not an internal node).
The squares of all internal nodes at the same depth are disjoint.
There are \( n \) points in total distributed among these squares.

There are \( \leq n/2 \) internal nodes at the depth.

There are \( O((d + 1)n) \) internal nodes in the tree.
(We include 1 in the Big-O in case \( d = 0 \).)

The quadtree has \( O((d + 1)n) \) nodes.

The amount of time spent at an internal node is linear in the number of points in its associated square.

\( n \) points are distributed among the squares at the same depth.

\( O(n) \) work at each depth.

Total construction work \( O((d + 1)n) \).
Proof (cont’d)

The associated square of any internal node contains $\geq 2$ points (otherwise it would be a leaf not an internal node).
The squares of all internal nodes at the same depth are disjoint.
There are $n$ points in total distributed among these squares.

There are $\leq n/2$ internal nodes at the depth.

There are $O((d + 1)n)$ internal nodes in the tree.
(We include 1 in the Big-O in case $d = 0$.)

The quadtree has $O((d + 1)n)$ nodes.

The amount of time spent at an internal node is linear in the number of points in its associated square.

$n$ points are distributed among the squares at the same depth.

$O(n)$ work at each depth.

Total construction work $O((d + 1)n)$. 
Problem Given a node $v$ and a direction (north, east, south, or west), find the deepest node $v'$ with depth $\leq$ the depth of $v$ such that its associate square $\sigma(v')$ is adjacent to the associate square $\sigma(v)$ of $v$ in the given direction.

Returns nil if no adjacent square in the direction (e.g., $\sigma(v)$’s edge is part of the edge of the bounding square in that direction).
IV. Neighbor Finding

**Problem** Given a node \( v \) and a direction (north, east, south, or west), find the deepest node \( v' \) with depth \( \leq \) the depth of \( v \) such that its associate square \( \sigma(v') \) is adjacent to the associate square \( \sigma(v) \) of \( v \) in the given direction.

Returns **nil** if no adjacent square in the direction (e.g., \( \sigma(v) \)'s edge is part of the edge of the bounding square in that direction).

Suppose we want to find the north neighbor of \( v \).
IV. Neighbor Finding

**Problem** Given a node \( v \) and a direction (north, east, south, or west), find the deepest node \( v' \) with depth \( \leq \) the depth of \( v \) such that its associate square \( \sigma(v') \) is adjacent to the associate square \( \sigma(v) \) of \( v \) in the given direction.

Returns **nil** if no adjacent square in the direction (e.g., \( \sigma(v) \)'s edge is part of the edge of the bounding square in that direction).

Suppose we want to find the north neighbor of \( v \).

- \( v \) is the SE or SW-child of its parent.
IV. Neighbor Finding

Problem Given a node $v$ and a direction (north, east, south, or west), find the deepest node $v'$ with depth $\leq$ the depth of $v$ such that its associate square $\sigma(v')$ is adjacent to the associate square $\sigma(v)$ of $v$ in the given direction.

Returns nil if no adjacent square in the direction (e.g., $\sigma(v)$’s edge is part of the edge of the bounding square in that direction).

Suppose we want to find the north neighbor of $v$.

- $v$ is the SE or SW-child of its parent.

\[
\downarrow
\]

The north neighbor is the NE or NW-child of the parent.
IV. Neighbor Finding

**Problem** Given a node $v$ and a direction (north, east, south, or west), find the deepest node $v'$ with depth $\leq$ the depth of $v$ such that its associate square $\sigma(v')$ is adjacent to the associate square $\sigma(v)$ of $v$ in the given direction.

Returns nil if no adjacent square in the direction (e.g., $\sigma(v)$’s edge is part of the edge of the bounding square in that direction).

Suppose we want to find the north neighbor of $v$.

- $v$ is the SE or SW-child of its parent.

\[ \Downarrow \]

The north neighbor is the NE or NW-child of the parent.
Problem Given a node $v$ and a direction (north, east, south, or west), find the deepest node $v'$ with depth $\leq$ the depth of $v$ such that its associate square $\sigma(v')$ is adjacent to the associate square $\sigma(v)$ of $v$ in the given direction.

Returns nil if no adjacent square in the direction (e.g., $\sigma(v)$’s edge is part of the edge of the bounding square in that direction).

Suppose we want to find the north neighbor of $v$.

- $v$ is the SE or SW-child of its parent. 
  \[ \Downarrow \]
  The north neighbor is the NE or NW-child of the parent.

- $v$ is the NE or NW-child of its parent.
  Recursively finds the north neighbor, $\mu$, of the parent node.
Problem Given a node $v$ and a direction (north, east, south, or west), find the deepest node $v'$ with depth $\leq$ the depth of $v$ such that its associate square $\sigma(v')$ is adjacent to the associate square $\sigma(v)$ of $v$ in the given direction.

Returns nil if no adjacent square in the direction (e.g., $\sigma(v)$’s edge is part of the edge of the bounding square in that direction).

Suppose we want to find the north neighbor of $v$.

- $v$ is the SE or SW-child of its parent.  

  \[ \downarrow \]

  The north neighbor is the NE or NW-child of the parent.

- $v$ is the NE or NW-child of its parent.

  Recursively finds the north neighbor, $\mu$, of the parent node.

  * If $\mu$ is an internal node, then the neighbor of $v$ is its SE or SW-child.
IV. Neighbor Finding

**Problem** Given a node \( v \) and a direction (north, east, south, or west), find the deepest node \( v' \) with depth \( \leq \) the depth of \( v \) such that its associate square \( \sigma(v') \) is adjacent to the associate square \( \sigma(v) \) of \( v \) in the given direction.

Returns \textit{nil} if no adjacent square in the direction (e.g., \( \sigma(v) \)'s edge is part of the edge of the bounding square in that direction).

Suppose we want to find the north neighbor of \( v \).

- \( v \) is the SE or SW-child of its parent.

  \[
  \text{The north neighbor is the NE or NW-child of the parent.}
  \]

- \( v \) is the NE or NW-child of its parent.

  Recursively finds the north neighbor, \( \mu \), of the parent node.
  - \( \mu \) is an internal node, then the neighbor of \( v \) is its SE or SW-child.
  - \( \mu \) is a leaf, then it is the neighbor we are looking for.
Searching for the North Neighbor

NorthNeighbor(v, T)

1. if v = root(T)
2. then return nil
3. u ← parent(v)
4. if v is the SW-child of u
5. then return the NW-child of u
6. if v is the SE-child of u
7. then return the NE-child of u
8. μ ← NorthNeighbor(u, T)
9. if μ = nil or μ is a leaf
10. then return μ
11. else if v is the NW-child of u
12. then return the SW-child of μ
13. else return the SE-child of μ
Searching for the North Neighbor

NorthNeighbor($v, T$)

1. if $v = \text{root}(T)$
2. then return nil
3. $u \leftarrow \text{parent}(v)$
4. if $v$ is the SW-child of $u$
5. then return the NW-child of $u$
6. if $v$ is the SE-child of $u$
7. then return the NE-child of $u$
8. $\mu \leftarrow \text{NorthNeighbor}(u, T)$
9. if $\mu = \text{nil}$ or $\mu$ is a leaf
10. then return $\mu$
11. else if $v$ is the NW-child of $u$
12. then return the SW-child of $\mu$
13. else return the SE-child of $\mu$

* The call does not necessarily return a leaf node.
Searching for the North Neighbor

NorthNeighbor($v, T$)

1. if $v = \text{root}(T)$
2. then return nil
3. $u \leftarrow \text{parent}(v)$
4. if $v$ is the SW-child of $u$
5. then return the NW-child of $u$
6. if $v$ is the SE-child of $u$
7. then return the NE-child of $u$
8. $\mu \leftarrow \text{NorthNeighbor}(u, T)$
9. if $\mu = \text{nil}$ or $\mu$ is a leaf
10. then return $\mu$
11. else if $v$ is the NW-child of $u$
12. then return the SW-child of $\mu$
13. else return the SE-child of $\mu$

• The call does not necessarily return a leaf node.

• To find one, needs to walk down the quadtree from the returned node, always proceeding to a south-child.
A Bigger Example

NorthNeighbor(\(v, \mathcal{T}\))

1. if \(v = \text{root}(\mathcal{T})\)
2. then return nil
3. \(u \leftarrow \text{parent}(v)\)
4. if \(v\) is the SW-child of \(u\)
5. then return the NW-child of \(u\)
6. if \(v\) is the SE-child of \(u\)
7. then return the NE-child of \(u\)
8. \(\mu \leftarrow \text{NorthNeighbor}(u, \mathcal{T})\)
9. if \(\mu = \text{nil}\) or \(\mu\) is a leaf
10. then return \(\mu\)
11. else if \(v\) is the NW-child of \(u\)
12. then return the SW-child of \(\mu\)
13. else return the SE-child of \(\mu\)
NorthNeighbor($v, T$)

1. if $v = \text{root}(T)$
2. then return nil
3. $u \leftarrow \text{parent}(v)$
4. if $v$ is the SW-child of $u$
5. then return the NW-child of $u$
6. if $v$ is the SE-child of $u$
7. then return the NE-child of $u$
8. $\mu \leftarrow \text{NorthNeighbor}(u, T)$
9. if $\mu = \text{nil}$ or $\mu$ is a leaf
10. then return $\mu$
11. else if $v$ is the NW-child of $u$
12. then return the SW-child of $\mu$
13. else return the SE-child of $\mu$

NorthNeighbor($v, T$)
A Bigger Example

\[
\text{NorthNeighbor}(v, \mathcal{T})
\]

1. if \( v = \text{root}(\mathcal{T}) \)
2. then return \( \text{nil} \)
3. \( u \leftarrow \text{parent}(v) \)
4. if \( v \) is the SW-child of \( u \)
5. then return the NW-child of \( u \)
6. if \( v \) is the SE-child of \( u \)
7. then return the NE-child of \( u \)
8. \( \mu \leftarrow \text{NorthNeighbor}(u, \mathcal{T}) \)
9. if \( \mu = \text{nil} \) or \( \mu \) is a leaf
10. then return \( \mu \)
11. else if \( v \) is the NW-child of \( u \)
12. then return the SW-child of \( \mu \)
13. else return the SE-child of \( \mu \)
A Bigger Example

NorthNeighbor(v, T)

1. if v = root(T) then return nil
2. u ← parent(v)
3. if v is the SW-child of u then return the NW-child of u
4. if v is the SE-child of u then return the NE-child of u
5. μ ← NorthNeighbor(u, T)
6. if μ = nil or μ is a leaf then return μ
7. else if v is the NW-child of u then return the SW-child of μ
8. else return the SE-child of μ

NorthNeighbor(v, T)  
\n
NorthNeighbor(u, T)  
\n
NorthNeighbor(t, T)  
\n
A Bigger Example

NorthNeighbor($v, T$)

1. \textbf{if} $v = \text{root}(T)$
2. \hspace{1em} then return nil
3. \hspace{1em} $u \leftarrow \text{parent}(v)$
4. \hspace{1em} \textbf{if} $v$ is the SW-child of $u$
5. \hspace{2em} then return the NW-child of $u$
6. \hspace{1em} \textbf{if} $v$ is the SE-child of $u$
7. \hspace{2em} then return the NE-child of $u$
8. $\mu \leftarrow \text{NorthNeighbor}(u, T)$
9. \hspace{1em} \textbf{if} $\mu = \text{nil}$ or $\mu$ is a leaf
10. \hspace{2em} then return $\mu$
11. \hspace{1em} \textbf{else if} $v$ is the NW-child of $u$
12. \hspace{2em} then return the SW-child of $\mu$
13. \hspace{2em} \textbf{else return} the SE-child of $\mu$
A Bigger Example

NorthNeighbor(\(v, T\))
1. if \(v = \text{root}(T)\)
2. then return nil
3. \(u \leftarrow \text{parent}(v)\)
4. if \(v\) is the SW-child of \(u\)
5. then return the NW-child of \(u\)
6. if \(v\) is the SE-child of \(u\)
7. then return the NE-child of \(u\)
8. \(\mu \leftarrow \text{NorthNeighbor}(u, T)\)
9. if \(\mu = \text{nil}\) or \(\mu\) is a leaf
10. then return \(\mu\)
11. else if \(v\) is the NW-child of \(u\)
12. then return the SW-child of \(\mu\)
13. else return the SE-child of \(\mu\)

\[ \sigma(r) \]

\[ \sigma(s) \]

\[ \sigma(w) \]

\[ \sigma(t) \]

\[ \sigma(q) \]

\[ r \]

\[ s \]

\[ q \]

\[ u \]

\[ v \]

\[ w \]

\[ t \]

\[ \mu = \sigma \]

NorthNeighbor(\(v, T\))
NorthNeighbor(\(u, T\))
NorthNeighbor(\(t, T\))
NorthNeighbor(\(q, T\))
A Bigger Example

NorthNeighbor(v, T)

1. if v = root(T)
2. then return nil
3. u ← parent(v)
4. if v is the SW-child of u
5. then return the NW-child of u
6. if v is the SE-child of u
7. then return the NE-child of u
8. μ ← NorthNeighbor(u, T)
9. if μ = nil or μ is a leaf
10. then return μ
11. else if v is the NW-child of u
12. then return the SW-child of μ
13. else return the SE-child of μ
A Bigger Example

NorthNeighbor(v, T)

1. if v = root(T)  
   2. then return nil  
   3. u ← parent(v)  
   4. if v is the SW-child of u  
      5. then return the NW-child of u  
   6. if v is the SE-child of u  
      7. then return the NE-child of u  
   8. μ ← NorthNeighbor(u, T)  
   9. if μ = nil or μ is a leaf  
      10. then return μ  
   11. else if v is the NW-child of u  
      12. then return the SW-child of μ  
   13. else return the SE-child of μ

NorthNeighbor(v, T)
  NorthNeighbor(u, T)
  NorthNeighbor(t, T)
  NorthNeighbor(q, T)  
  μ = σ
  returns s

Need to descend from s to locate the neighbor w.
V. Balanced Quadtree

In its subdivision, any two neighboring squares differ by a factor of one or two in size (as measured by the side length not by area).

This implies that any two leaves whose squares are neighbors can differ at most one in depth.

Unbalanced subdivision and Corresponding quadtree.
Example
Example

balancing
Example
BalanceQuadTree(\(T\))

1. insert all the leaves of \(T\) into a linear list \(\mathcal{L}\)
2. while \(\mathcal{L}\) is not empty
3. do remove a leaf \(\mu\) from \(\mathcal{L}\)
4. if \(\sigma(\mu)\) has to be split
5. then make \(\mu\) an internal node with four new leaves
6. if \(\mu\) stores a point
7. then stores it in the correct new leaf
8. insert the four new leaves into \(\mathcal{L}\)
9. if \(\sigma(\mu)\) had neighbors that now need to be split
10. then insert them into \(\mathcal{L}\)
First Issue to Settle

On line 4 of the algorithm, how to check if a leaf $\sigma(\mu)$ needs to be split?

BalanceQuadTree($T$)

1. insert all the leaves of $T$ into a linear list $\mathcal{L}$
2. while $\mathcal{L}$ is not empty
3. do remove a leaf $\mu$ from $\mathcal{L}$
4. if $\sigma(\mu)$ has to be split
5. then make $\mu$ an internal node with four new leaves
6. if $\mu$ stores a point
7. then stores it in the correct new leaf
8. insert the four new leaves into $\mathcal{L}$
9. if $\sigma(\mu)$ had neighbors that now need to be split
10. then insert them into $\mathcal{L}$
First Issue to Settle

On line 4 of the algorithm, how to check if a leaf $\sigma(\mu)$ needs to be split?

- Check if $\sigma(\mu)$ has a neighboring square less than half its size.
- Employ the earlier introduced neighbor finding algorithm.

```
BalanceQuadTree(T')
1. insert all the leaves of $T$ into a linear list $L$
2. while $L$ is not empty
3. do remove a leaf $\mu$ from $L$
4. if $\sigma(\mu)$ has to be split
5. then make $\mu$ an internal node with four new leaves
6. if $\mu$ stores a point
7. then stores it in the correct new leaf
8. insert the four new leaves into $L$
9. if $\sigma(\mu)$ had neighbors that now need to be split
10. then insert them into $L$
```
First Issue to Settle

On line 4 of the algorithm, how to check if a leaf $\sigma(\mu)$ needs to be split?

- Check if $\sigma(\mu)$ has a neighboring square less than half its size.
- Employ the earlier introduced neighbor finding algorithm.

Such a small neighbor in the north exists iff $\text{NorthNeighbor}(\mu, \mathcal{T})$ returns a node that has a SW- or SE-child that is not a leaf.

BalanceQuadTree($\mathcal{T}$)

1. insert all the leaves of $\mathcal{T}$ into a linear list $\mathcal{L}$
2. while $\mathcal{L}$ is not empty
3. do remove a leaf $\mu$ from $\mathcal{L}$
4. if $\sigma(\mu)$ has to be split
5. then make $\mu$ an internal node with four new leaves
6. if $\mu$ stores a point
7. then stores it in the correct new leaf
8. insert the four new leaves into $\mathcal{L}$
9. if $\sigma(\mu)$ had neighbors that now need to be split
10. then insert them into $\mathcal{L}$
BalanceQuadTree($\mathcal{T}$)

1. insert all the leaves of $\mathcal{T}$ into a linear list $\mathcal{L}$
2. while $\mathcal{L}$ is not empty
3. do remove a leaf $\mu$ from $\mathcal{L}$
4. if $\sigma(\mu)$ has to be split
5. then make $\mu$ an internal node with four new leaves
6. if $\mu$ stores a point
7. then stores it in the correct new leaf
8. insert the four new leaves into $\mathcal{L}$
9. if $\sigma(\mu)$ had neighbors that now need to be split
10. then insert them into $\mathcal{L}$
BalanceQuadTree($\mathcal{T}$)

1. insert all the leaves of $\mathcal{T}$ into a linear list $\mathcal{L}$
2. while $\mathcal{L}$ is not empty
3. do remove a leaf $\mu$ from $\mathcal{L}$
4. if $\sigma(\mu)$ has to be split
5. then make $\mu$ an internal node with four new leaves
6. if $\mu$ stores a point
7. then stores it in the correct new leaf
8. insert the four new leaves into $\mathcal{L}$
9. if $\sigma(\mu)$ had neighbors that now need to be split
10. then insert them into $\mathcal{L}$

On line 9, check if $\sigma(\mu)$, already split, had a neighbor that needs to be split.
BalanceQuadTree($T$)

1. insert all the leaves of $T$ into a linear list $L$
2. while $L$ is not empty
   3. do remove a leaf $\mu$ from $L$
   4. if $\sigma(\mu)$ has to be split
      5. then make $\mu$ an internal node with four new leaves
         if $\mu$ stores a point
            7. then stores it in the correct new leaf
       8. insert the four new leaves into $L$
   9. if $\sigma(\mu)$ had neighbors that now need to be split
      10. then insert them into $L$

On line 9, check if $\sigma(\mu)$, already split, had a neighbor that needs to be split.

- Again use the neighbor finding algorithm.
  - Such a neighbor exists to the north iff $\text{NorthNeighbor}(\mu, T)$ returns a node corresponding to a square larger than $\sigma(\mu)$. 

Second Issue to Settle

BalanceQuadTree($\mathcal{T}$)

1. insert all the leaves of $\mathcal{T}$ into a linear list $\mathcal{L}$
2. while $\mathcal{L}$ is not empty
3. do remove a leaf $\mu$ from $\mathcal{L}$
4. if $\sigma(\mu)$ has to be split
5. then make $\mu$ an internal node with four new leaves
6. if $\mu$ stores a point
7. then stores it in the correct new leaf
8. insert the four new leaves into $\mathcal{L}$
9. if $\sigma(\mu)$ had neighbors that now need to be split
10. then insert them into $\mathcal{L}$

On line 9, check if $\sigma(\mu)$, already split, had a neighbor that needs to be split.

- Again use the neighbor finding algorithm.
  - Such a neighbor exists to the north iff NorthNeighbor($\mu, \mathcal{T}$) returns a node corresponding to a square larger than $\sigma(\mu)$. 
BalanceQuadTree(\(T\))

1. insert all the leaves of \(T\) into a linear list \(L\)
2. while \(L\) is not empty
3. do remove a leaf \(\mu\) from \(L\)
4. if \(\sigma(\mu)\) has to be split
5. then make \(\mu\) an internal node with four new leaves
6. if \(\mu\) stores a point
7. then stores it in the correct new leaf
8. insert the four new leaves into \(L\)
9. if \(\sigma(\mu)\) had neighbors that now need to be split
10. then insert them into \(L\)

On line 9, check if \(\sigma(\mu)\), already split, had a neighbor that needs to be split.

- Again use the neighbor finding algorithm.
  - Such a neighbor exists to the north iff NorthNeighbor(\(\mu, T\)) returns a node corresponding to a square larger than \(\sigma(\mu)\).
  - Such a neighbor would be more than twice the size of each of the four children from splitting of \(\mu\) on line 5.
Cost of Balancing

**Theorem 2** Let $\mathcal{T}$ be a quadtree with $m$ nodes. Then the balanced version of $\mathcal{T}$ has $O(m)$ nodes and can be constructed in $O((d + 1)m)$ time.

**Proof** Omitted.
Cost of Balancing

**Theorem 2** Let $T$ be a quadtree with $m$ nodes. Then the balanced version of $T$ has $O(m)$ nodes and can be constructed in $O((d + 1)m)$ time.

**Proof** Omitted.

Mesh generation from a set of disjoint polygons.
Cost of Balancing

**Theorem 2** Let $\mathcal{T}$ be a quadtree with $m$ nodes. Then the balanced version of $\mathcal{T}$ has $O(m)$ nodes and can be constructed in $O((d + 1)m)$ time.

**Proof** Omitted.

Mesh generation from a set of disjoint polygons.

- Construct a quadtree subdivision of the polygon vertices.
Cost of Balancing

**Theorem 2**  Let $\mathcal{T}$ be a quadtree with $m$ nodes. Then the balanced version of $\mathcal{T}$ has $O(m)$ nodes and can be constructed in $O((d + 1)m)$ time.

**Proof**  Omitted.

Mesh generation from a set of disjoint polygons.

- Construct a quadtree subdivision of the polygon vertices.
  - Stop splitting a square when it is no longer intersected by a polygon edge, or when it has unit size.
Cost of Balancing

**Theorem 2** Let $\mathcal{T}$ be a quadtree with $m$ nodes. Then the balanced version of $\mathcal{T}$ has $O(m)$ nodes and can be constructed in $O((d + 1)m)$ time.

**Proof** Omitted.

Mesh generation from a set of disjoint polygons.

- Construct a quadtree subdivision of the polygon vertices.
  - Stop splitting a square when it is no longer intersected by a polygon edge, or when it has unit size.
- Balance the quadtree subdivision.
- Triangulate the balanced quadtree subdivision (adding Steiner points).