Quadtrees

Outline:

I. Meshing

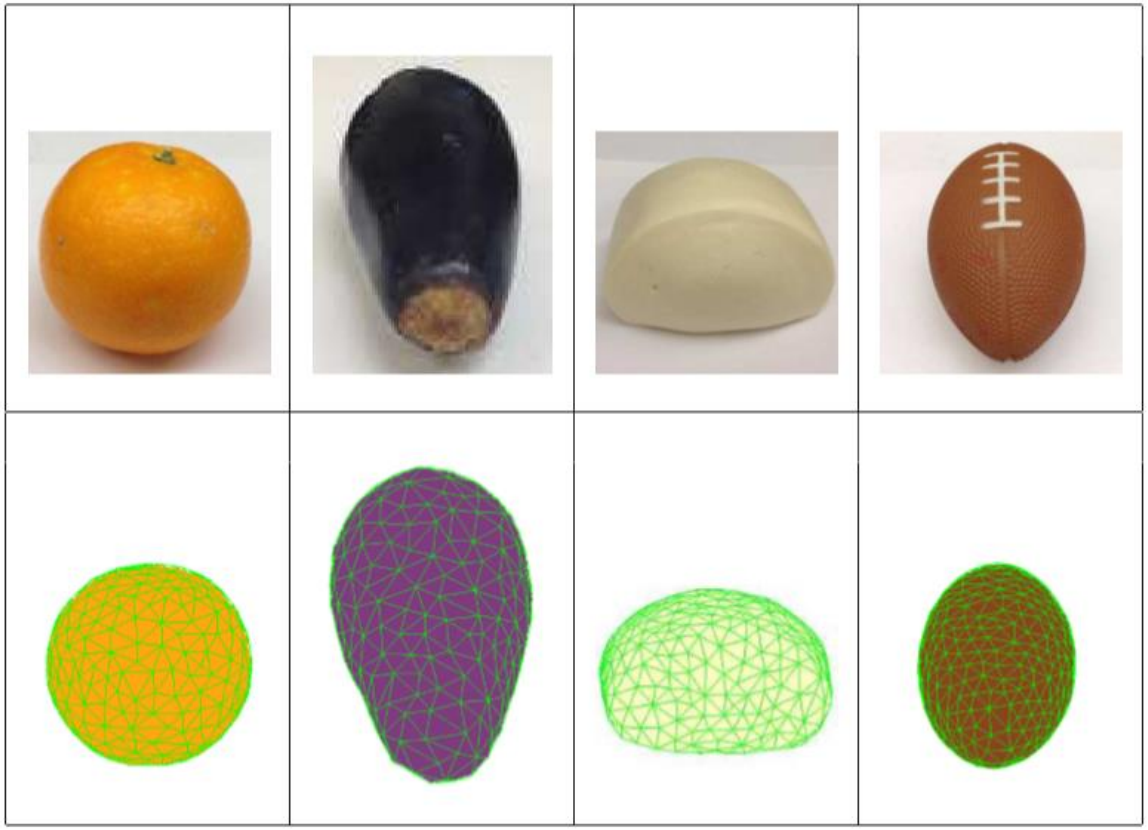
II. Definition of a quadtree

III. Tree height and size

IV. Search for a neighbor

V. Balancing a quadtree

I. Meshing



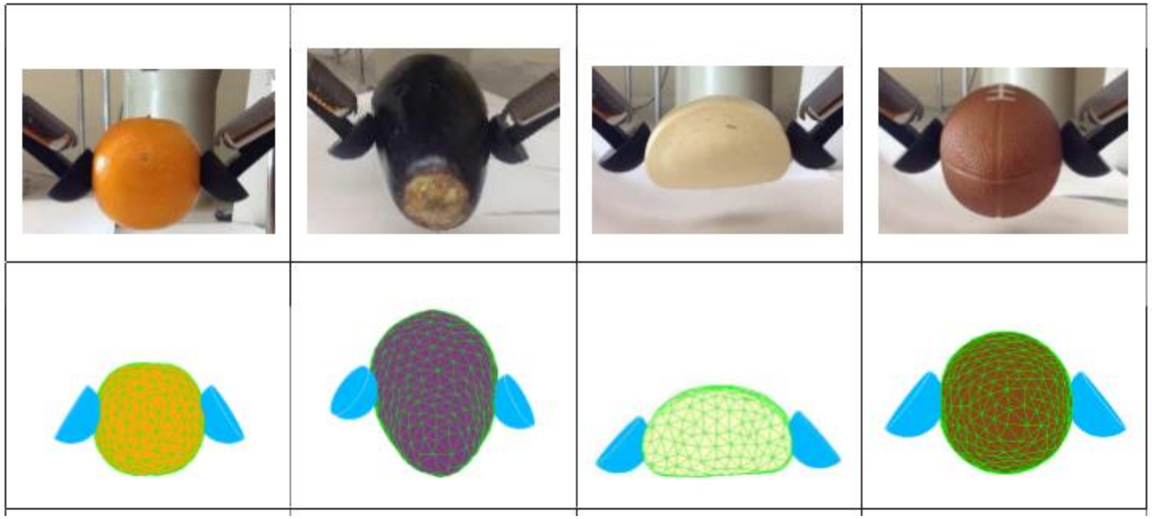
Modeling with the finite element methods (FEMs)

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

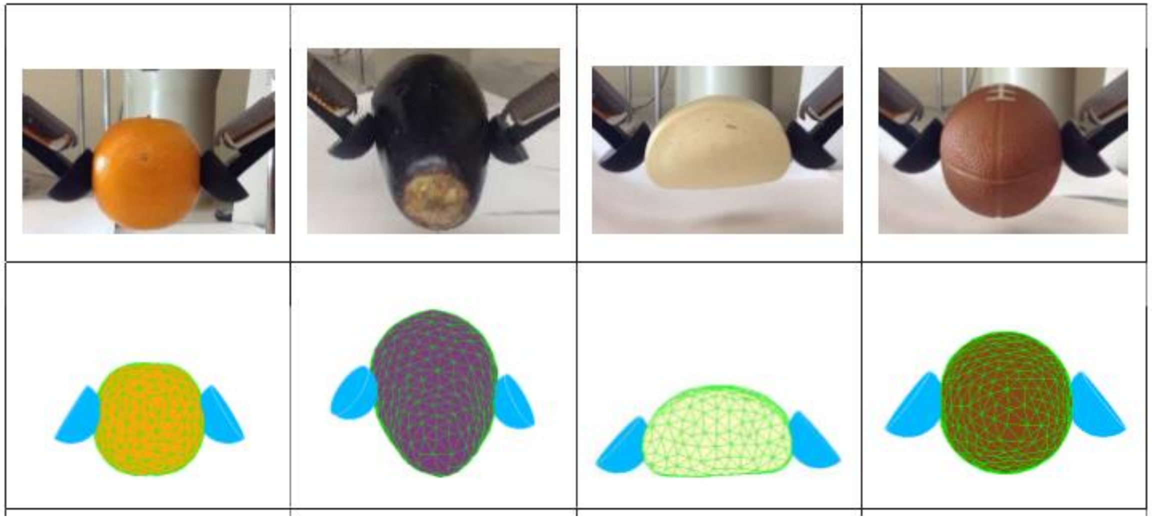
I. Meshing

Modeling with the finite element methods (FEMs)

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



I. Meshing



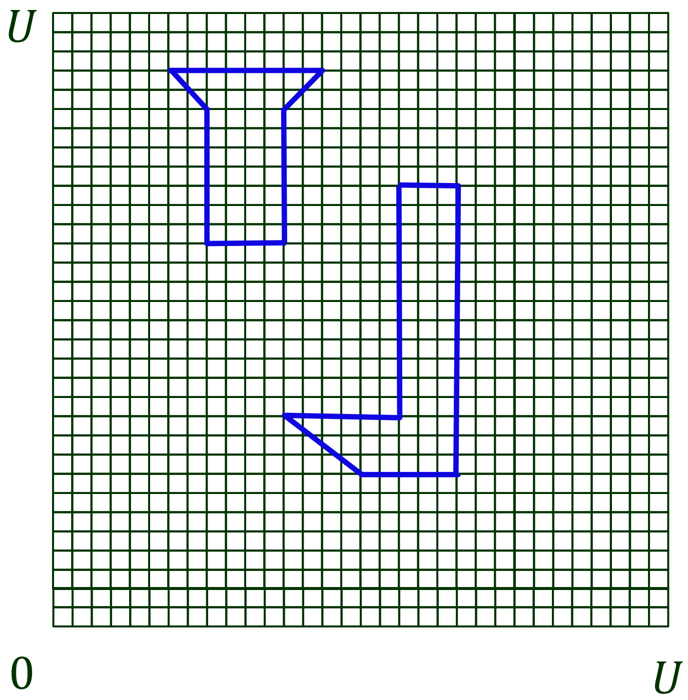
Modeling with the finite element methods (FEMs)

- 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

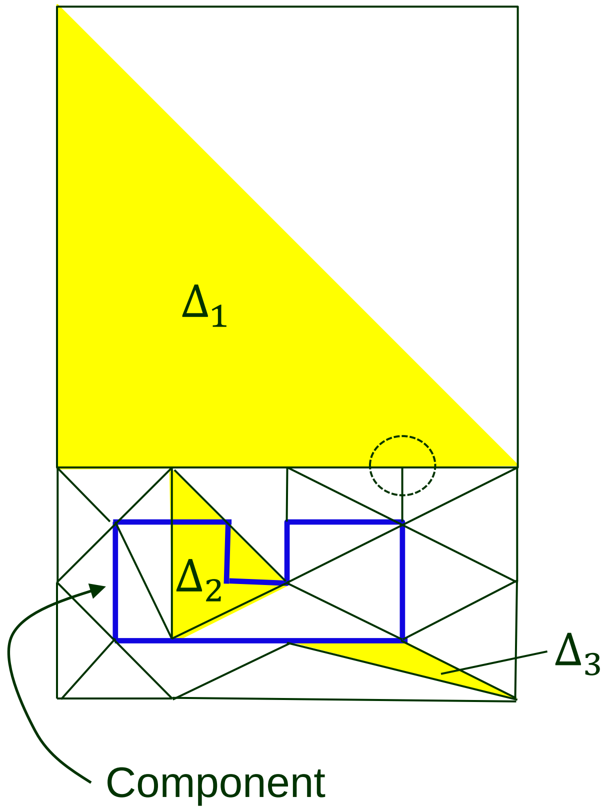
Compute a triangular mesh of the square that is

- *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)

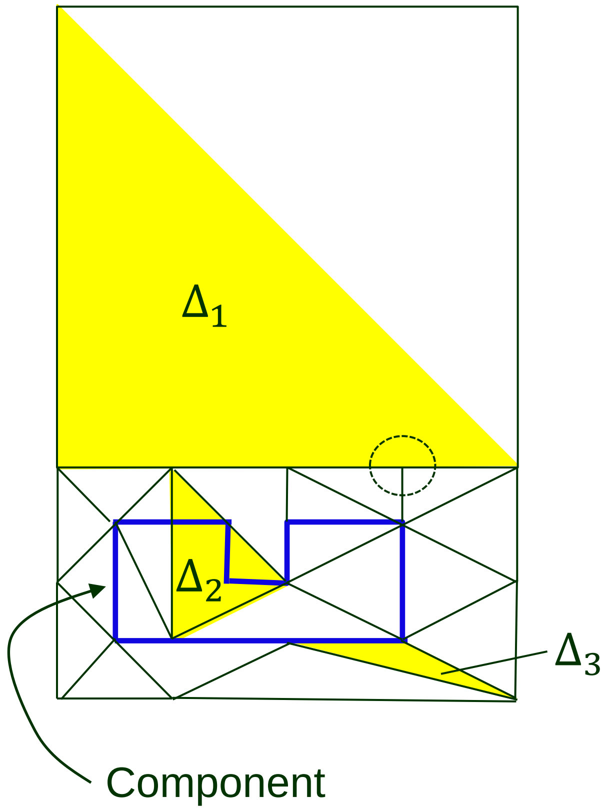


Triangulation Task

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 Δ_1



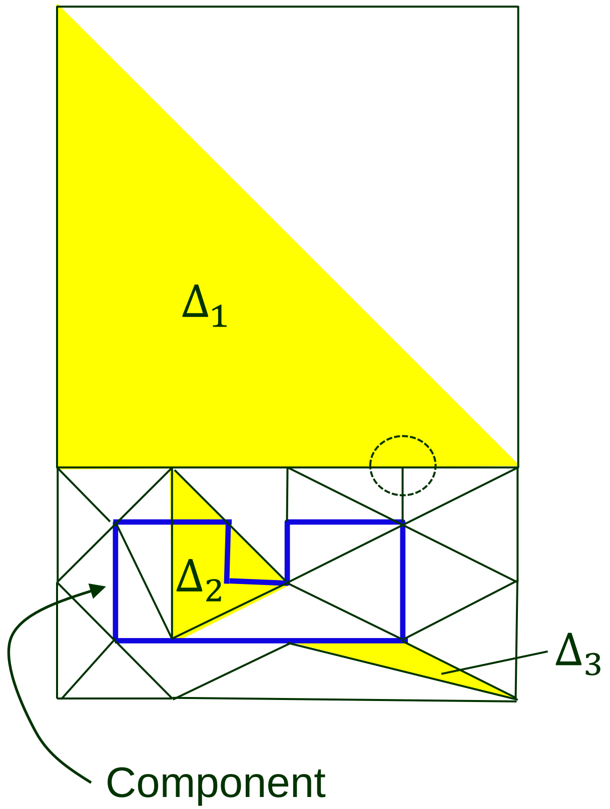
Triangulation Task

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 Δ_1

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



Triangulation Task

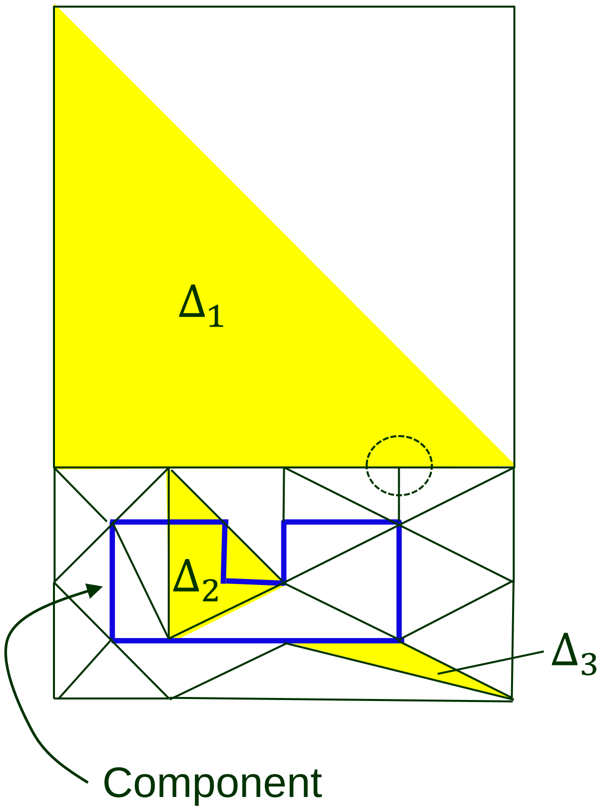
Compute a triangular mesh of the square that is

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

  violated by Δ_1

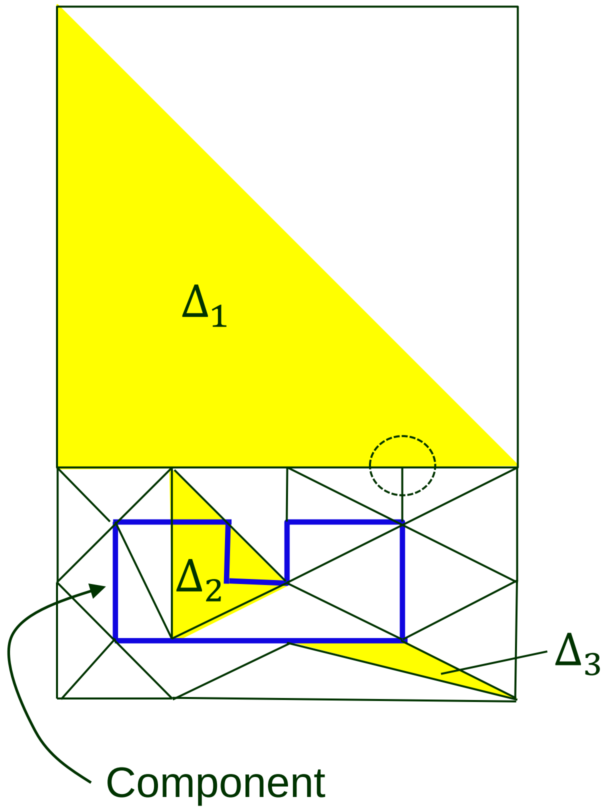
- *Input respecting* (component edges must be contained in the union of the edges of the mesh triangles)

  violated by Δ_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 Δ_1

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

  violated by Δ_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)

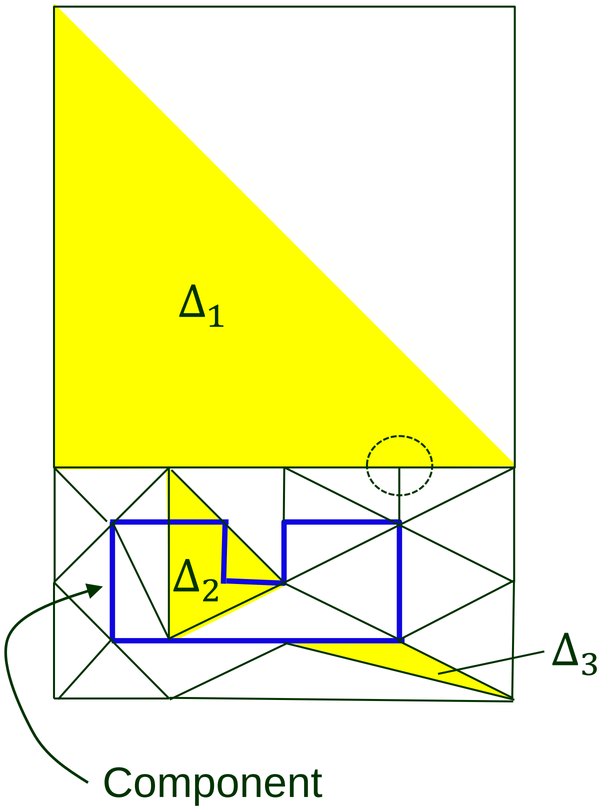
  violated by Δ_1

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

  violated by Δ_2

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

  violated by Δ_3



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 Δ_1

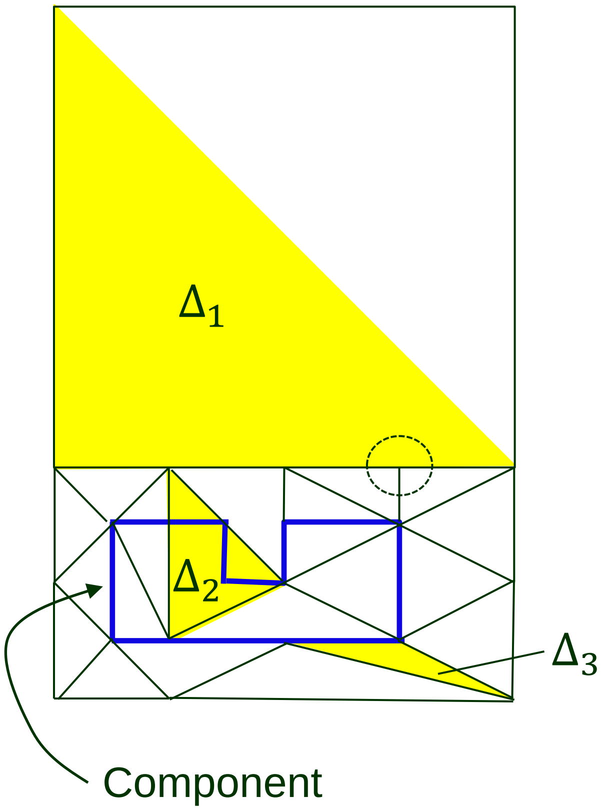
- *Input respecting* (component edges must be contained in the union of the edges of the mesh triangles)

  violated by Δ_2

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

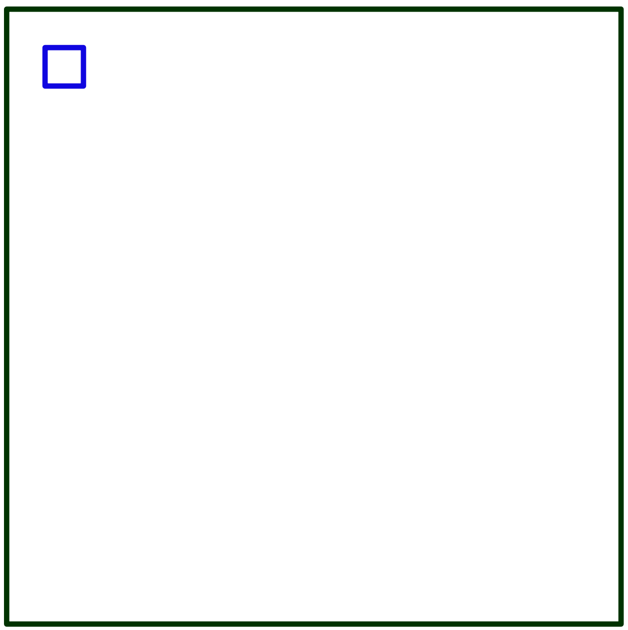
  violated by Δ_3

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



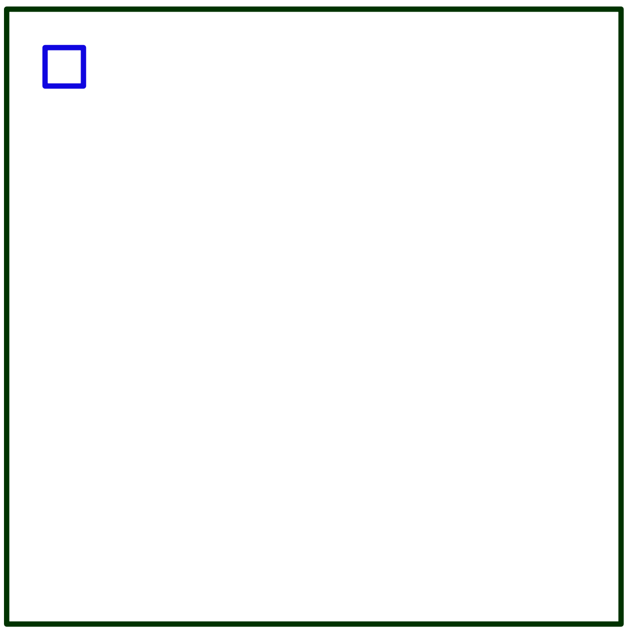
Steiner Triangulation

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



Steiner Triangulation

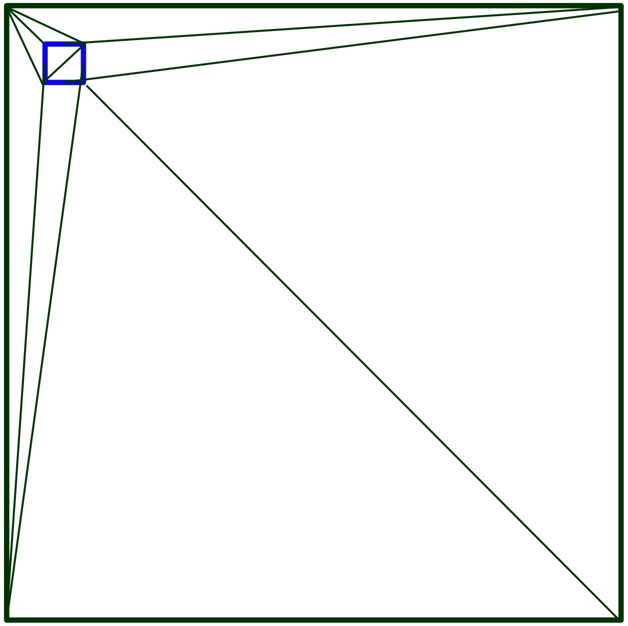
1×1 component located at $(1,1)$ in a 16×16 grid.



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

Steiner Triangulation

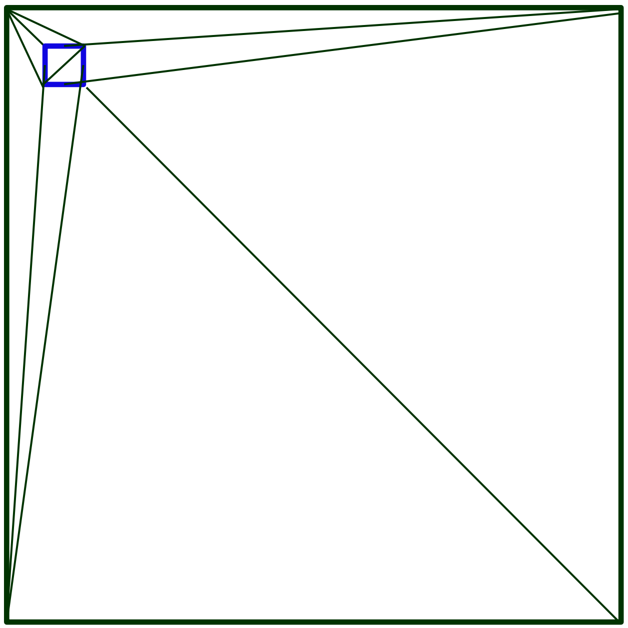
1×1 component located at $(1,1)$ in a 16×16 grid.



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.

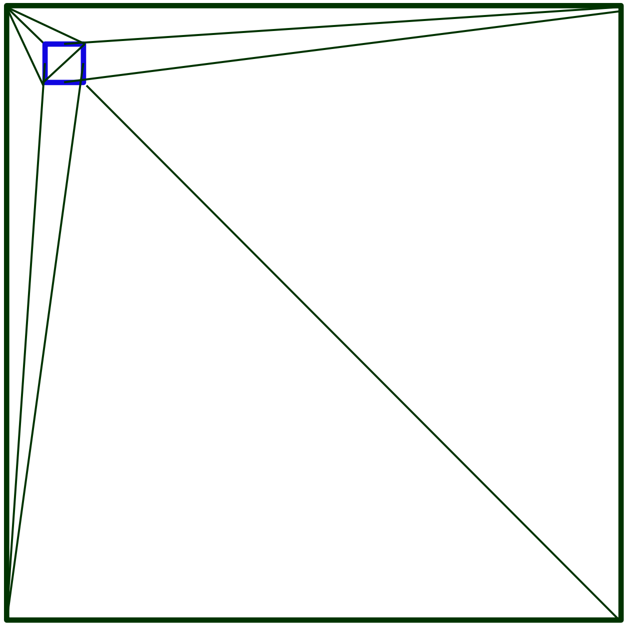


Delaunay triangulation of the eight vertices of the square component and grid's bounding box.

Angles too small!

Steiner Triangulation

1×1 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.

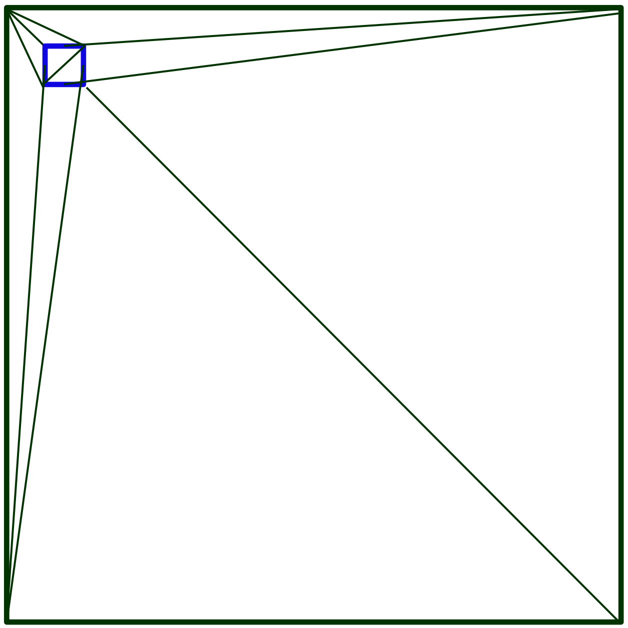
Angles too small!

Solution:

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

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.

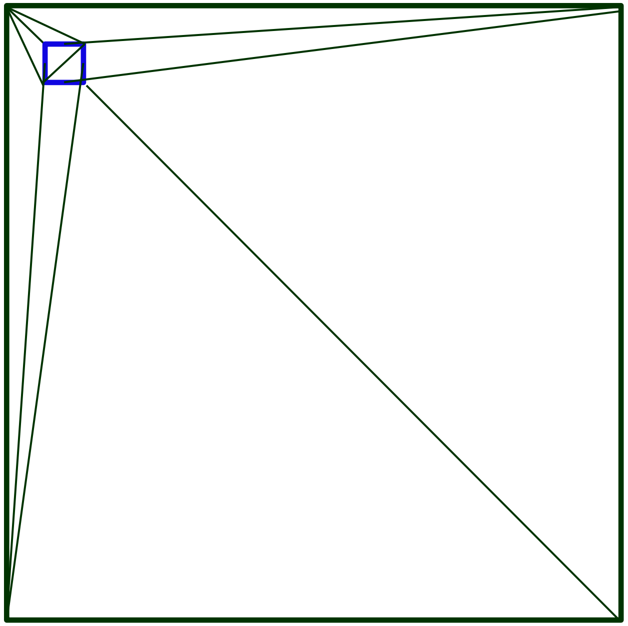
Angles too small!

Solution:

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

Steiner Triangulation

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

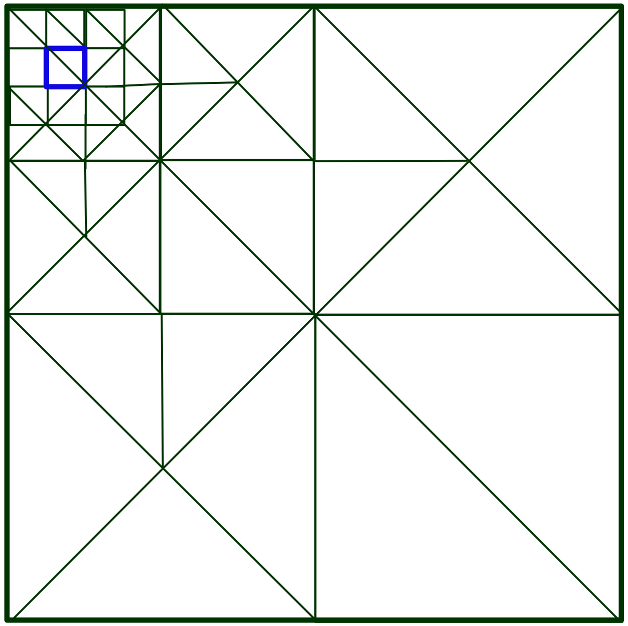


Delaunay triangulation of the eight vertices of the square component and grid's bounding box.

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.

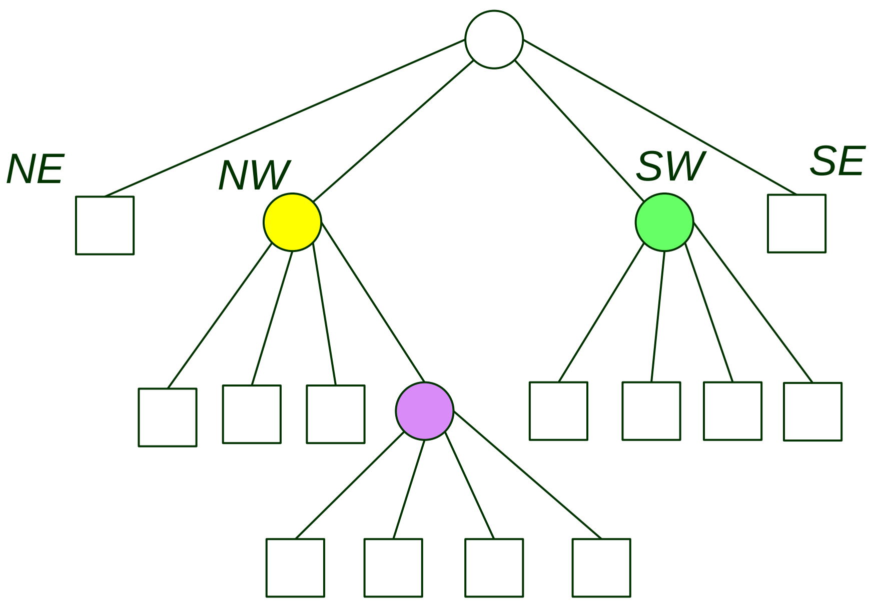


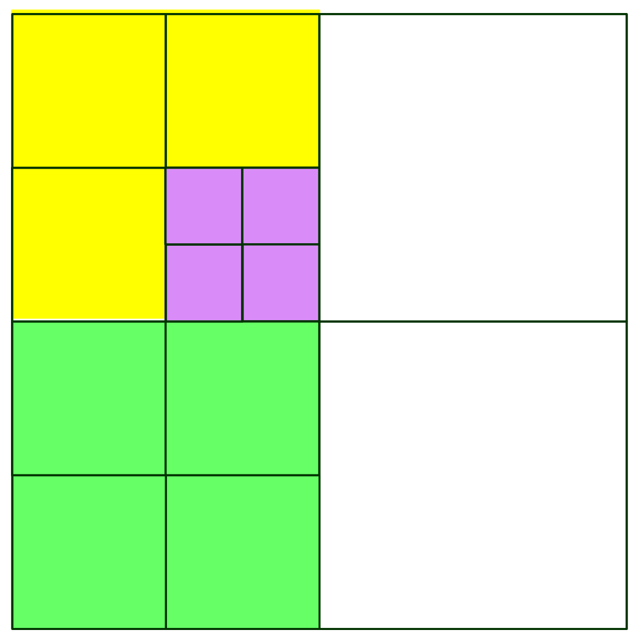
52 triangles (down from 512 uniform triangles)

II. Quadtree

A rooted tree in which

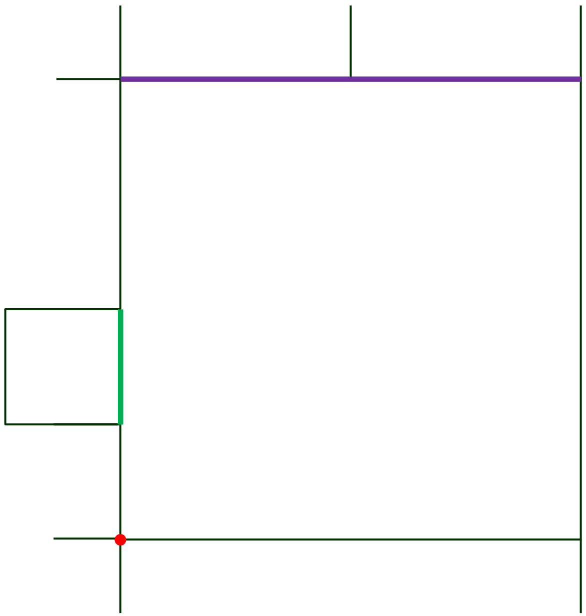
- every node corresponds to a square;
- every internal node v has four children which represent the four quadrants of the node's corresponding square.



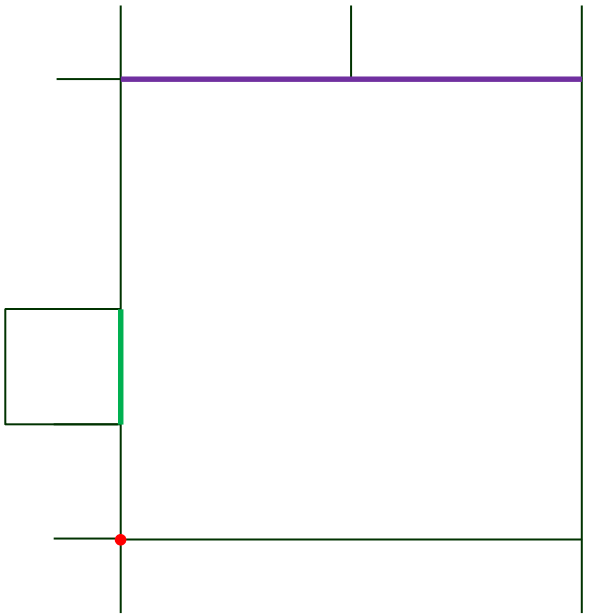


Quadtree subdivision

More Terminology

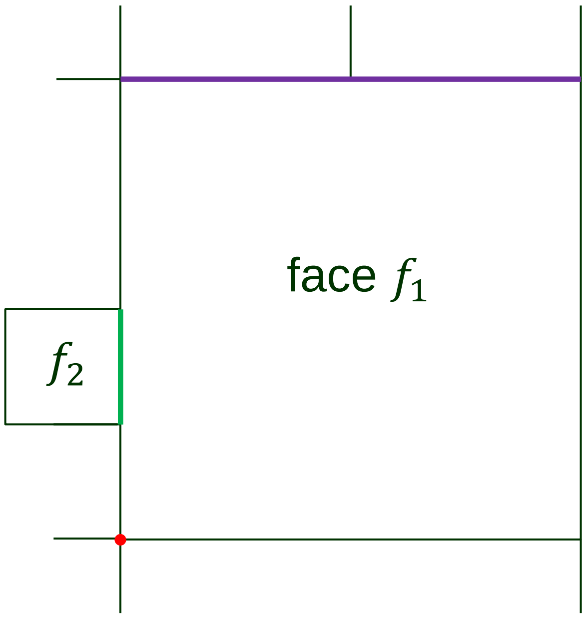


More Terminology



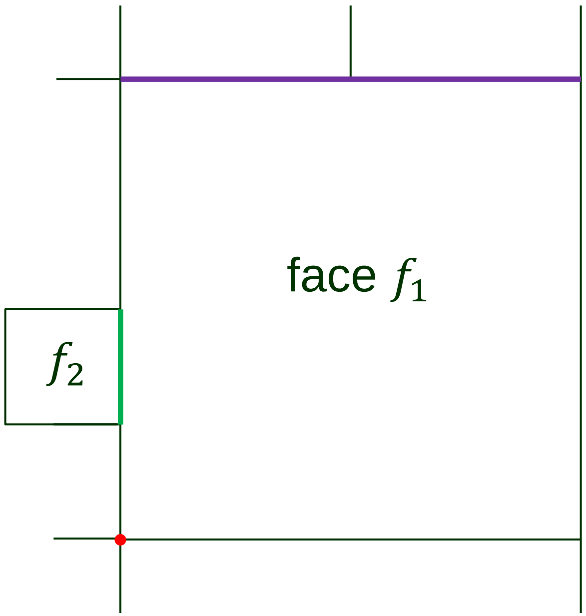
Face: a square in the quadtree subdivision (although it may have ≥ 4 vertices).

More Terminology



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

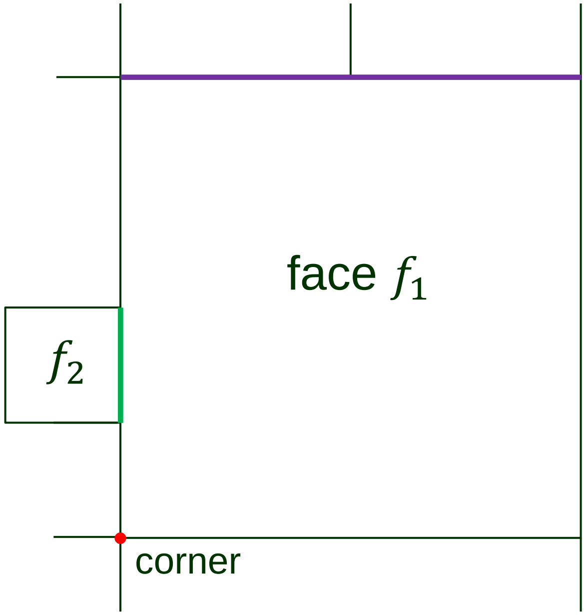
More Terminology



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

Corner: a vertex at the corner of the square.

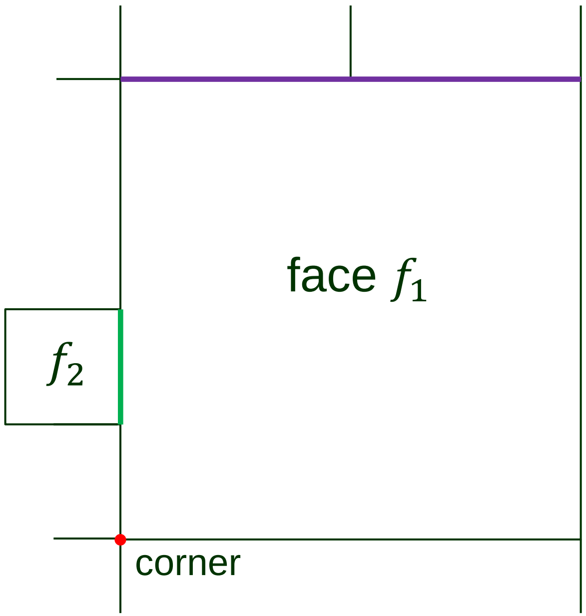
More Terminology



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

Corner: a vertex at the corner of the square.

More Terminology

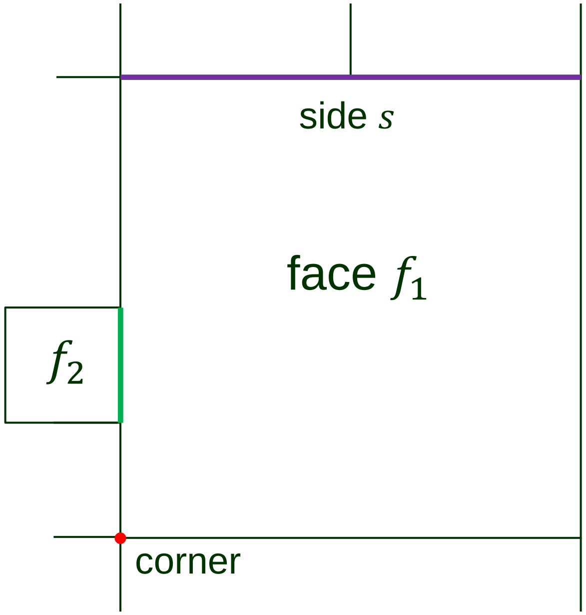


Face: a square in the quadtree subdivision (although it may have ≥ 4 vertices).

Corner: a vertex at the corner of the square.

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

More Terminology

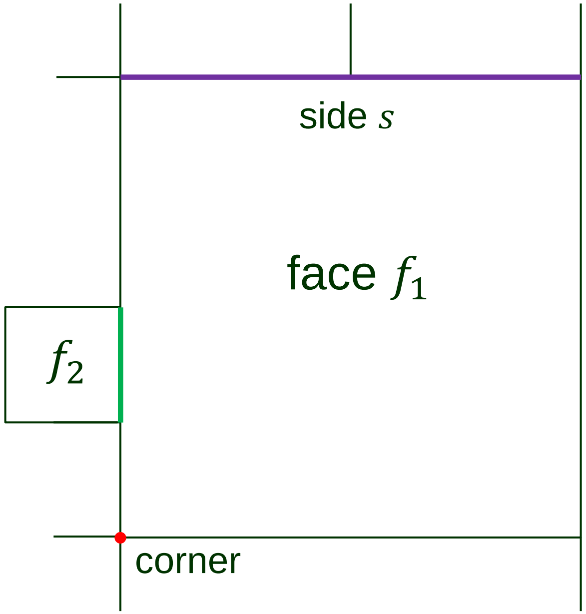


Face: a square in the quadtree subdivision (although it may have ≥ 4 vertices).

Corner: a vertex at the corner of the square.

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

More Terminology



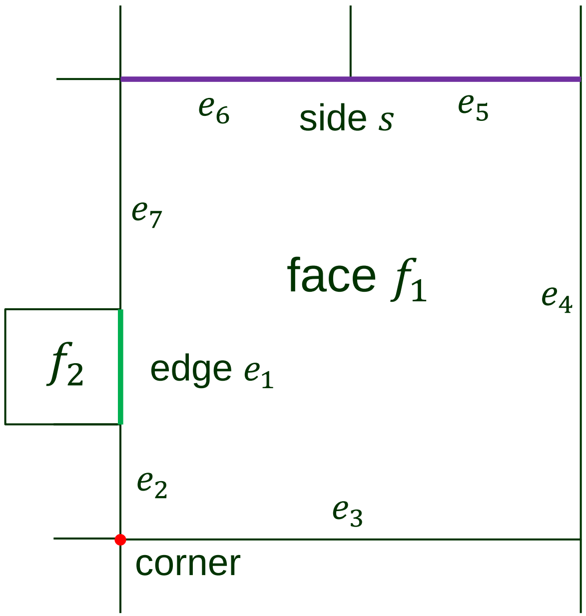
Face: a square in the quadtree subdivision (although it may have ≥ 4 vertices).

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



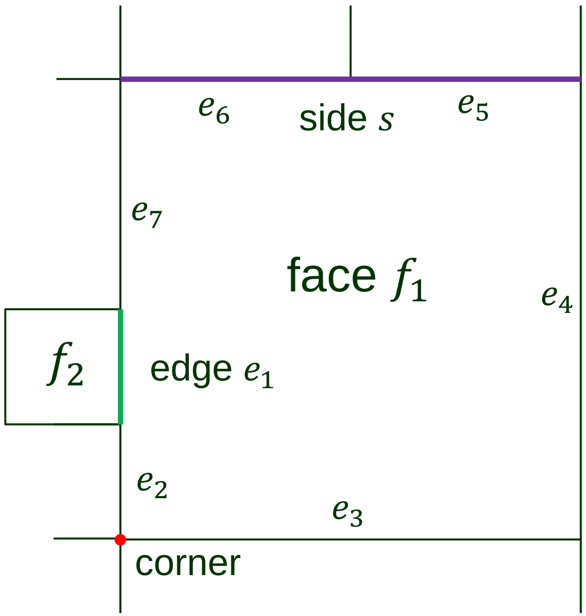
Face: a square in the quadtree subdivision (although it may have ≥ 4 vertices).

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 ≥ 4 vertices).

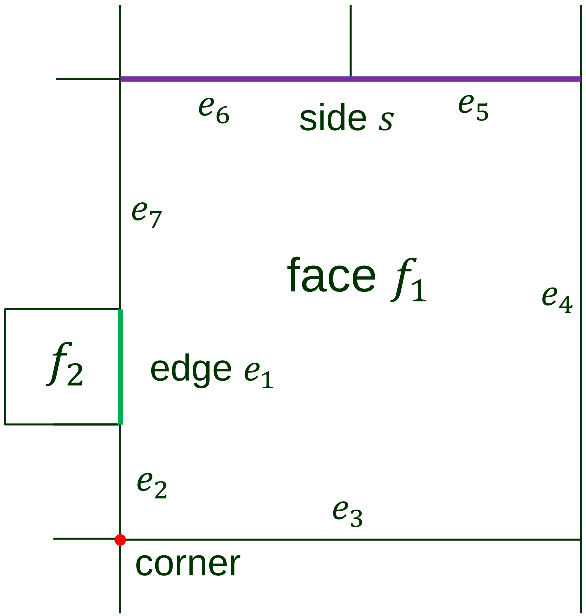
Corner: a vertex at the corner of the square.

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 ≥ 1 edges.

More Terminology



Face: a square in the quadtree subdivision (although it may have ≥ 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 ≥ 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 σ.

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 σ.

- Otherwise, let

$$x_{mid} = \frac{x_\sigma + x'_\sigma}{2} \qquad y_{mid} = \frac{y_\sigma + y'_\sigma}{2}$$

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 σ.

- Otherwise, let

$$x_{mid} = \frac{x_\sigma + x'_\sigma}{2} \qquad y_{mid} = \frac{y_\sigma + y'_\sigma}{2}$$

  and define

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

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 σ.
- Otherwise, let

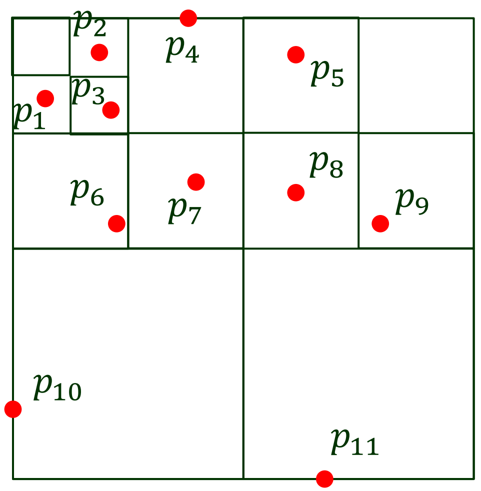
$$x_{mid} = \frac{x_\sigma + x'_\sigma}{2} \qquad y_{mid} = \frac{y_\sigma + y'_\sigma}{2}$$

and define

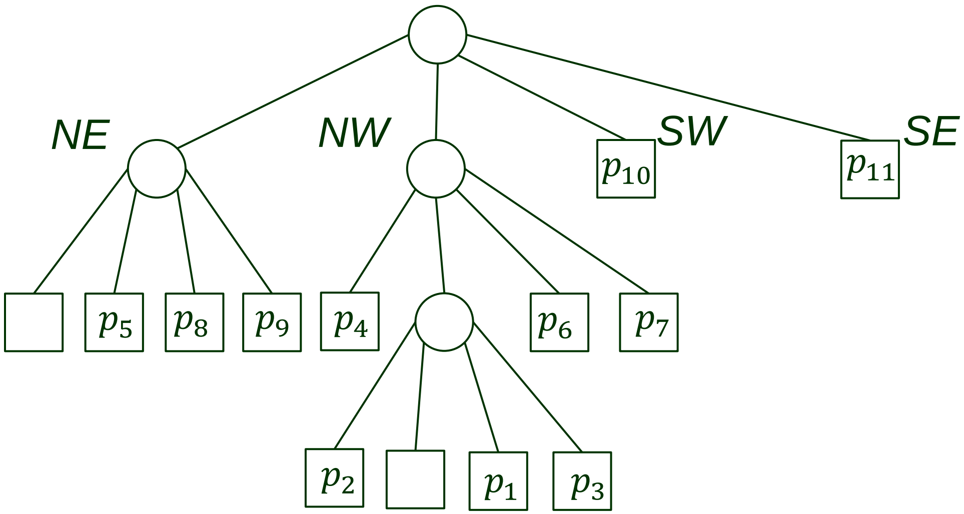
$$P_{NE} = \{p \in P \mid p_x > x_{mid} \text{ and } p_y > y_{mid}\}$$
$$P_{NW} = \{p \in P \mid p_x \leq x_{mid} \text{ and } p_y > y_{mid}\}$$
$$P_{SW} = \{p \in P \mid p_x \leq x_{mid} \text{ and } p_y \leq y_{mid}\}$$
$$P_{SE} = \{p \in P \mid p_x > x_{mid} \text{ and } p_y \leq y_{mid}\}$$

Store P_{NE}, P_{NW}, P_{SW}, P_{SE} in the four quadrants σ_{NE}, σ_{NW}, σ_{SW}, σ_{SE} of σ, respectively.

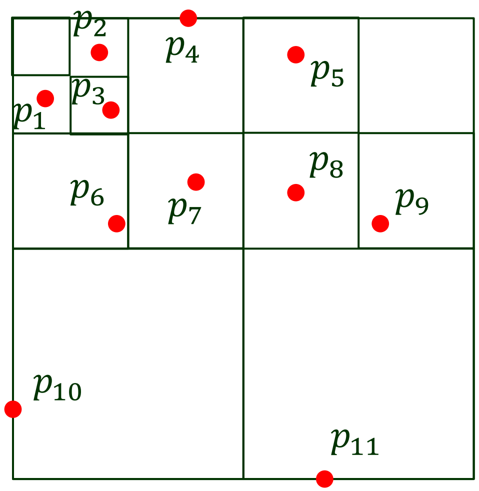
Recursive Construction



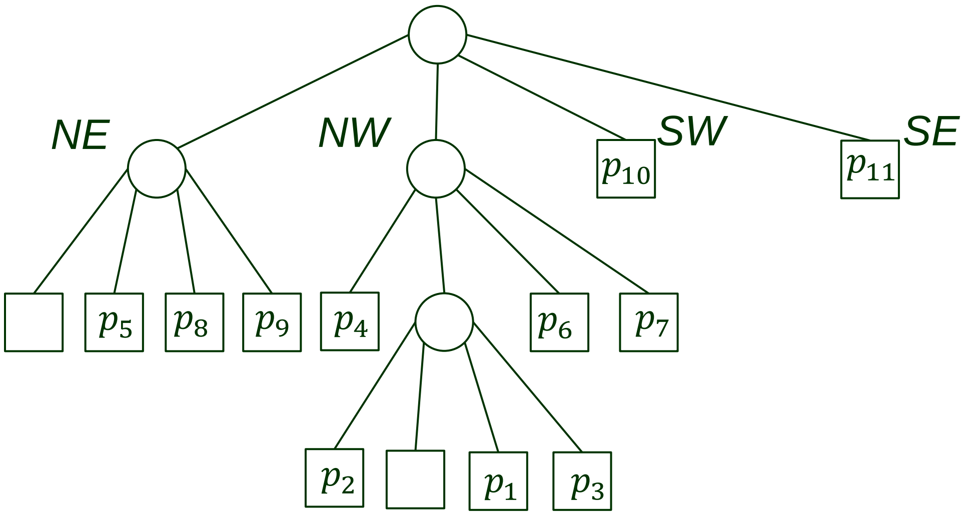
- Compute the smallest enclosing square for the point set P.



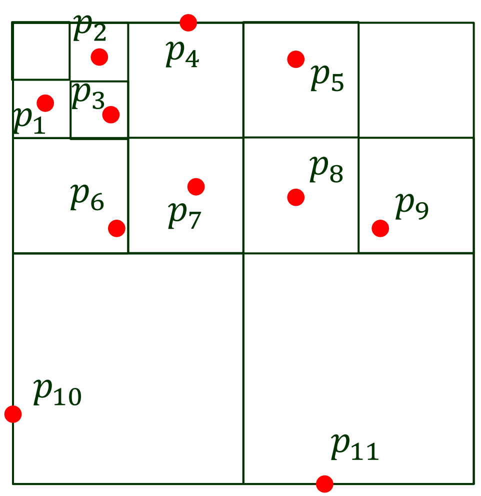
Recursive Construction



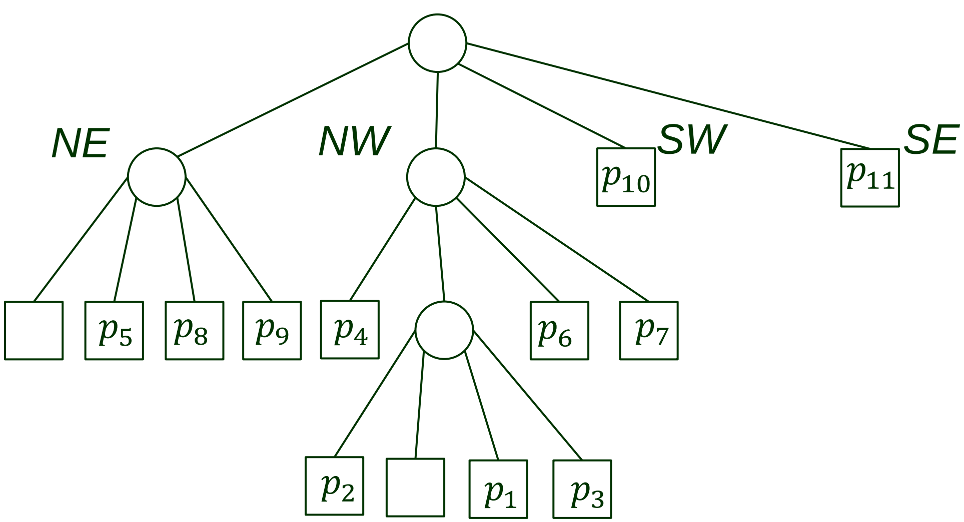
- Compute the smallest enclosing square for the point set P.
- Split the square into four quadrants: NE, NW, SW, and SE.



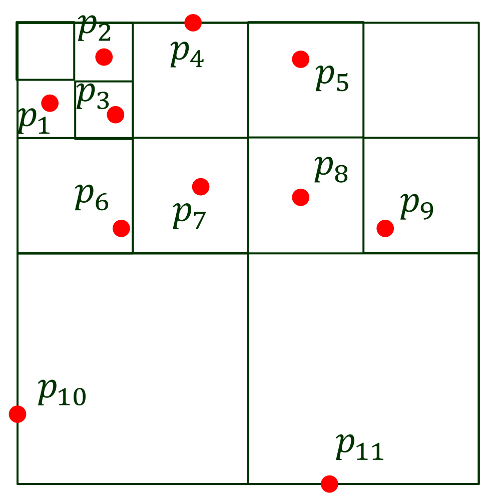
Recursive Construction



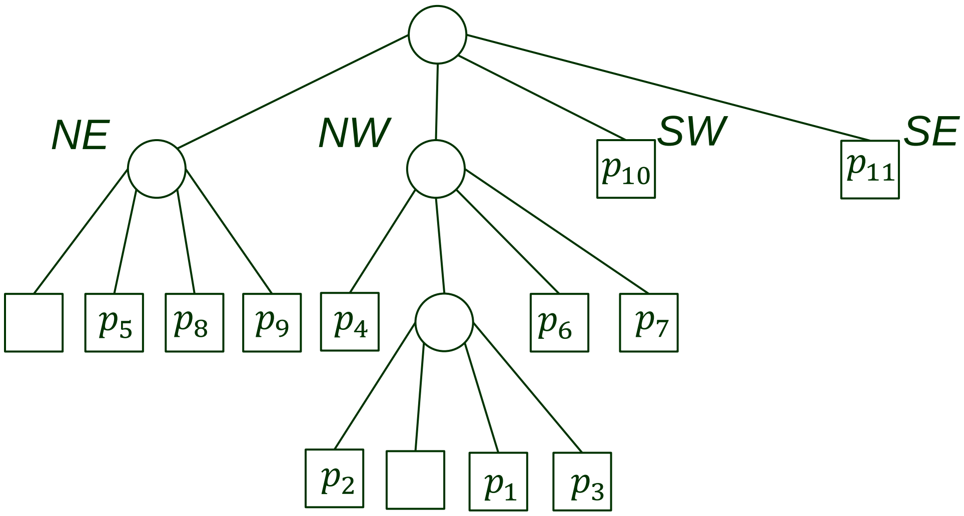
- Compute the smallest enclosing square for the point set P.
- Split the square into four quadrants: NE, NW, SW, and SE.
- Partition P accordingly into four subsets, one for each quadrant.



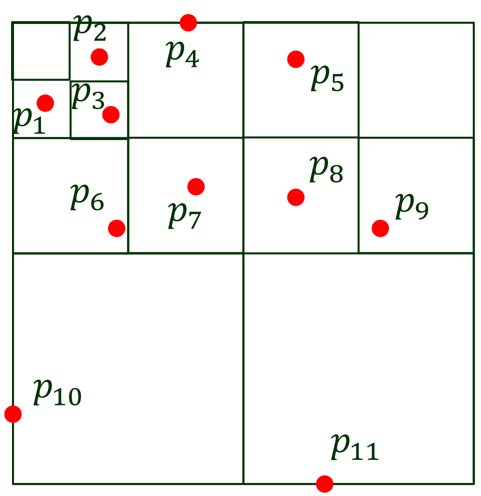
Recursive Construction



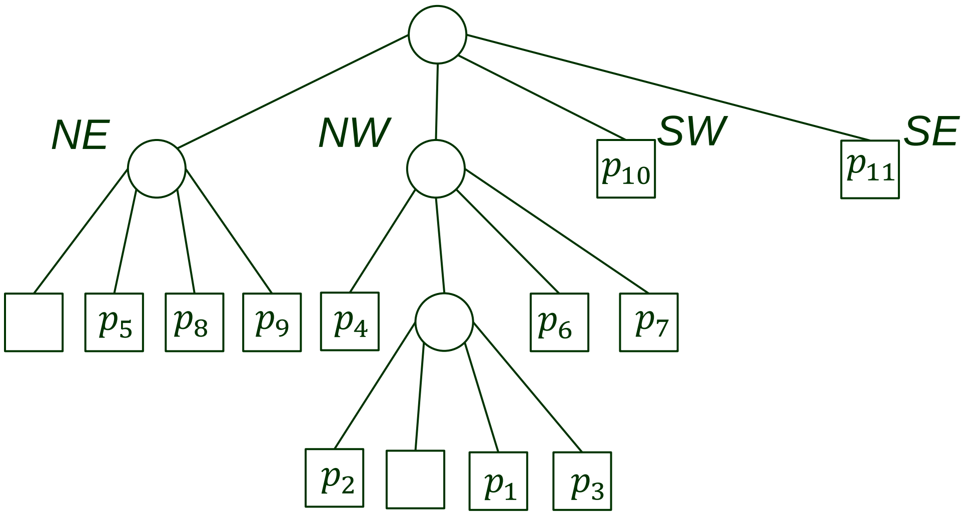
- Compute the smallest enclosing square for the point set P.
- Split the square into four quadrants: NE, NW, SW, and SE.
- Partition P accordingly into four subsets, one for each quadrant.
- Recursively construct quadtrees for each quadrant.



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 ≤ 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 ≥ 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 ≥ 2 points.

$\Downarrow$

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

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 ≥ 2 points.

$\Downarrow$

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

$\Downarrow$

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

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 ≥ 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.

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 ≥ 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 ≥ 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 $s/2^i$.

$\Longrightarrow$ $s/2^i \geq c/\sqrt{2}$

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 ≥ 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 $s/2^i$.

$\Longrightarrow s/2^i \geq c/\sqrt{2}$

⇓

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

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 ≥ 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 $s/2^i$.

$\Longrightarrow$ $s/2^i \geq c/\sqrt{2}$

$\Downarrow$

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

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

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 ≥ 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 $s/2^i$.

$\Longrightarrow$ $s/2^i \geq c/\sqrt{2}$

$\Downarrow$

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

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

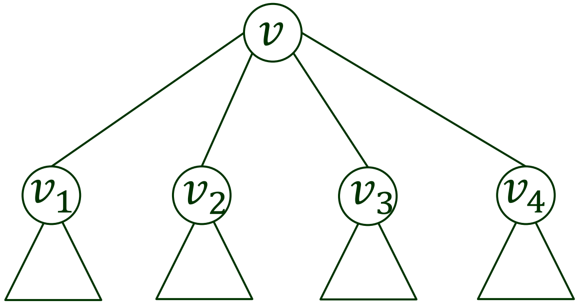
Number of Nodes

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

Number of Nodes

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

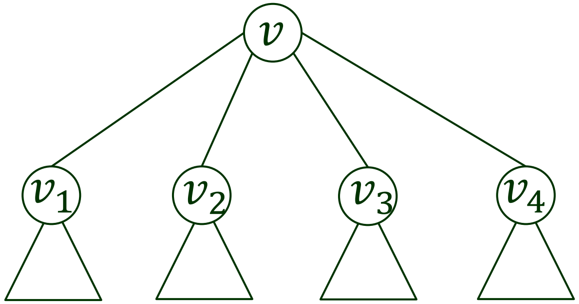


Number of Nodes

Theorem 1 A quadtree of depth d storing P has $O\big((d+1)n\big)$ nodes and can be constructed in $O\big((d+1)n\big)$ 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 $\eta(v) = 1 + 3\mu(v)$.

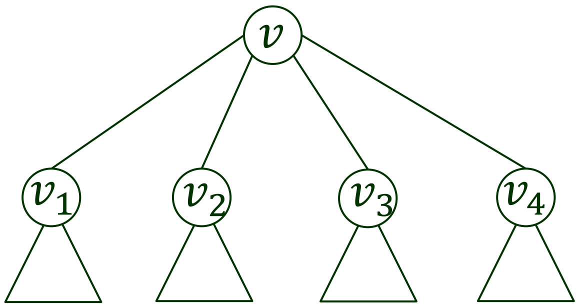


Number of Nodes

Theorem 1 A quadtree of depth d storing P has $O\big((d+1)n\big)$ nodes and can be constructed in $O\big((d+1)n\big)$ 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 $\eta(v) = 1 + 3\mu(v)$.



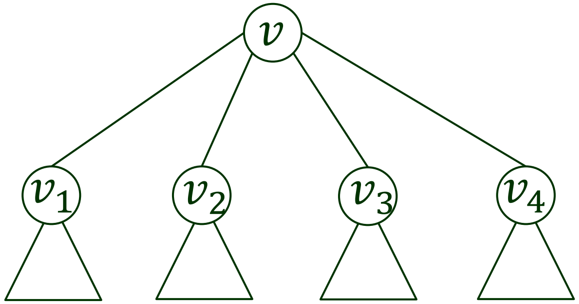
- This is true in the base case where the four children v_1, v_2, v_3, v_4 of v are leaves, i.e., $\mu(v) = 1$.

Number of Nodes

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

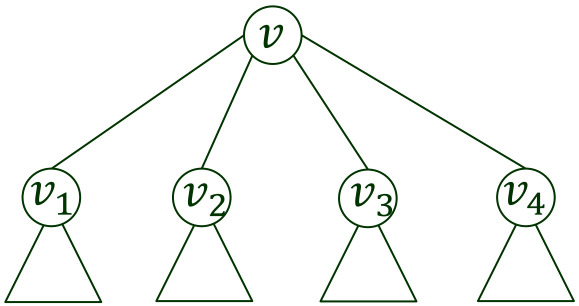
$$\eta(v) = \eta(v_1) + \eta(v_2) + \eta(v_3) + \eta(v_4)$$

Number of Nodes

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

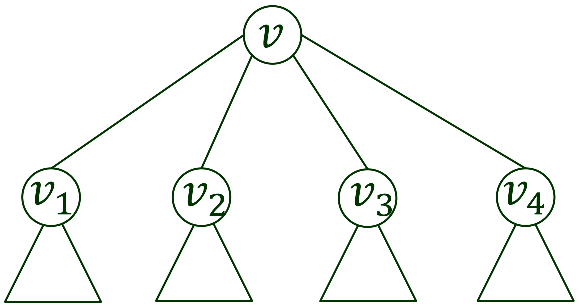
$$\eta(v) = \eta(v_1) + \eta(v_2) + \eta(v_3) + \eta(v_4)$$
$$= 3\big(\mu(v_1) + \mu(v_2) + \mu(v_3) + \mu(v_4)\big) + 4$$

Number of Nodes

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

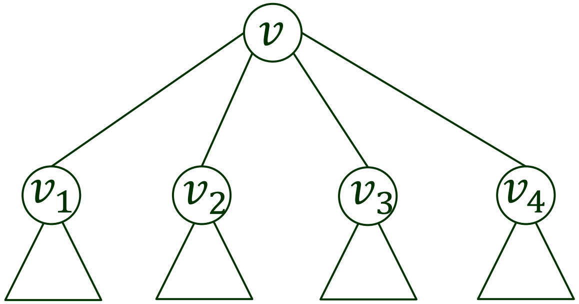
$$
\begin{aligned}
\eta(v) &= \eta(v_1) + \eta(v_2) + \eta(v_3) + \eta(v_4) \\
&= 3\big(\mu(v_1) + \mu(v_2) + \mu(v_3) + \mu(v_4)\big) + 4 \\
&= 3(\mu(v) - 1) + 4
\end{aligned}
$$

Number of Nodes

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

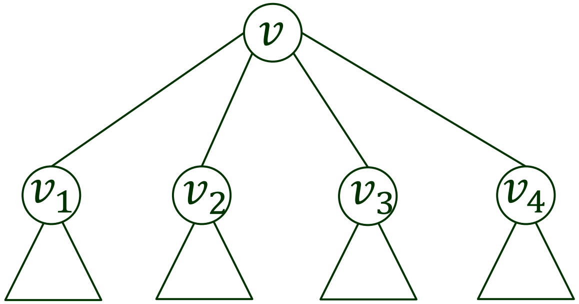
$$\begin{aligned}
\eta(v) &= \eta(v_1) + \eta(v_2) + \eta(v_3) + \eta(v_4) \\
&= 3\big(\mu(v_1) + \mu(v_2) + \mu(v_3) + \mu(v_4)\big) + 4 \\
&= 3(\mu(v) - 1) + 4 \\
&= 3\mu(v) + 1
\end{aligned}$$

Number of Nodes

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

$$\begin{aligned}\eta(v) &= \eta(v_1) + \eta(v_2) + \eta(v_3) + \eta(v_4) \\ &= 3\big(\mu(v_1) + \mu(v_2) + \mu(v_3) + \mu(v_4)\big) + 4 \\ &= 3(\mu(v) - 1) + 4 \\ &= 3\mu(v) + 1\end{aligned}$$

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

Proof (cont'd)

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

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.

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.

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.

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

⇓

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

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.

$\Downarrow$

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

$\Downarrow$

The quadtree has $O\big((d+1)n\big)$ 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 ≥ 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 $O\big((d+1)n\big)$ internal nodes in the tree.
(We include 1 in the Big-O in case $d = 0$.)

⇓

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

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

⇓

The quadtree has $O\big((d+1)n\big)$ 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 $O\big((d+1)n\big)$ internal nodes in the tree.
(We include 1 in the Big-O in case $d = 0$.)

⇓

The quadtree has $O\big((d+1)n\big)$ 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 ≥ 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 $O\big((d+1)n\big)$ internal nodes in the tree.
(We include 1 in the Big-O in case $d = 0$.)

$\Downarrow$

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

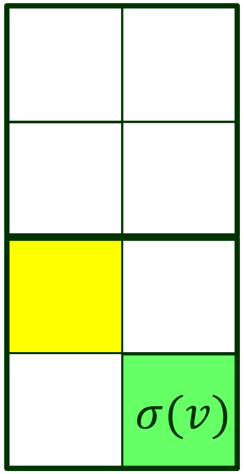
$\Downarrow$

Total construction work $O((d+1)n)$.

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).

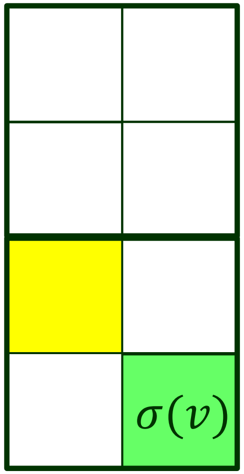


IV. Neighbor Finding

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.



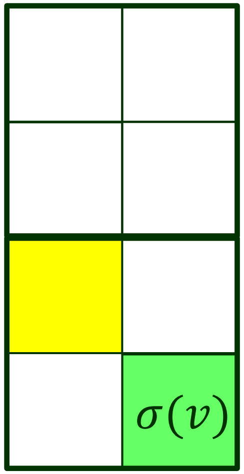
IV. Neighbor Finding

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.

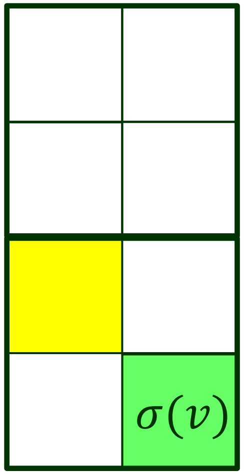
- v is the SE or SW-child of its parent.



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.

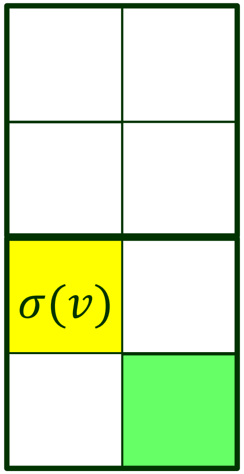
⇩

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.

⇩

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

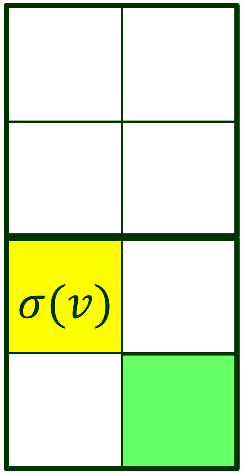
- v is the NE or NW-child of its parent.

Recursively finds the north neighbor, μ, of the parent node.

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.

  ⇩

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

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

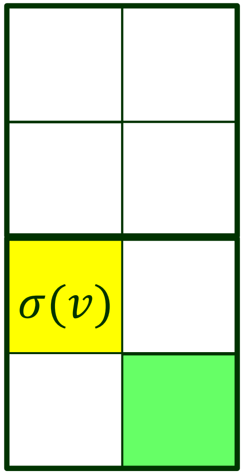
  Recursively finds the north neighbor, μ, of the parent node.

  ♣ If μ 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 $\le$ 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.

  ⇩

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

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

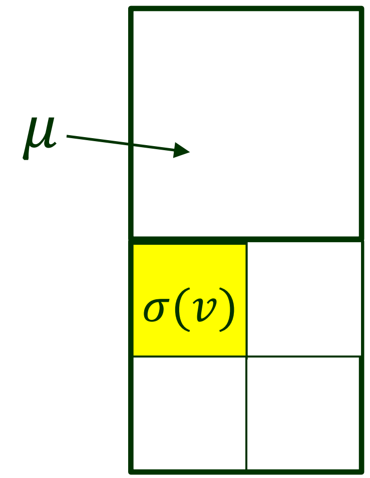
  Recursively finds the north neighbor, μ, of the parent node.

  ♣ If μ is an internal node, then the neighbor of v is its SE or SW-child.

  ♣ If μ is a leaf, then it is the neighbor we are looking for.

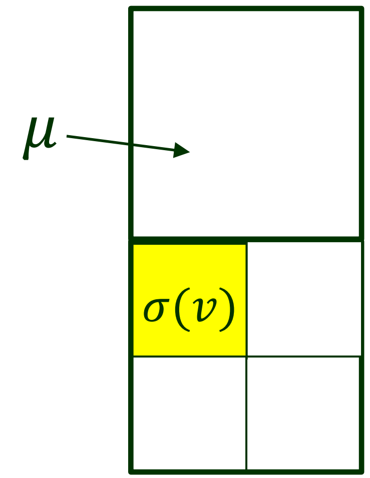
Searching for the North Neighbor

```
NorthNeighbor(v,𝒯)

if v = root(𝒯)
     then return nil
 u ← parent(v)
 if v is the SW-child of u
     then return the NW-child of u
 if v is the SE-child of u
     then return the NE-child of u
 μ ← NorthNeighbor(u, 𝒯)
 if μ =nil or μ is a leaf
     then return μ
     else if v is the NW-child of u
          then return the SW-child of μ
          else return the SE-child of μ
```



Searching for the North Neighbor

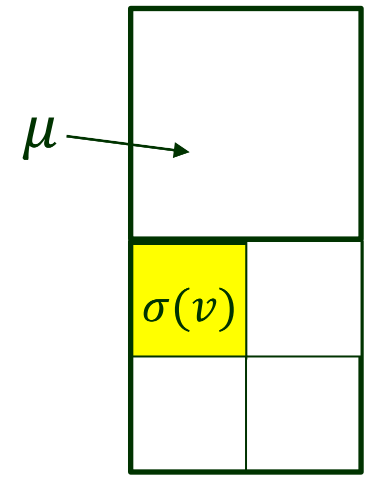
```
NorthNeighbor(v,𝒯)

if v = root(𝒯)
    then return nil
u ← parent(v)
if v is the SW-child of u
    then return the NW-child of u
if v is the SE-child of u
    then return the NE-child of u
μ ← NorthNeighbor(u, 𝒯)
if μ =nil or μ is a leaf
    then return μ
    else if v is the NW-child of u
        then return the SW-child of μ
        else return the SE-child of μ
```



♣ The call does not necessarily return a leaf node.

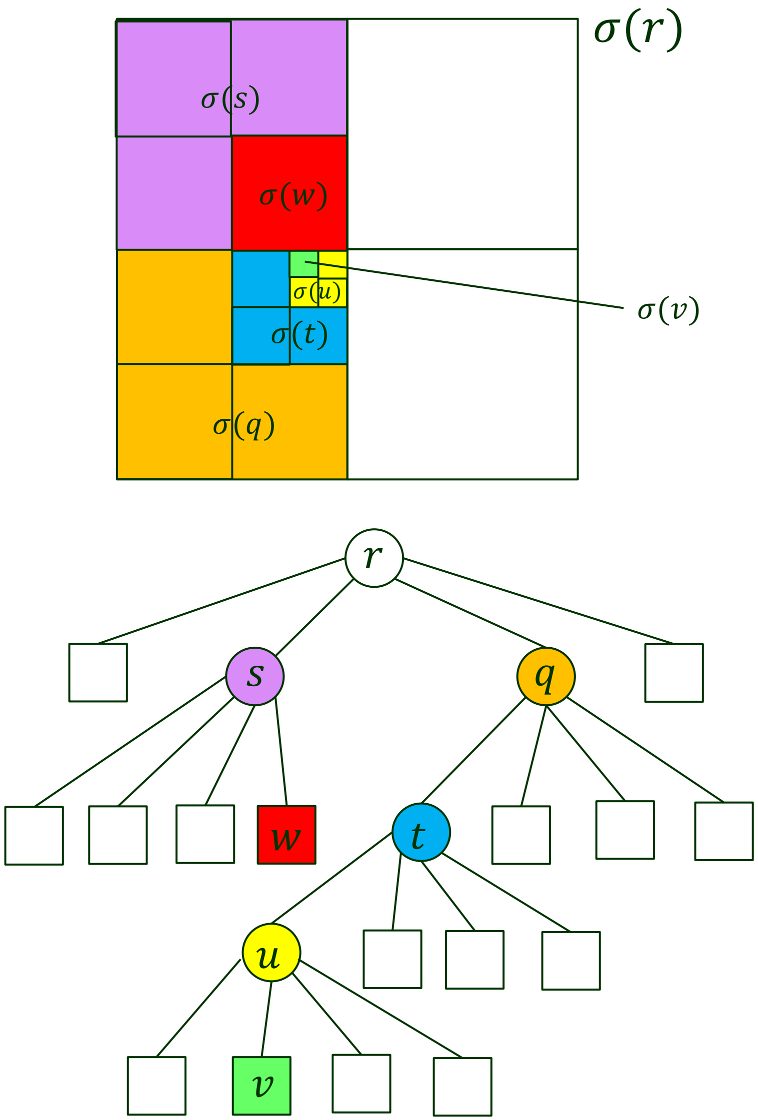
Searching for the North Neighbor

```
NorthNeighbor(v,𝒯)

if v = root(𝒯)
     then return nil
u ← parent(v)
if v is the SW-child of u
     then return the NW-child of u
if v is the SE-child of u
     then return the NE-child of u
μ ← NorthNeighbor(u, 𝒯)
if μ =nil or μ is a leaf
     then return μ
     else if v is the NW-child of u
          then return the SW-child of μ
          else return the SE-child of μ
```



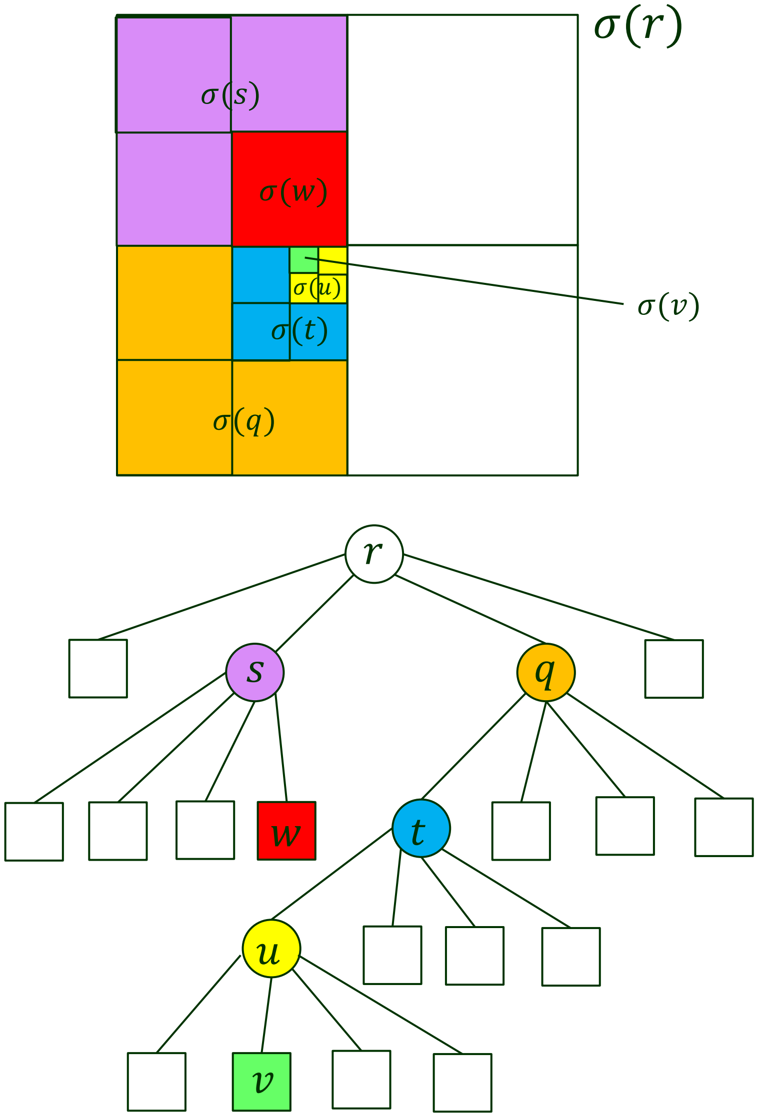
- ♣ The call does not necessarily return a leaf node.
- ♣ To find one, needs to walk down the quadtree from the returned node, always proceeding to a south-child.

A Bigger Example




```
NorthNeighbor(v,𝒯)

if v = root(𝒯)
    then return nil
u ← parent(v)
if v is the SW-child of u
    then return the NW-child of u
if v is the SE-child of u
    then return the NE-child of u
μ ← NorthNeighbor(u, 𝒯)
if μ =nil or μ is a leaf
    then return μ
    else if v is the NW-child of u
        then return the SW-child of μ
        else return the SE-child of μ
```

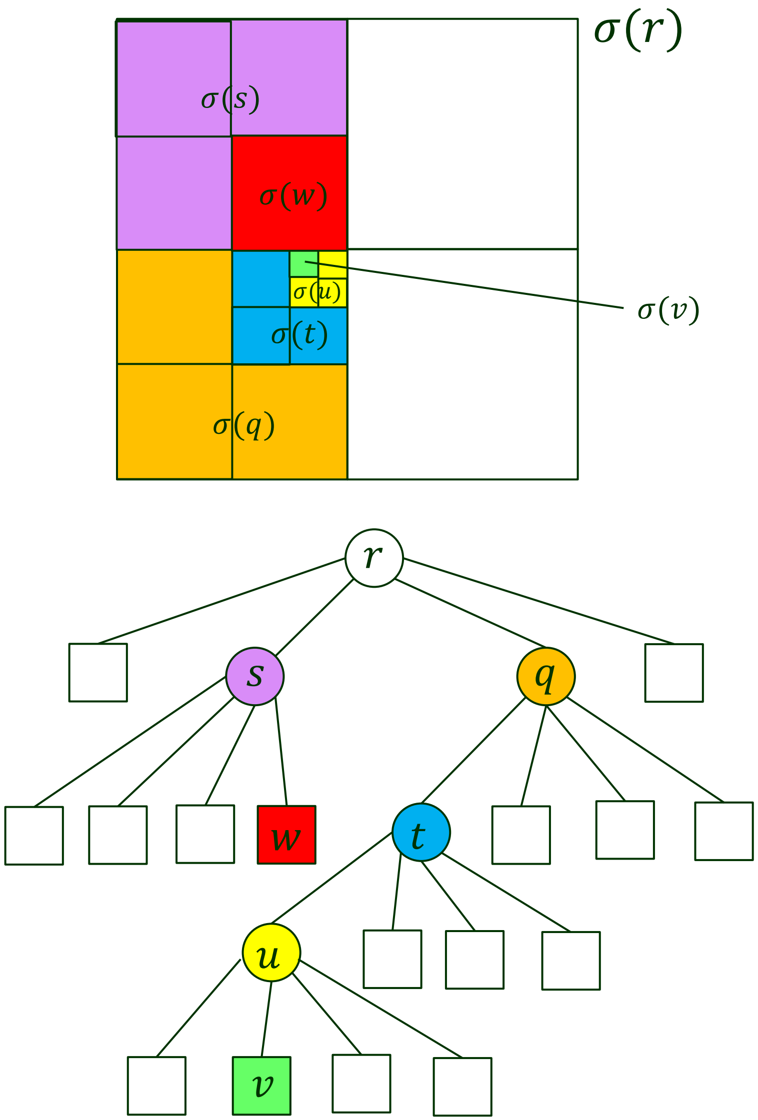
A Bigger Example




```
NorthNeighbor(v,𝒯)

if v = root(𝒯)
    then return nil
u ← parent(v)
if v is the SW-child of u
    then return the NW-child of u
if v is the SE-child of u
    then return the NE-child of u
μ ← NorthNeighbor(u, 𝒯)
if μ =nil or μ is a leaf
    then return μ
    else if v is the NW-child of u
        then return the SW-child of μ
        else return the SE-child of μ
```

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

A Bigger Example



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

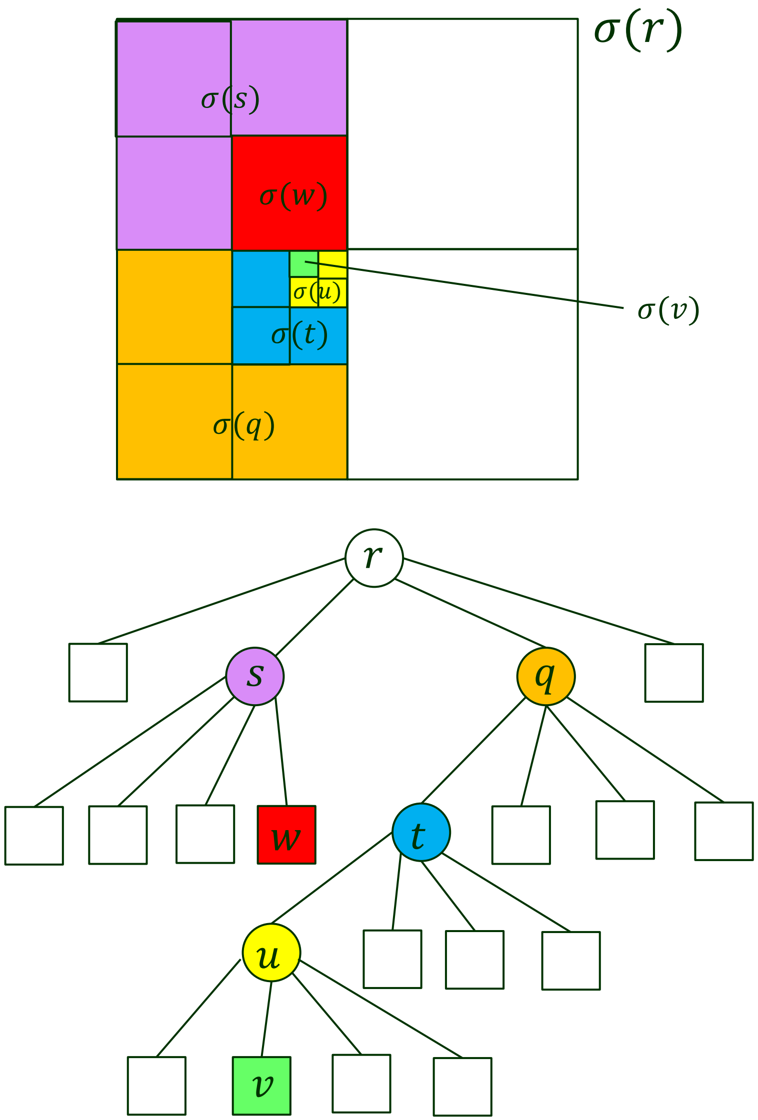
```
if v = root(T)
    then return nil
u ← parent(v)
if v is the SW-child of u
    then return the NW-child of u
if v is the SE-child of u
    then return the NE-child of u
μ ← NorthNeighbor(u, T)
if μ = nil or μ is a leaf
    then return μ
    else if v is the NW-child of u
        then return the SW-child of μ
        else return the SE-child of μ
```

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

|

NorthNeighbor(u,$\mathcal{T}$)

A Bigger Example




```
NorthNeighbor(v,T)

if v = root(T)
    then return nil
u ← parent(v)
if v is the SW-child of u
    then return the NW-child of u
if v is the SE-child of u
    then return the NE-child of u
μ ← NorthNeighbor(u, T)
if μ =nil or μ is a leaf
    then return μ
    else if v is the NW-child of u
        then return the SW-child of μ
        else return the SE-child of μ
```

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

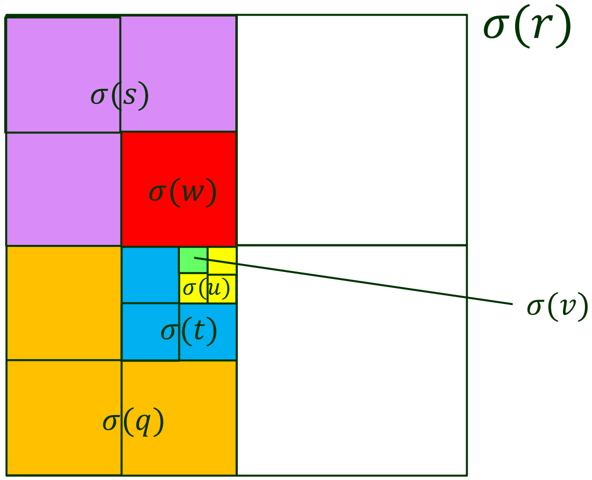
|

NorthNeighbor(u,$\mathcal{T}$)

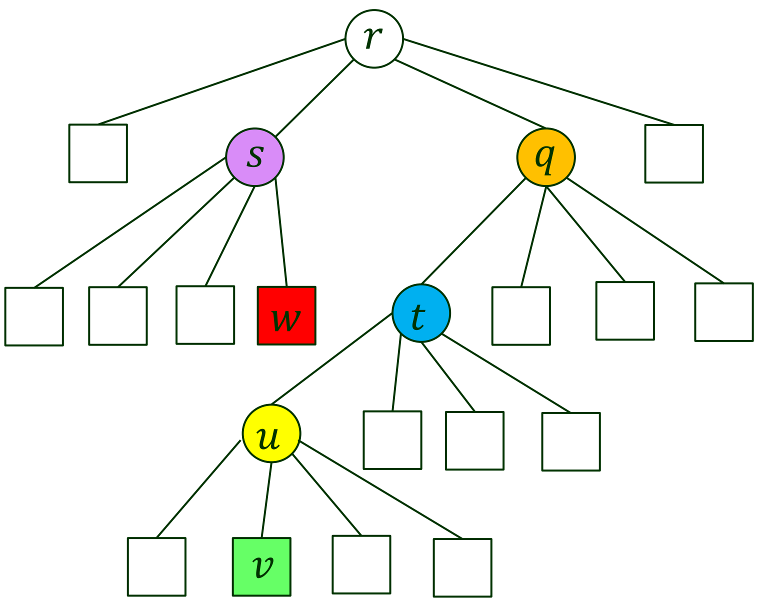
|

NorthNeighbor(t,$\mathcal{T}$)

A Bigger Example




```
NorthNeighbor(v,𝒯)

    if v = root(𝒯)
        then return nil
    u ← parent(v)
    if v is the SW-child of u
        then return the NW-child of u
    if v is the SE-child of u
        then return the NE-child of u
    μ ← NorthNeighbor(u, 𝒯)
    if μ =nil or μ is a leaf
        then return μ
        else if v is the NW-child of u
            then return the SW-child of μ
            else return the SE-child of μ
```



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

|

NorthNeighbor(u,$\mathcal{T}$)

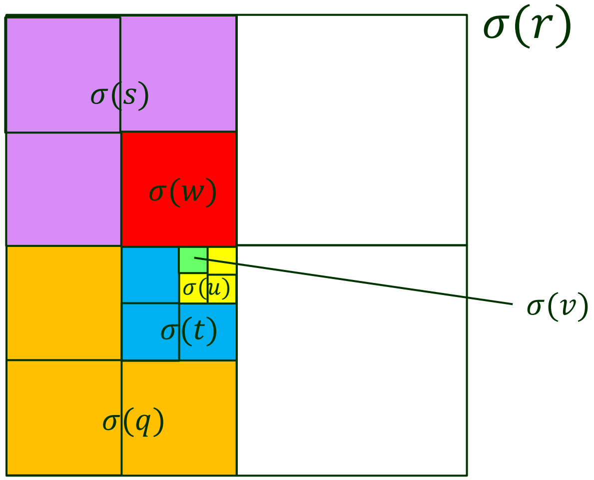
|

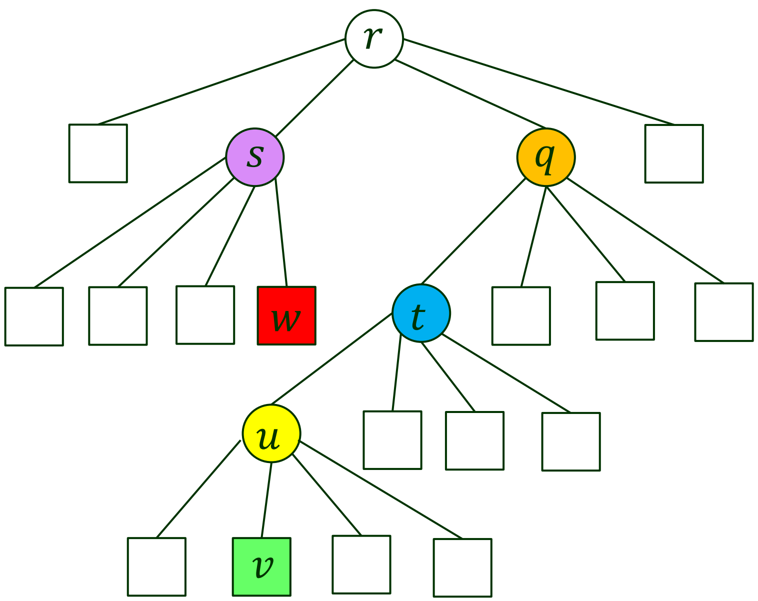
NorthNeighbor(t,$\mathcal{T}$)

|

NorthNeighbor(q,$\mathcal{T}$)

A Bigger Example






```
NorthNeighbor(v,T)

if v = root(T)
    then return nil
u ← parent(v)
if v is the SW-child of u
    then return the NW-child of u
if v is the SE-child of u
    then return the NE-child of u
μ ← NorthNeighbor(u, T)
if μ =nil or μ is a leaf
    then return μ
    else if v is the NW-child of u
        then return the SW-child of μ
        else return the SE-child of μ
```

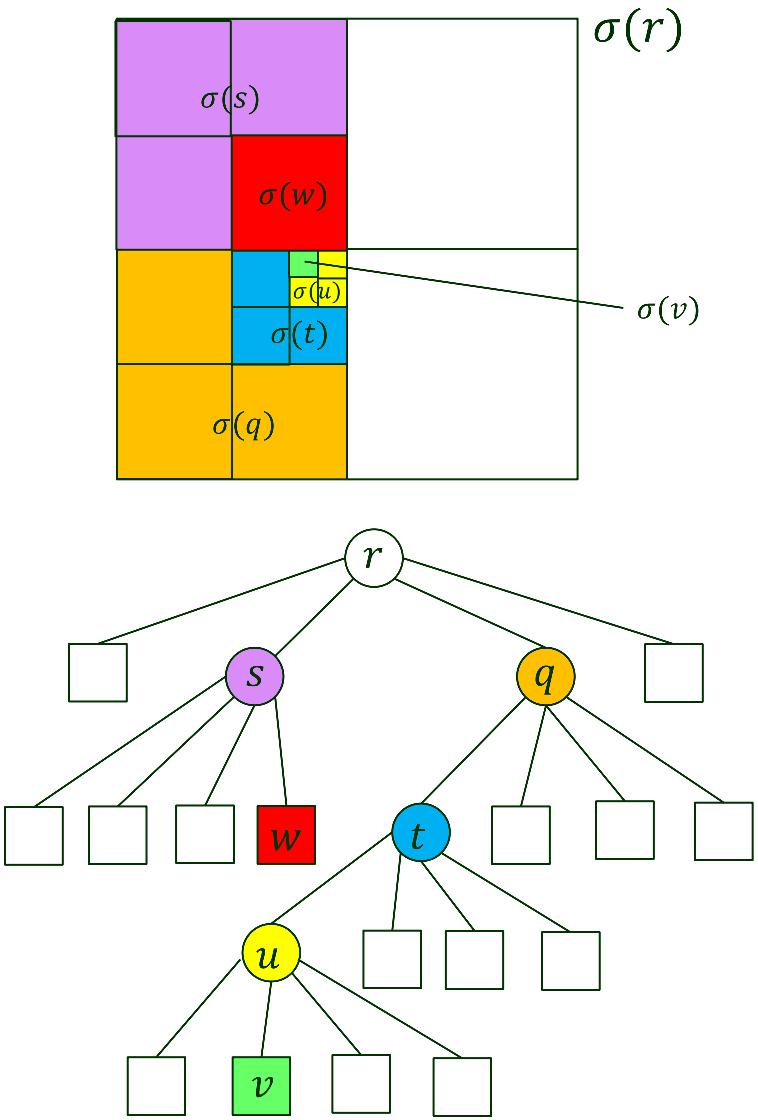
NorthNeighbor(v,$\mathcal{T}$)

NorthNeighbor(u,$\mathcal{T}$)

NorthNeighbor(t,$\mathcal{T}$)

NorthNeighbor(q,$\mathcal{T}$) returns s

A Bigger Example




```
NorthNeighbor(v,T)

if v = root(T)
    then return nil
u ← parent(v)
if v is the SW-child of u
    then return the NW-child of u
if v is the SE-child of u
    then return the NE-child of u
μ ← NorthNeighbor(u, T)
if μ =nil or μ is a leaf
    then return μ
    else if v is the NW-child of u
        then return the SW-child of μ
        else return the SE-child of μ
```

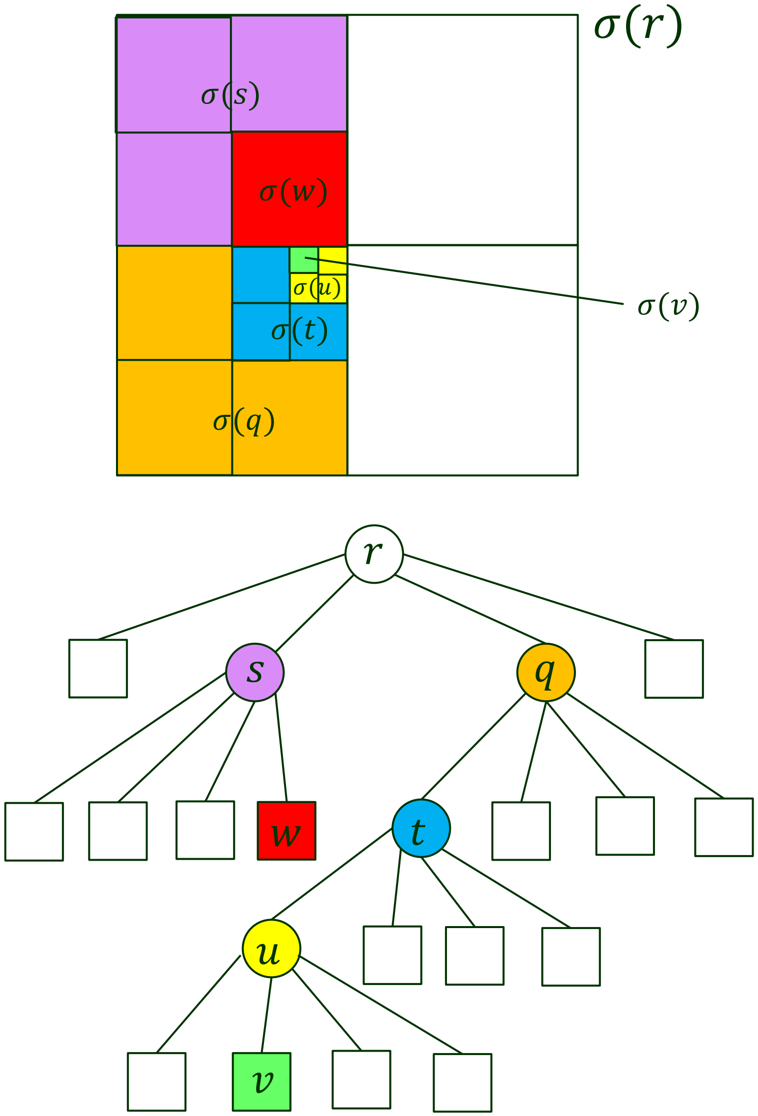
NorthNeighbor(v,$\mathcal{T}$)

NorthNeighbor(u,$\mathcal{T}$)

NorthNeighbor(t,$\mathcal{T}$) $\mu \leftarrow s$; returns w

NorthNeighbor(q,$\mathcal{T}$) returns s

A Bigger Example




```
NorthNeighbor(v,𝒯)

    if v = root(𝒯)
        then return nil
    u ← parent(v)
    if v is the SW-child of u
        then return the NW-child of u
    if v is the SE-child of u
        then return the NE-child of u
    μ ← NorthNeighbor(u, 𝒯)
    if μ =nil or μ is a leaf
        then return μ
        else if v is the NW-child of u
            then return the SW-child of μ
            else return the SE-child of μ
```

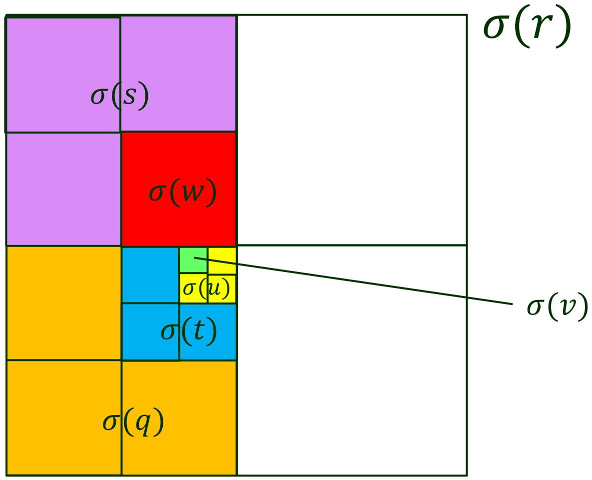
NorthNeighbor(v,$\mathcal{T}$)

NorthNeighbor(u,$\mathcal{T}$) — $\mu \leftarrow w$; return w

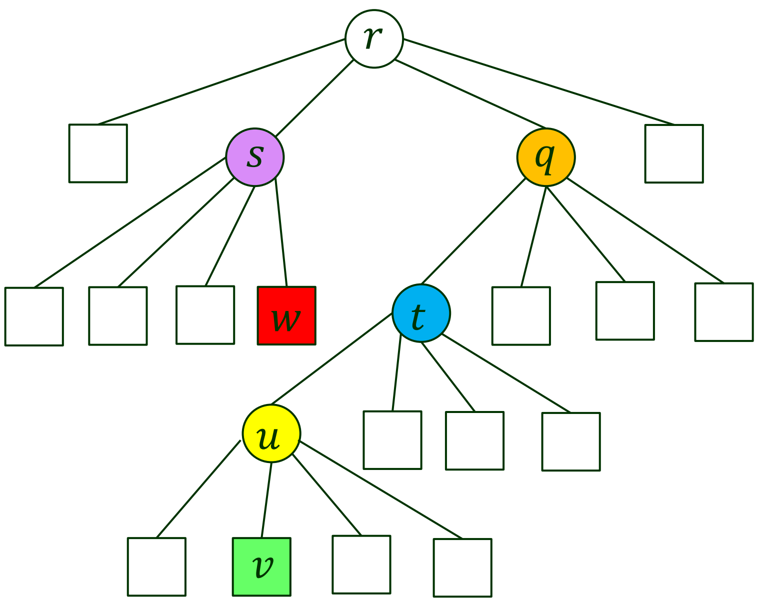
NorthNeighbor(t,$\mathcal{T}$) — $\mu \leftarrow s$; returns w

NorthNeighbor(q,$\mathcal{T}$) — returns s

A Bigger Example




```
NorthNeighbor(v,T)

if v = root(T)
    then return nil
u ← parent(v)
if v is the SW-child of u
    then return the NW-child of u
if v is the SE-child of u
    then return the NE-child of u
μ ← NorthNeighbor(u, T)
if μ =nil or μ is a leaf
    then return μ
    else if v is the NW-child of u
        then return the SW-child of μ
        else return the SE-child of μ
```



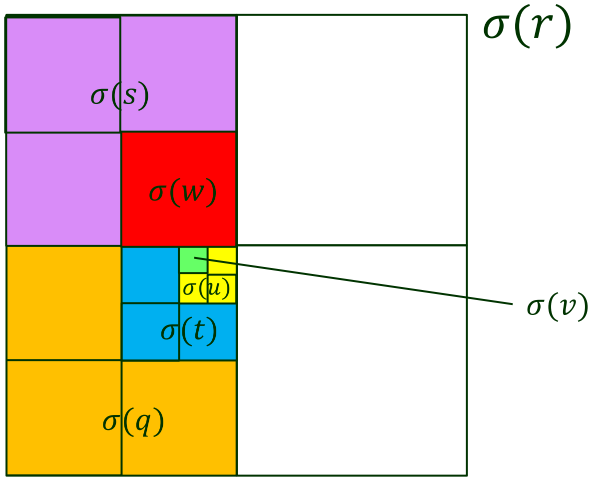
NorthNeighbor(v,$\mathcal{T}$) — $\mu \leftarrow w$; return w

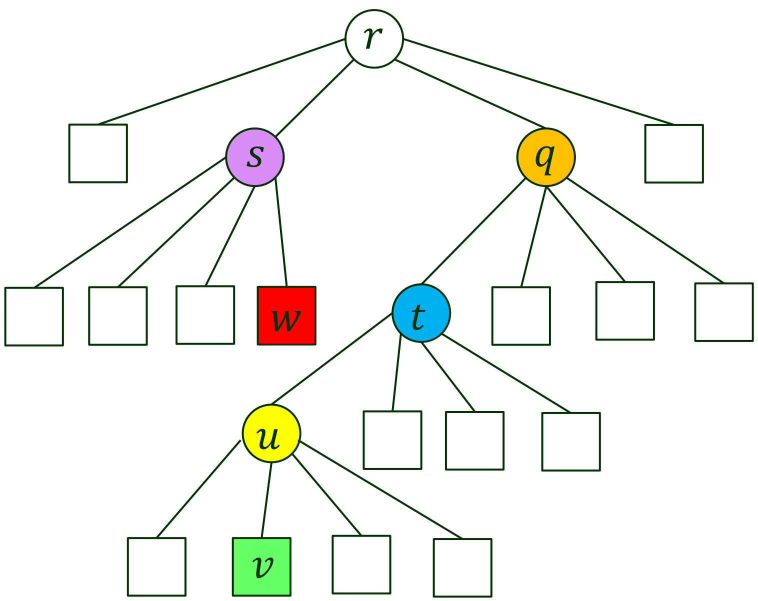
NorthNeighbor(u,$\mathcal{T}$) — $\mu \leftarrow w$; return w

NorthNeighbor(t,$\mathcal{T}$) — $\mu \leftarrow s$; returns w

NorthNeighbor(q,$\mathcal{T}$) — returns s

A Bigger Example






```
NorthNeighbor(v,T)

if v = root(T)
    then return nil
u ← parent(v)
if v is the SW-child of u
    then return the NW-child of u
if v is the SE-child of u
    then return the NE-child of u
μ ← NorthNeighbor(u, T)
if μ =nil or μ is a leaf
    then return μ
    else if v is the NW-child of u
        then return the SW-child of μ
        else return the SE-child of μ
```

NorthNeighbor(v,$\mathcal{T}$) — $\mu \leftarrow w$; return w

NorthNeighbor(u,$\mathcal{T}$) — $\mu \leftarrow w$; return w

NorthNeighbor(t,$\mathcal{T}$) — $\mu \leftarrow s$; returns w

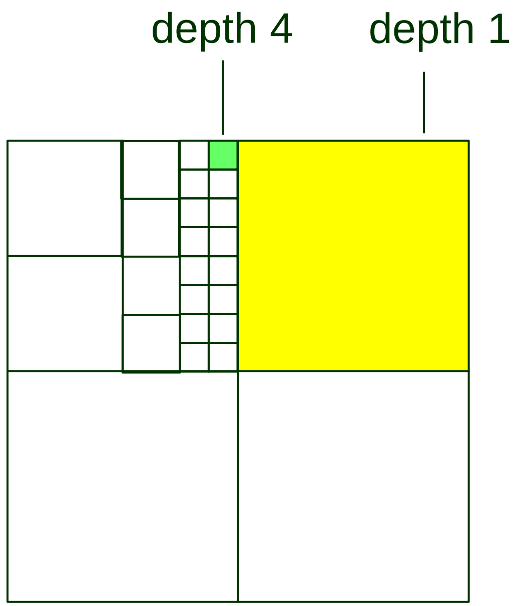
NorthNeighbor(q,$\mathcal{T}$) — returns s

Need to descend from s to locate the neighbor w.

V. Balanced Quadtree

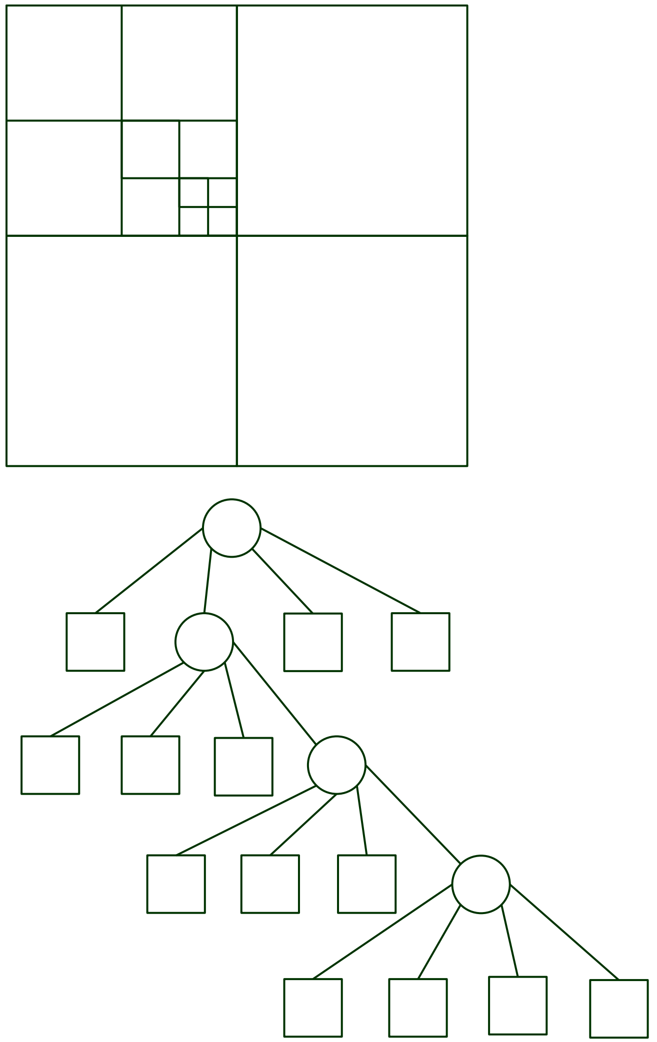
A quadtree subdivision is *balanced* if any two neighboring squares differ by *a factor of one or two* in size (as measured by the side length not by area).

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

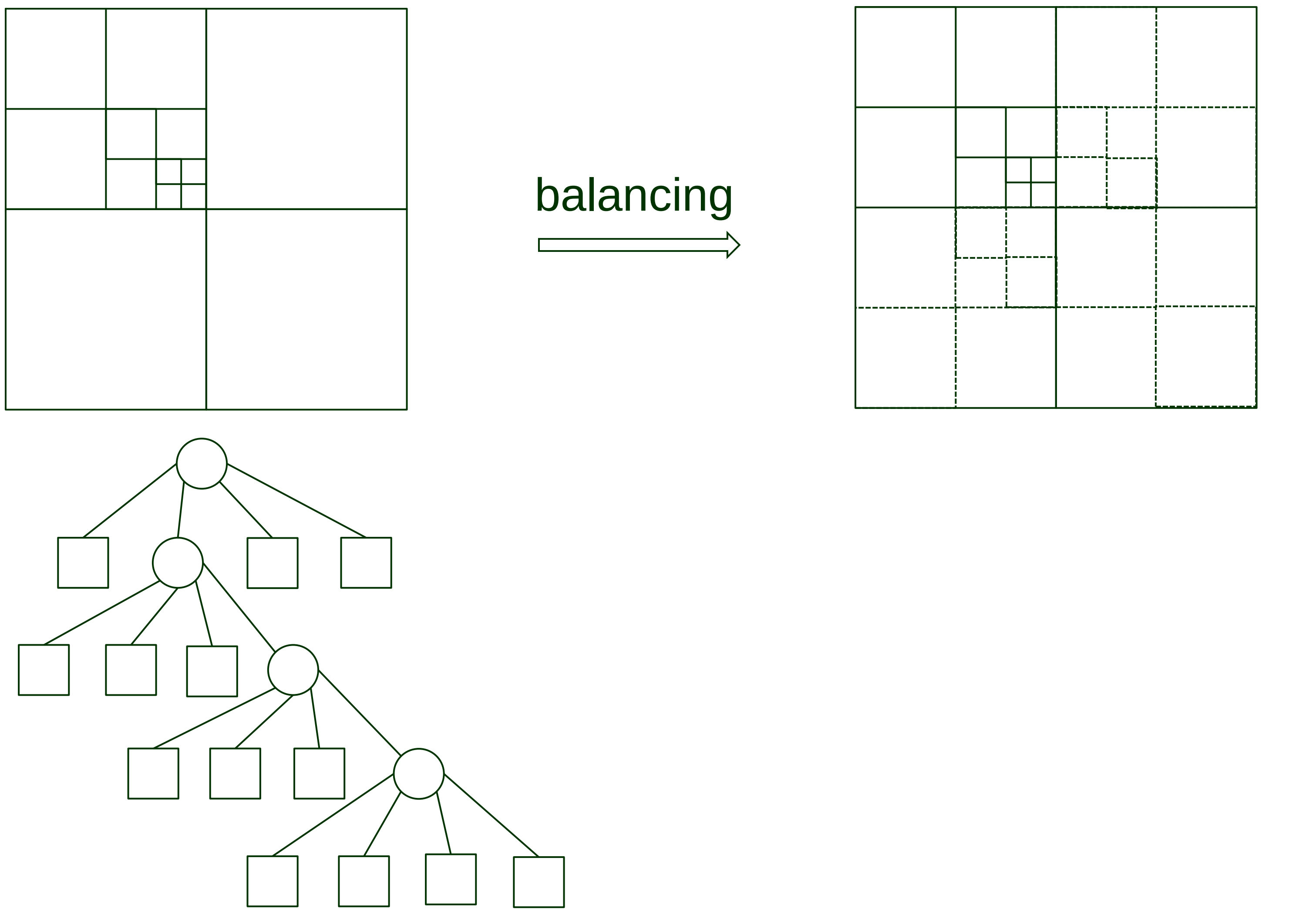


Unbalanced subdivision and
Corresponding quadtree.

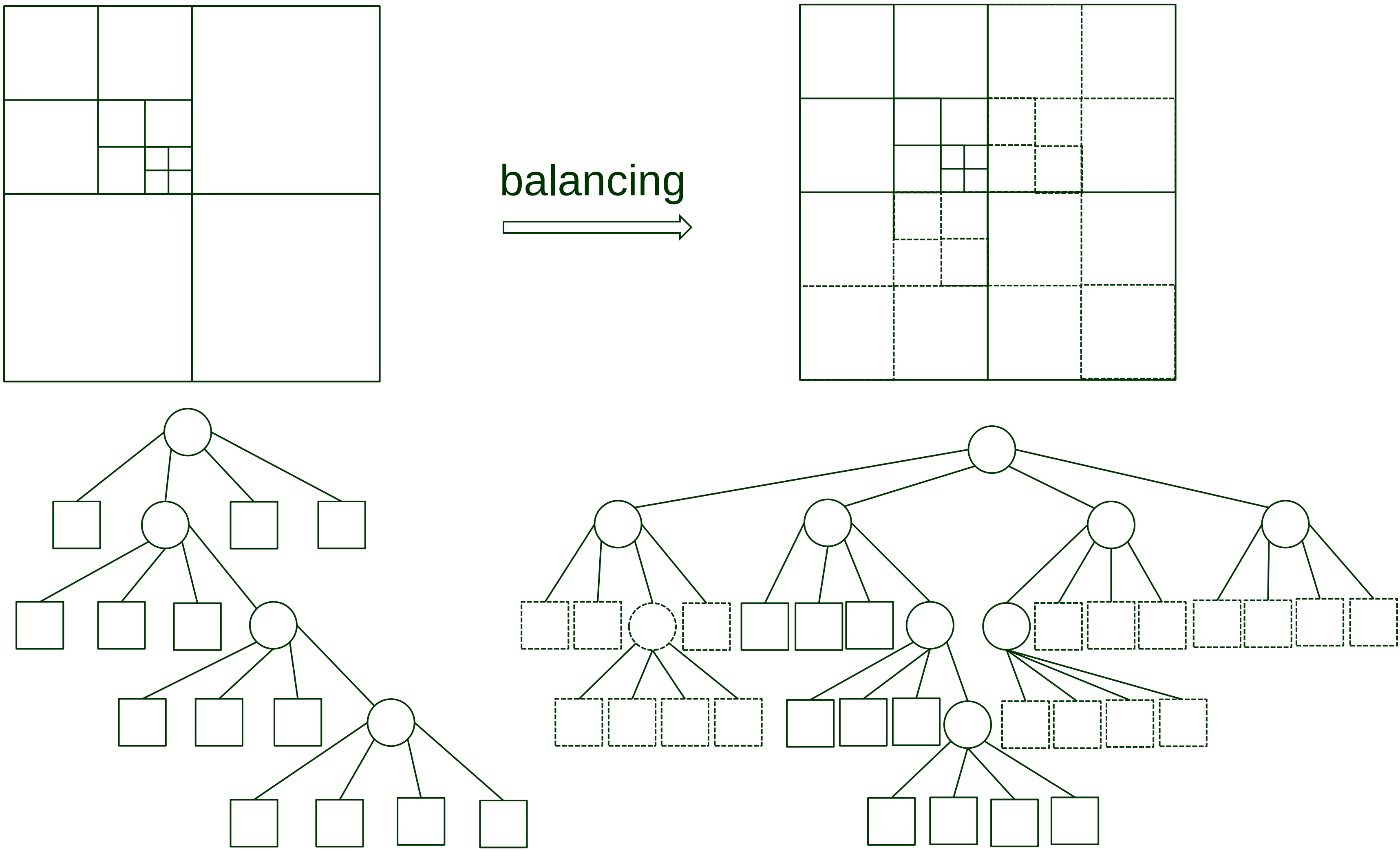
Example



Example



Example



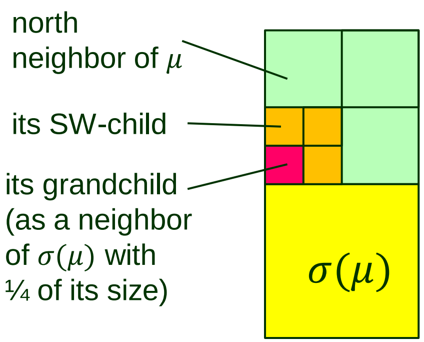
Balancing Algorithm

BalanceQuadTree($\mathcal{T}$)

insert all the leaves of $\mathcal{T}$ into a linear list $\mathcal{L}$
while $\mathcal{L}$ is not empty
    do remove a leaf μ from $\mathcal{L}$
        if $\sigma(\mu)$ has to be split
        then make μ an internal node with four new leaves
            if μ stores a point
                then stores it in the correct new leaf
            insert the four new leaves into $\mathcal{L}$
            if $\sigma(\mu)$ had neighbors that now need to be split
                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?

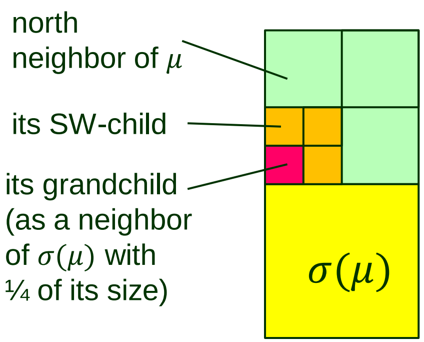



```
BalanceQuadTree(𝒯)

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

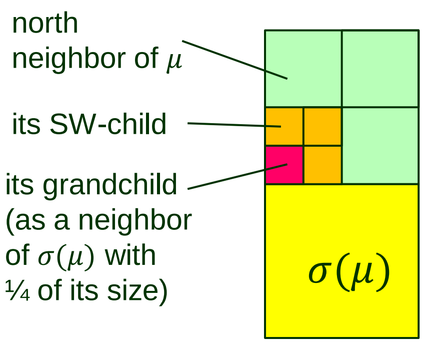



```
BalanceQuadTree(𝒯)

1.   insert all the leaves of 𝒯 into a linear list ℒ
2.   while ℒ is not empty
3.       do remove a leaf μ from ℒ
4.          if σ(μ) has to be split
5.          then make μ an internal node with four new leaves
6.              if μ stores a point
7.                  then stores it in the correct new leaf
8.              insert the four new leaves into ℒ
9.              if σ(μ) 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 NorthNeighbor(μ,$\mathcal{T}$) returns a node that has a SW- or SE-child that is not a leaf.

```
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 μ from ℒ
4.          if σ(μ) has to be split
5.          then make μ an internal node with four new leaves
6.               if μ stores a point
7.                   then stores it in the correct new leaf
8.               insert the four new leaves into ℒ
9.               if σ(μ) had neighbors that now need to be split
10.                  then insert them into ℒ
```

Second Issue to Settle

```
BalanceQuadTree(𝒯)

insert all the leaves of 𝒯 into a linear list ℒ
while ℒ is not empty
    do remove a leaf μ from ℒ
       if σ(μ) has to be split
       then make μ an internal node with four new leaves
           if μ stores a point
               then stores it in the correct new leaf
           insert the four new leaves into ℒ
           if σ(μ) had neighbors that now need to be split
               then insert them into ℒ
```

Second Issue to Settle

```
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 μ from ℒ
4.          if σ(μ) has to be split
5.          then make μ an internal node with four new leaves
6.              if μ stores a point
7.                  then stores it in the correct new leaf
8.              insert the four new leaves into ℒ
9.              if σ(μ) 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.

Second Issue to Settle

```
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 μ from ℒ
4.          if σ(μ) has to be split
5.          then make μ an internal node with four new leaves
6.              if μ stores a point
7.                  then stores it in the correct new leaf
8.              insert the four new leaves into ℒ
9.              if σ(μ) 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.

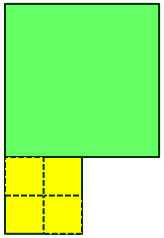
- Again use the neighbor finding algorithm.
  - Such a neighbor exists to the north iff `NorthNeighbor`$(\mu, \mathcal{T})$ returns a node corresponding to a square larger than $\sigma(\mu)$.

Second Issue to Settle

```
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 μ from ℒ
4.         if σ(μ) has to be split
5.         then make μ an internal node with four new leaves
6.              if μ stores a point
7.                 then stores it in the correct new leaf
8.              insert the four new leaves into ℒ
9.              if σ(μ) 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(`$\mu, \mathcal{T}$`)` returns a node corresponding to a square larger than $\sigma(\mu)$.



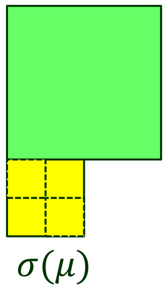
$\sigma(\mu)$

Second Issue to Settle

```
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 μ from ℒ
4.          if σ(μ) has to be split
5.          then make μ an internal node with four new leaves
6.              if μ stores a point
7.                  then stores it in the correct new leaf
8.              insert the four new leaves into ℒ
9.              if σ(μ) 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(μ,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 μ 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 $O\big((d+1)m\big)$ time.

Proof Omitted.

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 $O\big((d+1)m\big)$ 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 $O\big((d+1)m\big)$ 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 $O\big((d+1)m\big)$ 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 $O\big((d+1)m\big)$ 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).