

Region Quadrees

Finkel and Bentley (1974)





Background

- › A **region quadtree** represents a 2-dimensional image by recursively dividing the image into four equal quadrants or regions
- › Each quadrant is divided further if the region it represents needs to be refined. For example, if the region is entirely **green** then there is no reason to divide further.



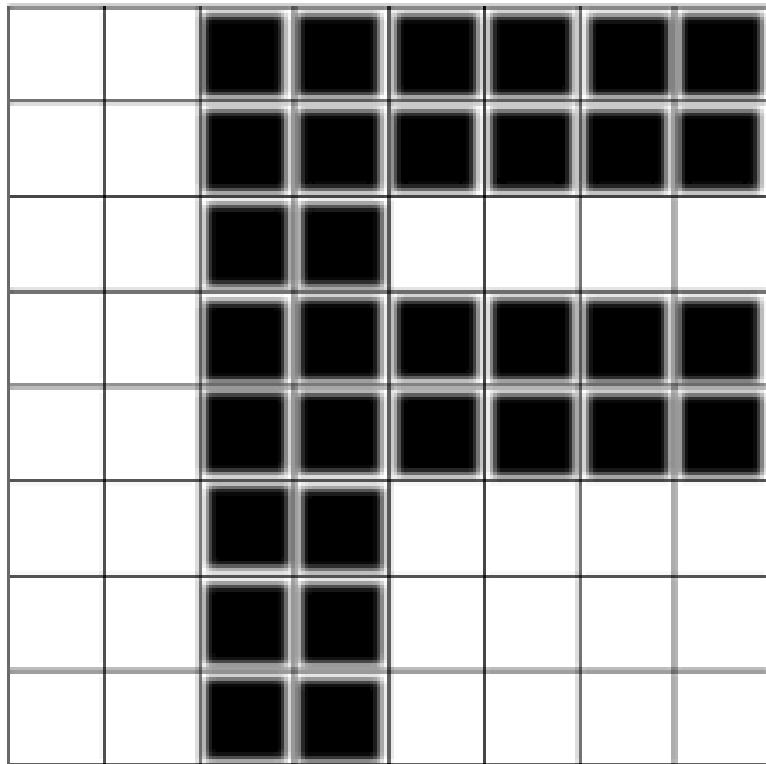
- › To simplify our discussion, assume that
 - only two colours are used (white and black)
 - the size of the image is a 2^k by 2^k , $k > 0$
 - the smallest region is 1×1 or one pixel



Example of an 8x8 image

NW

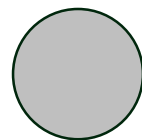
NE



The image is NOT a single colour
=> break the image into quadrants

SW

SE

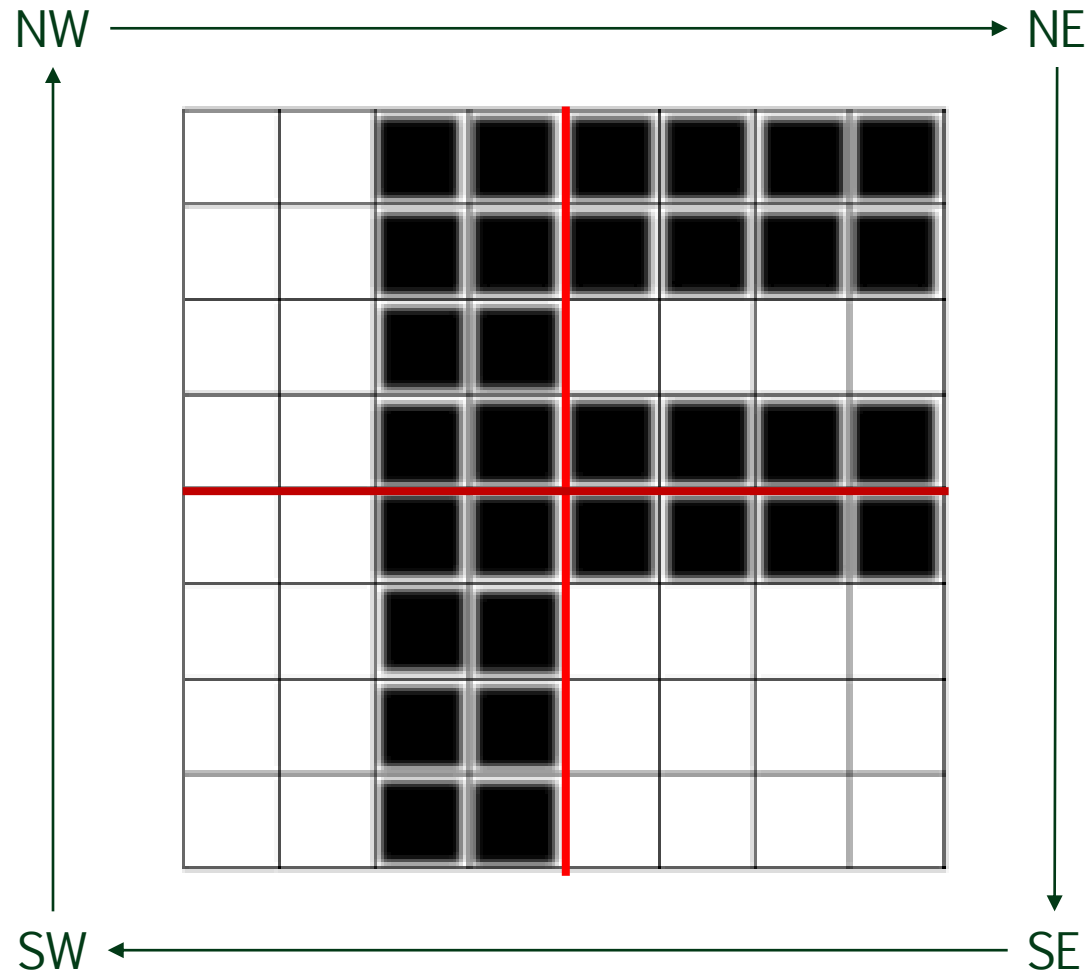


← Gray

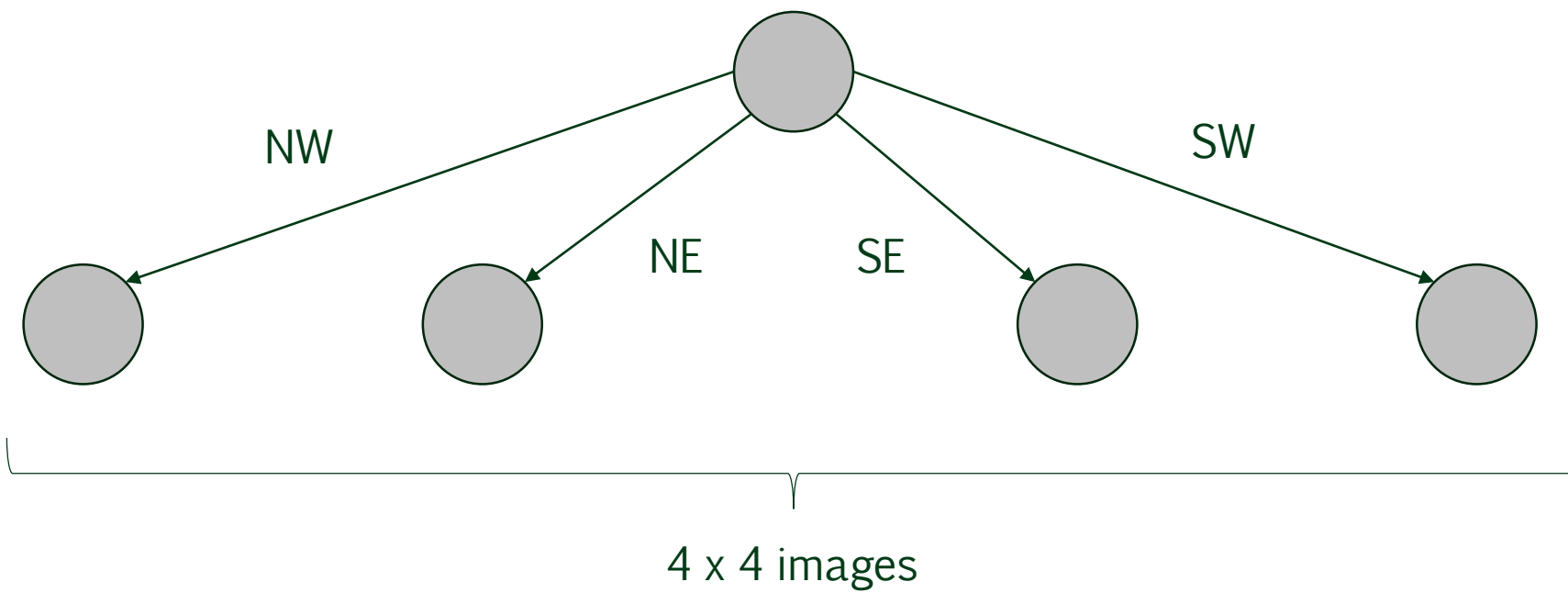
8 x 8 Image



Example of an 8x8 image



Each quadrant is NOT a single color
=> break each one into subquadrants

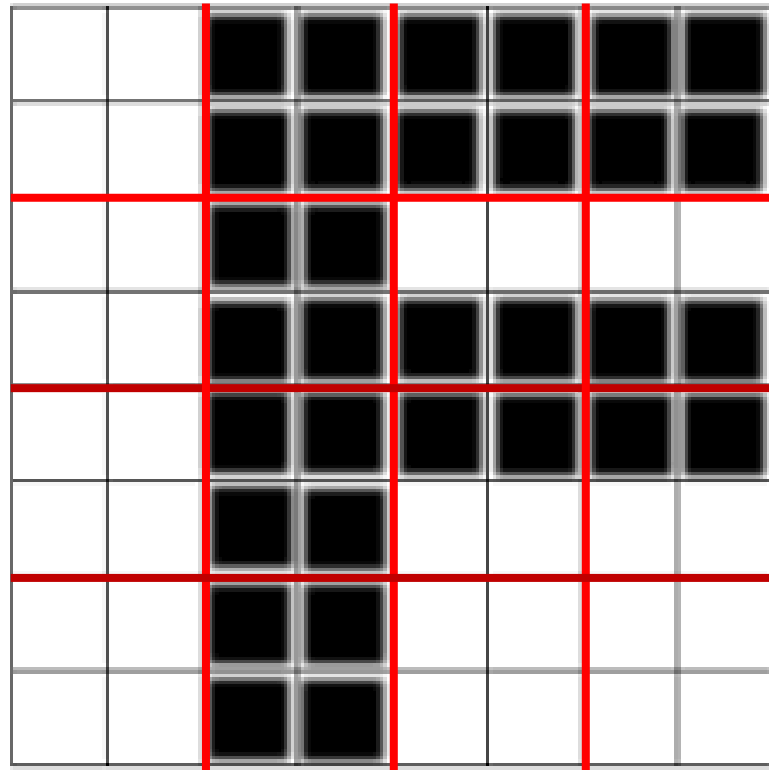




Example of an 8x8 image

NW

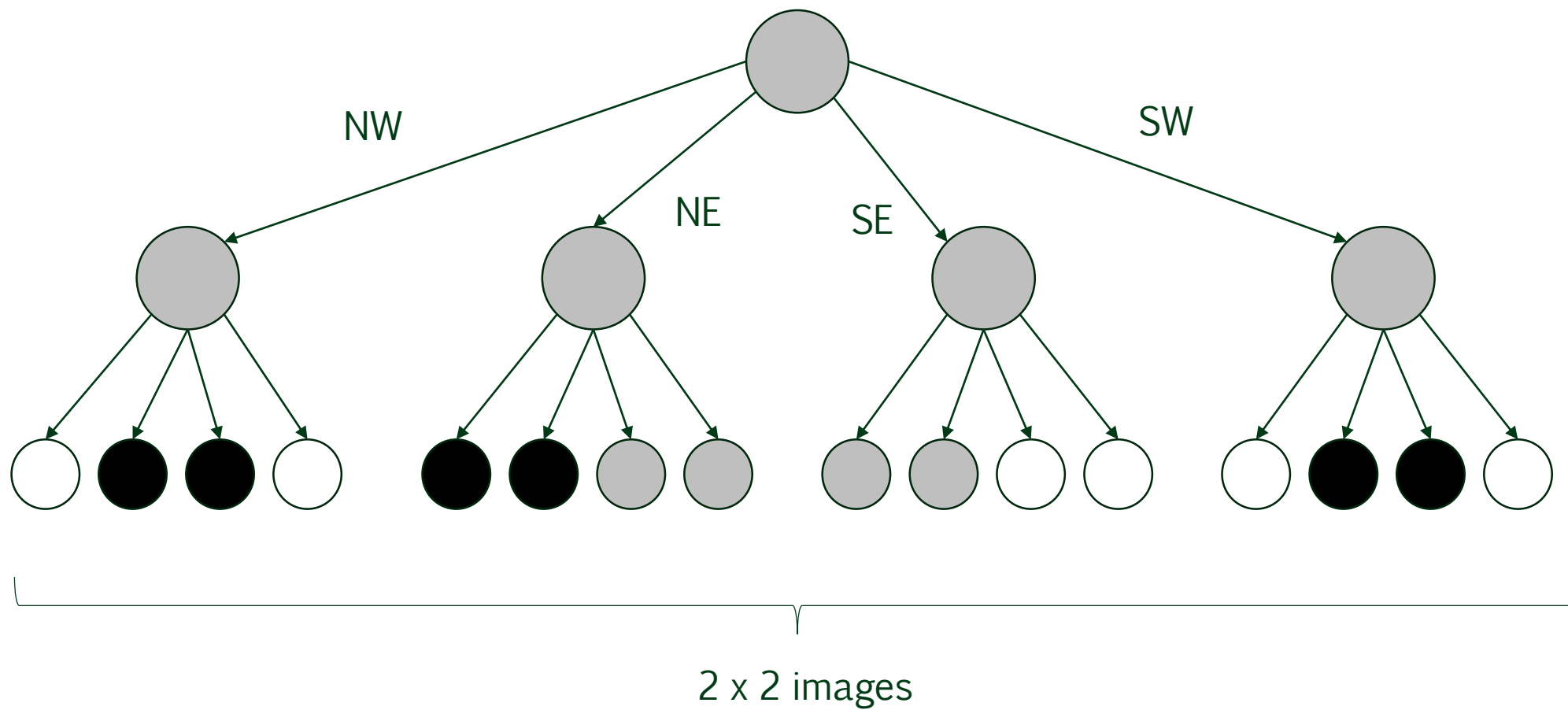
NE



All subquadrants are a single color except for four (which ones?)
=> break these four into subquadrants

SW

SE

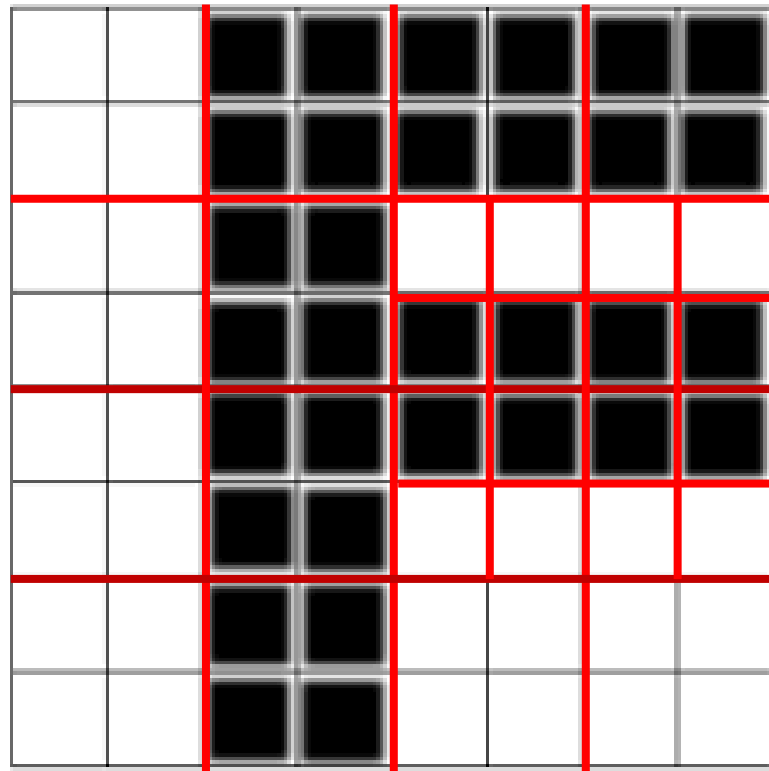




Example of an 8x8 image

NW

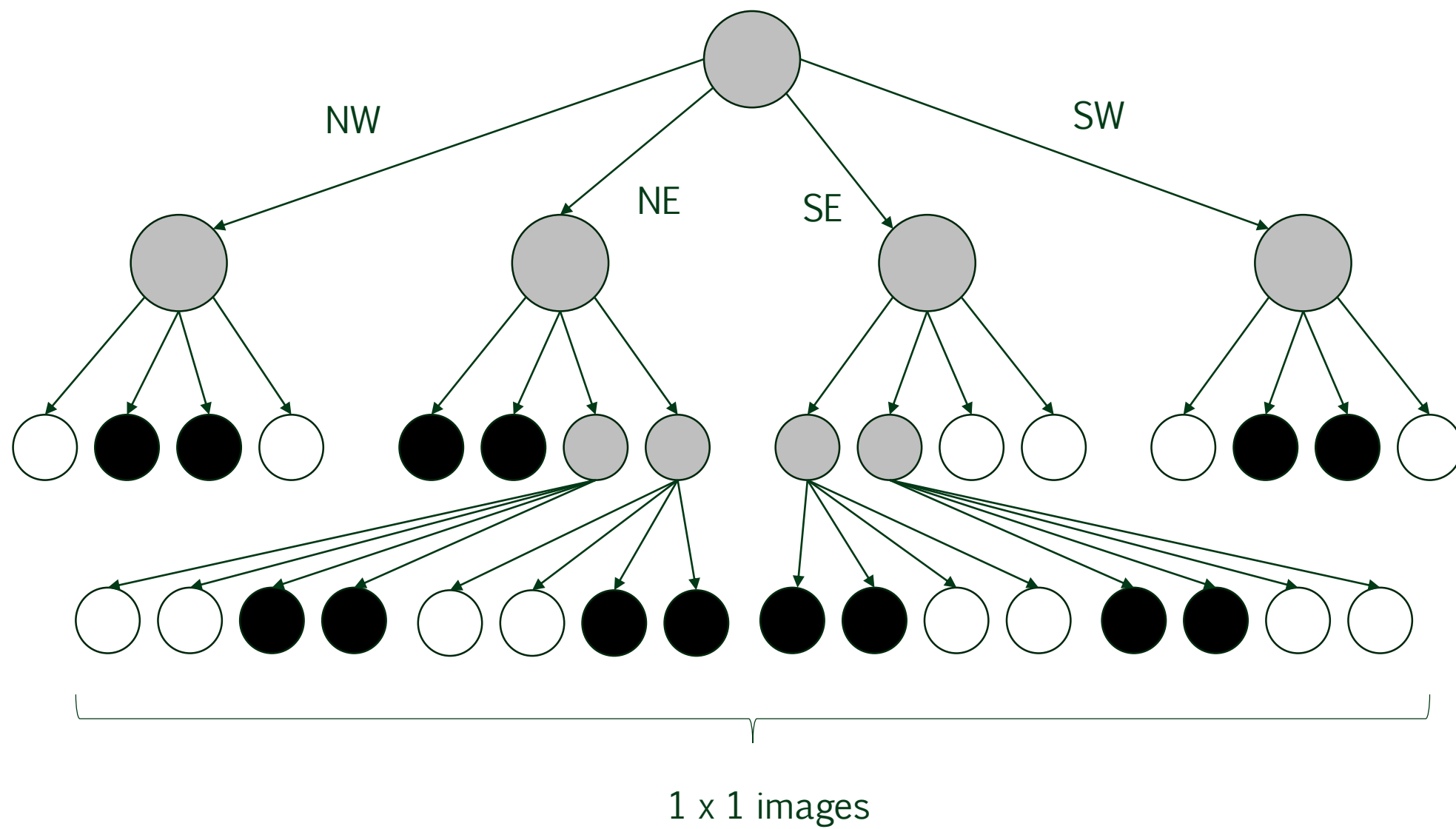
NE



Now all quadrants and subquadrants
are a single colour

SW

SE





Observations

- › All leaf nodes are either BLACK or WHITE
- › All internal nodes are GRAY
- › If the NW child of a node is null then the NE, SE and SW children of the node are also null
- › If the NW child of a node is not null, then the NE, SE and SW children of the node are also not null
- › The maximum height of the region quadtree is $O(\log n)$.
Why?



Exercises

- › What image would lead to a quadtree where all leaf nodes are at maximum depth?
- › Calculate in terms of k the maximum number of nodes to represent a quadtree of a $2^k \times 2^k$ image.
- › If the image has more than two colours, does the maximum height of the quadtree increase?



Data structure (Node)

```
// Leaf nodes are BLACK or WHITE
// Interior nodes are GRAY
public enum Color { BLACK, WHITE, GRAY};

// Quadtree Node

public class Node
{
    public Color C { get; set; }           // Color of a node (BLACK, WHITE, or GRAY)
    public Node NW { get; set; }           // Northwest quadrant
    public Node NE { get; set; }           // Northeast quadrant
    public Node SE { get; set; }           // Southeast quadrant
    public Node SW { get; set; }           // Southwest quadrant

    // Constructor
    // Creates a node with color c and four empty children
    // Time complexity: O(1)

    public Node(Color c)
    {
        C = c;
        NW = NE = SE = SW = null;
    }
}
```

four children {



Data structure (Quadtree)

```
public class Quadtree
{
    // Assumptions:
    // 1) The image is a square with dimensions  $2^k \times 2^k$ 
    // 2) The image is colored either BLACK or WHITE

    private Node root;        // Root of the quadtree
    private int size;         // Length of its side ( $n = 2^k$ )

    // Constructor A
    // Creates an empty quadtree
    // Time complexity:  $O(1)$ 

    public Quadtree()
    {
        root = null;
    }
    ...
}
```



Primary methods

- › `public Quadtree(Color[,] image, int size, int option = 1)`
 - builds a quadtree with compression (option 1) or initially without compression (option 2) based on a given size x size image
 - for option 2, the final quadtree is compressed
- › `public void Compress()`
 - compresses the quadtree such that no node has children that are all BLACK or all WHITE



Primary methods (cont.)

- › public void Switch(int i, in j)
 - modifies the quadtree when a single region (pixel) at index [i,j] changes color from BLACK to WHITE (or vice versa)
- › public Quadtree Union(Quadtree Q)
 - returns the union of the current and given quadtrees
 - a fuller description follows later



Primary methods (cont.)

- › `public void Print()`
 - prints the image represented by the current quadtree
- › `public void PrintQuadtree()`
 - prints the inorder traversal of the current quadtree



Support methods

- › private bool Region(Color[,] image, int x, int y, int n)
 - returns true if the square image starting at index [x,y] with side length n is a single colour; false otherwise
- › private Node Clone (Node p)
 - returns a clone of the quadtree rooted at p
- › private void FillIn(Color[,] image, int x, int y, int n, Node p)
 - fills in a square image starting at index [x,y] with side length n with the proper colors (used by Print)

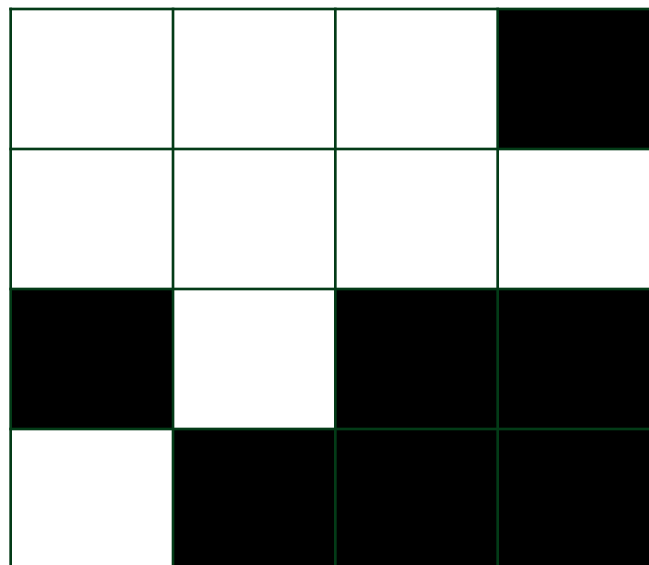


Constructors

- › The first constructor sets the root of the quadtree to null
- › The second constructor builds the quadtree given a $2^k \times 2^k$ image according to one of two options (1 or 2)
 - The first option builds the tree as just presented. As soon as a region is defined by a single colour, a leaf node is created and defined by that colour.
 - The second option breaks the image down to the its smallest region (1x1) regardless of colour. Then it calls the Compress method to yield the same tree as option 1.

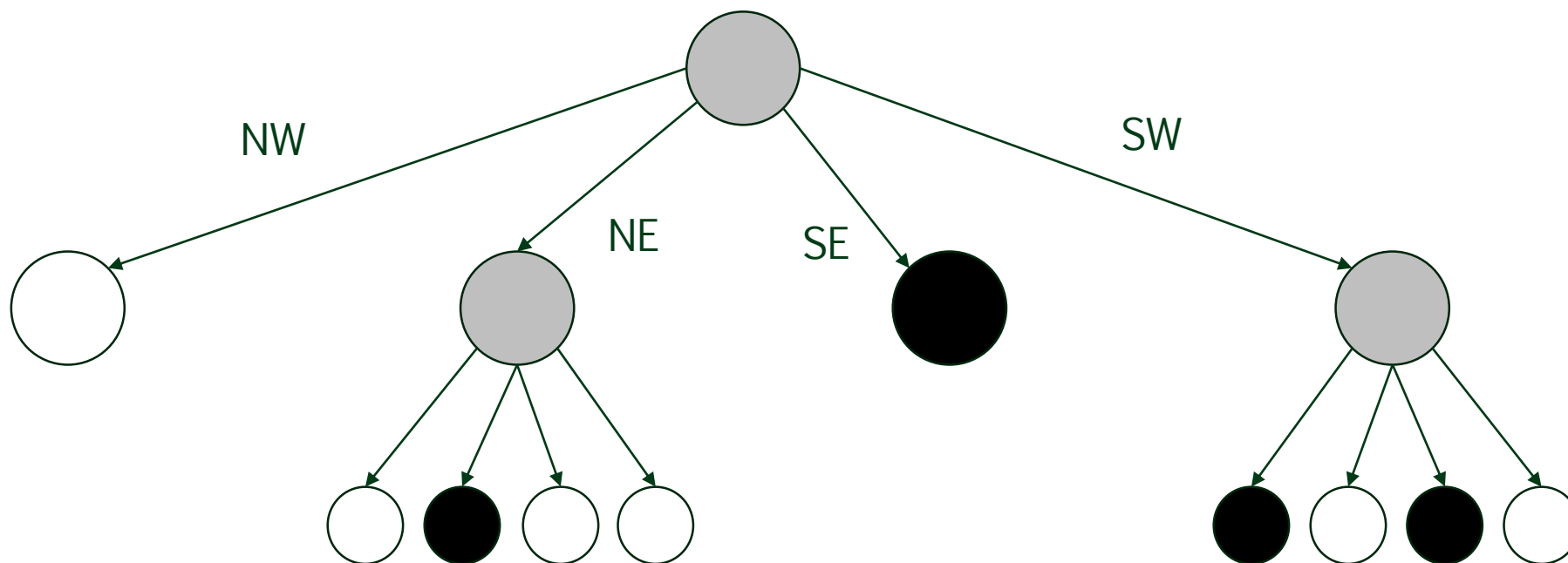


Consider the following 4x4 image



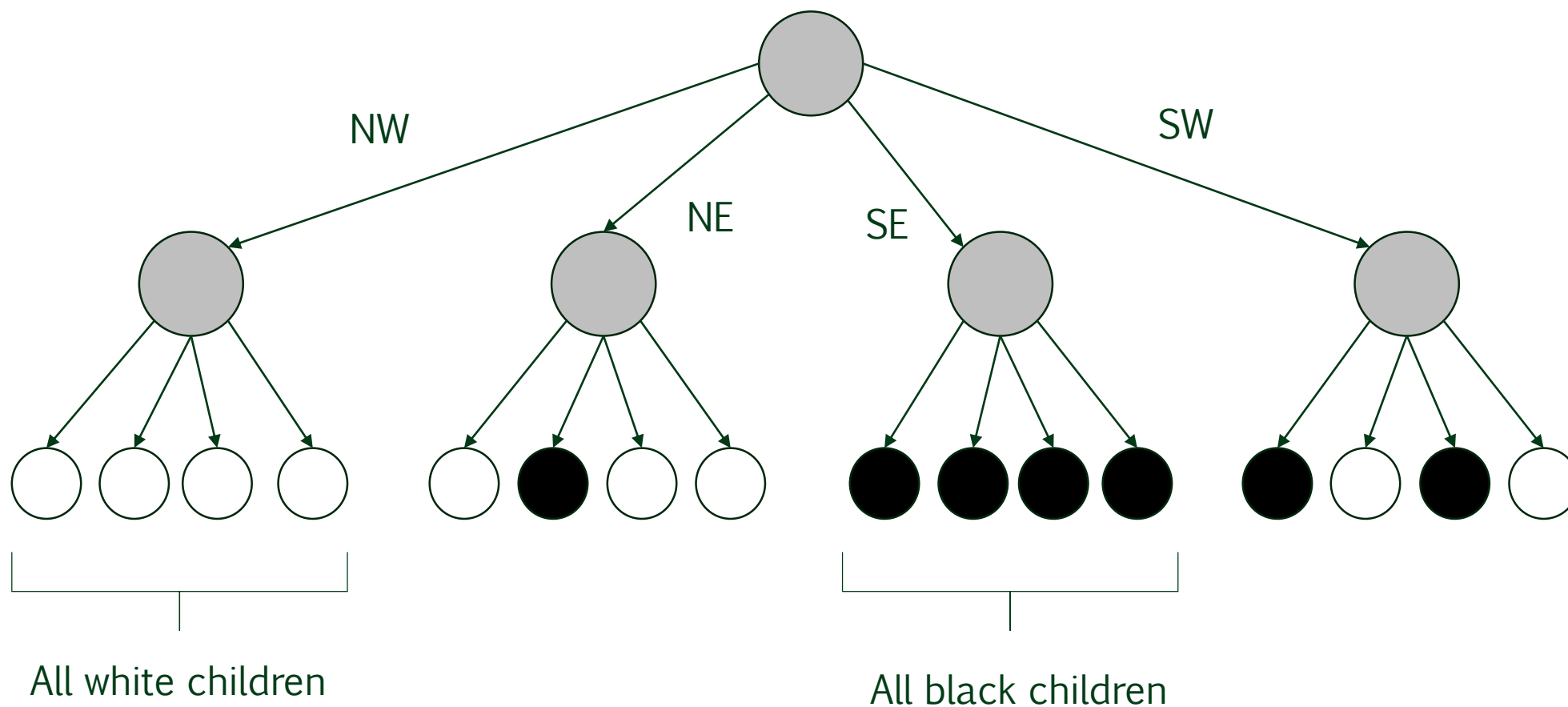


Option 1: Build with compression



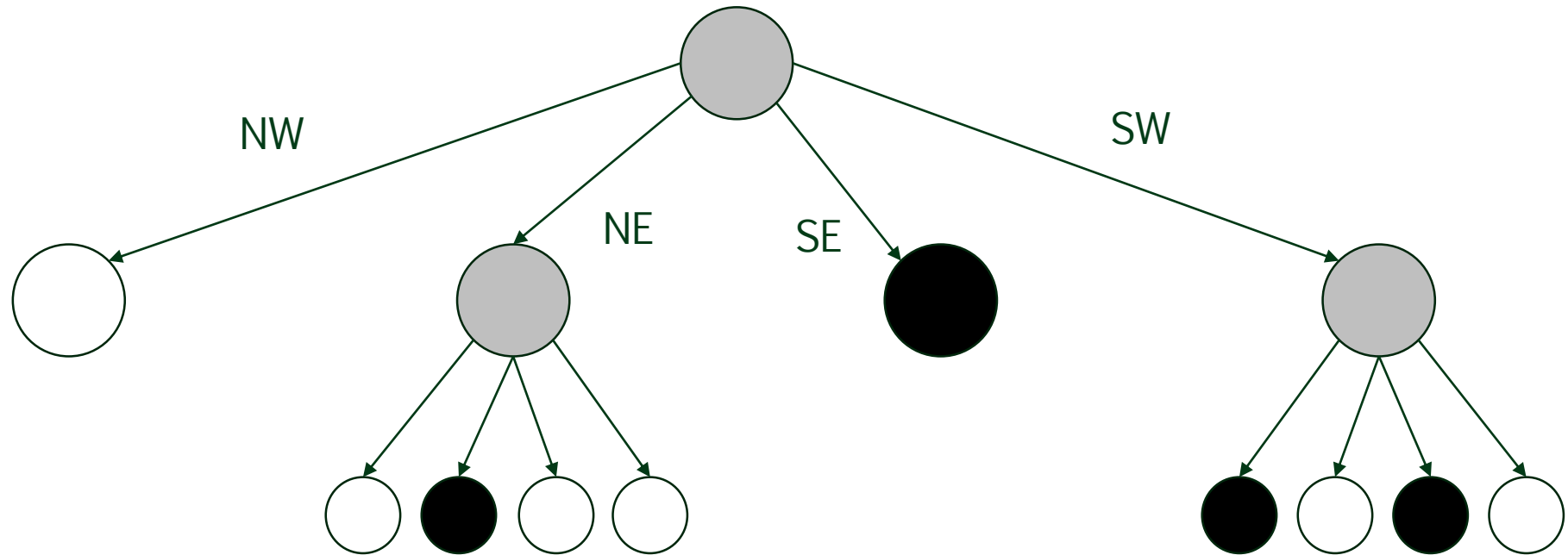


Option 2: Build without compression



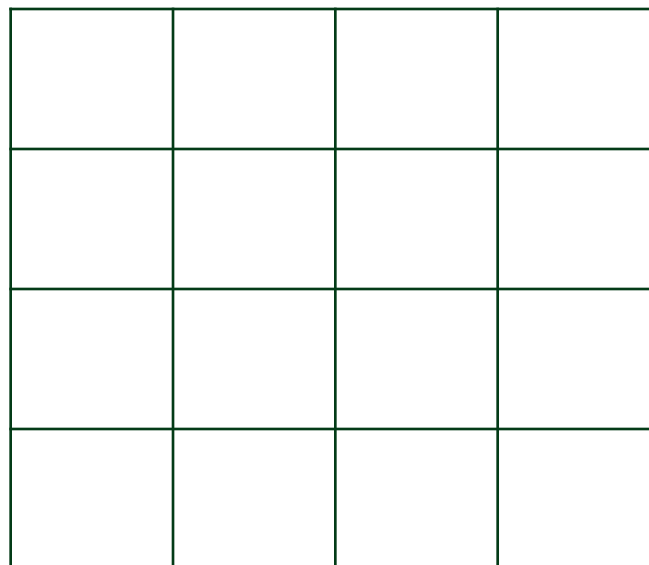


After compression (same result as Option 1)



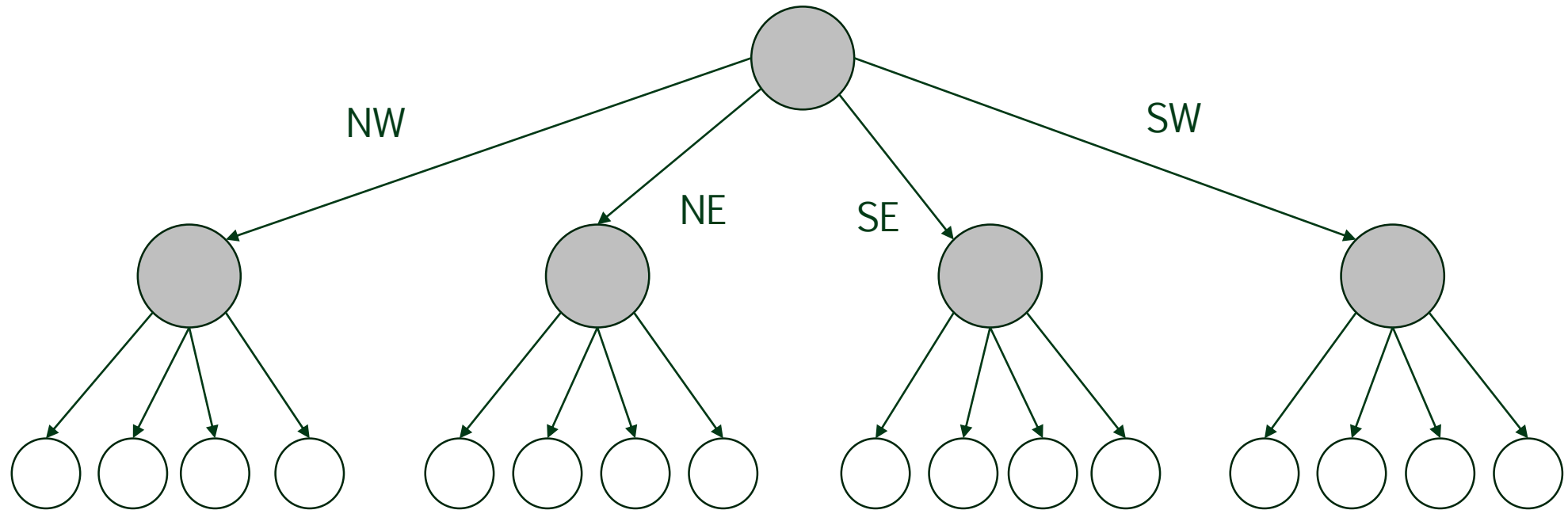


Consider the following 4x4 white image



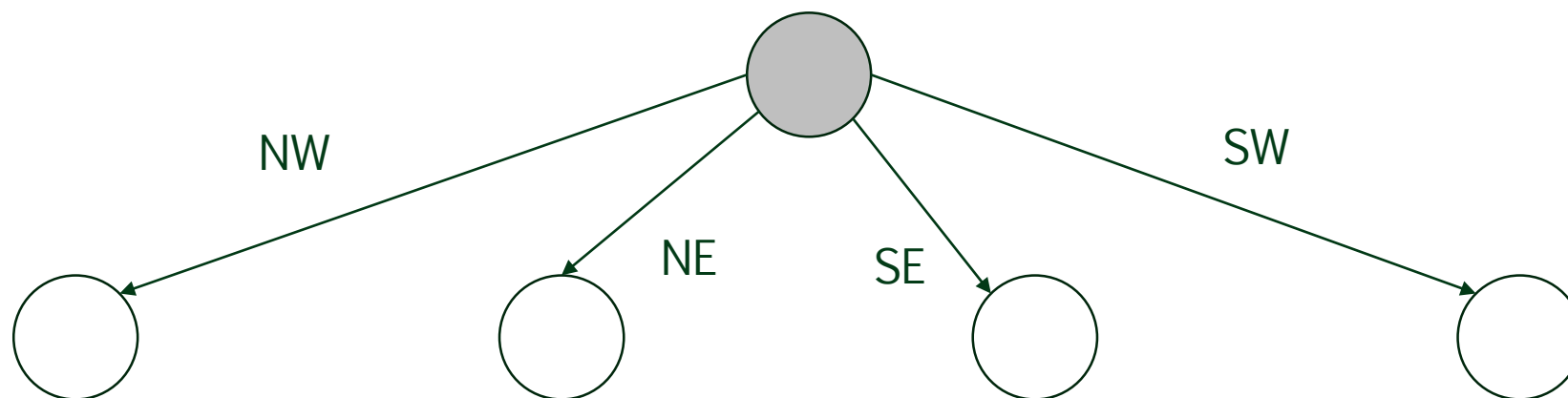


Option 2: Build without compression



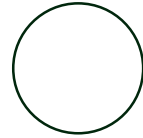


Compress





Compress (same result as Option 1)



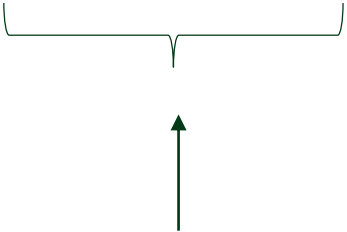


Option 2

```
private Node ConstructWithoutCompression(Color[,] image, int x, int y, int n)
{
    Node p;
    int h;

    → { if (n == 1) // Leaf node representing one pixel
        p = new Node(image[x, y]);
        else
        {
            h = n / 2;
            p = new Node(Color.GRAY); // Set the color of the node to GRAY

            // Recursively build each subtree (quadrant)
            p.NW = ConstructWithoutCompression(image, x, y, h);
            p.NE = ConstructWithoutCompression(image, x, y + h, h);
            p.SE = ConstructWithoutCompression(image, x + h, y + h, h);
            p.SW = ConstructWithoutCompression(image, x + h, y, h);
        }
    }
    return p;
}
```





NW

NE

y

$y+h$

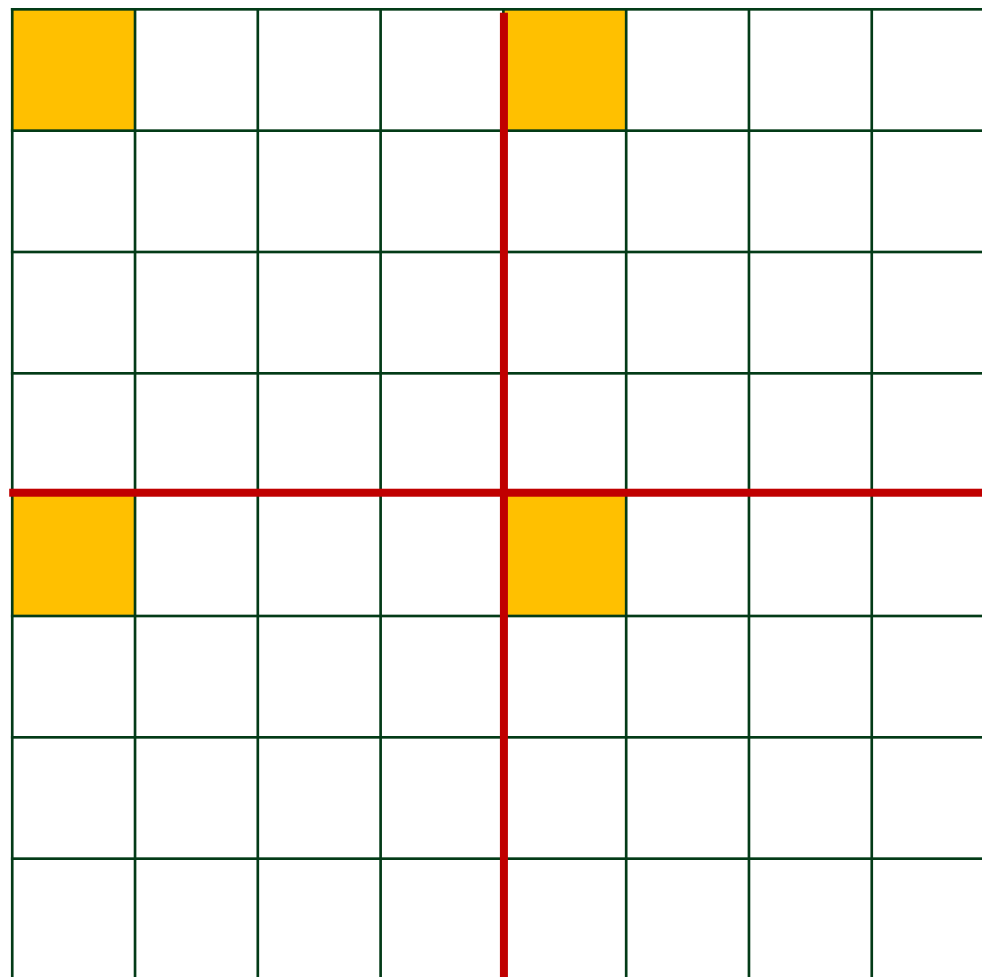
x

h

$x+h$

SW

SE





Option 1

```
private Node ConstructWithCompression(Color[,] image, int x, int y, int n)
{
    Node p;
    int h;

    —————> if (Region(image, x, y, n))           // Check if a region is a single color
        p = new Node(image[x, y]);           // If so, create a leaf node with that color
    else
    {
        h = n / 2;
        p = new Node(Color.GRAY);           // Set the color of the node to GRAY

        // Recursively build each subtree (quadrant)
        p.NW = ConstructWithCompression(image, x, y, h);
        p.NE = ConstructWithCompression(image, x, y + h, h);
        p.SE = ConstructWithCompression(image, x + h, y + h, h);
        p.SW = ConstructWithCompression(image, x + h, y, h);
    }
    return p;
}
```



Time complexity

- › The time complexity of both constructor options is $O(n^2)$ where n is the side length of the image

Compress

```
private void Compress(Node p)
{
    if (p.NW != null)
    {
        // Compress each GRAY (non-leaf) quadrant of p
        if (p.NW.C == Color.GRAY)
            Compress(p.NW);
        if (p.NE.C == Color.GRAY)
            Compress(p.NE);
        if (p.SE.C == Color.GRAY)
            Compress(p.SE);
        if (p.SW.C == Color.GRAY)
            Compress(p.SW);

        // If all children are the same color (BLACK or WHITE)
        // Then set p to the color its children and set all children to null

        if (p.NW.C != Color.GRAY)
            if (p.NW.C == p.NE.C && p.NW.C == p.SE.C && p.NW.C == p.SW.C)
            {
                p.C = p.NW.C;
                p.NW = p.NE = p.SE = p.SW = null;
            }
    }
}
```

Not a leaf node

All four children are not null



Time complexity

- › The time complexity of Compress is $O(n^2)$ which is the maximum depth of the quadtree



Switch(i,j)

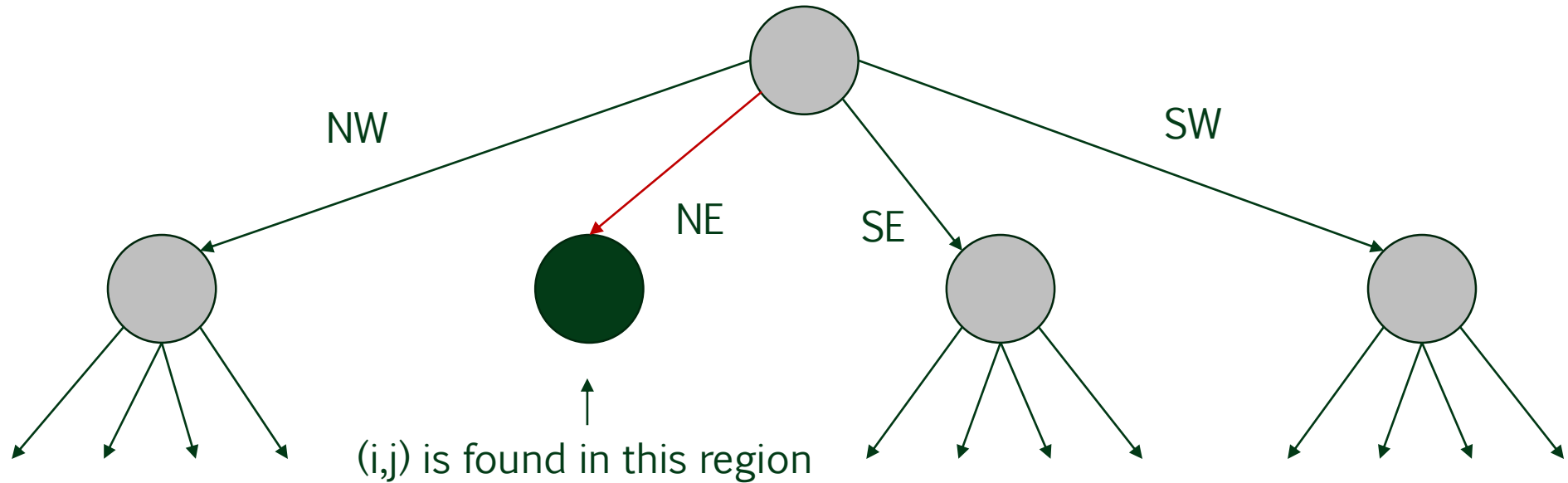
- › Basic strategy (two scenarios)

- First scenario

- › If index (i,j) is found in a region defined by a leaf node p then p is expanded, and all its children are set to the colour of p
 - › p is then set to GRAY and descends to the quadrant that contains (i,j)
 - › Once p defines the 1x1 singular region (i,j), the colour of p is set to its opposite (BLACK to WHITE or WHITE to BLACK)

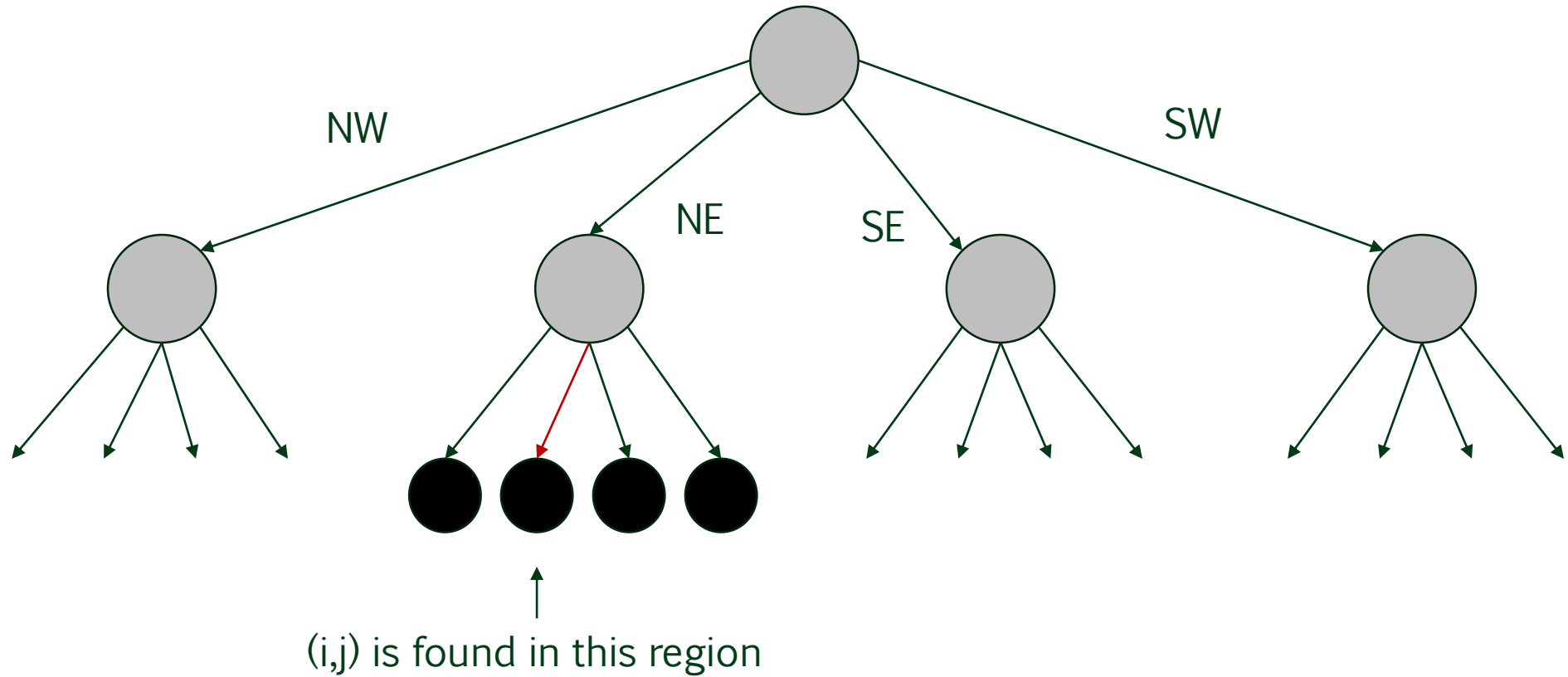


8x8 image (maximum depth of 3)



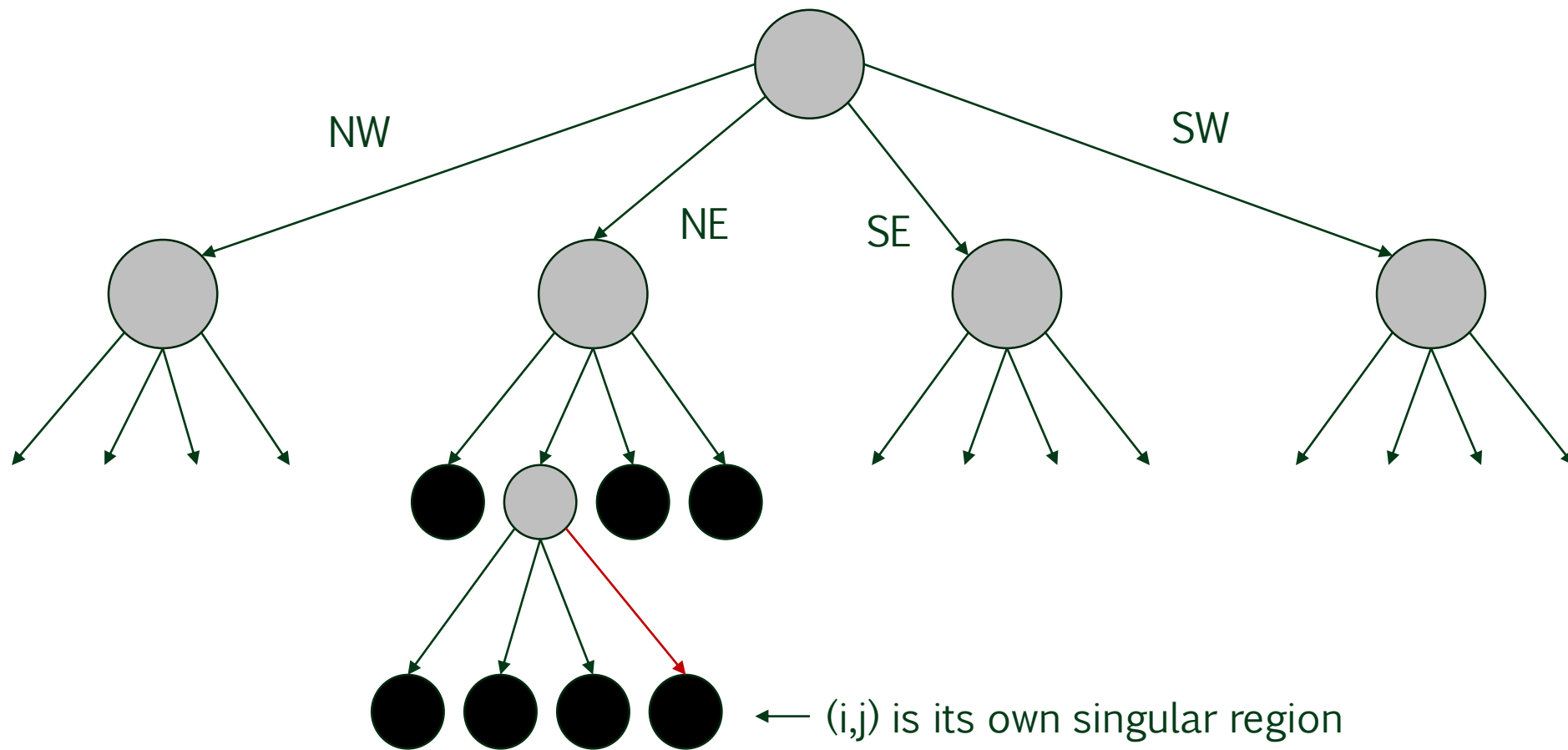


8x8 image (maximum depth of 3)



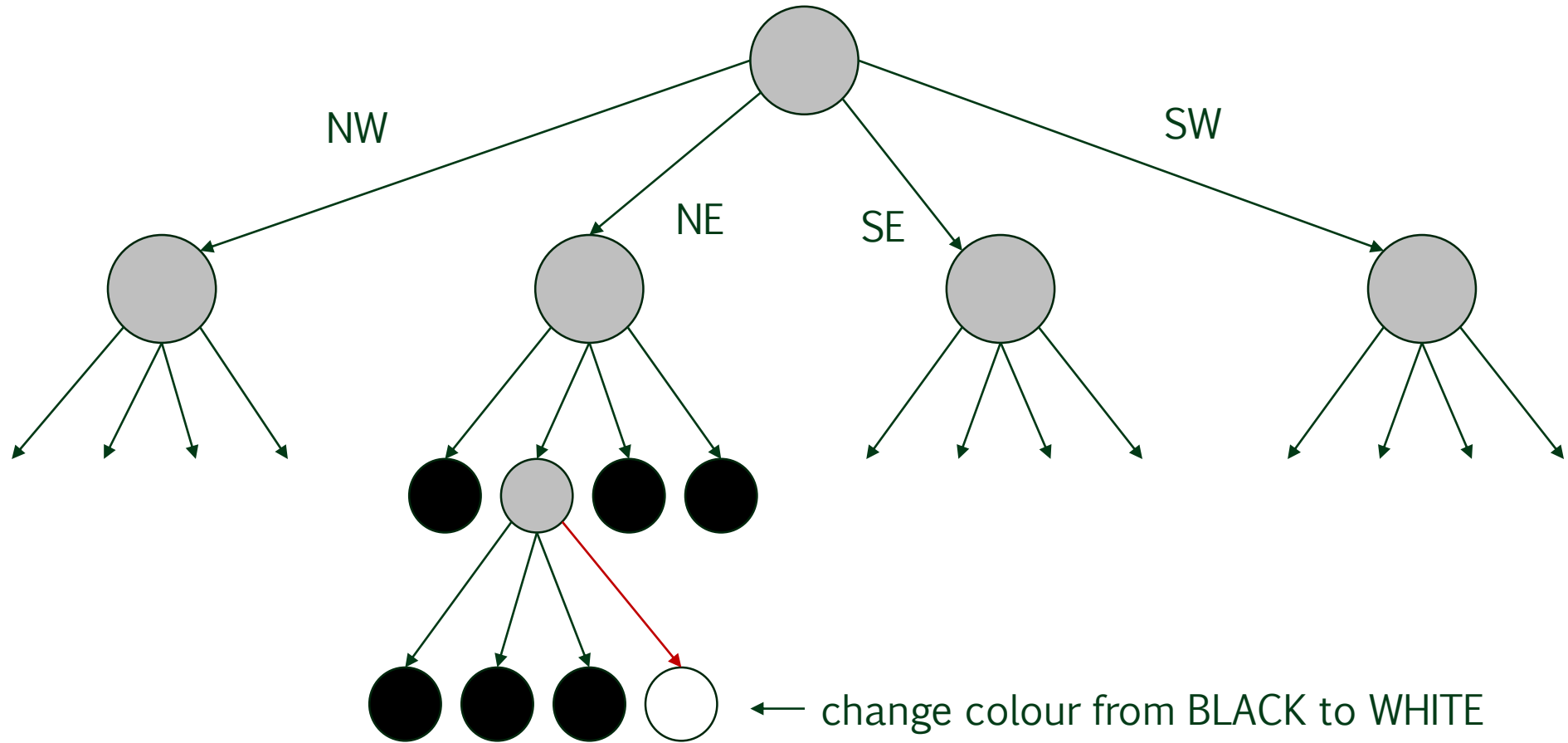


8x8 image (maximum depth of 3)





8x8 image (maximum depth of 3)





Switch(i,j)

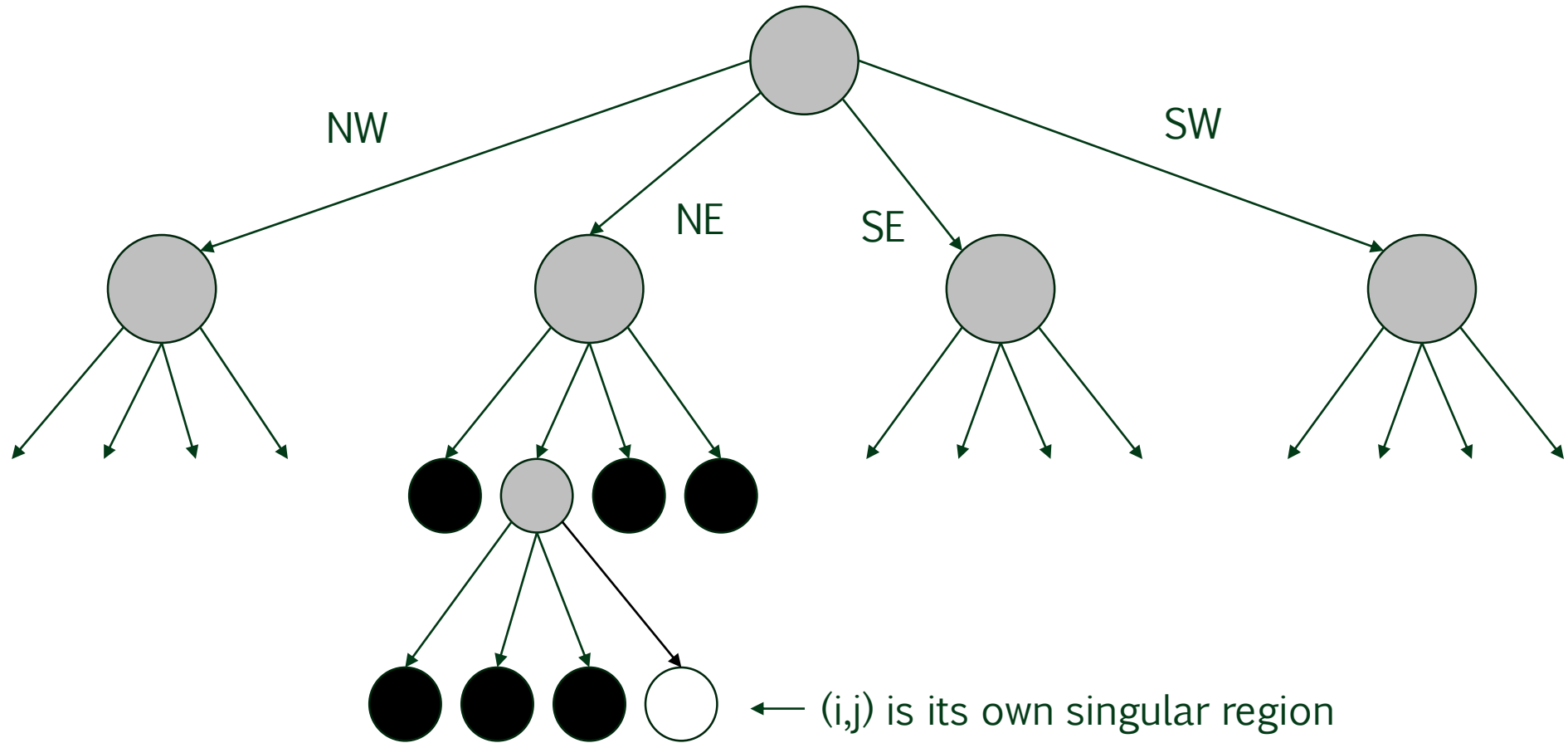
- › Basic strategy (two scenarios)

- Second scenario

- › If index (i,j) is found in its own 1x1 singular region then the colour of the leaf node is set to its opposite (BLACK to WHITE or WHITE to BLACK) and the quadtree is compressed (if possible) along to path from (i,j) to the root

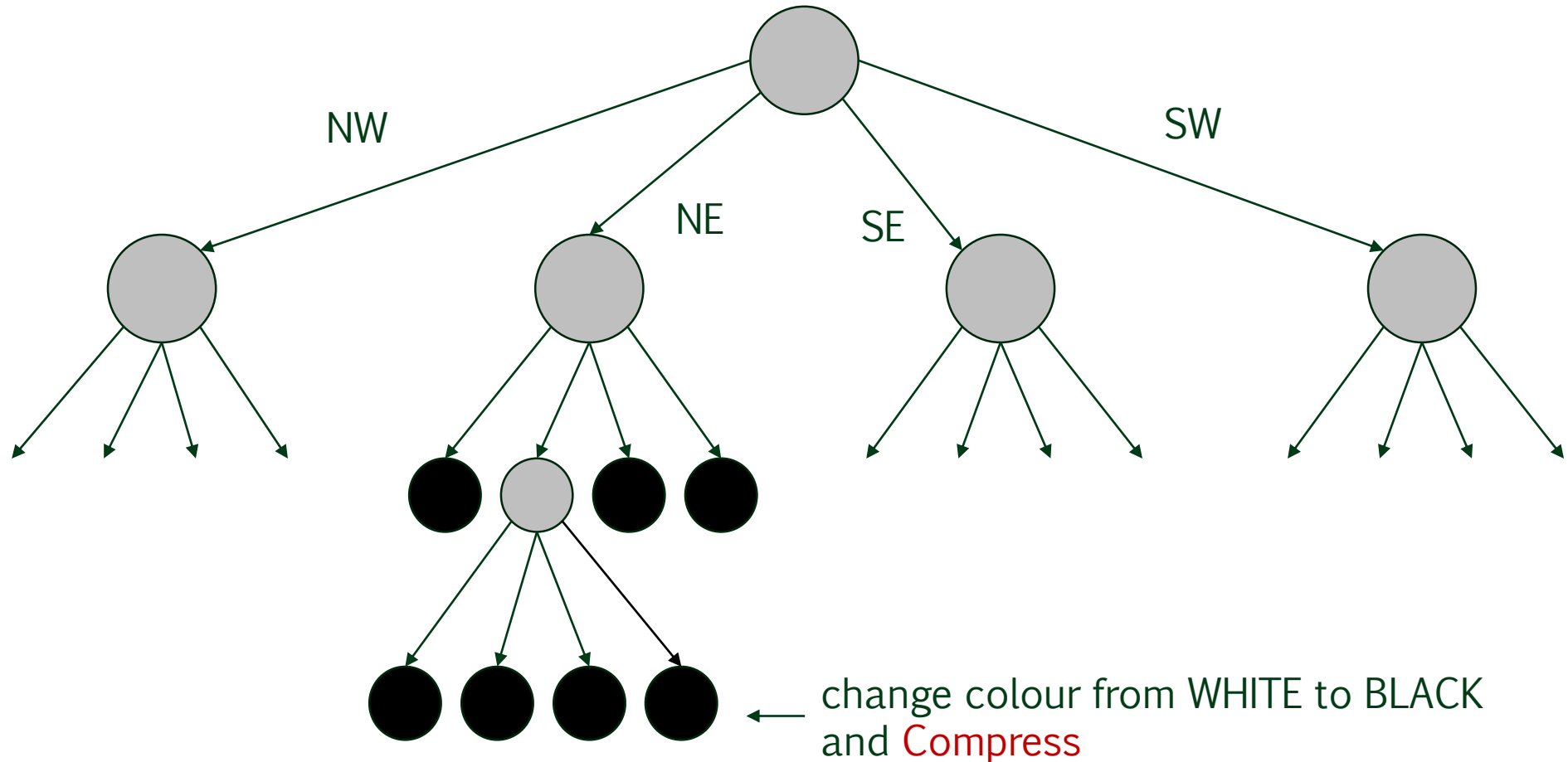


8x8 image (maximum depth of 3)



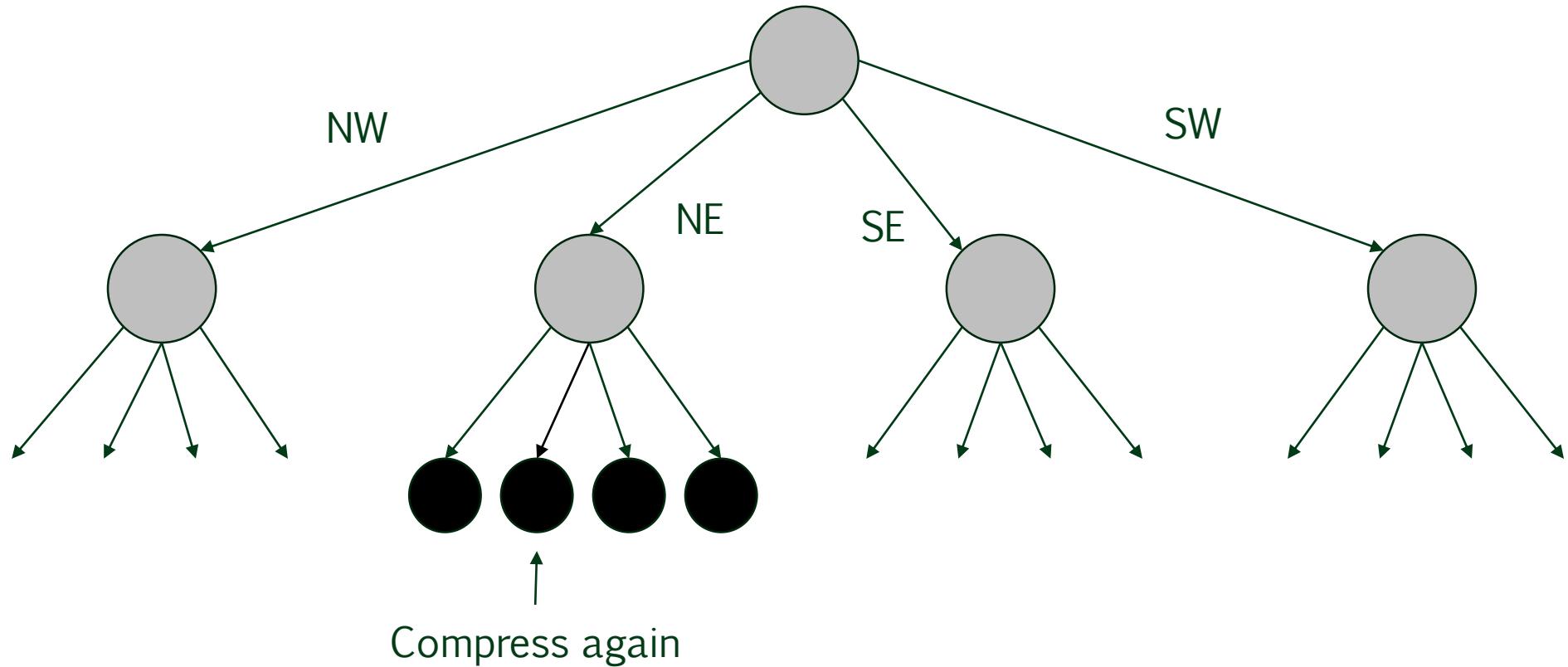


8x8 image (maximum depth of 3)



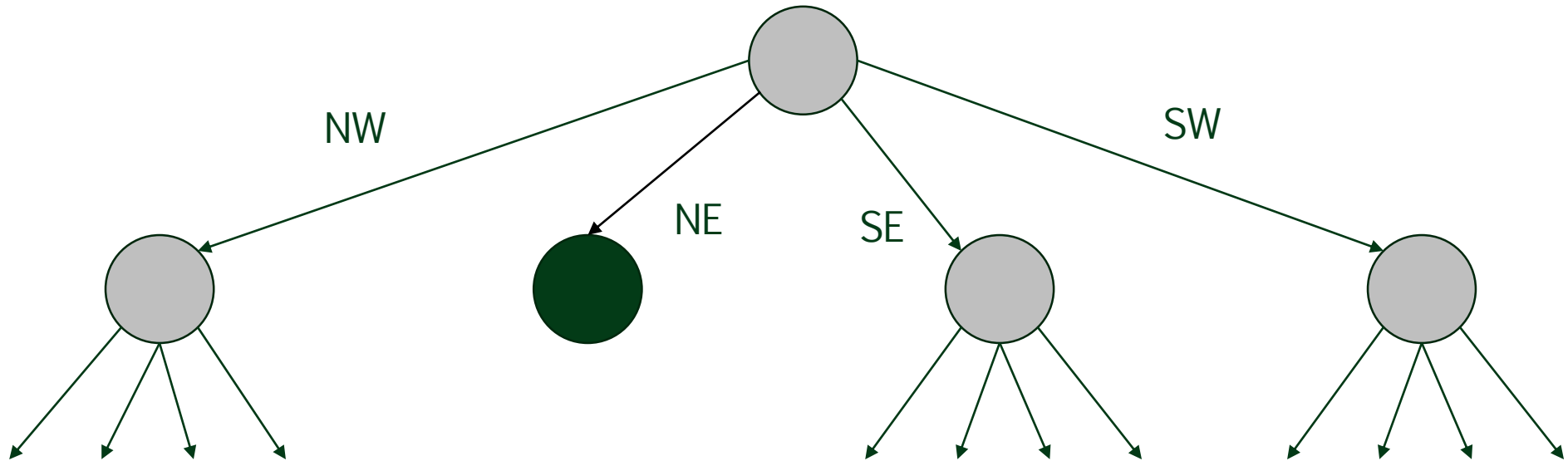


8x8 image (maximum depth of 3)





8x8 image (maximum depth of 3)





Time complexity

- › The time complexity of Switch is $O(\log n)$ which is the maximum depth of the quadtree



Union(Q)

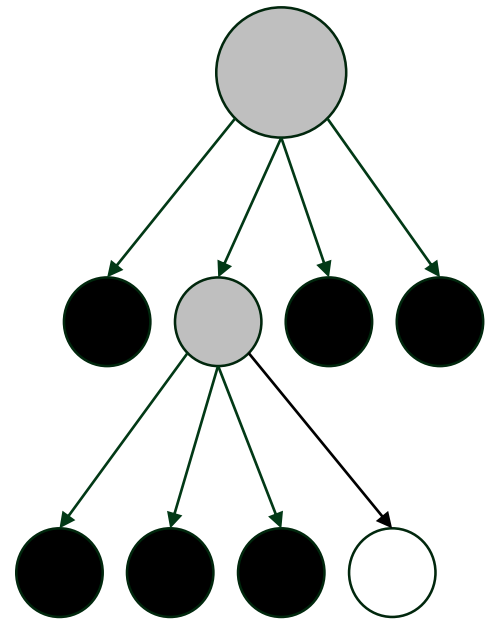
› Basic strategy

- Return a quadtree T where each 1x1 region (say, pixel) of T is defined by the union of corresponding pixels of the current quadtree and given quadtree Q in the following way:

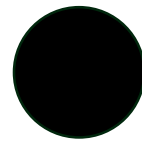
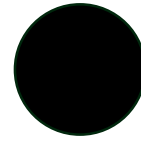
- › the union of two BLACK pixels is a BLACK pixel
- › the union of a BLACK and WHITE pixel is a BLACK pixel
- › the union of two WHITE pixels is a WHITE pixel

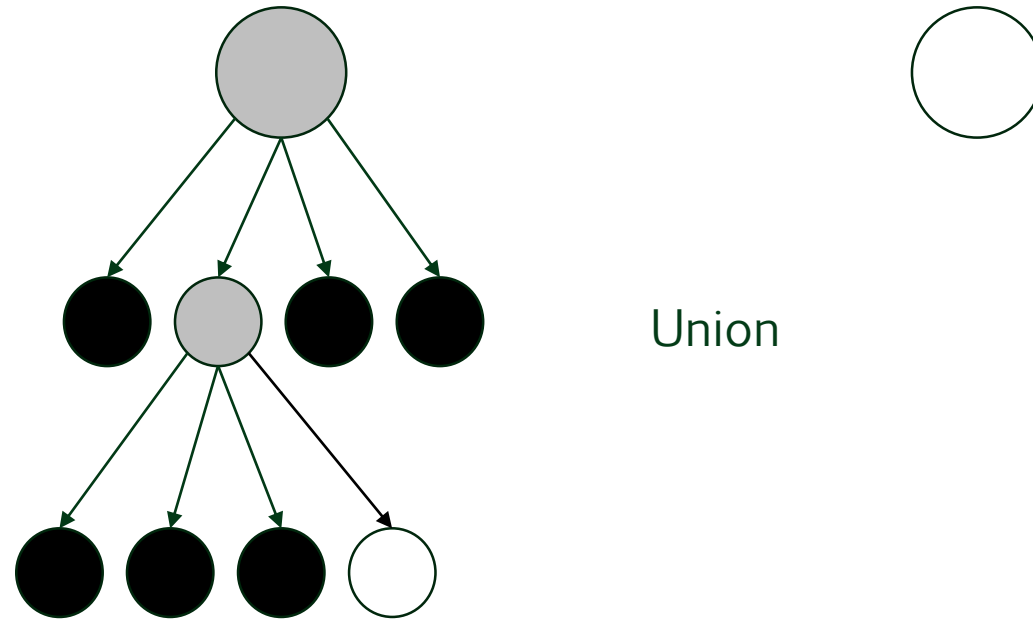
●	●	●
●	○	●
○	○	○

- The union defined in this way implies that:
 - › the union of a BLACK region with any corresponding region is a BLACK region
 - › the union of a WHITE region with any corresponding region R is the region R
 - › the union of two GRAY regions is the union of the corresponding children

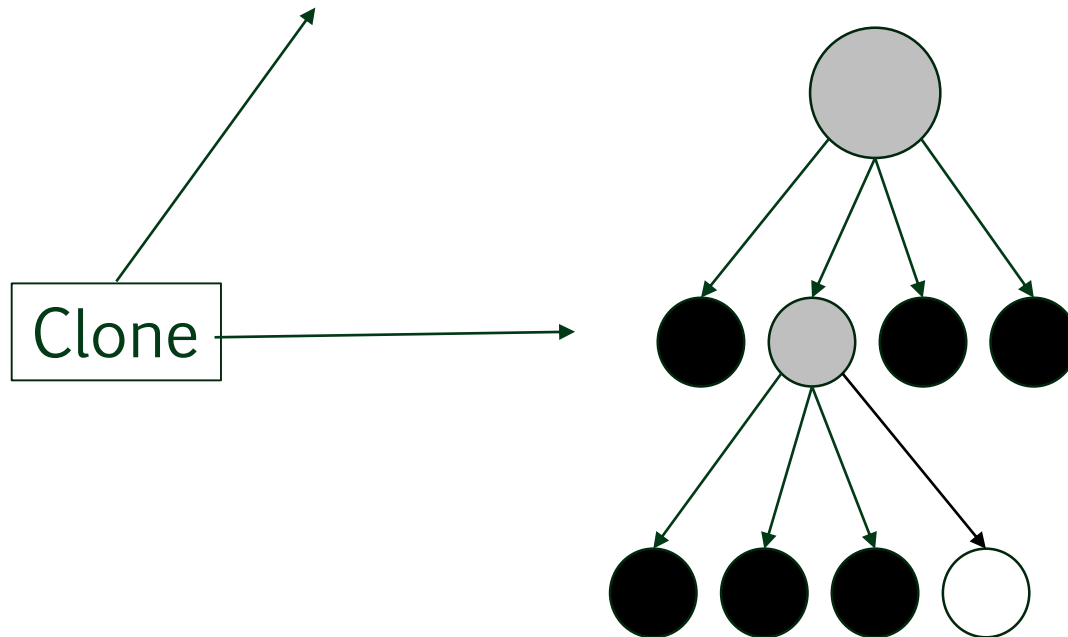


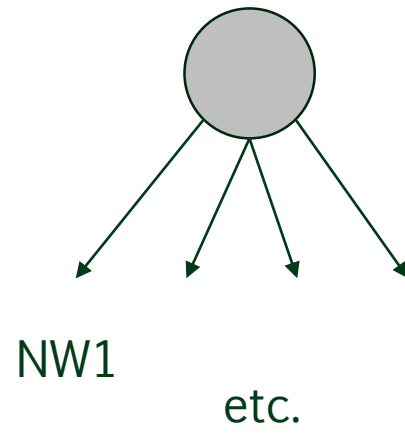
Union



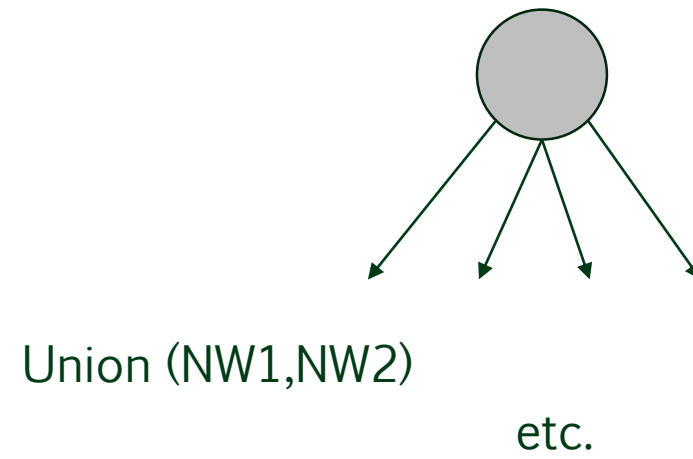
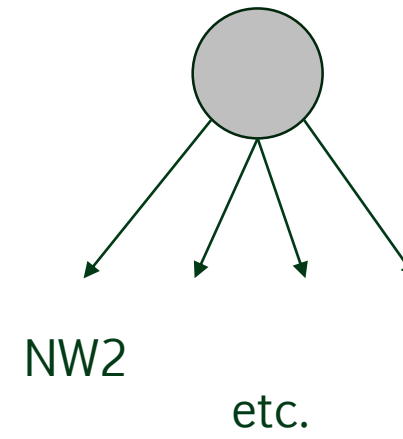


Union





Union





Time complexity

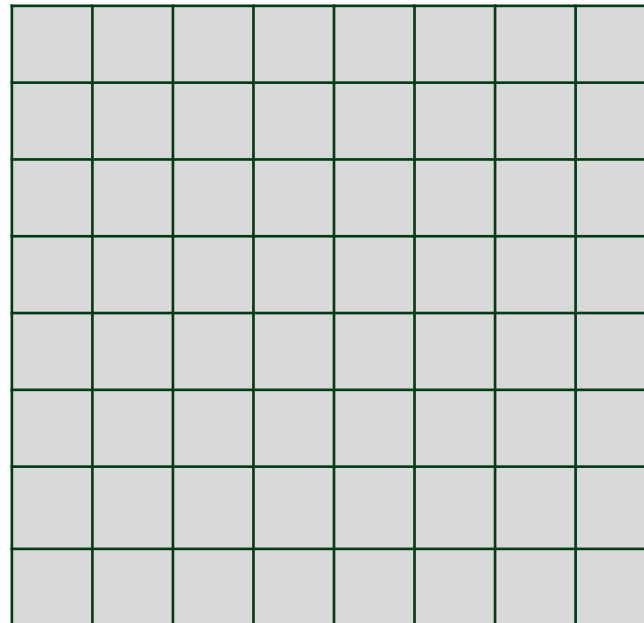
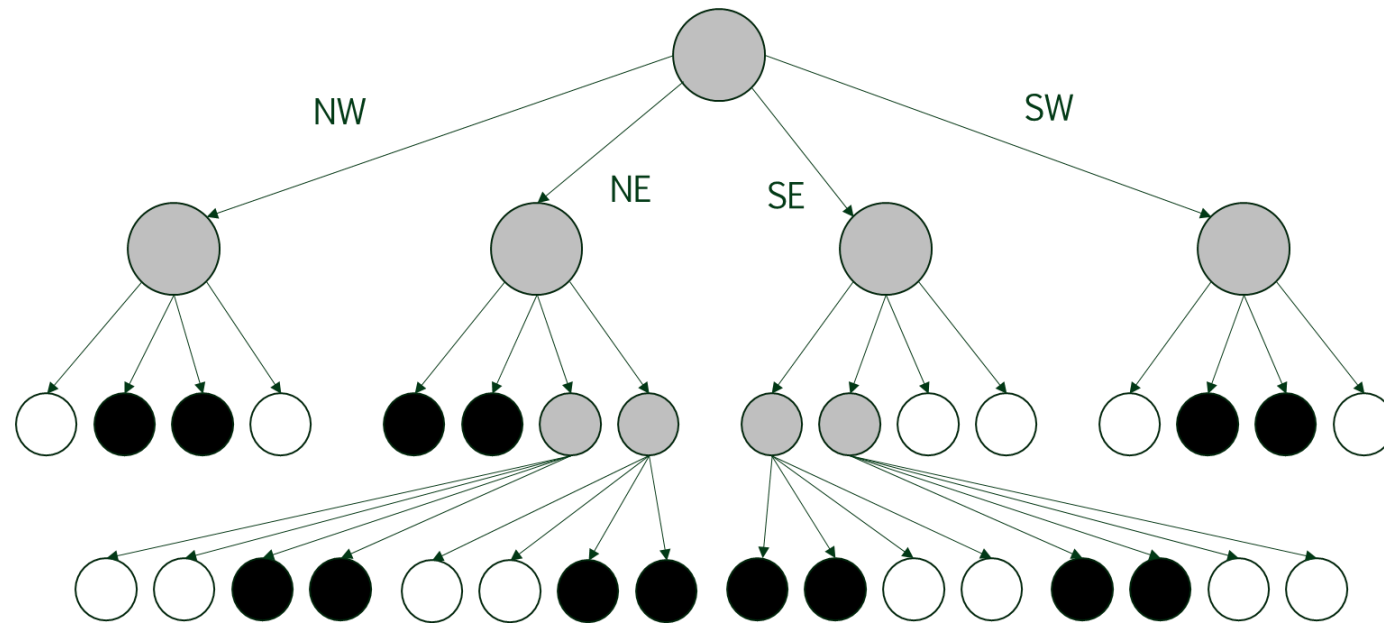
- › The time complexity of Union is $O(n^2)$ where n is the side length of the image

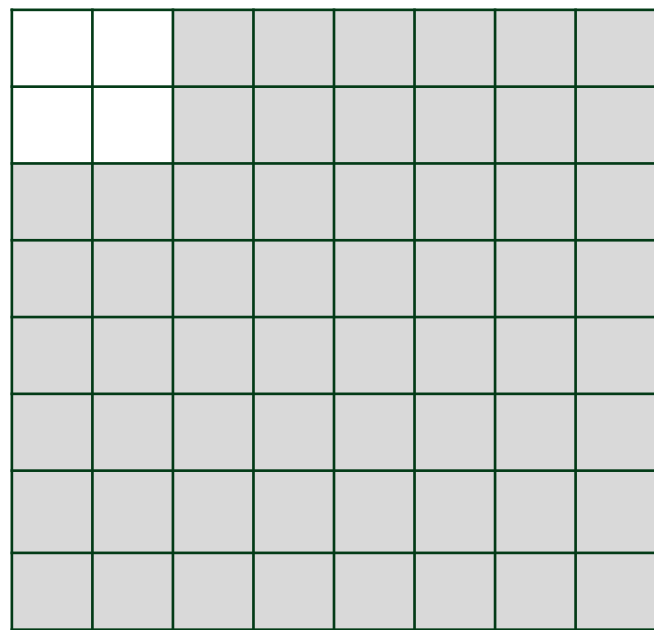
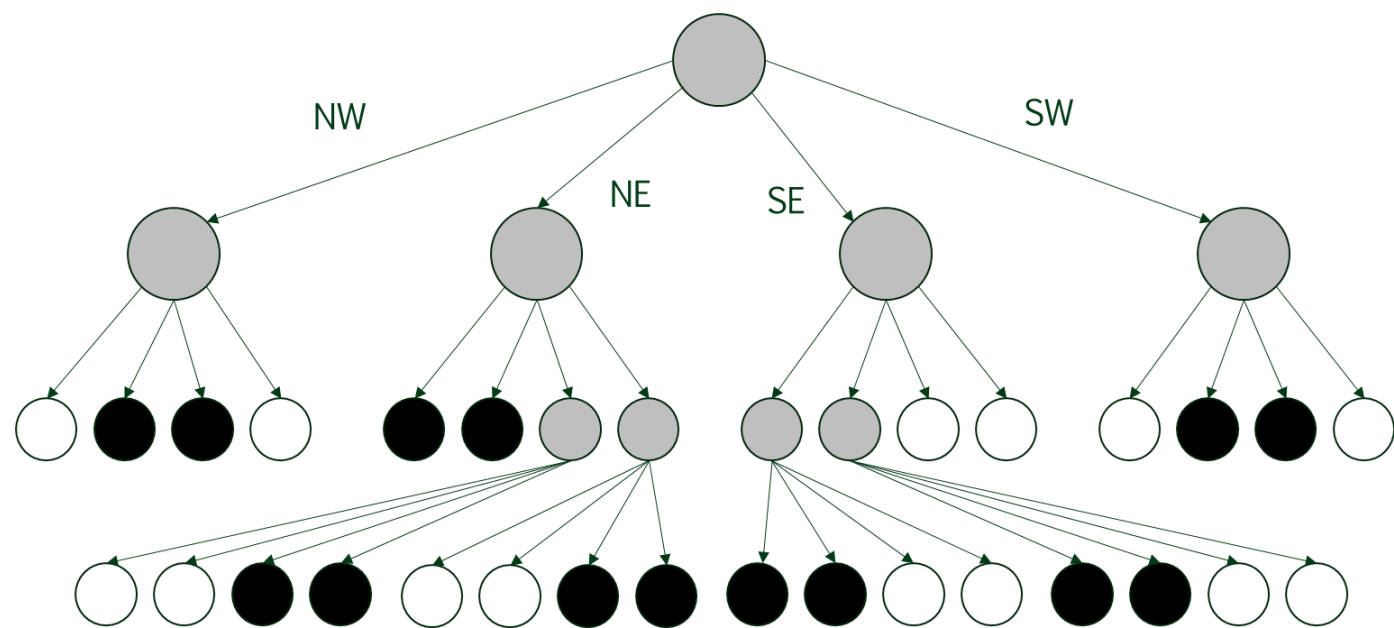


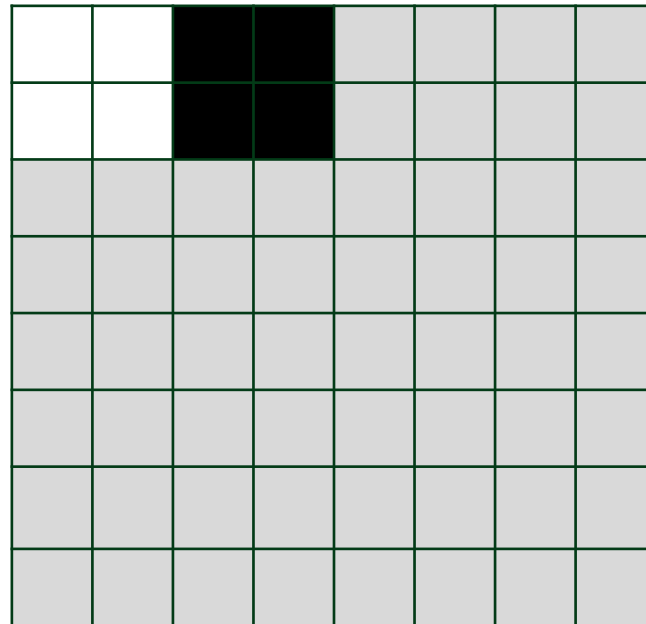
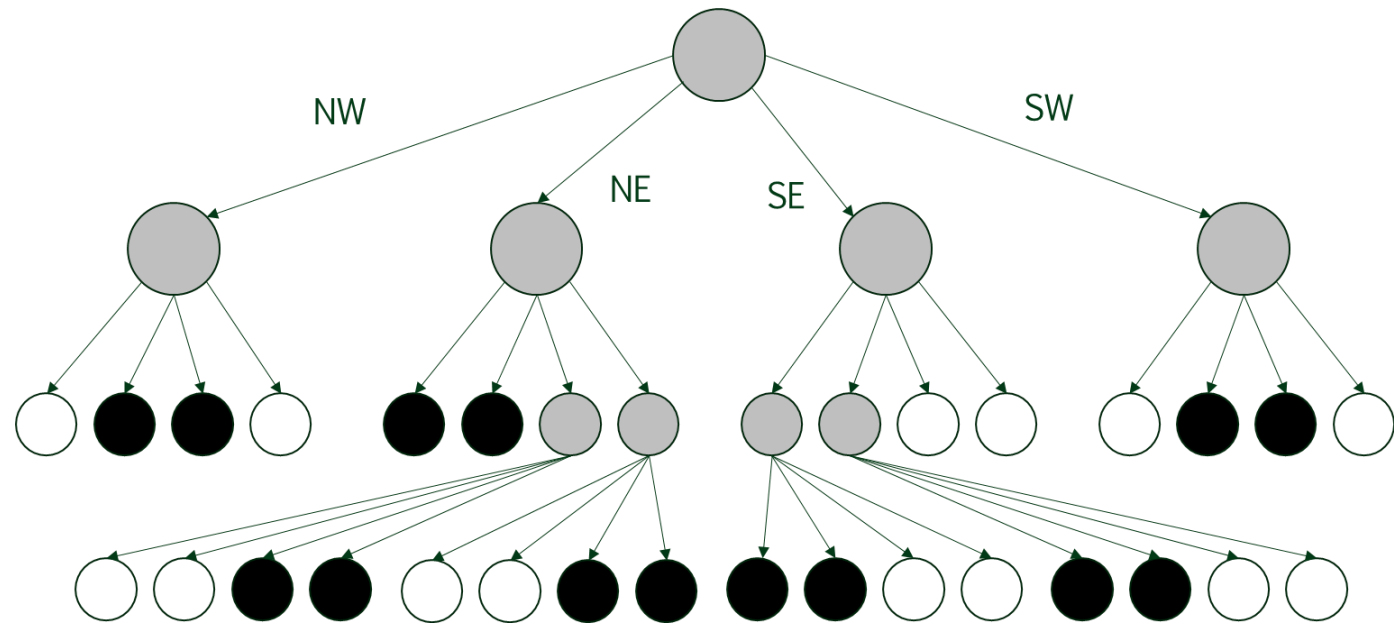
Print (using FillIn and ColorRegion)

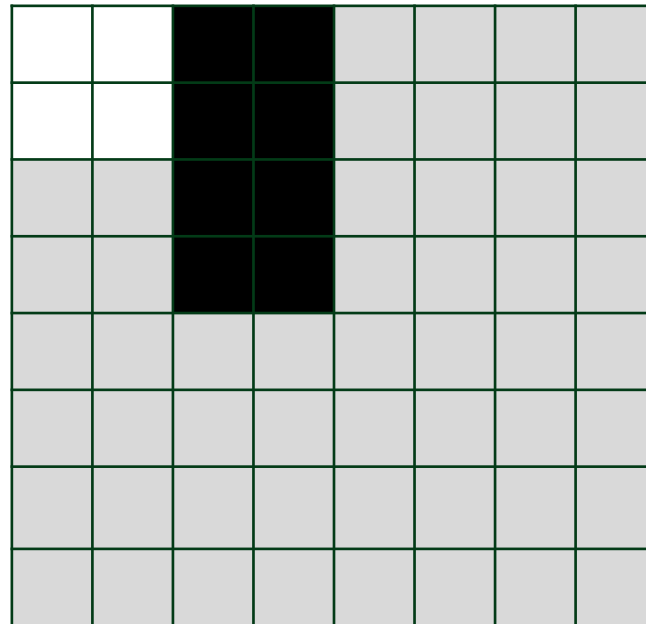
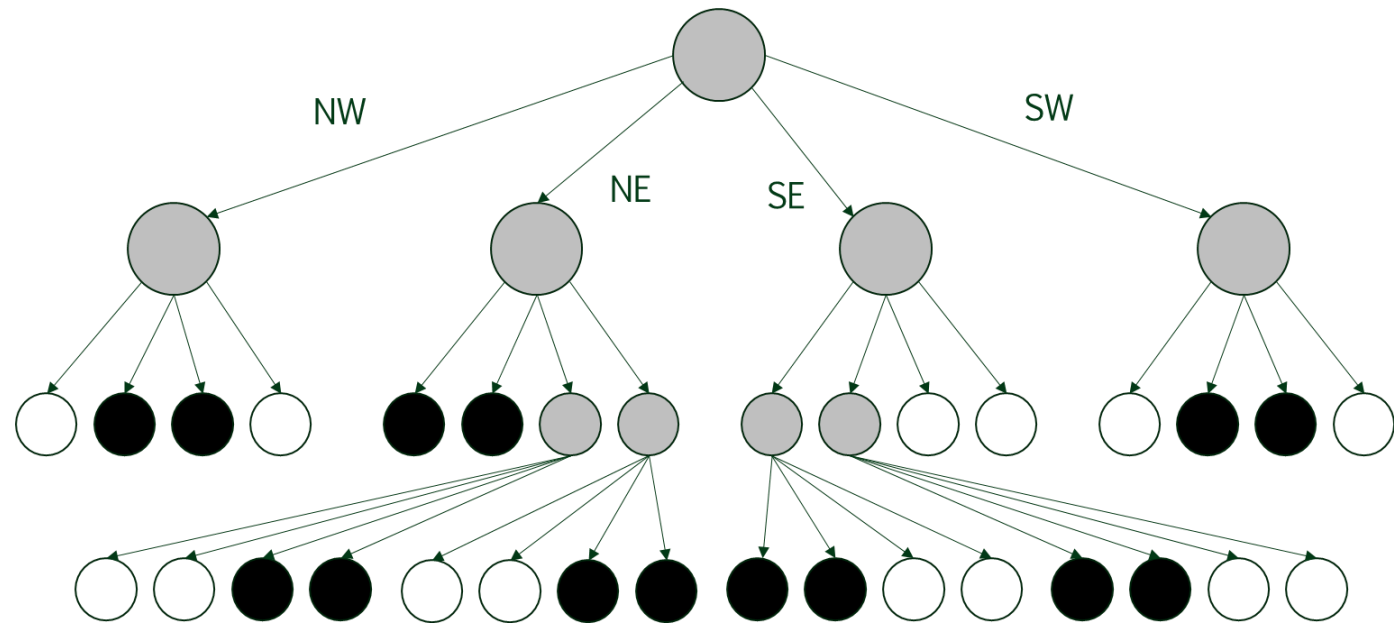
› Basic strategy

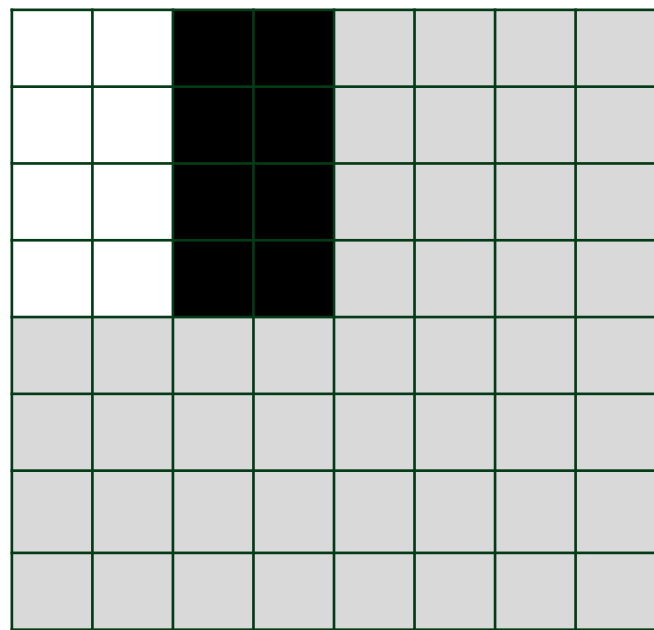
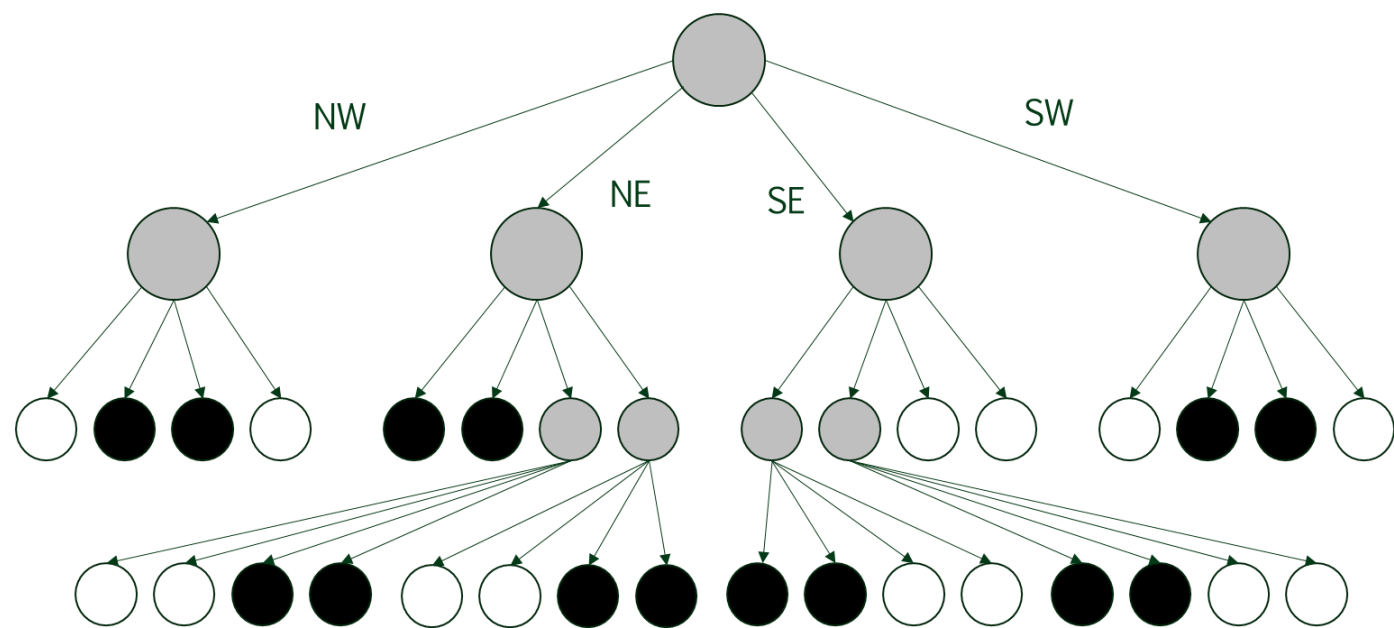
- Recursively fill in each quadrant of the image
- Once a leaf node is reached, the image is coloured either BLACK or WHITE

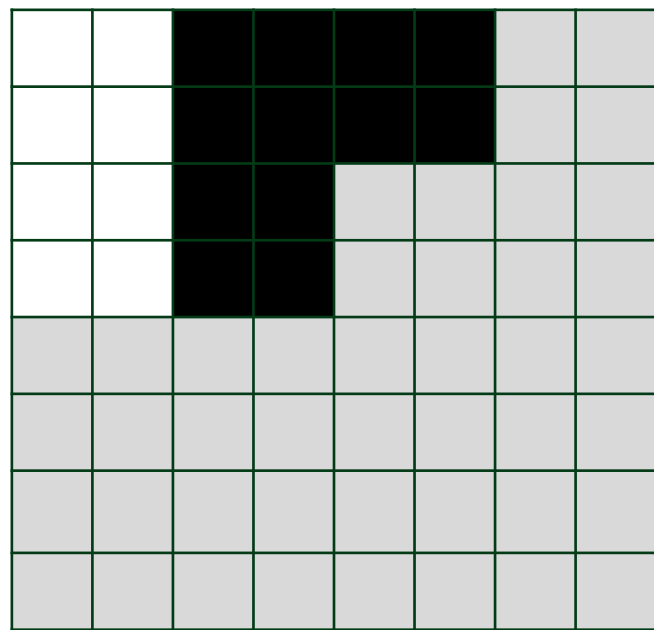
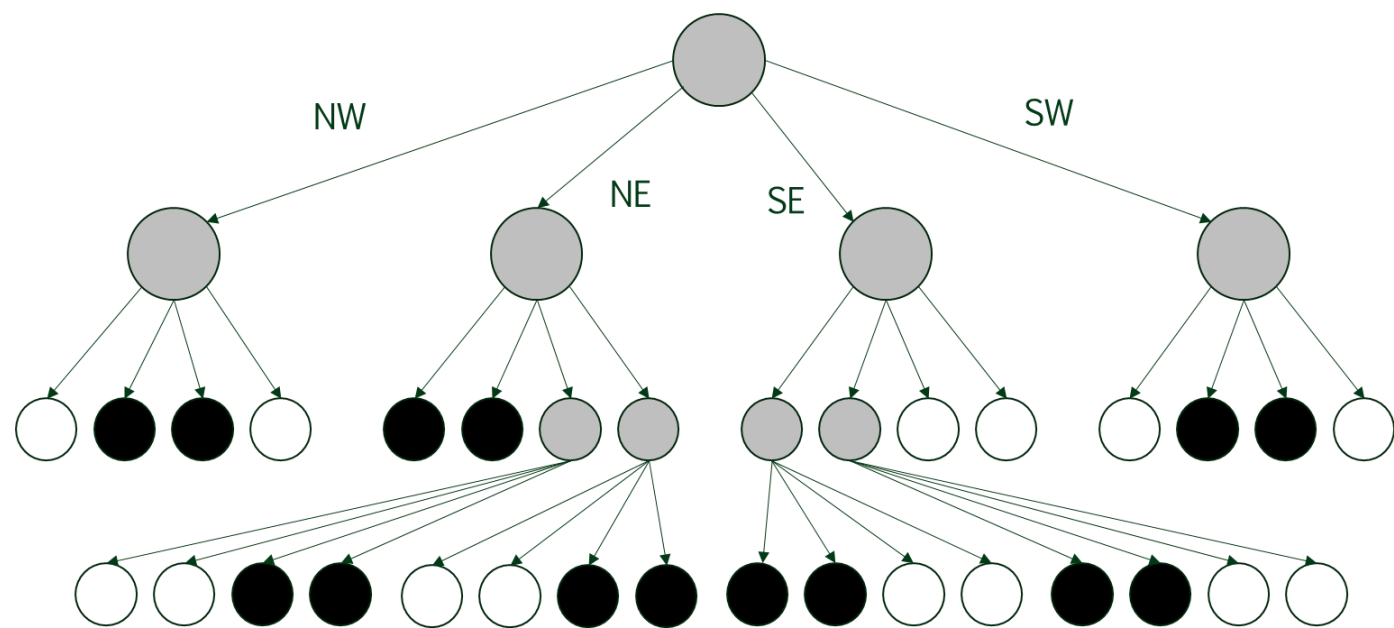


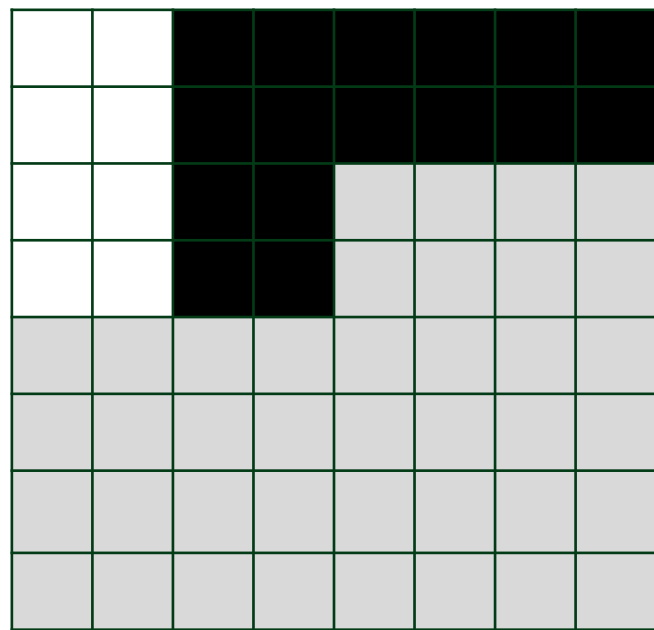
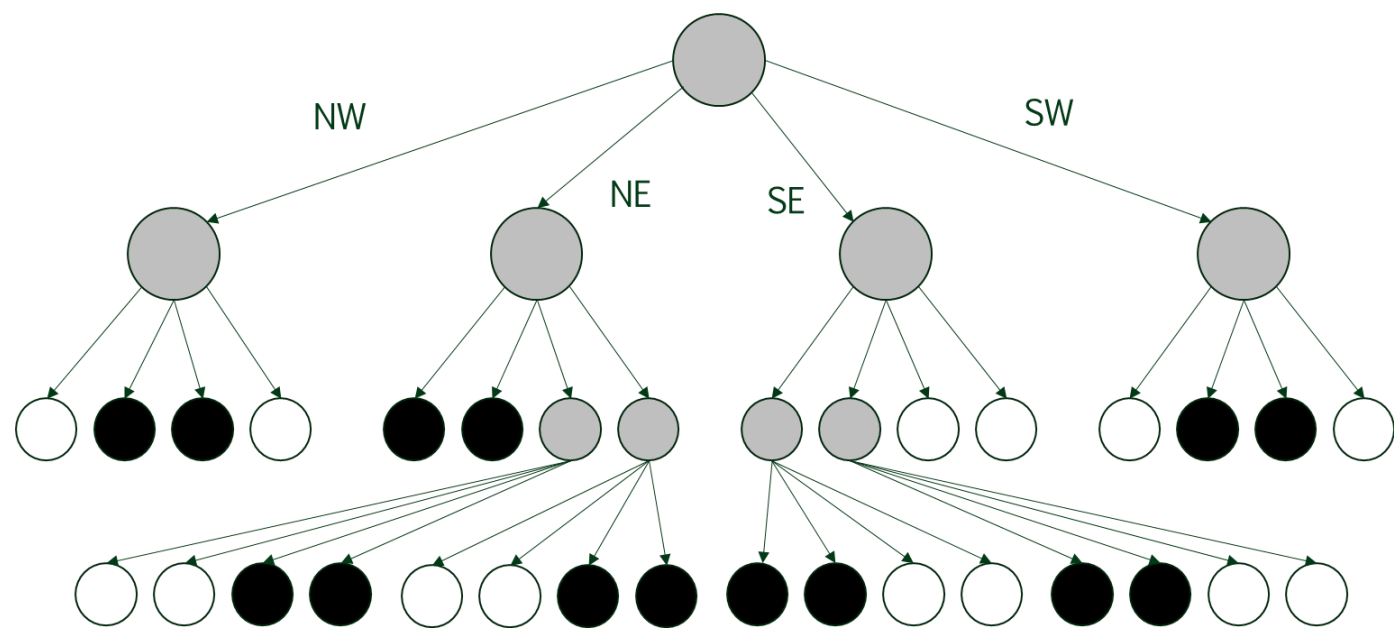


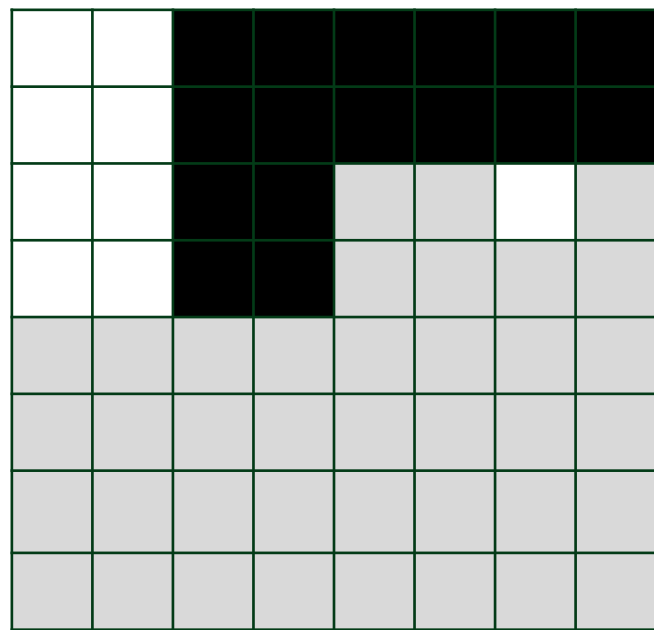
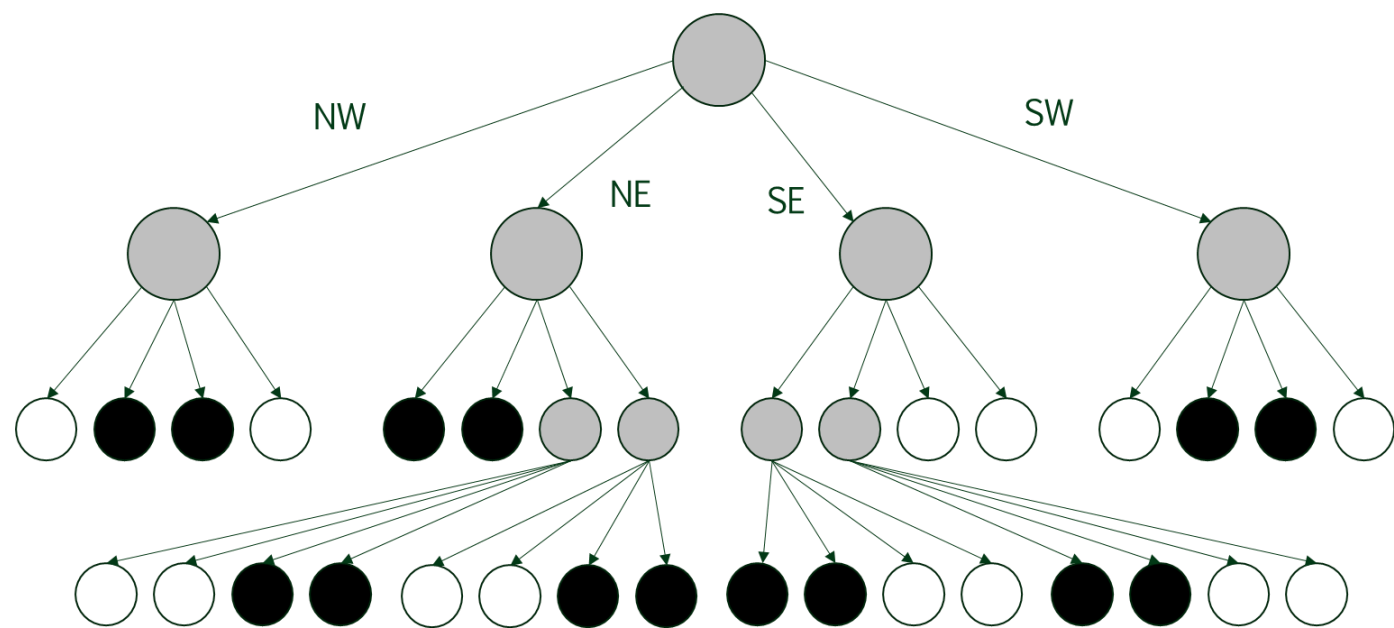


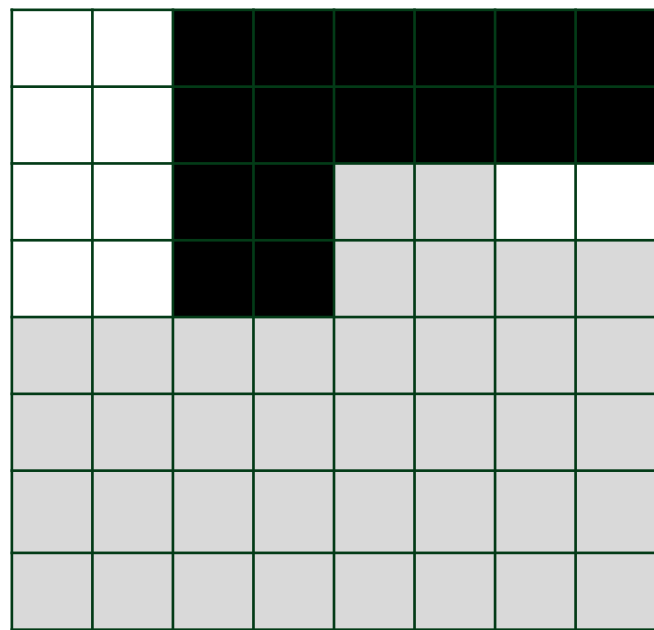
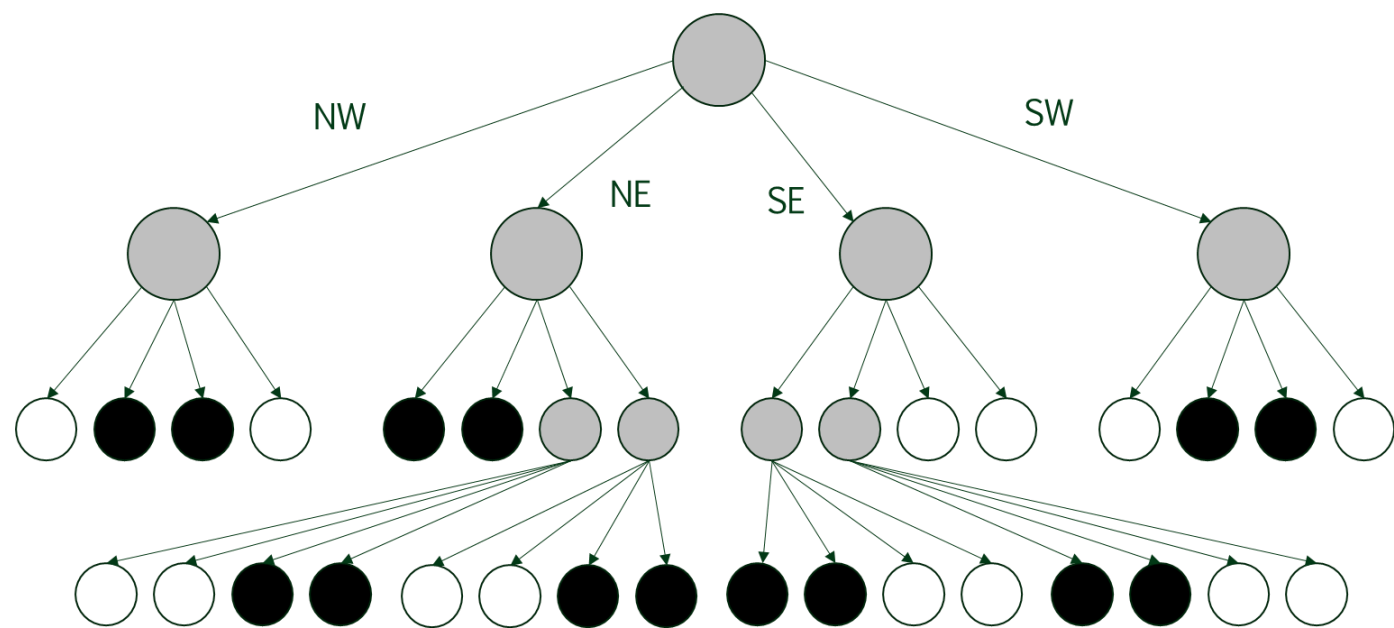


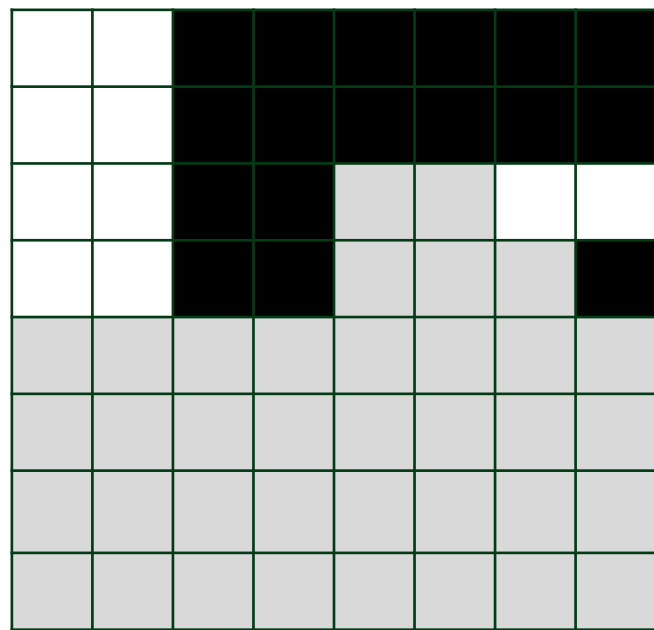
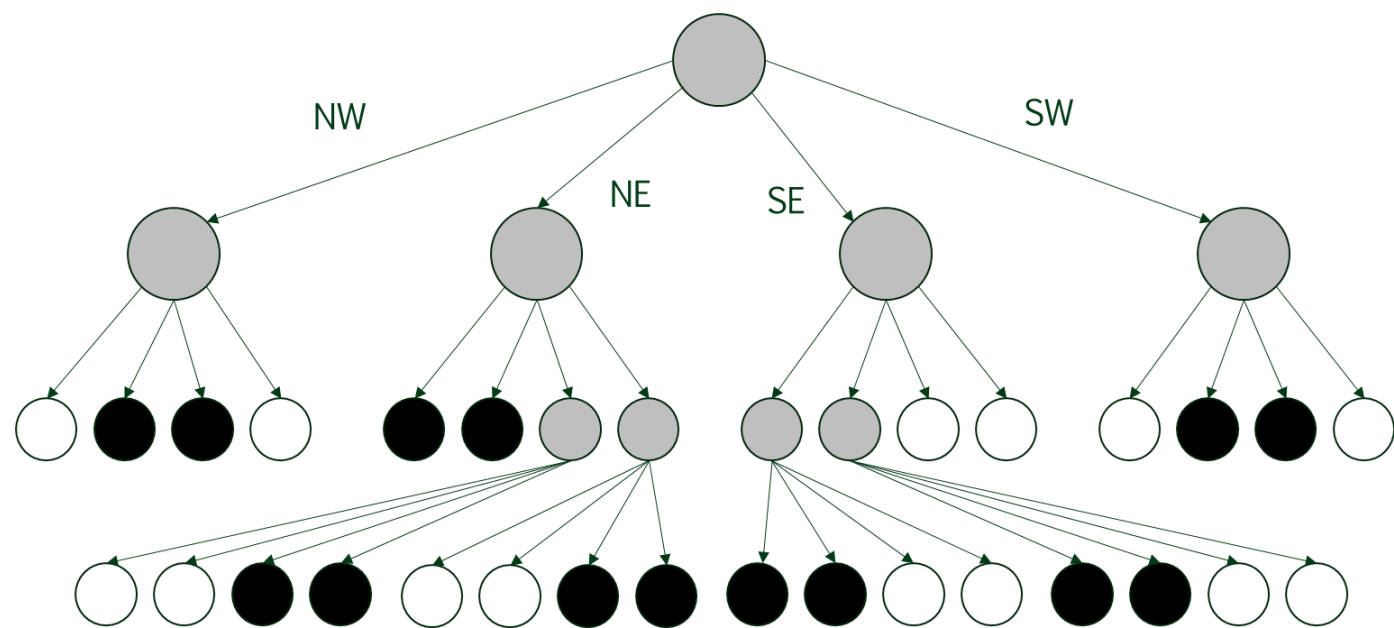


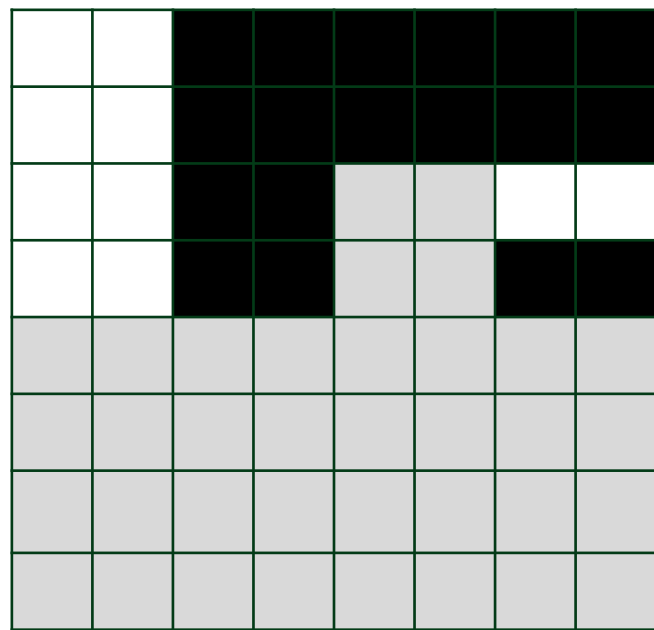
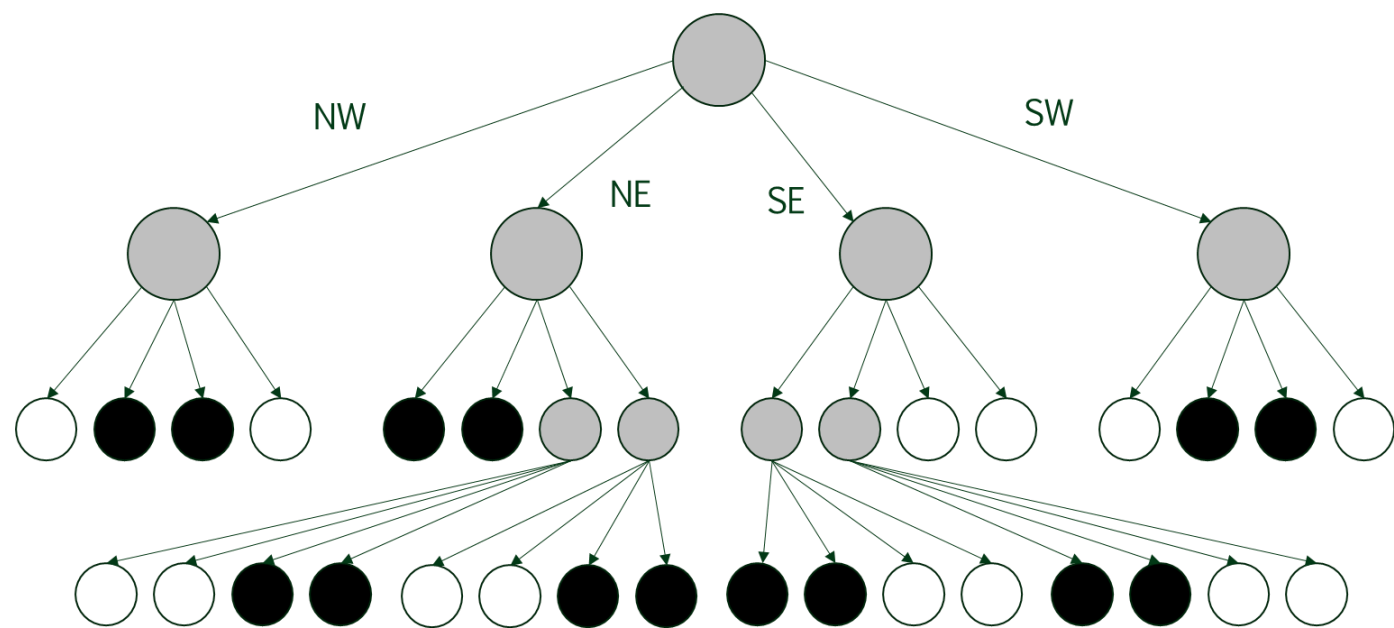


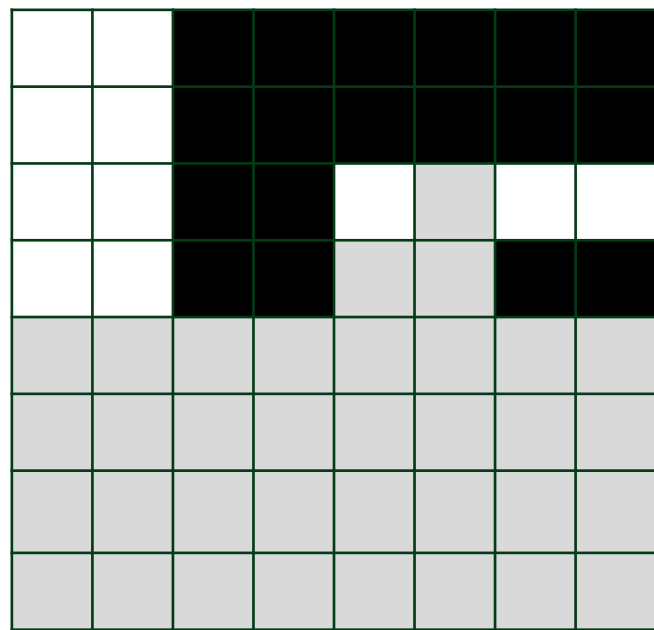
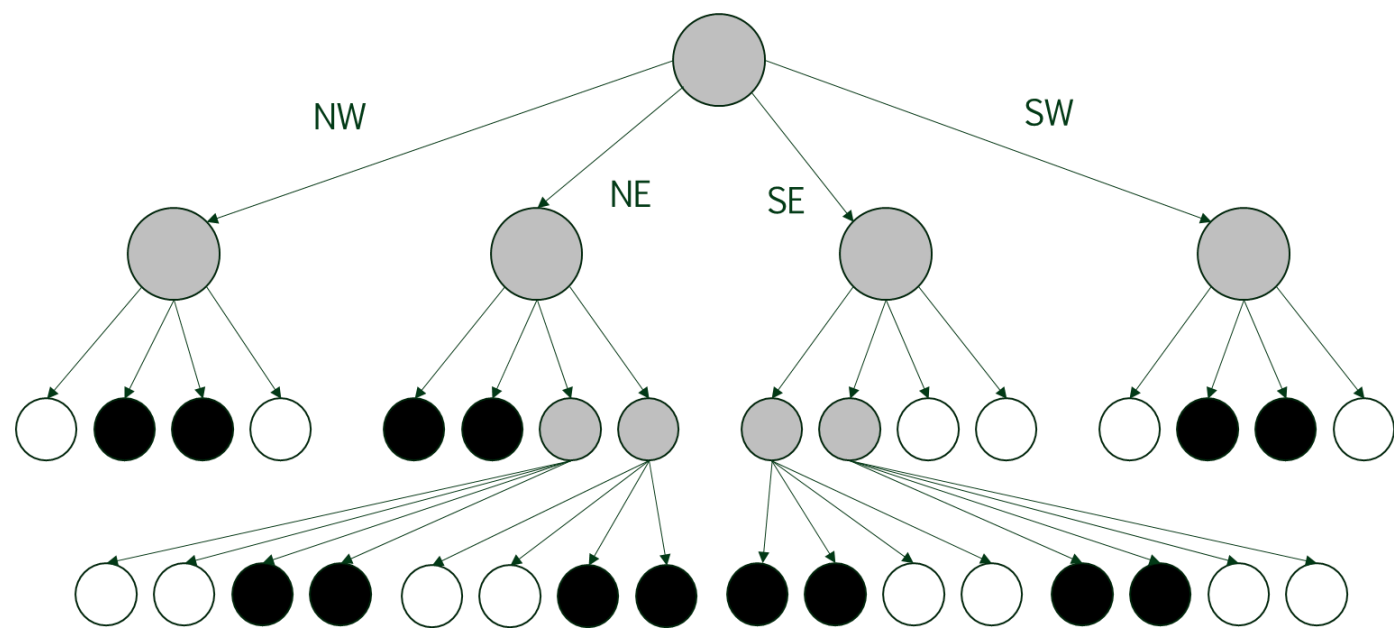


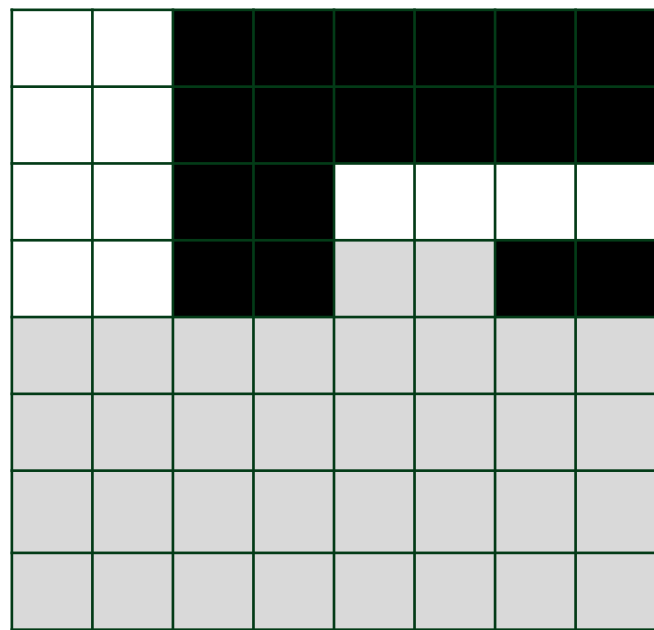
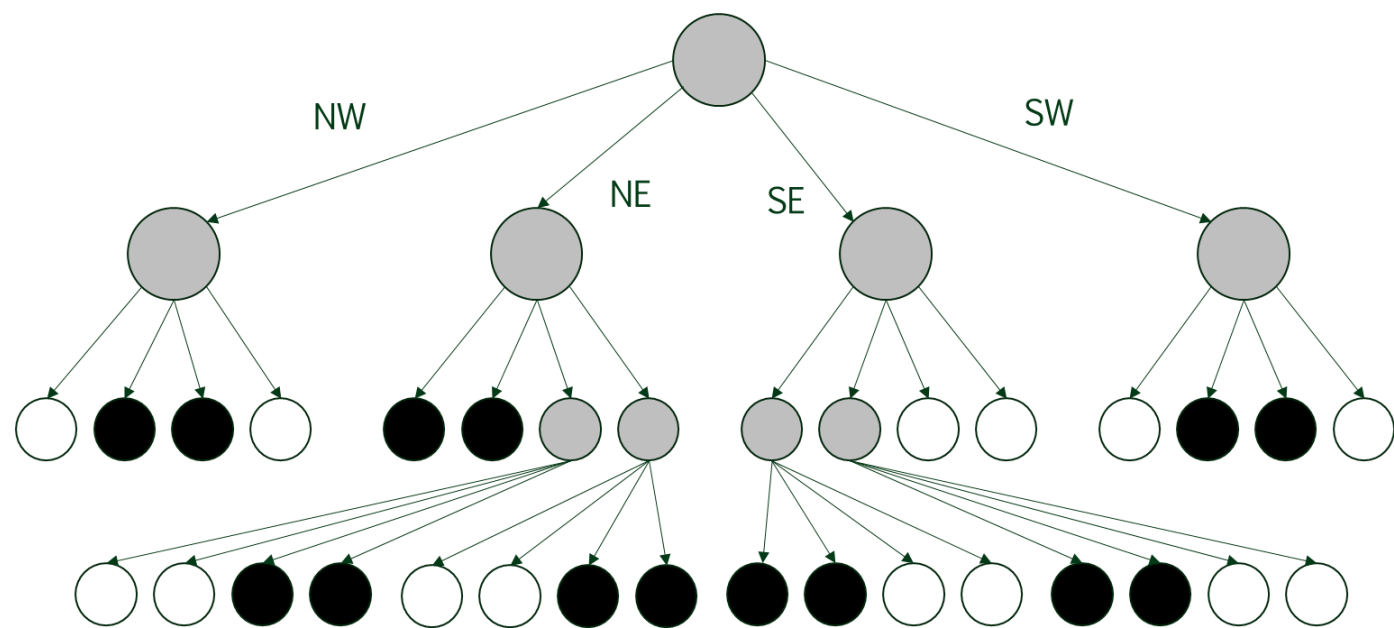


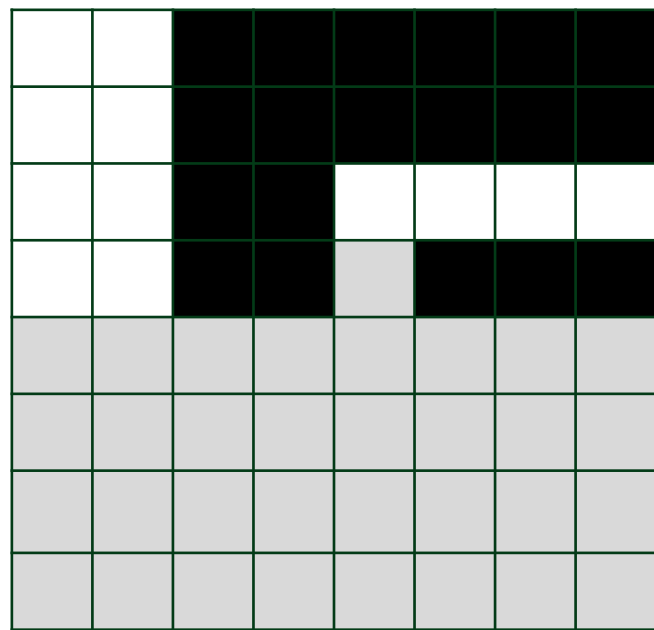
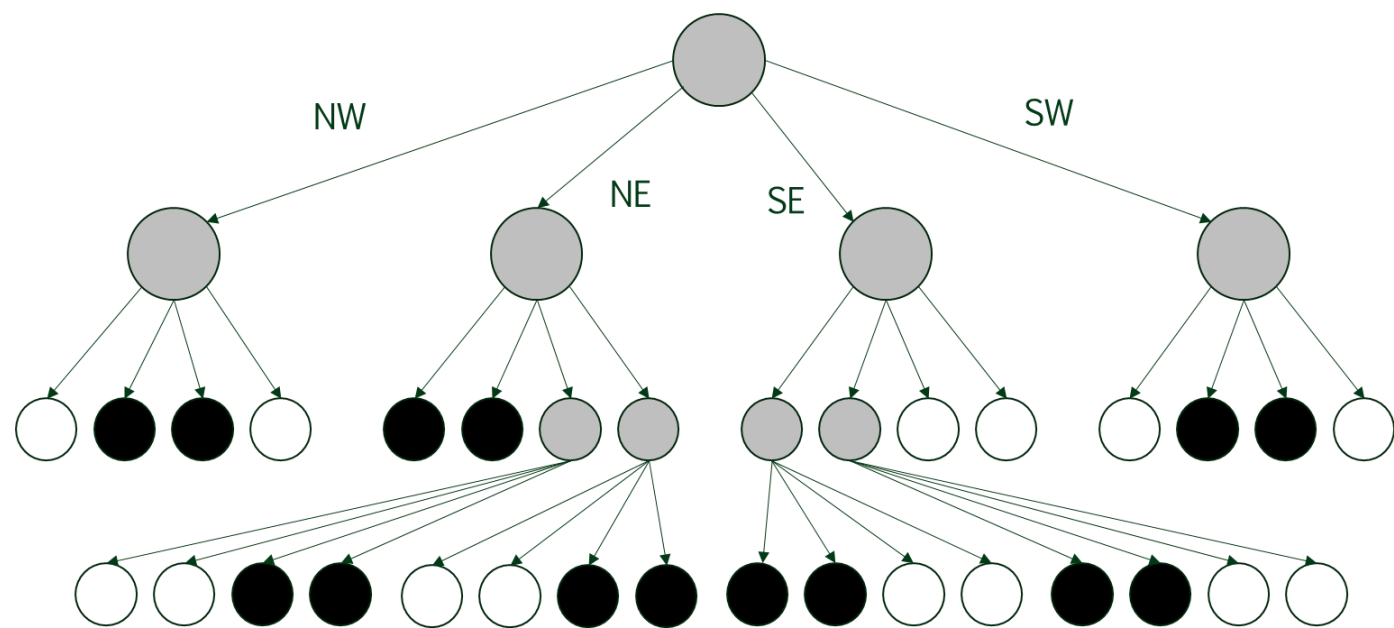


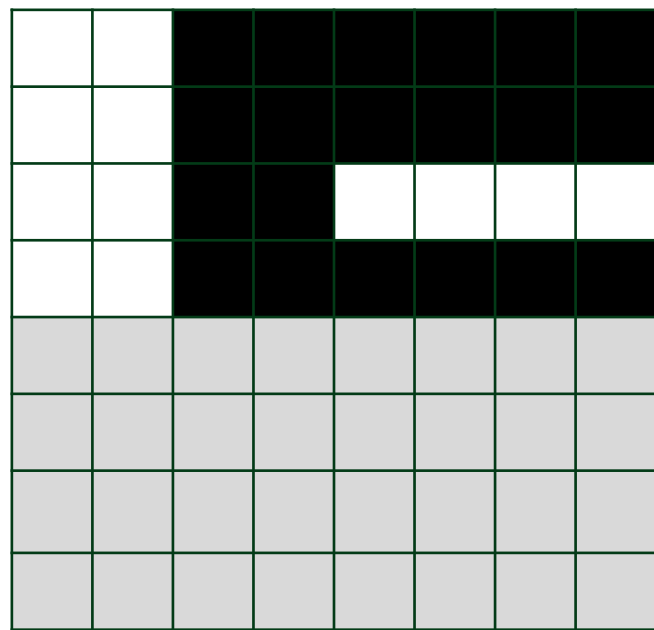
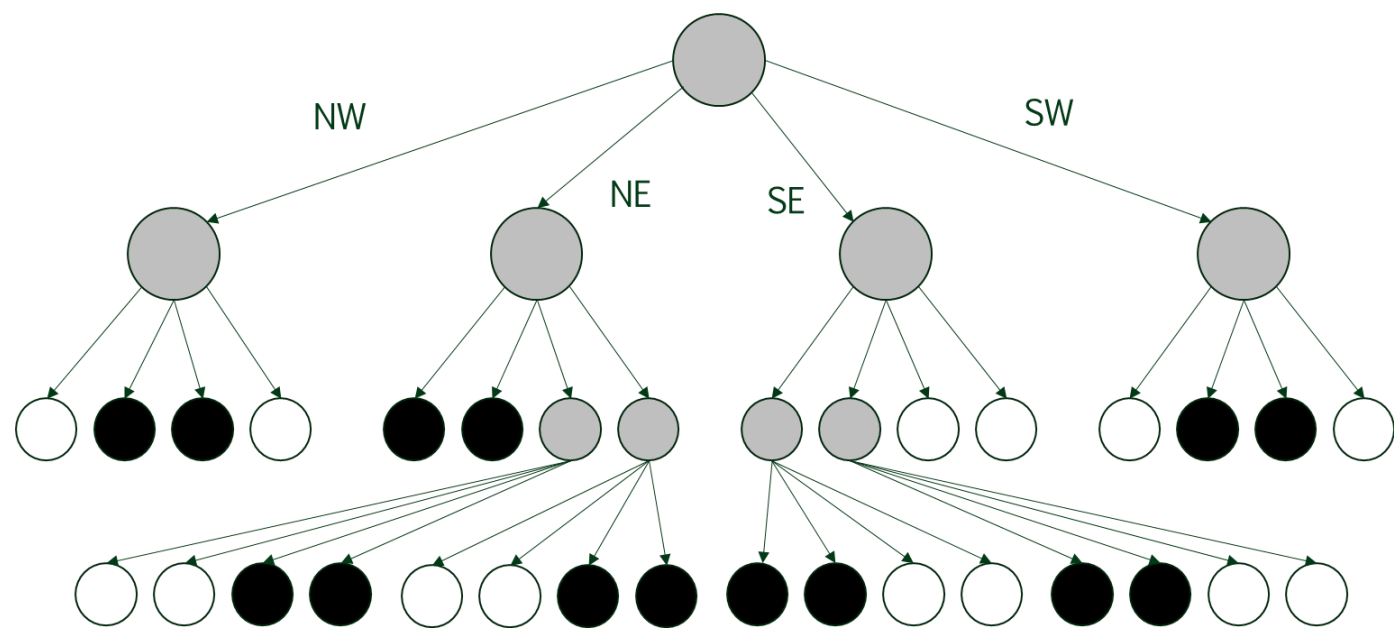


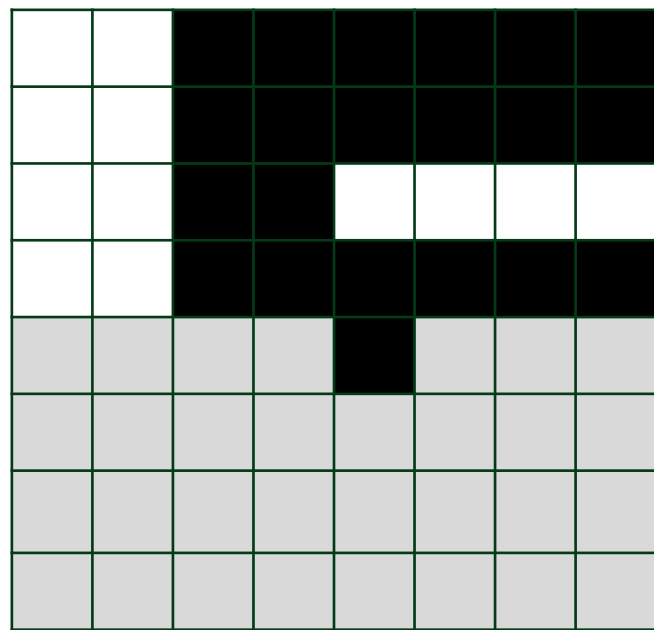
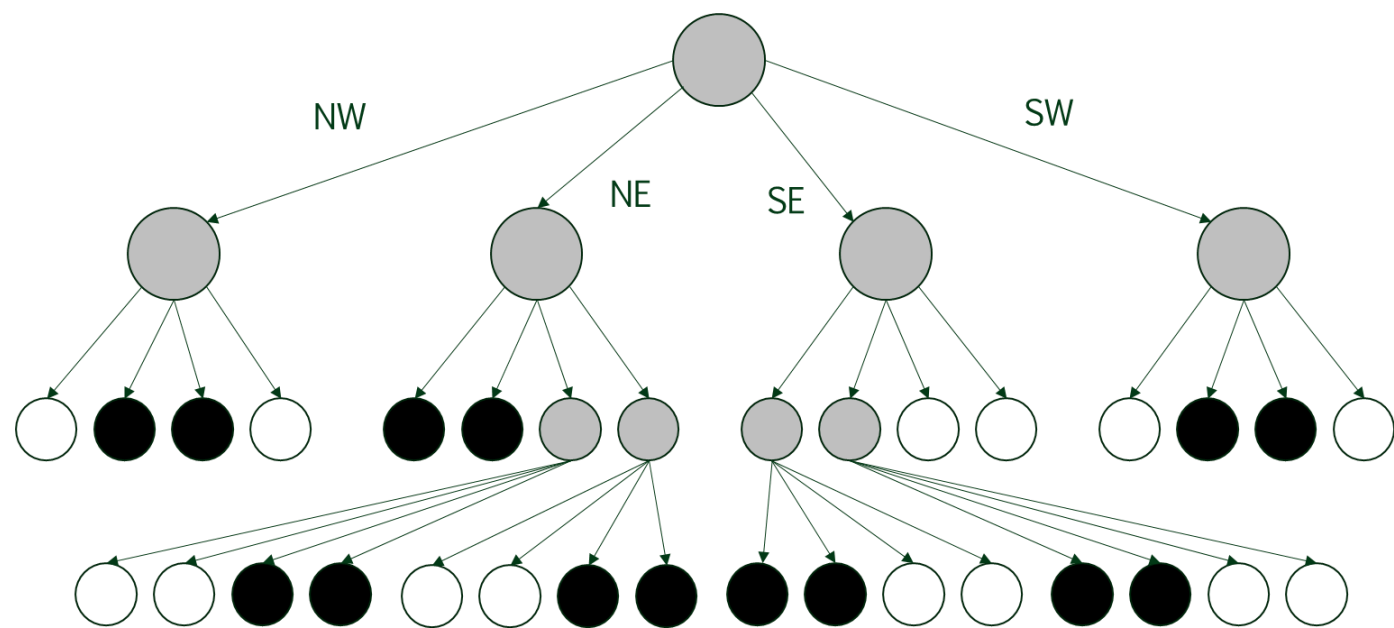


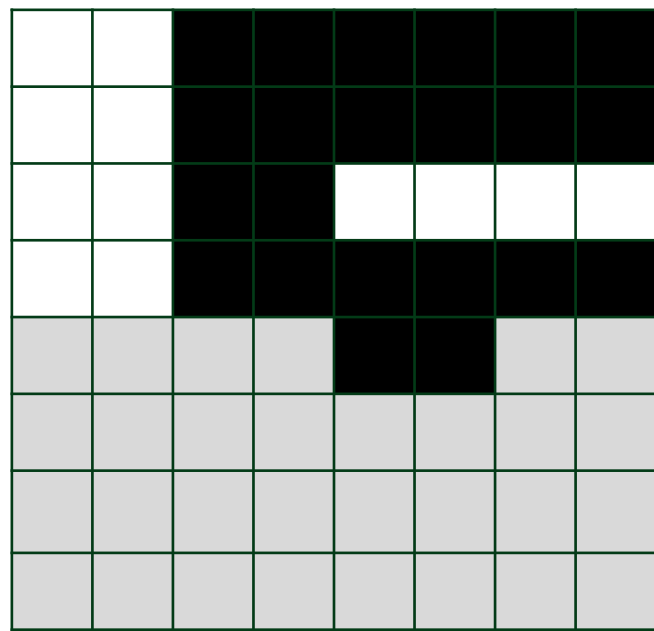
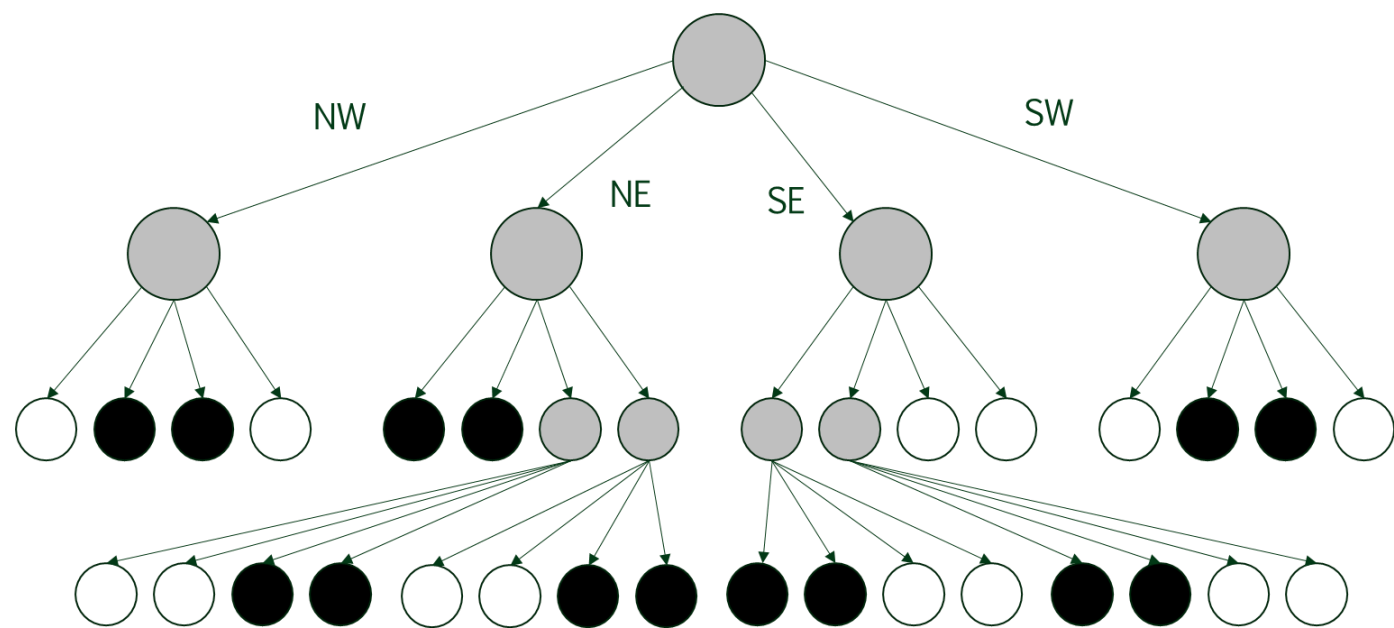


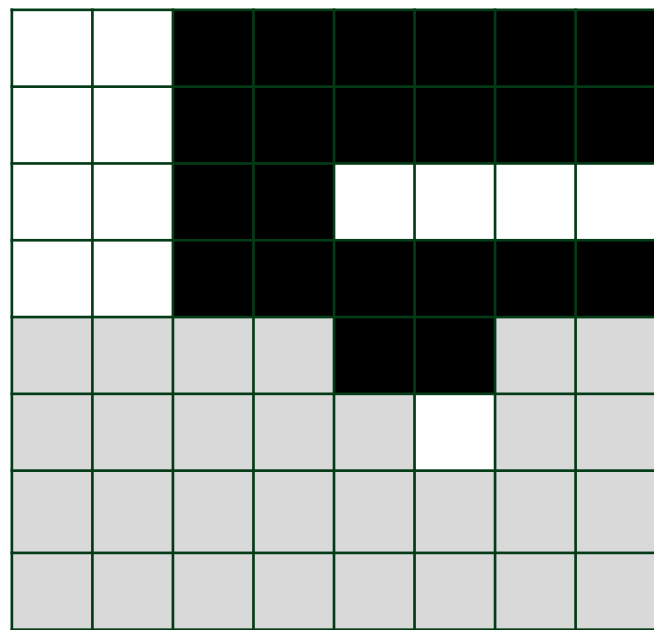
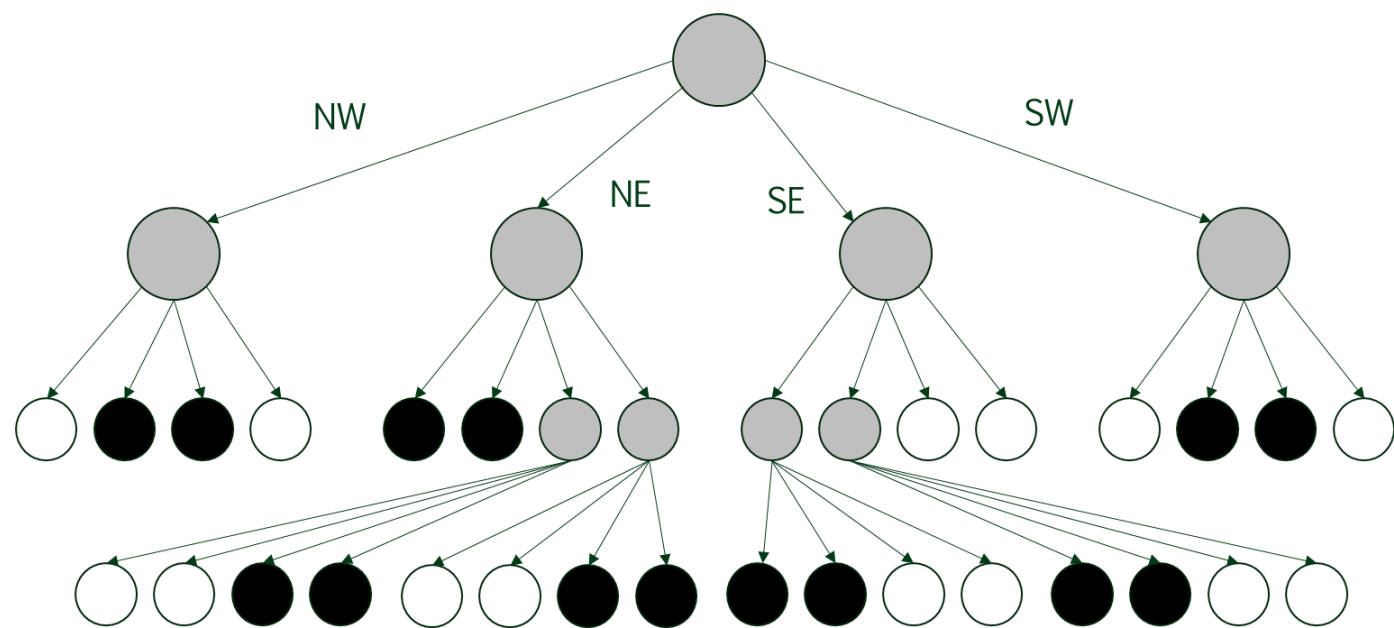


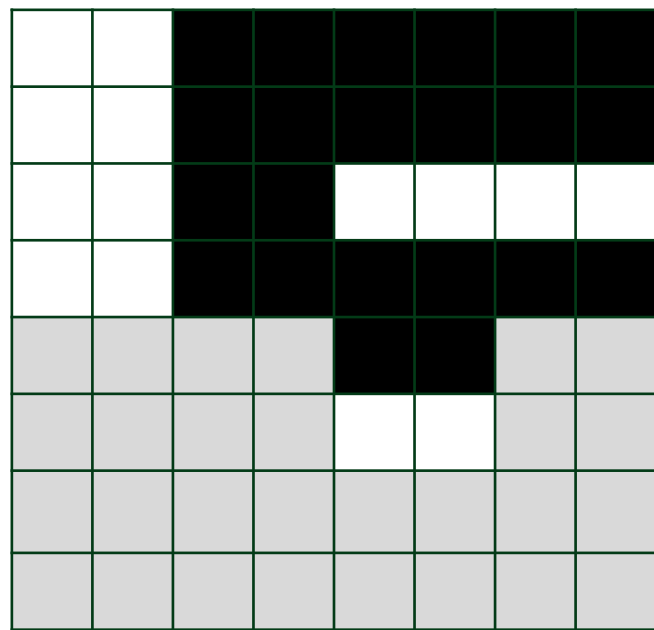
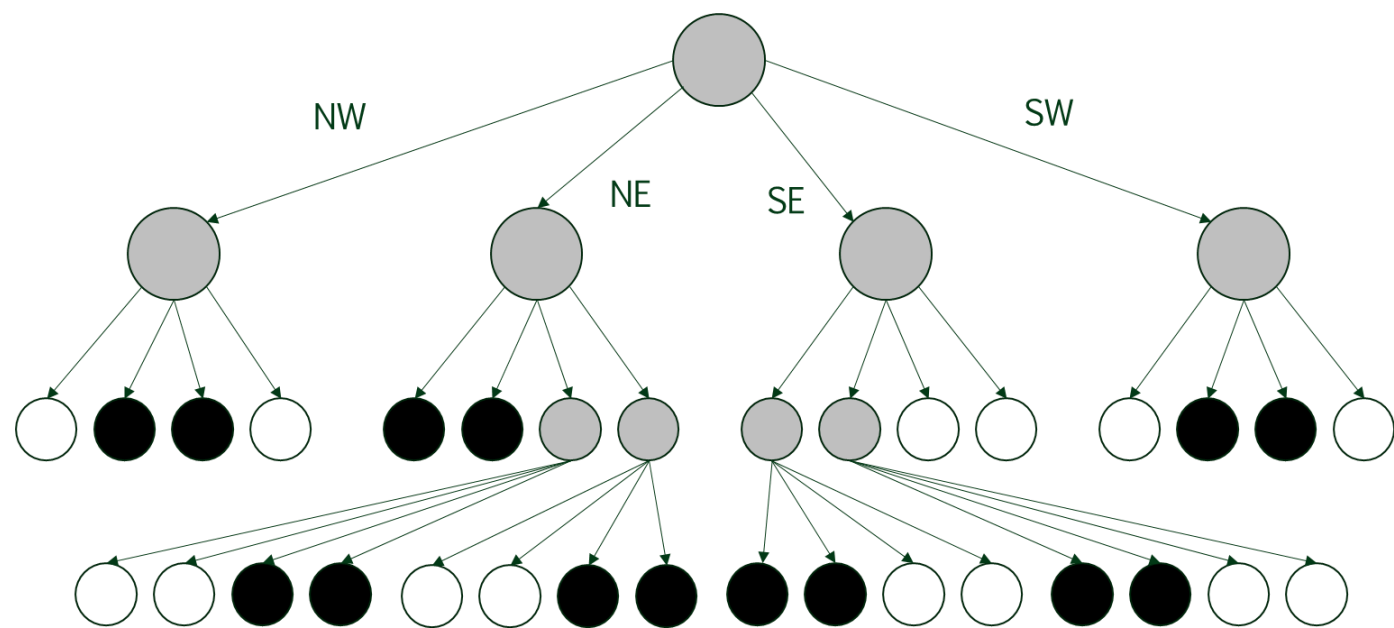


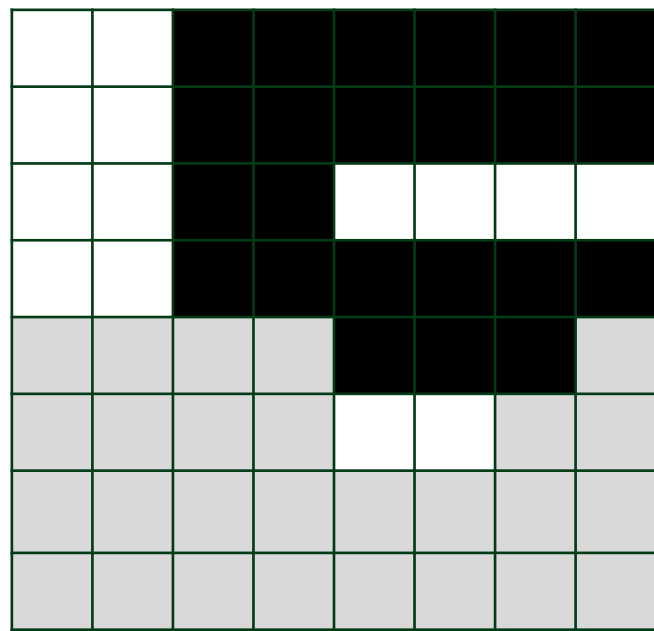
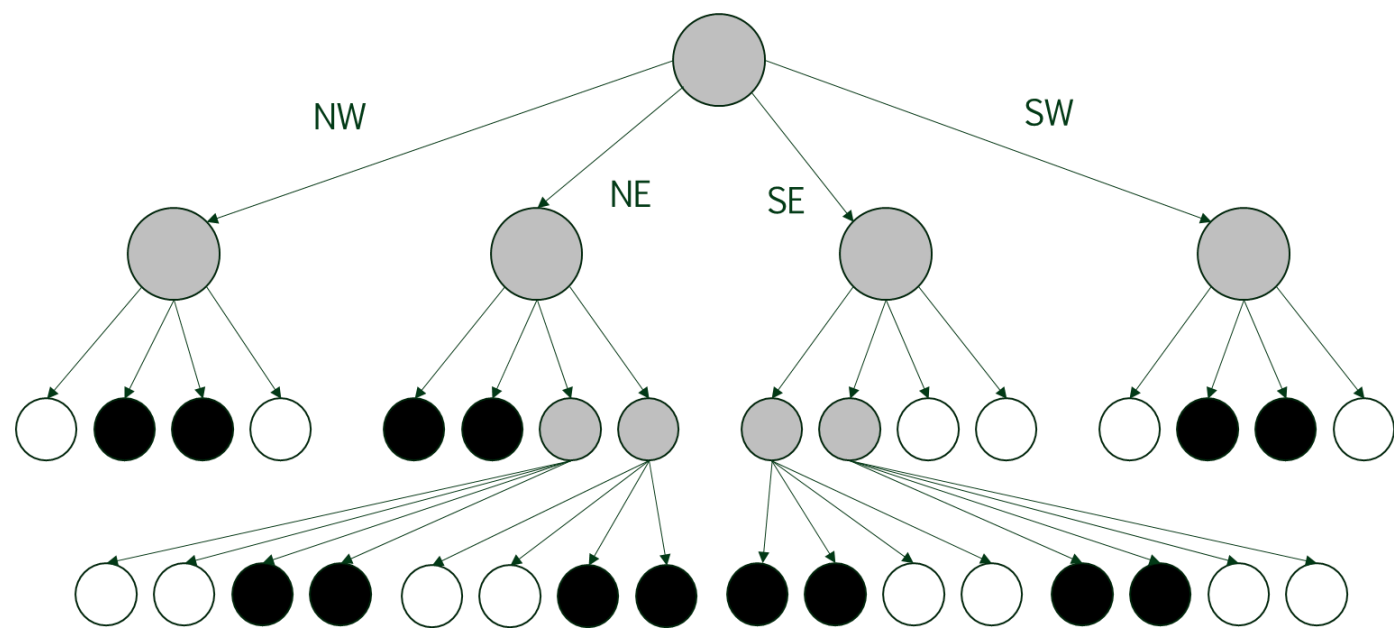


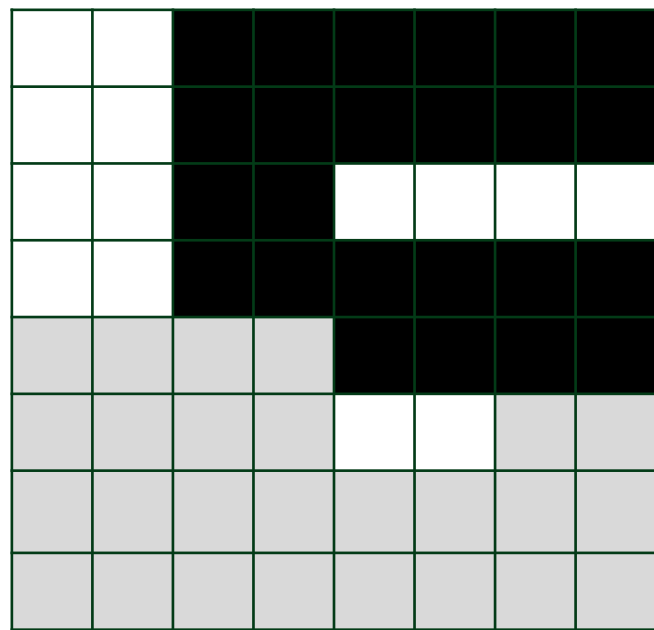
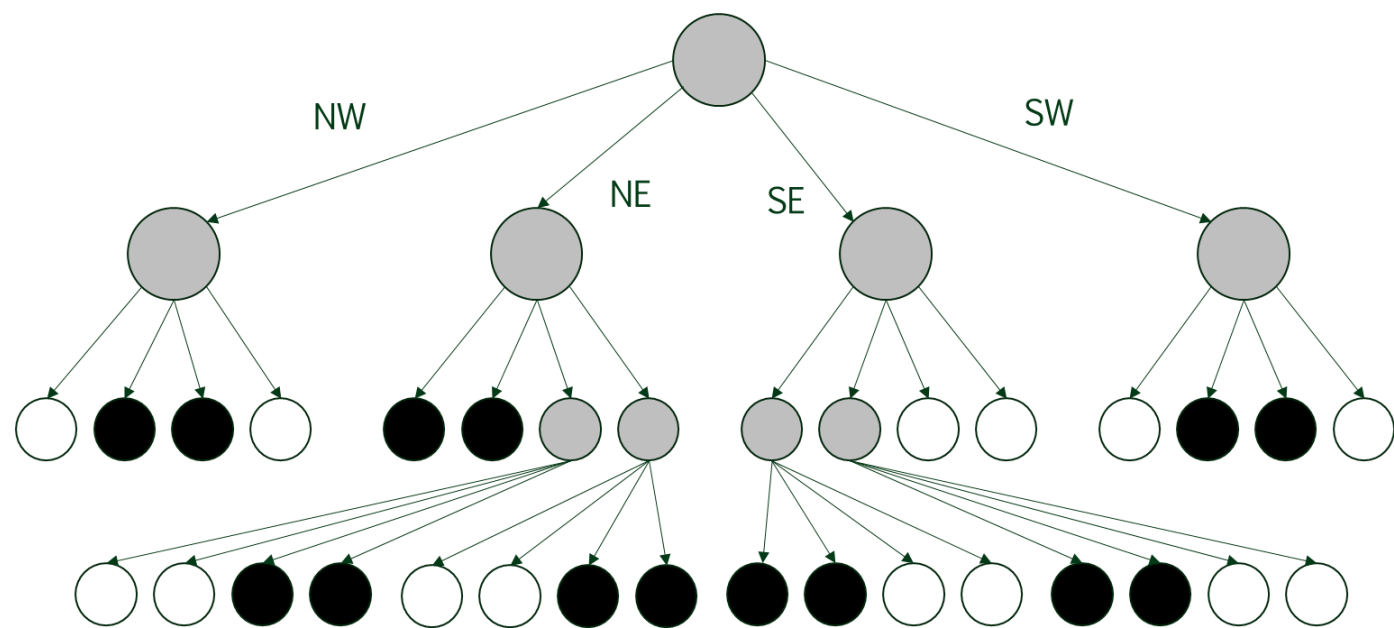


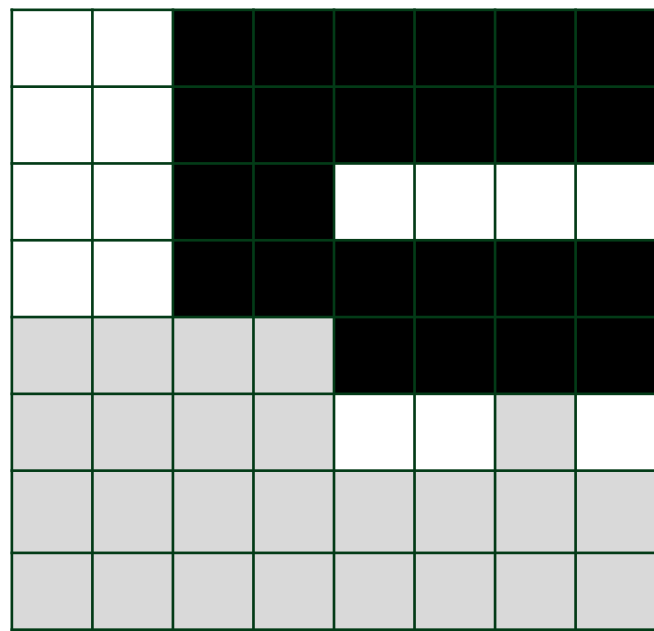
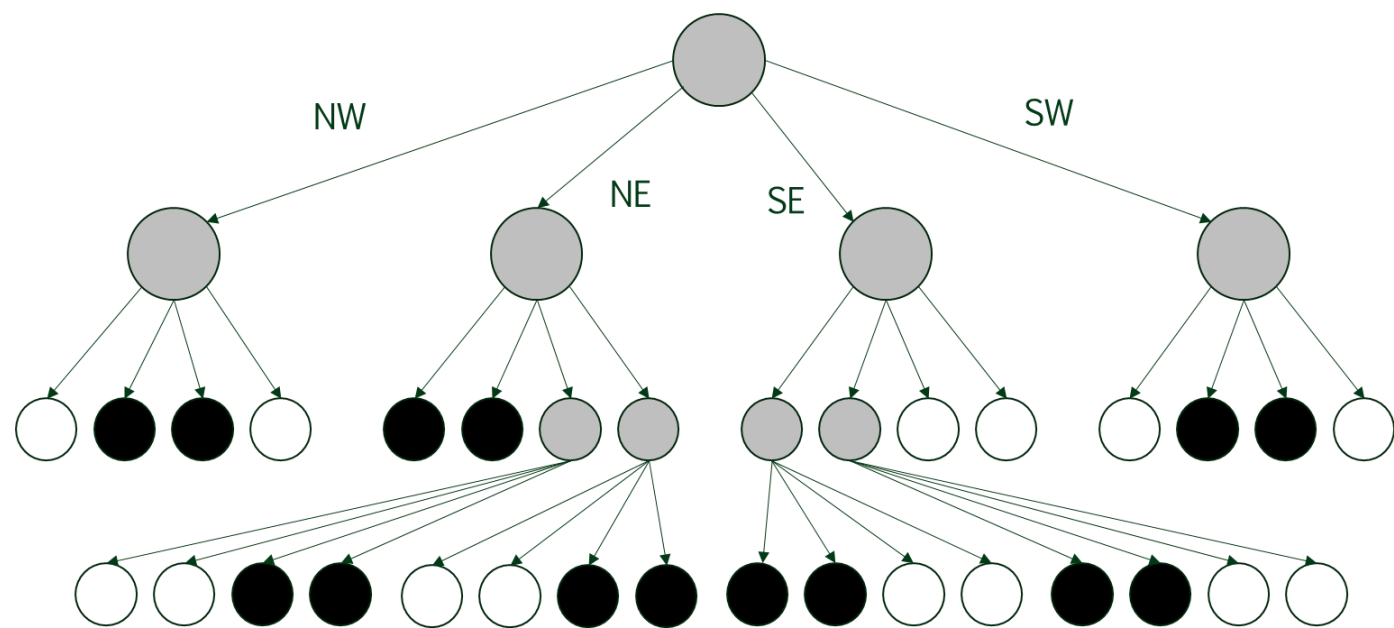


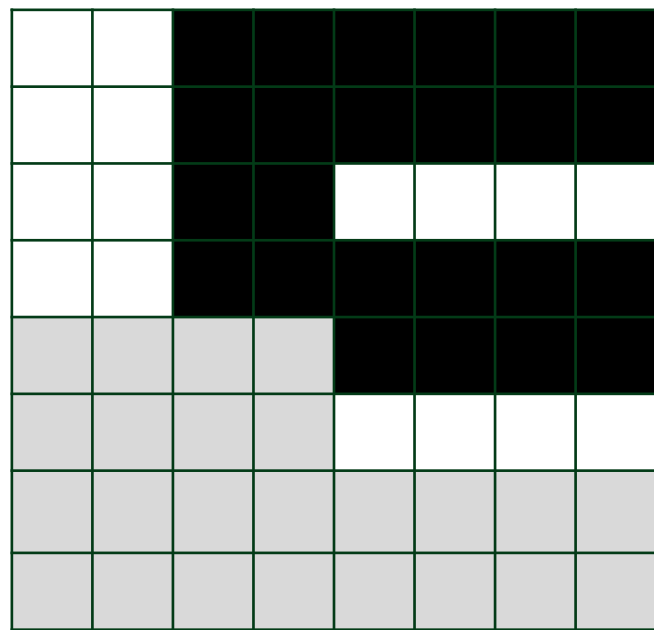
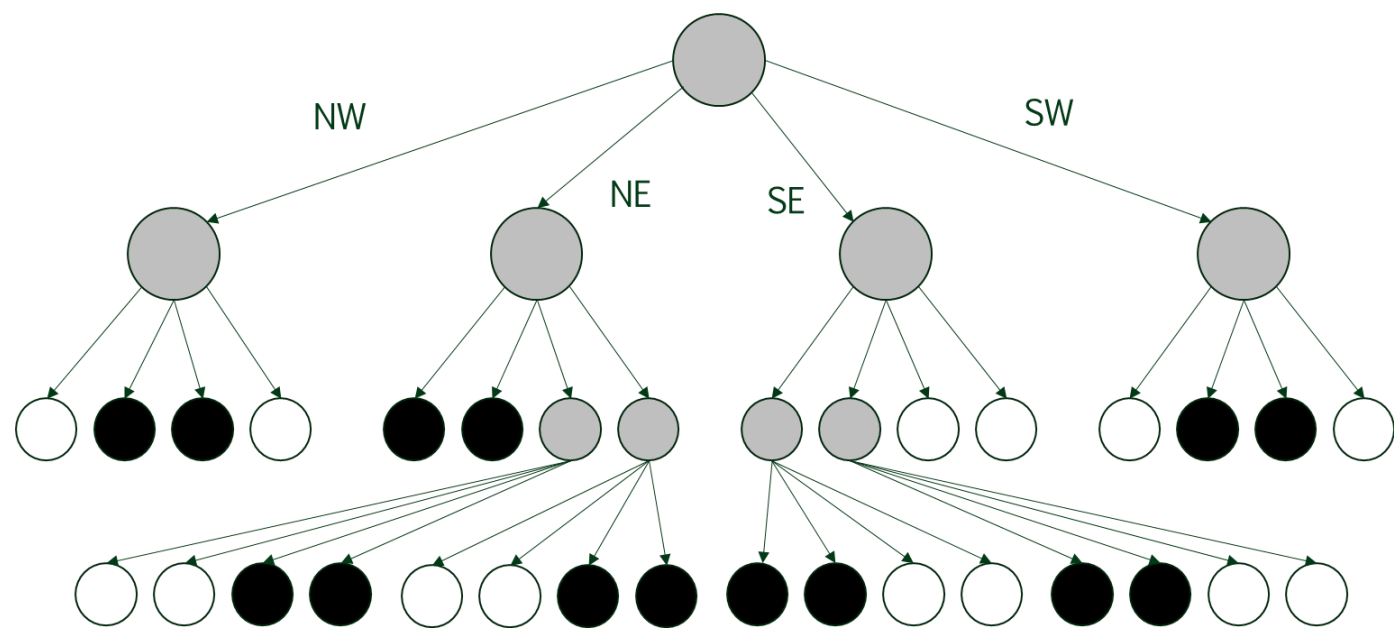


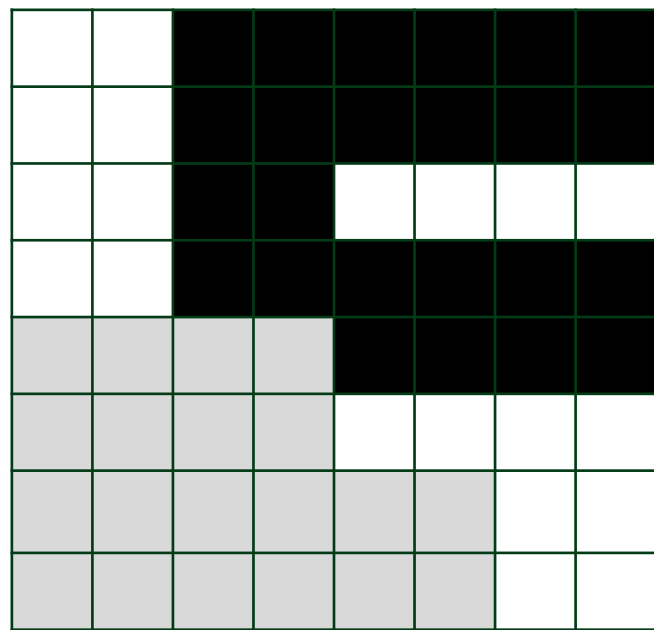
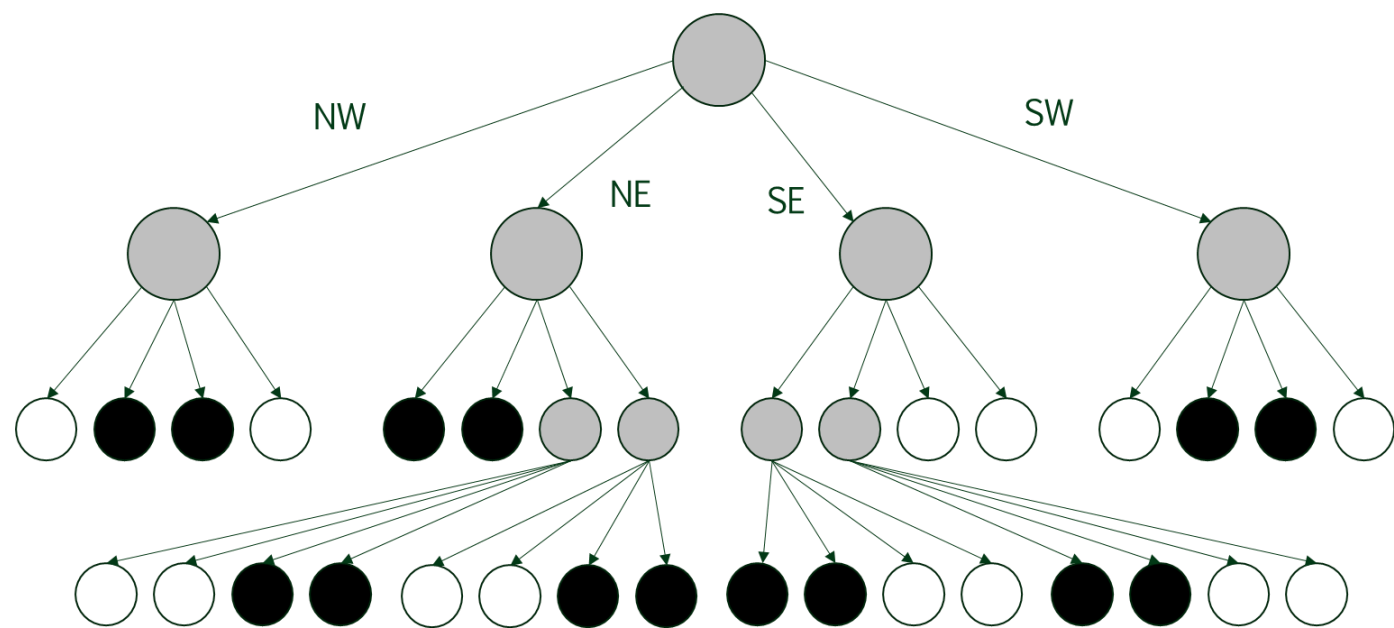


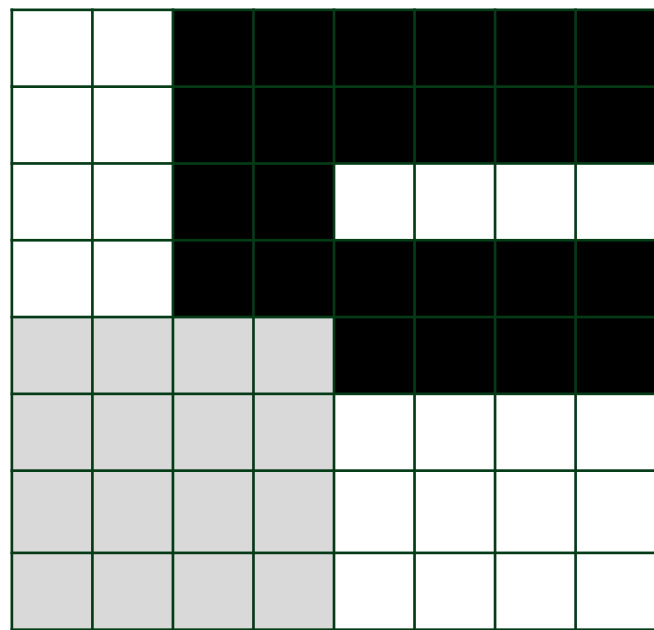
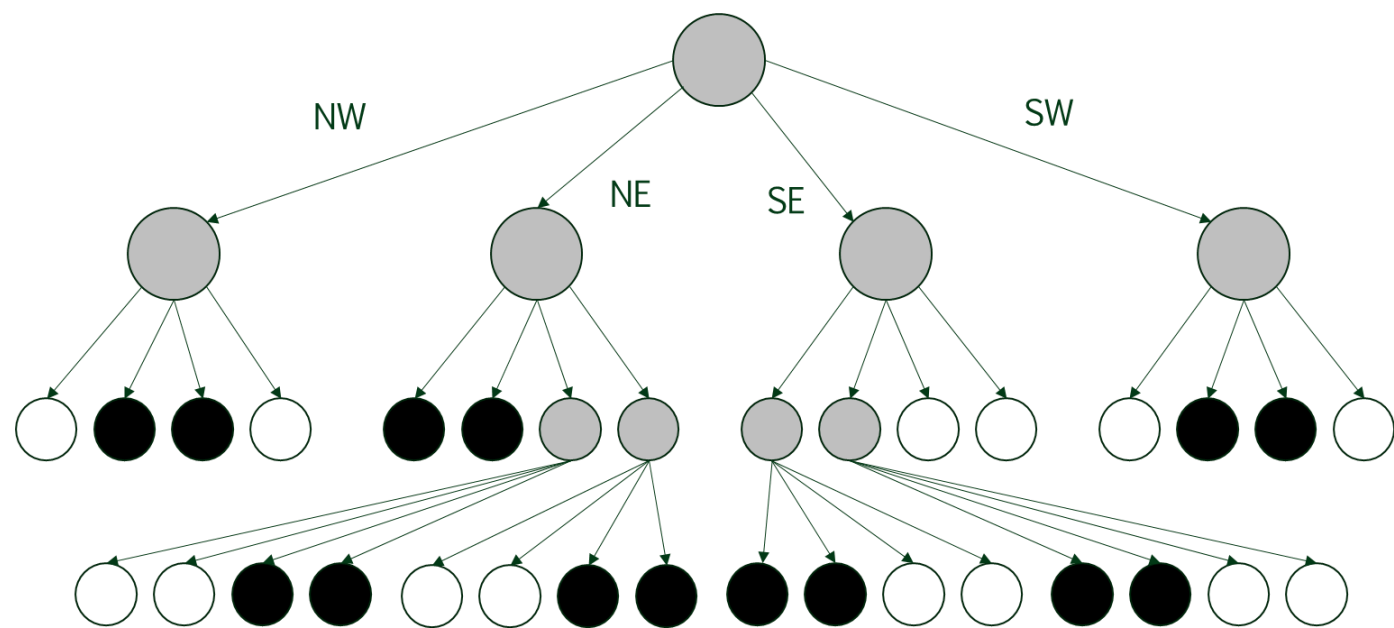


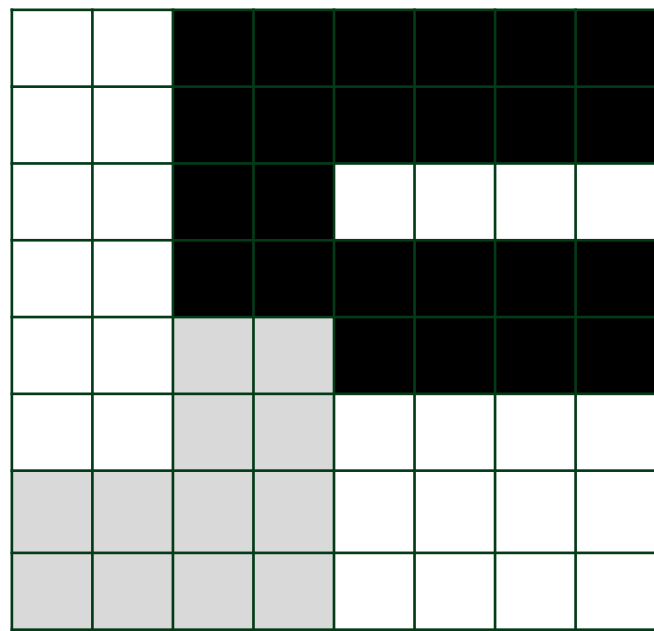
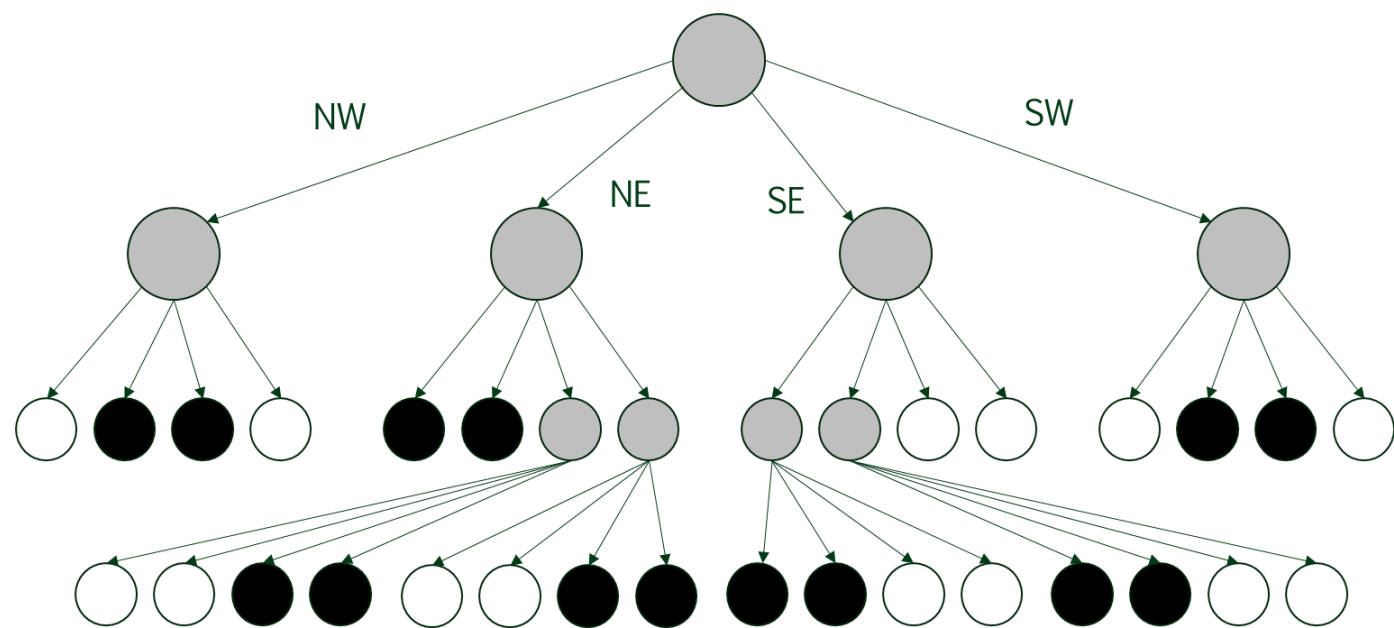


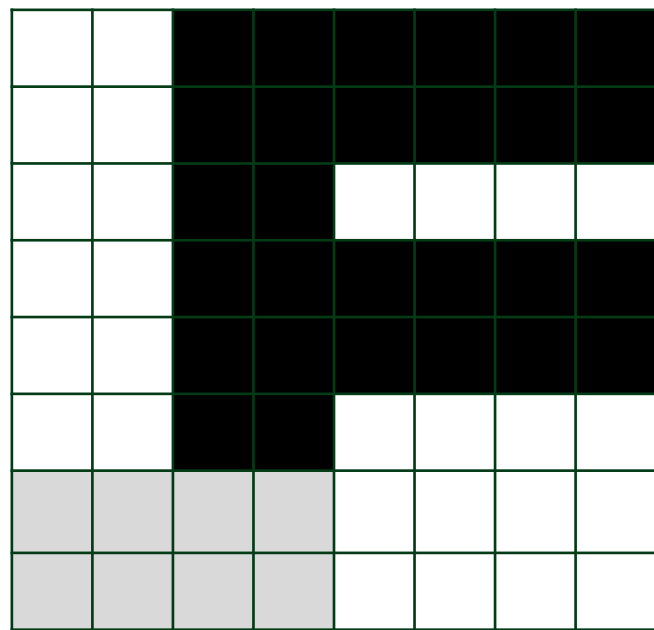
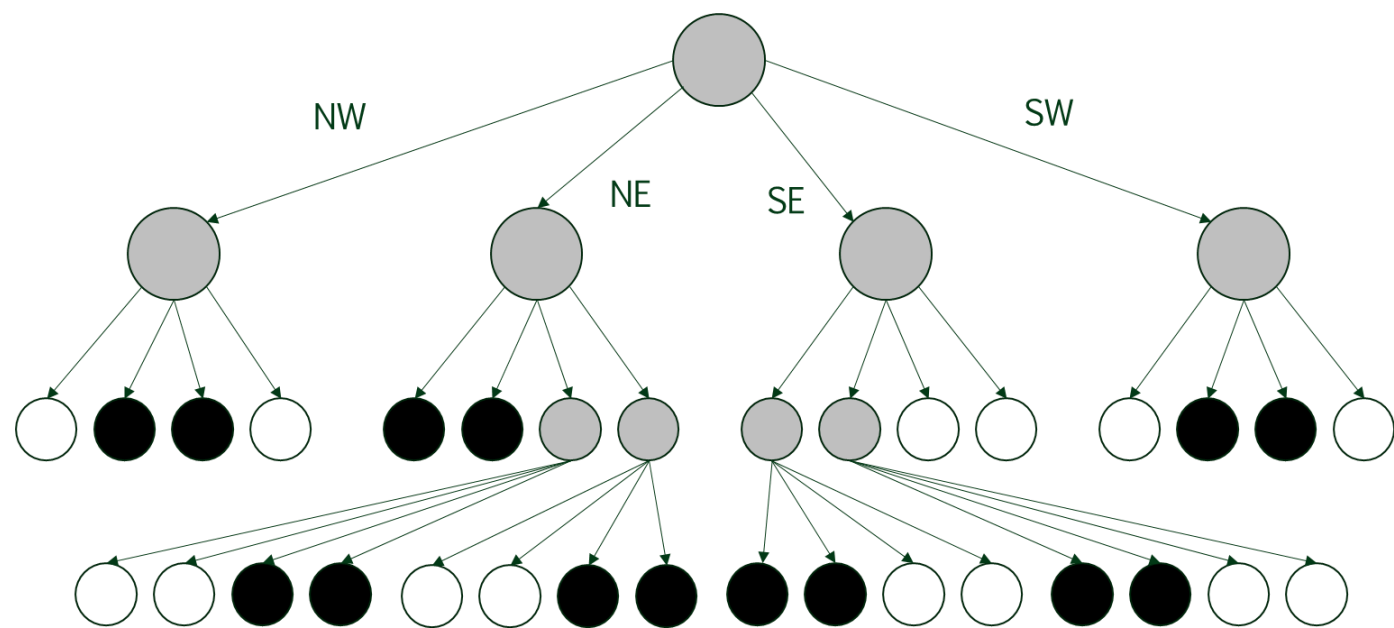


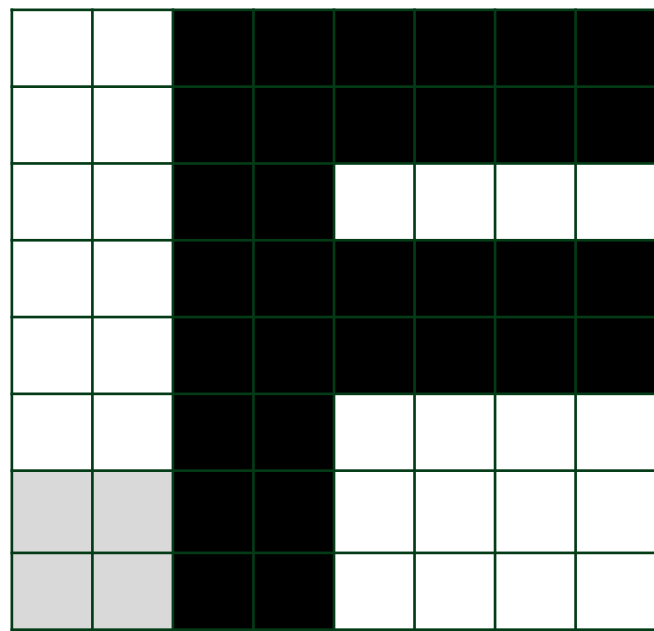
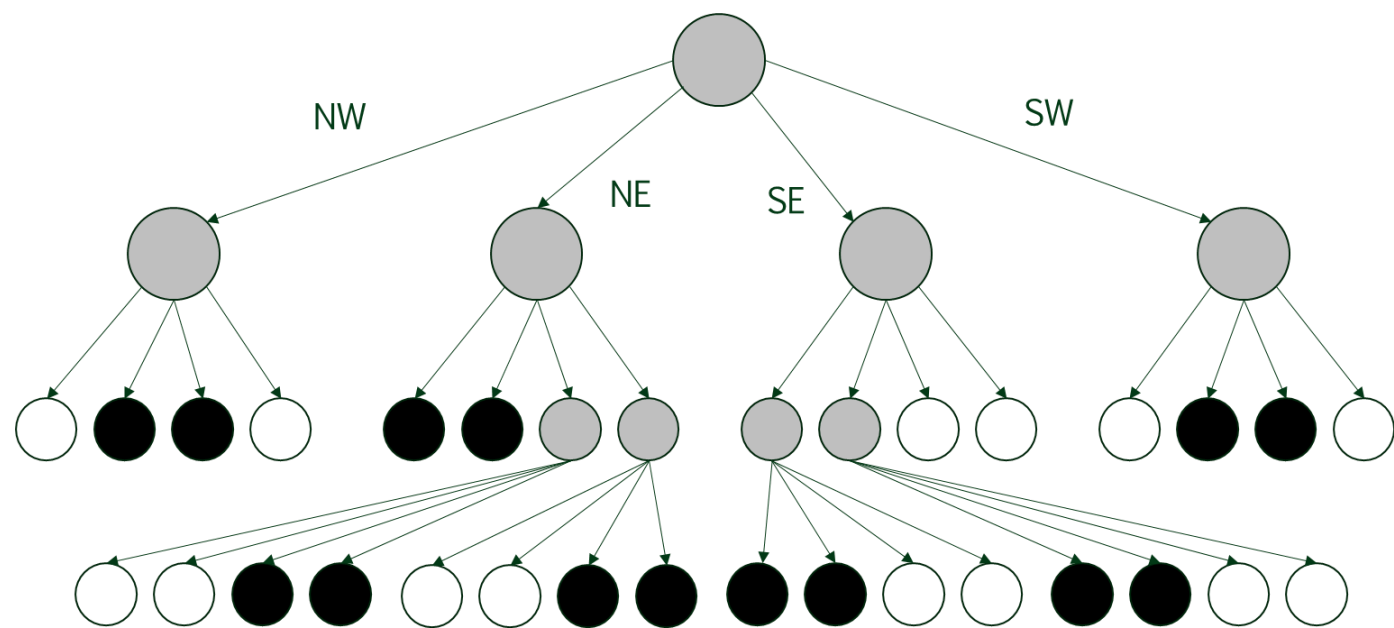


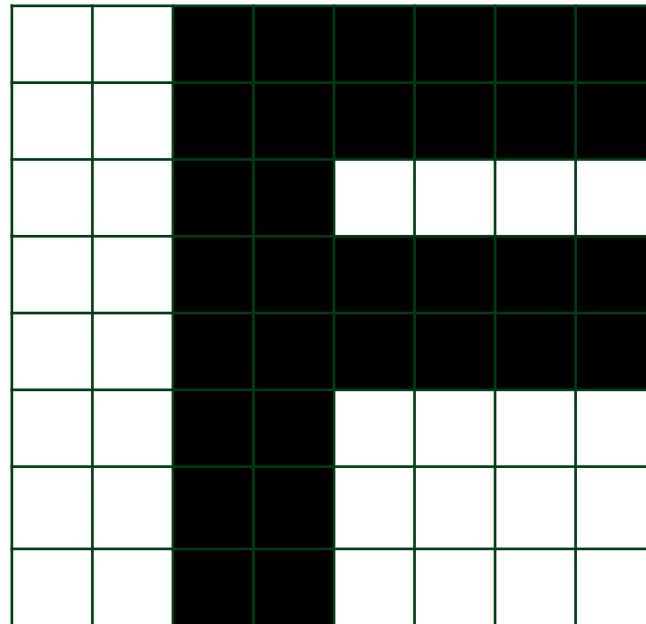
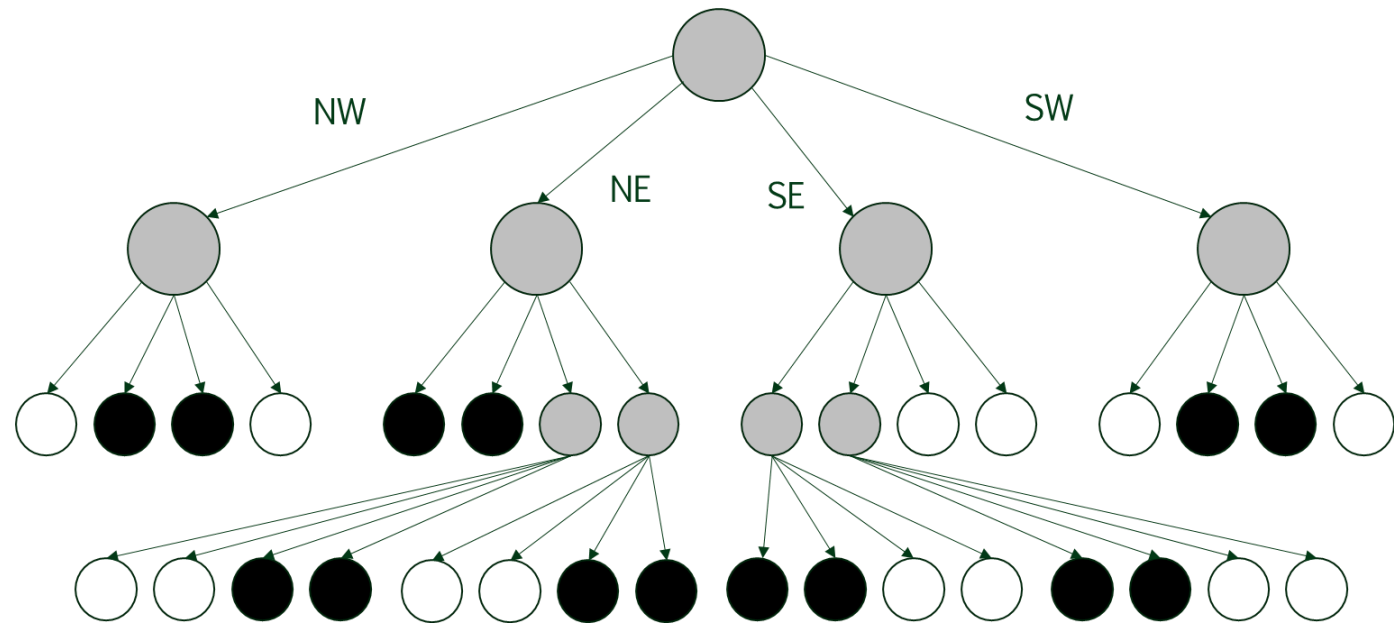












TADA !!



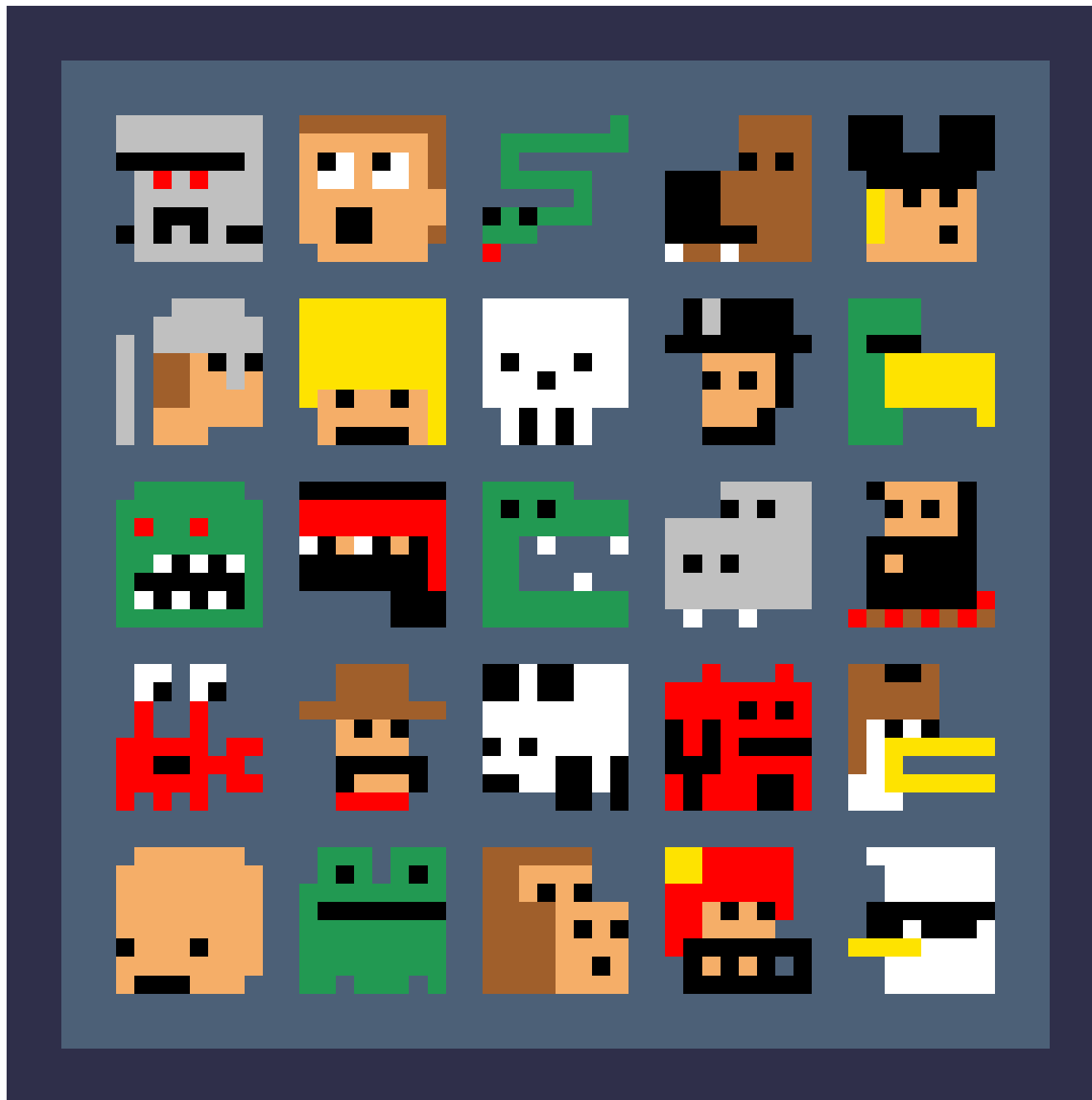
Time complexity

- › The time complexity of Print is $O(n^2)$ where n is the side length of the image



Exercises

- › Choose your favourite 8x8 pixel art and draw the corresponding quadtree. See next slide for a few examples.
- › Describe an algorithm that switches an entire quadrant to either BLACK or WHITE.
- › Describe two complete quadtrees whose union yields a single node.





What we call the beginning is often the end.
And to make an end is to make a beginning.
The end is where we start from.

T. S. Eliot