CARV, ICS-FORTH

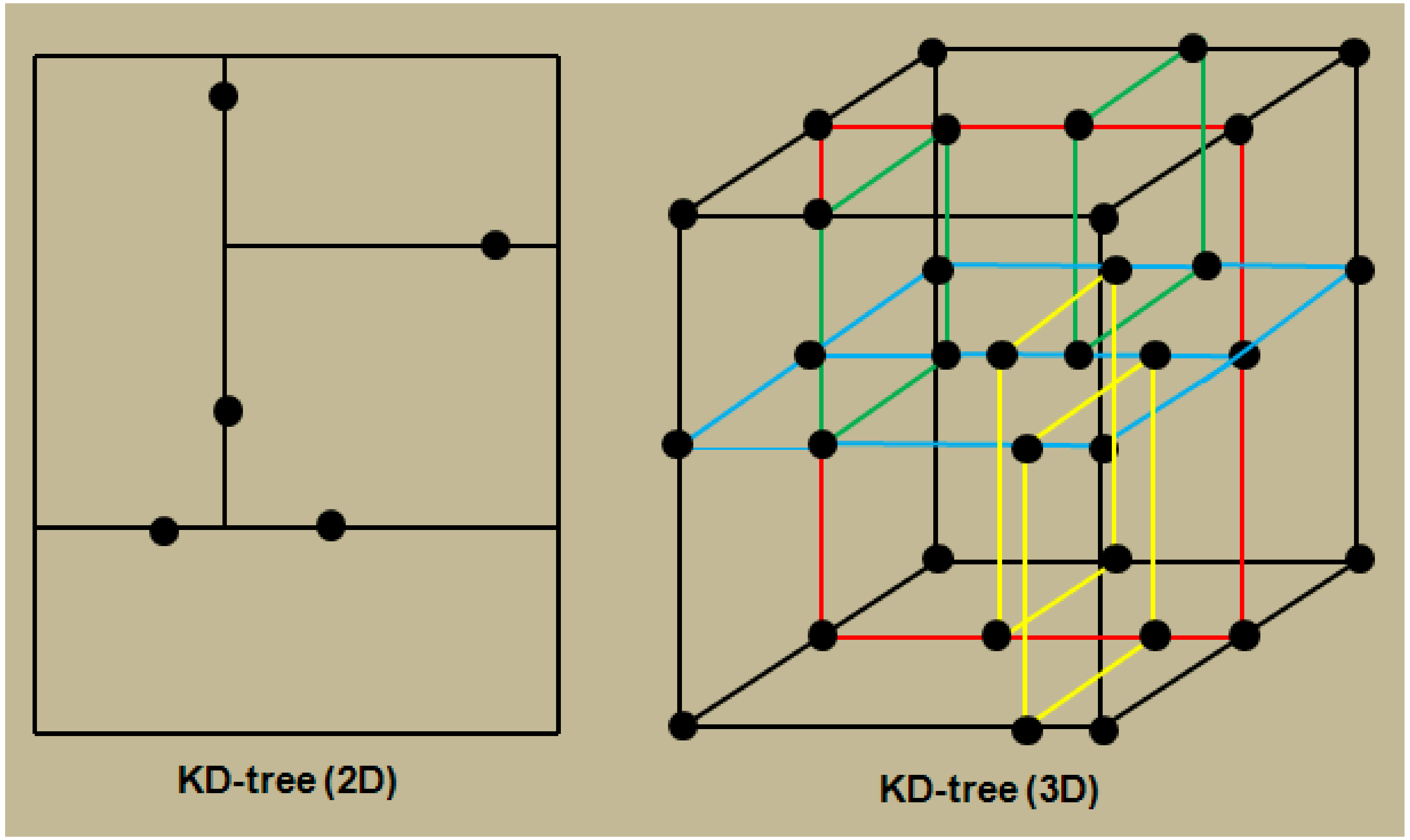
CONCURRENT QUADTREES

Presented By Kelly Kiouri Kyparisi

10/09/2024

ABOUT:

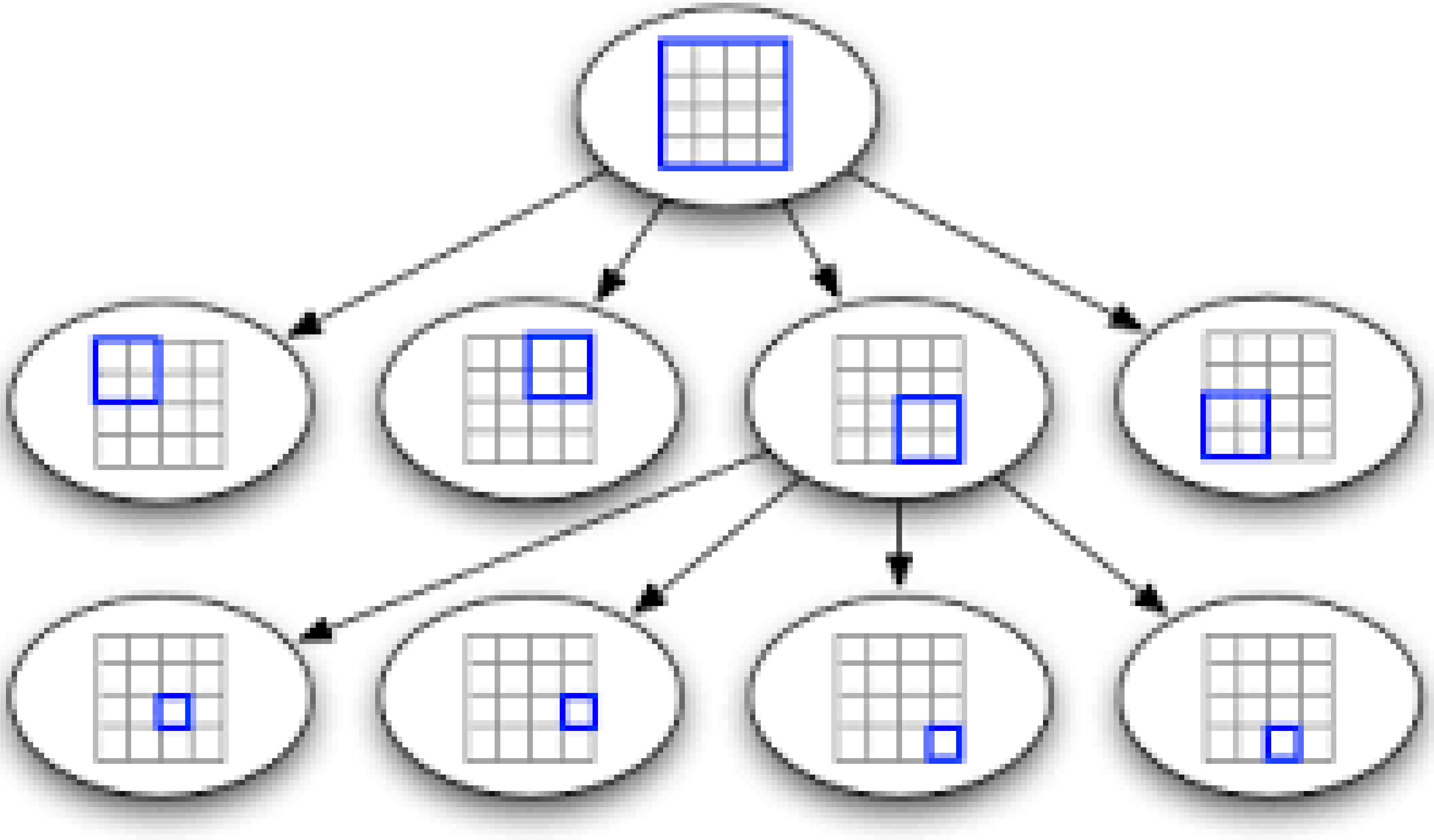
Reminder



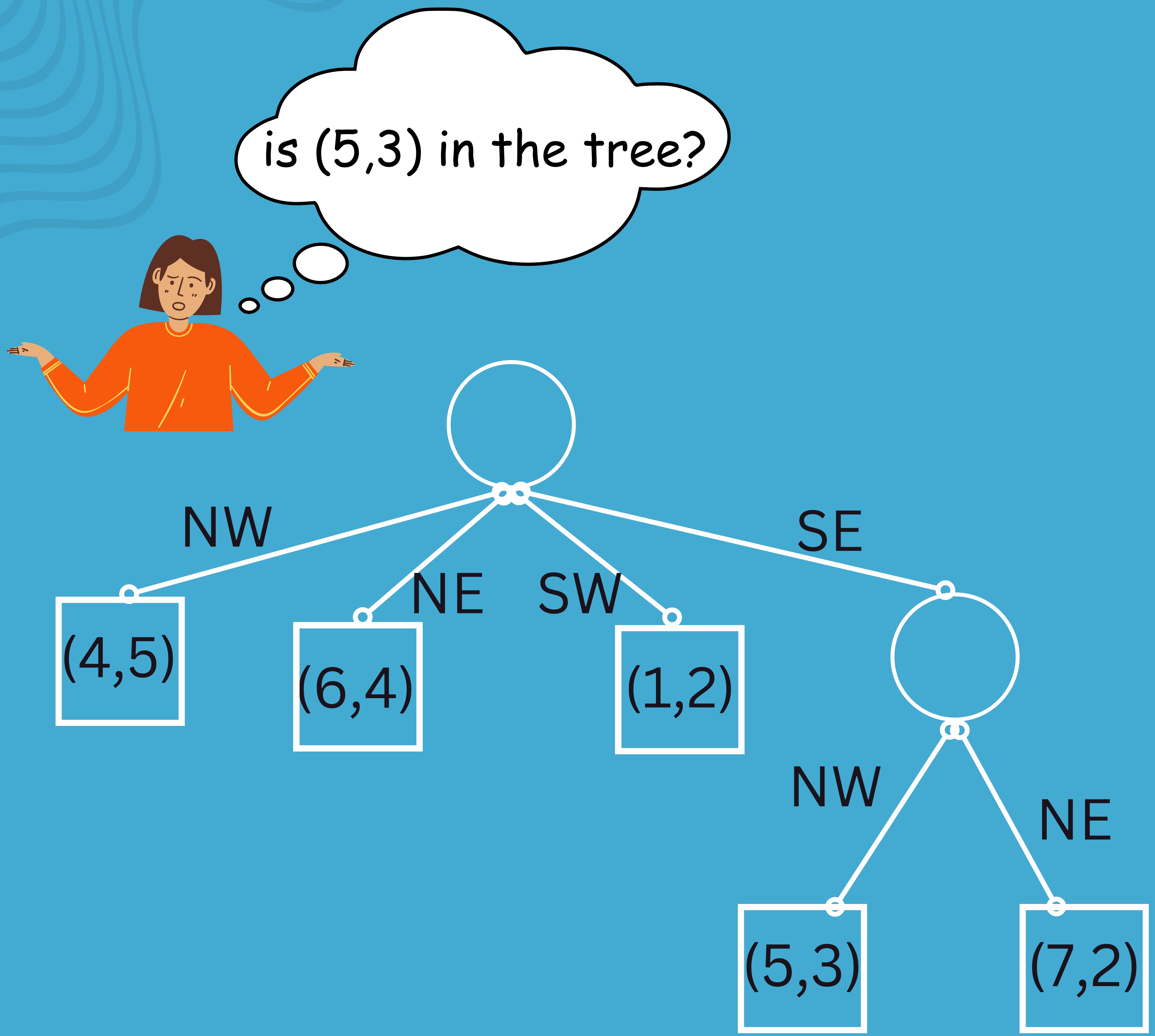
Quadtrees are part of a broader category of data structures known as **k-d trees** (k-dimensional trees).

They are trees used to efficiently store **data of points** on a two-dimensional space. In this tree, each node has at most four children.

STRUCTURE OF QUADTREES

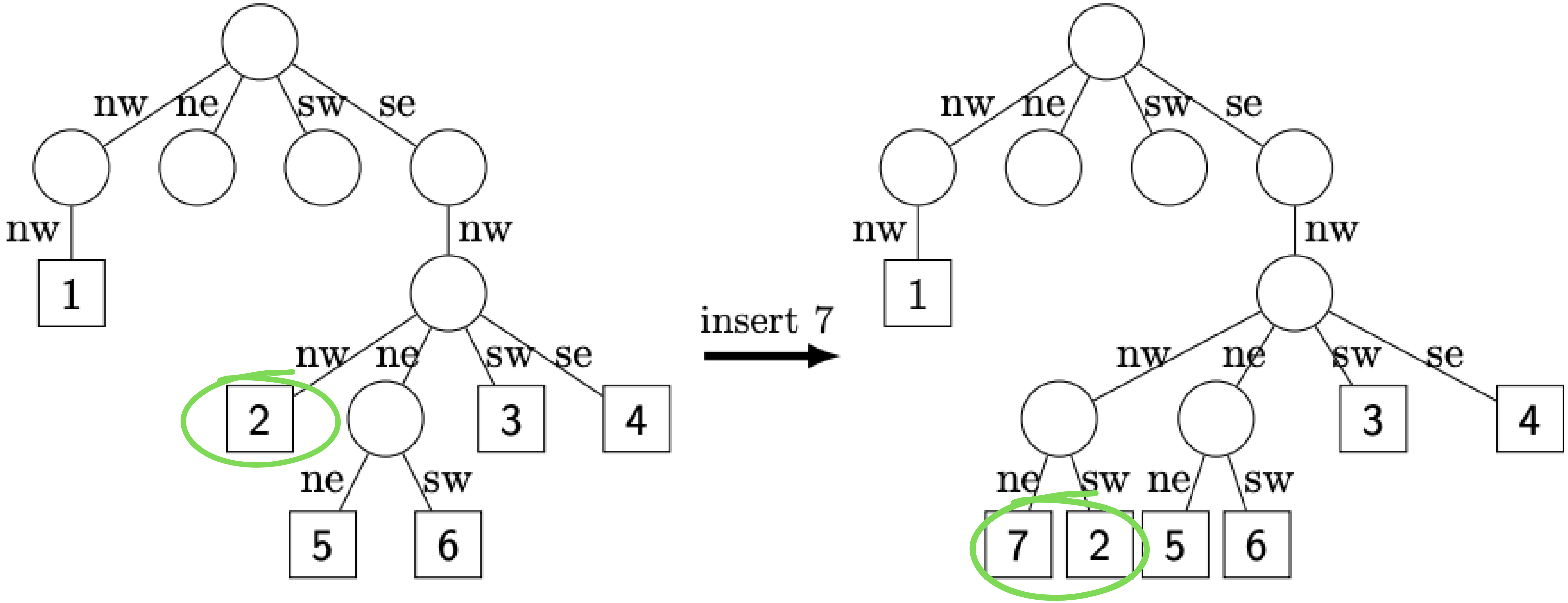


CONTAIN ALGORITHM




```
bool Contain (float kx , float ky) {
   if( root==NULL ) {return NULL }
   node l = root
   while( l is Internal ){  // leaves contain keys
       l = findQuadrant(l, kx, ky)
   }
   if( l.keyX== kx  &&  l.keyY==ky ) { return true }
   return false
}
```

INSERT

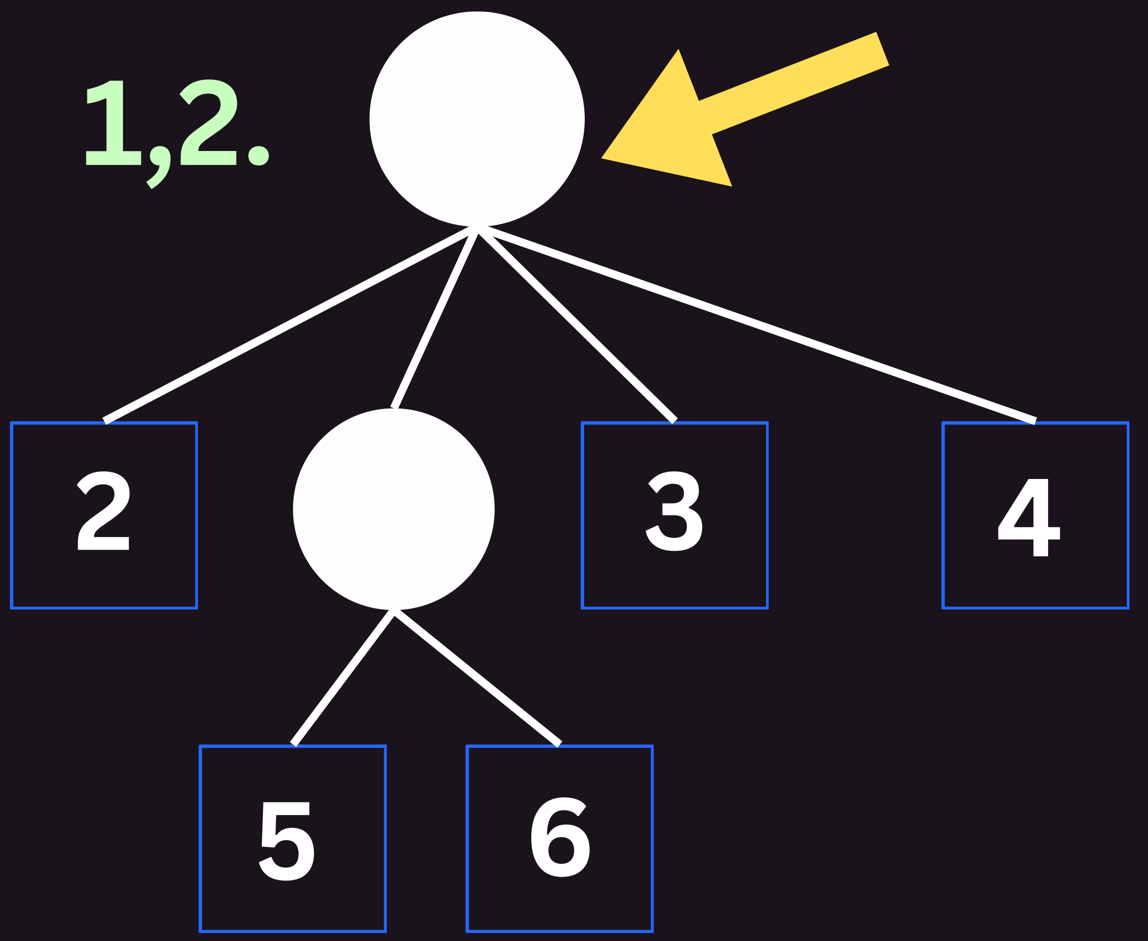


(a) Insert node 7 into the sample quadtree

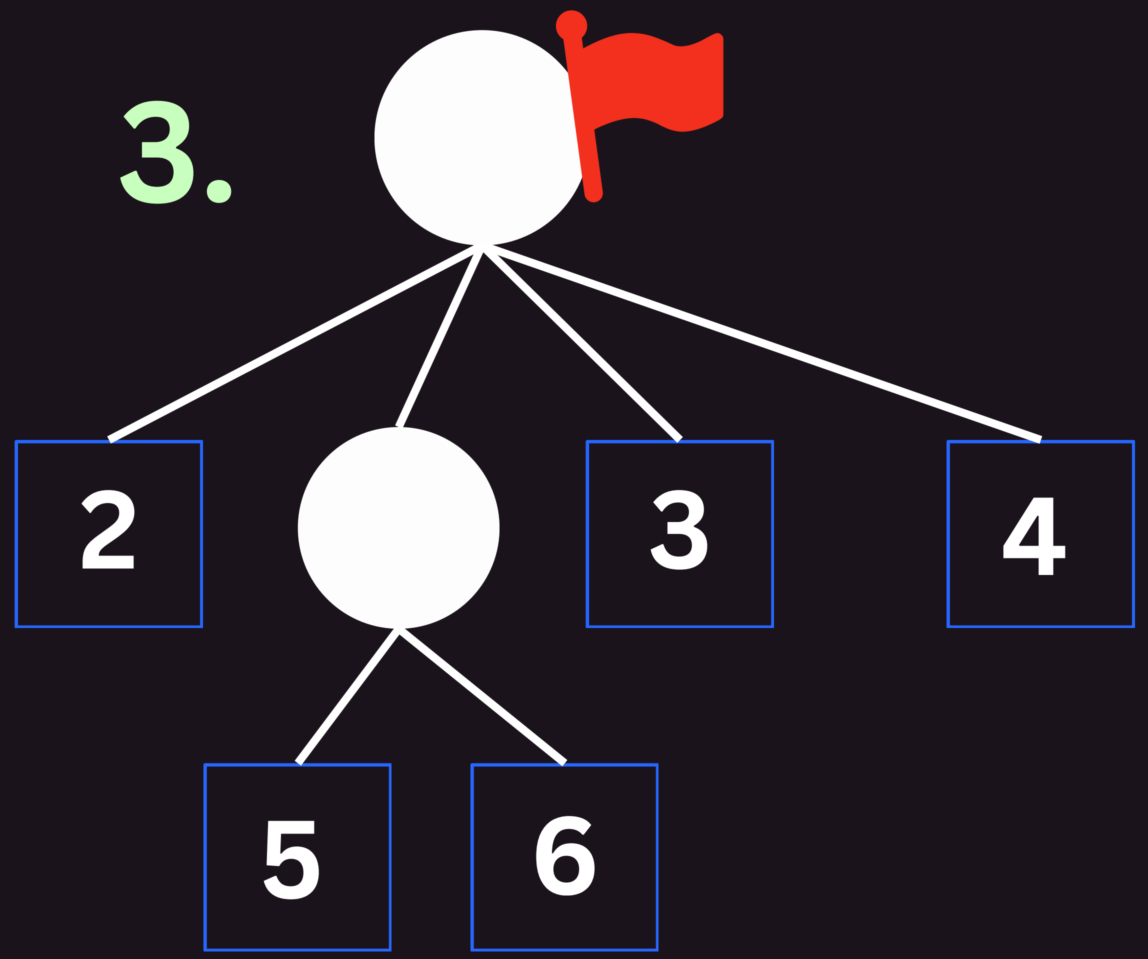
CONCURRENT INSERT

1. Is it **unique ?**
2. **Search** parent node
3. **Flag** the parent node to insert mode (CAS)
4. **Create subtree** with the new input
5. **Attach** subtree to the parent node (CAS)
6. **Unflag** the parent node (CAS)

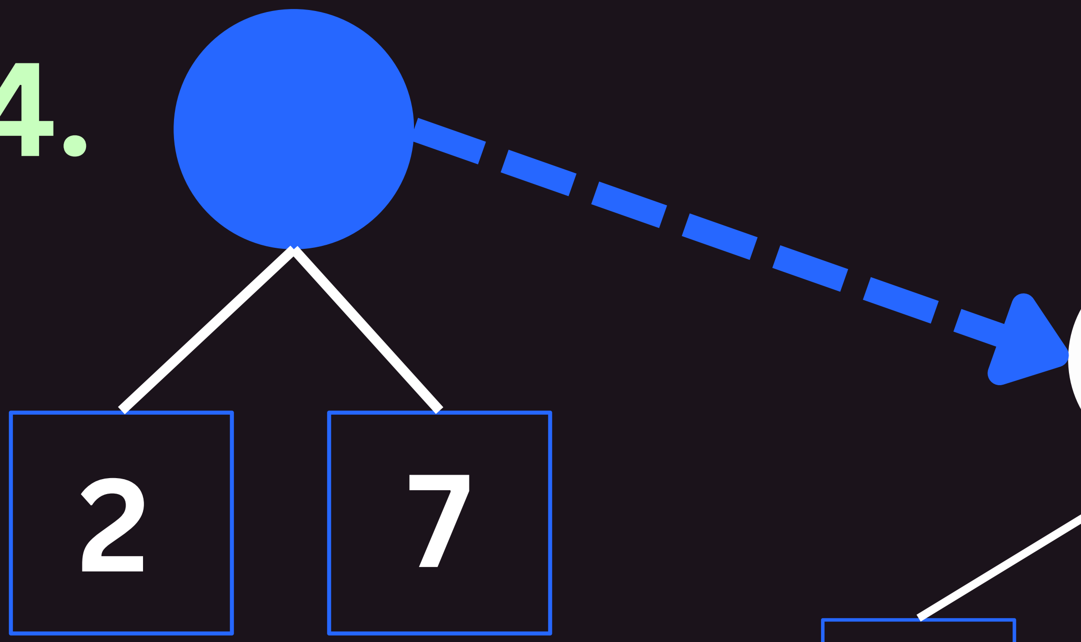
1,2.



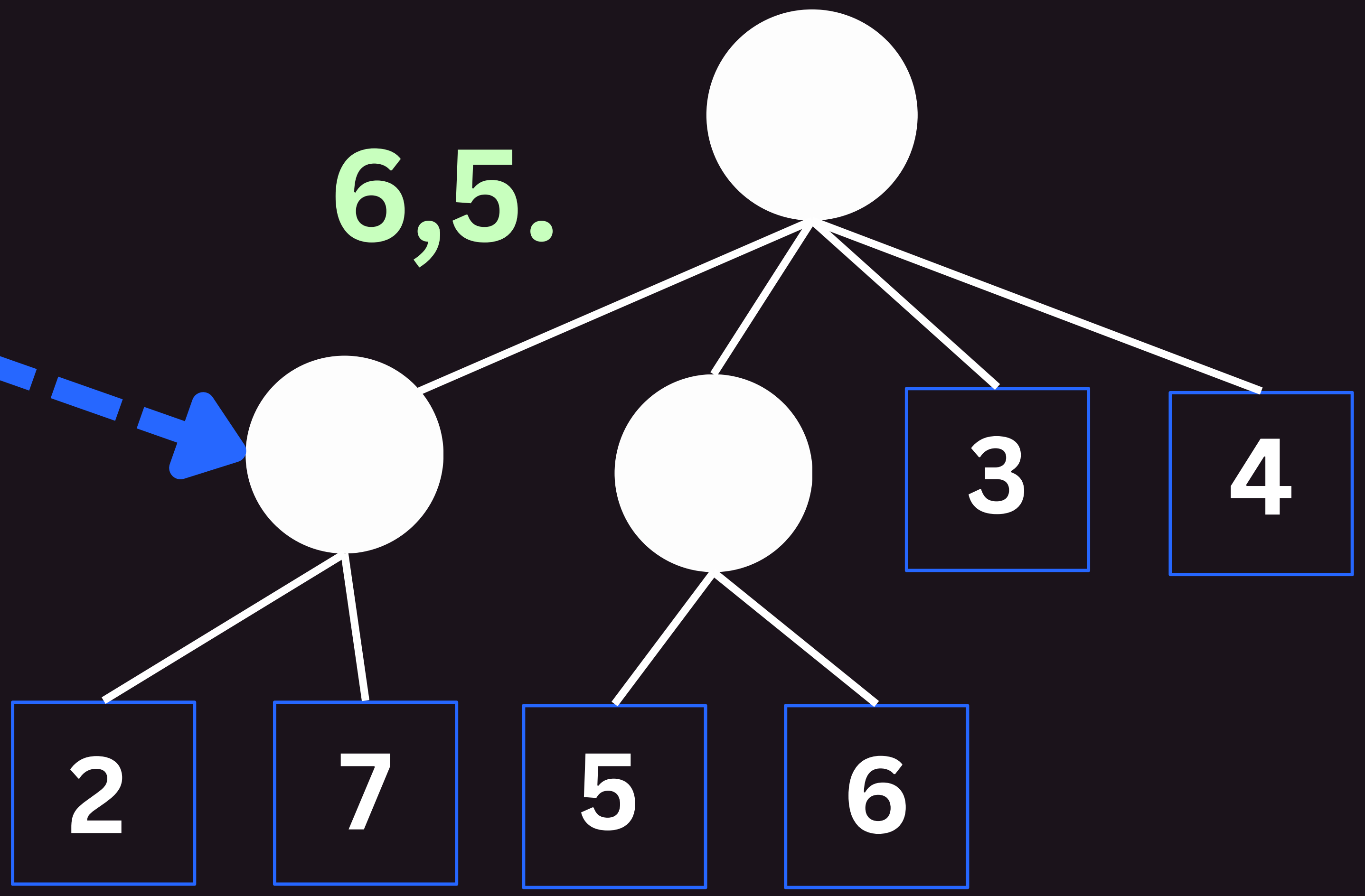
3.

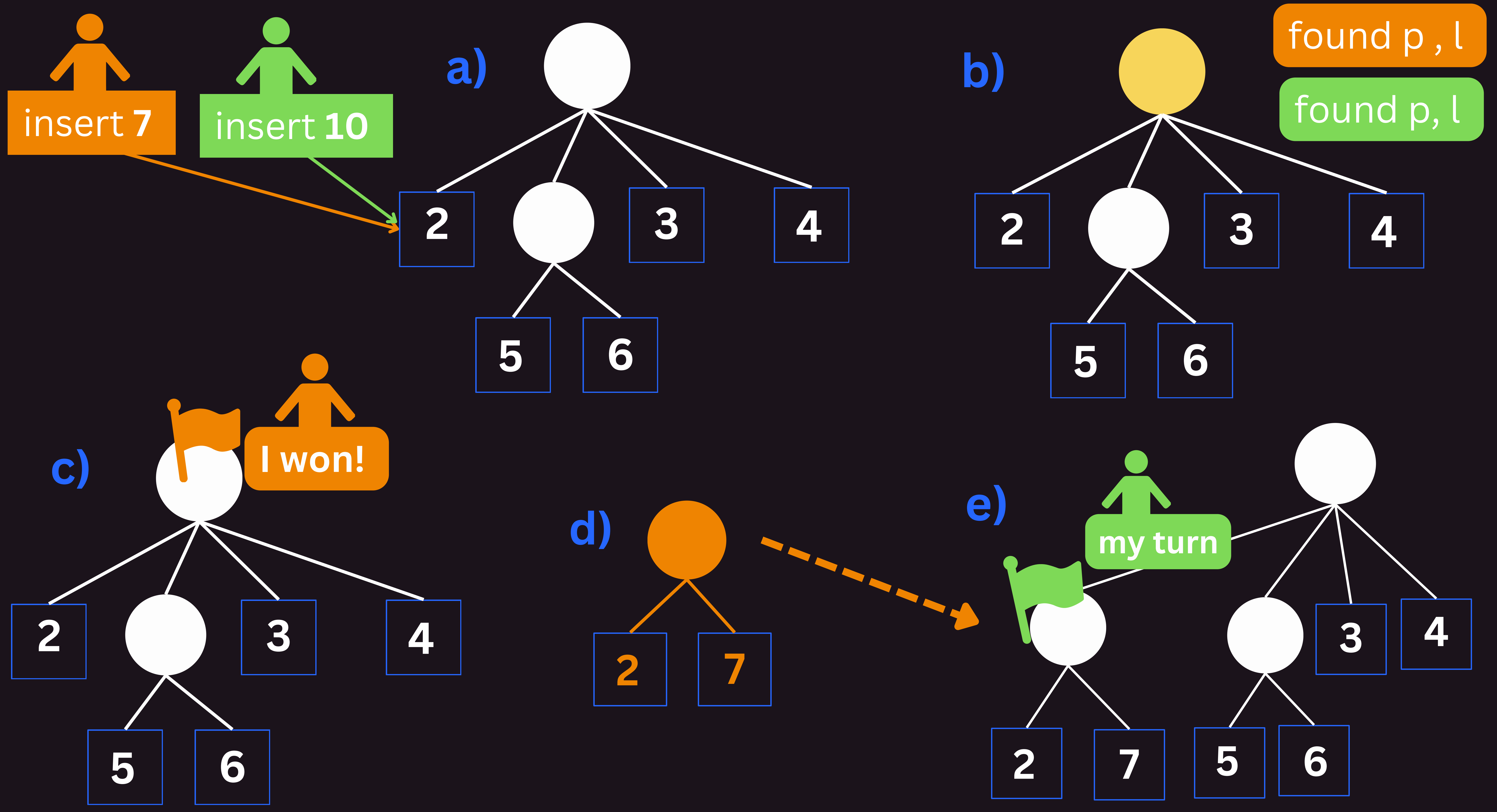


4.



6,5.




insert 7
insert 10
a)
2
3
4
5
6
b)
found p , l
found p, l
2
3
4
5
6
c)
I won!
2
3
4
5
6
d)
2
7
e)
my turn
2
7
5
6
3
4

CONCURRENT INSERT (1/2)

```
struct Internal {   ◃ subtype of Node
  float bounds xmin, ymin, xmax, ymax;
  struct Node *nw, *sw, *ne, *se;
  struct Info info ;
}
struct Leaf {  ◃ subtype of Node
  float keyX, keyY ;
}
```

```
struct Info {
  float keyX, keyY;
  enum state { Clean, Iflag } ;
  int seq;
}
```

CONCURRENT INSERT (2/2)

```
bool function Insert (float kx, float ky){
 if( Contain(kx, ky) ) { return false }
 //unique key
 while(True){   //continue until it's inserted
  Node p, l = root ;
  <p, l> Search (p, l, kx, ky)//returns
                    //parent & old child
  curr_seq=p.Info.seq
  if (p.info.state == Clean && p.info.seq ==
  curr_seq) {
    If( CAS( p.Info, {Clean, curr_seq}, {Iflag,
    curr_seq+1} ) ){   //iflag CAS
     p.Info.keyX= keyX; // the helper threads
     p.Info.keyY= keyY; // can find the data here
     //create subtree with a new leaf
     Node subtree = createSubtree( p, l, ,kx, ky)
     //attach subtree CAS
     if(p.nw==l)  {CAS( p.nw, l, subtree )}
     if(p.sw==l)   {CAS( p.sw, l, subtree )}
     if(p.ne==l)  {CAS( p.ne, l, subtree )}
     if(p.se==l)   {CAS( p.se, l, subtree )}
     //uniflag CAS
     CAS( p.Info, {Iflag, curr_seq+1},
     {Clean, curr_seq+2} )
     return true
   } else { Help (p, l) }//The Iflag CAS failed,
          //help the operation that caused failure
  }
 }
}
```

```
void Help( Node p, Node l ) {
    float kx = p.Info.keyX;  // Retrieve the key from p.info
    float ky = p.Info.keyY;
    curr_seq=p.Info.seq
    if (p.Info.state == Iflag && p.Info.seq == curr_seq) {
        // a subtree has already been attached,
        //so no need to create a new one
        if (p.nw != l && p.sw != l && p.ne != l && p.se != l) { return; }
        //create subtree with a new leaf
        Node subtree = createSubtree( p, l, ,kx, ky)
        //attach subtree CAS
        if(p.nw==l) {CAS( p.nw, l, subtree )}
        if(p.sw==l) {CAS( p.sw, l, subtree )}
        if(p.ne==l) {CAS( p.ne, l, subtree )}
        if(p.se==l) {CAS( p.se, l, subtree )}
        //uniflag CAS
        CAS( p.Info, {Iflag, curr_seq+1}, {Clean, curr_seq+2} )
    }
}
```

//takes the info state of parent and helps the other insert operation to finish

Limitations on Efficiency

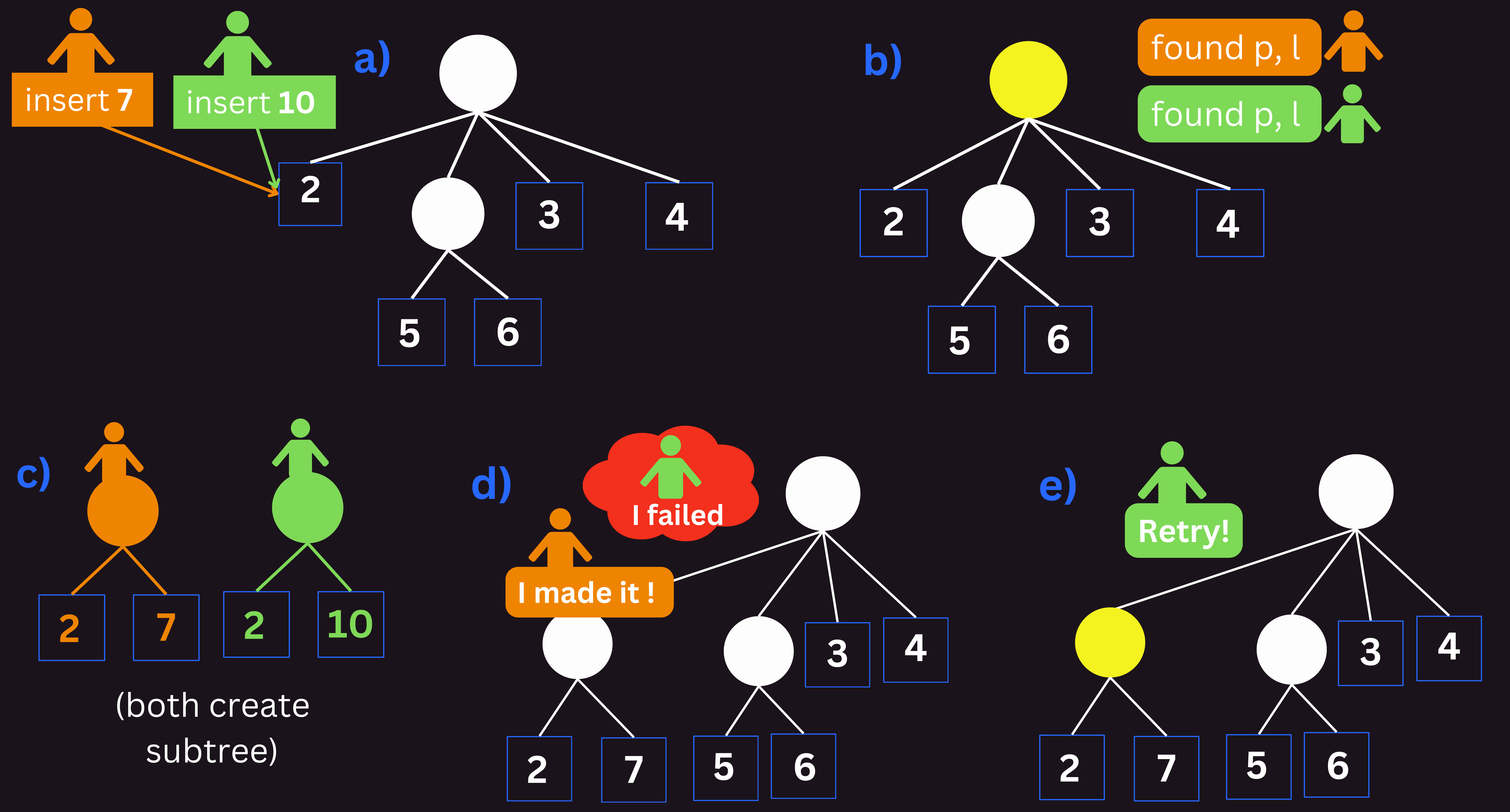
- It requires **multiple CAS operations** to set the Iflag, attach the subtree, and then reset the flag.
- **Keeping track** of flags and sequence numbers .

introduces **overhead** in terms of memory and additional checks.

SOLUTION?

We can use One CAS Insertion

- **Removing the Iflag and the Help function.** Each thread is independent and retries if necessary.
- **Removing the need for multiple CAS** operations to set and unset flags. There's only a single CAS to attach the subtree.


a)
insert 7
insert 10
2
3
4
5
6
b)
found p, l
found p, l
2
3
4
5
6
c)
2
7
2
10
(both create subtree)
d)
I failed
I made it !
2
7
5
6
3
4
e)
Retry!
2
7
5
6
3
4

CONCURRENT INSERT USING ONE CAS

```
bool InsertOneCas(float kx, float ky) {
  if (Contain(kx, ky)) {return false}// Point exists
 // Create a subtree with kx, ky
 Node subtree = createSubtree(NULL,NULL, kx, ky)
  while (true) {
      Node p, l = root;
      <p, l> Search (p, l, kx, ky)
      // Update the subtree with the correct parent
      //and leaf node after each search (fail CAS)
      updateSubtree(subtree, p, l);
  // Attempt to atomically replace
  //the old child lwith the new subtree
  if (p.nw == l && CAS(p.nw, l, subtree)) {return true}

  if (p.sw == l && CAS(p.sw, l, subtree)) {return true}

  if (p.ne == l && CAS(p.ne, l, subtree)) {return true}

  if (p.se == l && CAS(p.se, l, subtree)) {return true}
  }
}
```

Only one of the **CAS** operations will execute depending on which quadrant the old child l belongs to.

Why It Can Work With Only **One CAS & No Help Function ?**

- **CAS guarantees atomicity**: The CAS operation ensures that only one thread will successfully insert its subtree at a time, preventing race conditions.
- **Retry mechanism handles conflicts:** If one thread fails its CAS (because another thread inserted first), it will retry. No data is lost, and no locks are needed.
- **No need for flags:** Without Iflag or other state flags, there's no need for Help, since all threads simply retry until they succeed.

Limitations on Efficiency

- If many threads are inserting at the same time, there could be a lot of retries, which might hurt performance.
- Every time a retry happens, a new subtree is created. If subtree creation is expensive, it can lead to increased costs in terms of performance and resource utilization.

OVERALL:

For Low to Moderate Concurrency ->The Single CAS with Retries

For High Concurrency -> The Iflag with Help (it is optimized for reducing retry overhead)

HELPFUL FUNCTIONS(1/3)

```
<p,l> Search (Node p, Node l, float kx , float ky) {
  struct info
  while (l is internal){
    p=l  //record the parent node
    l = findQuadrant(l, kx, ky)
  }
    return <p,l>
}
```

/ / finds the parent node & the old child node of the new insertion position.

HELPFUL FUNCTIONS(3/3)

```
Node findQuadrant (Node l, float kx, float ky){
float midX = q.xmin + q.xmax /2 ;
float midY = q.ymin + q.ymax /2 ;
if (kx < midX ){      //top side
   if (ky <midY){    //left side
      q = q.nw;
   } else {  q= q.sw; }
} else {
   if (ky <midY){
      q = q.ne;
   } else { q = q.se; }
}
return q;
}
```

/ /find the right quadrant based on the point (kx,ky)

References :

Non-blocking Binary Search Trees by
Faith Ellen, Panagiota Fatourou, Eric Ruppert, Franck van Breugel

Quadboost: A Scalable Concurrent Quadtree by
Keren Zhou, Guangming Tan, Wei Zhou

Thank You!