LECTURE 21

ASSOCIATIVE ARRAYS

Associative arrays

- also known as maps or dictionaries
- are collections of (`key`, `value`) tuples, where
  - `key` could be any string of bits (integer, character string, other data)
  - `value` is any data
- that support
  - insertion (add a (`key`, `value`) tuple)
  - deletion (remove a (`key`, `value`) tuple)
  - lookup given a `key`,
    - find the corresponding `value`,
    - or determine that no such `key` has been added

Naive implementation

Just some list of (`key`, `value`) tuples:

```
(k0, v0)
(k1, v1)
(k2, v2)
(k3, v3)
(k4, v4)
(k5, v5)
...
```

	Insertion	Deletion (after lookup)	Lookup
Linked list	$O(1)$	$O(1)$	$O(n)$
Dynamic array	$O(1)$	$O(n)$	$O(n)$

ASSOCIATIVE ARRAYS:

IMPLEMENTATIONS USING A TOTAL ORDER ON KEYS

Total order on keys

- We assume that we can compare keys (i.e. evaluate `key_i ≤ key_j` for any `i, j`)
- Always possible in practice (reinterpret key bits as a big integer)
- Sometimes, a specialized ≤ operator makes sense (e.g. constant-size keys)
- key space may be infinite (arbitrary-sized keys)

Sorted dynamic array of (key, `value`) tuples

- Assume `key0 ≤ key1 ≤ ... ≤ keyn`.
- Use bisection for key lookup

	Insertion	**Deletion (after lookup)**	**Lookup**
Linked list	$O(1)$	$O(1)$	$O(n)$
Dynamic array	$O(1)$	$O(n)$	$O(n)$
Sorted dynamic array	$O(n)$	$O(n)$	$O(\log(n))$

Binary search tree

- Invariant: For any given node `i`,
  - `key_j ≤ key_i` for every descendant node `j` in the left subtree of `i`
  - `key_j > key_i` for every descendant node `j` in the right subtree of `i`

- Main concern: depending on insertion order, we may get

```
                    8
        4                        10
    1       5                9        12
```

```
    1
           4
               5
                    8
                         9
                             10
                                 12
```

good or bad

Self-balancing binary search tree

- AVL trees
- Red-black trees
- B-trees, splay trees, treaps, …

	Insertion	Deletion (after lookup)	Lookup
Linked list	$O(1)$	$O(1)$	$O(n)$
Dynamic array	$O(1)$	$O(n)$	$O(n)$
Sorted dynamic array	$O(n)$	$O(n)$	$O(\log(n))$
Binary search tree	$O(n)$	$O(n)$	$O((n))$
AVL tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
Red-black tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

Self-balancing binary search tree

- Cache behavior: ok, not great
  (similiar to other binary tree structures, e.g. heap)

```
                 8
      4                    10
   1     5              9       12
```

ASSOCIATIVE ARRAYS:

IMPLEMENTATIONS USING KEYS BITS

–

TRIES

Trie

- A trie (or prefix tree) is a tree of static arrays of size 2^T
- Key bits are divided into chunks of T bits: “letters”
- Each T-bits letter gives an index for one node’s static arrays
- Letters form a path in the tree (from root to leaf)

$T = 4$

```
T = 4
Insert (0x9f2, V1)   ->  letters 2, f, 9
```

```
T = 4
Insert (0x9f2, V1)   ->  letters 2, f, 9
```

0 1 2 3 4 5 6 7 8 9 a b c d e f

T = 4
Insert (0x9f2, V1)   -> letters 2, f, 9

```
                              0 1 2 3 4 5 6 7 8 9 a b c d e f
                                 /
                                /
0 1 2 3 4 5 6 7 8 9 a b c d e f
```

```
T = 4
Insert (0x9f2, V1)   ->  letters 2, f, 9



                                    0 1 2 3 4 5 6 7 8 9 a b c d e f
                                       /
                                      /
               0 1 2 3 4 5 6 7 8 9 a b c d e f
                                      \
                                       \
                                        \
                                         \
                                  0 1 2 3 4 5 6 7 8 9 a b c d e f
```

```
T = 4
Insert (0x9f2, V1)   ->  letters 2, f, 9




                                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                /
                                               /
               0 1 2 3 4 5 6 7 8 9 a b c d e f
                                              \
                                               \
                                                \
                                                 \
                                          0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                            |
                                                            V1
```

```
T = 4
Insert (0x9f2, V1)   ->  letters 2, f, 9
Insert (0xc8d, V2)   ->  letters d, 8, c



                                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                /
                                               /
               0 1 2 3 4 5 6 7 8 9 a b c d e f
                                              \
                                               \
                                                \
                                                 \
                                          0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                            |
                                                            V1
```

```
T = 4
Insert (0x9f2, V1)   ->  letters 2, f, 9
Insert (0xc8d, V2)   ->  letters d, 8, c



                                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                /                       \
                                               /                         \
               0 1 2 3 4 5 6 7 8 9 a b c d e f                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                              \
                                               \
                                                \
                                                 \
                                          0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                            |
                                                            V1
```

```
T = 4
Insert (0x9f2, V1)   ->  letters 2, f, 9
Insert (0xc8d, V2)   ->  letters d, 8, c



                                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                /                       \
                                               /                         \
               0 1 2 3 4 5 6 7 8 9 a b c d e f                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                              \                                             \
                                               \                                             \
                                                \                                             \
                                                 \                                             \
                                          0 1 2 3 4 5 6 7 8 9 a b c d e f                 0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                            |
                                                            V1
```

```
T = 4
Insert (0x9f2, V1)   ->  letters 2, f, 9
Insert (0xc8d, V2)   ->  letters d, 8, c


                                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                /                       \
                                               /                         \
               0 1 2 3 4 5 6 7 8 9 a b c d e f                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                              \                                             \
                                               \                                             \
                                                \                                             \
                                                 \                                             \
                                          0 1 2 3 4 5 6 7 8 9 a b c d e f                 0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                            |                                                     |
                                                            V1                                                    V2
```

```
T = 4
Insert (0x9f2, V1)   ->  letters 2, f, 9
Insert (0xc8d, V2)   ->  letters d, 8, c
Insert (0x532, V3)   ->  letters 2, 3, 5


                                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                /                       \
                                               /                         \
               0 1 2 3 4 5 6 7 8 9 a b c d e f                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                              \                                             \
                                               \                                             \
                                                \                                             \
                                                 \                                             \
                                          0 1 2 3 4 5 6 7 8 9 a b c d e f                 0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                            |                                                     |
                                                            V1                                                    V2
```

```
T = 4
Insert (0x9f2, V1)   ->  letters 2, f, 9
Insert (0xc8d, V2)   ->  letters d, 8, c
Insert (0x532, V3)   ->  letters 2, 3, 5


                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                /                         \
                               /                           \
        0 1 2 3 4 5 6 7 8 9 a b c d e f                     0 1 2 3 4 5 6 7 8 9 a b c d e f
                                      \                                       \
                                       \                                       \
                                        \                                       \
                                         \                                       \
                       0 1 2 3 4 5 6 7 8 9 a b c d e f            0 1 2 3 4 5 6 7 8 9 a b c d e f
                                         |                                                |
                                         V1                                               V2
```

```
T = 4
Insert (0x9f2, V1)   ->  letters 2, f, 9
Insert (0xc8d, V2)   ->  letters d, 8, c
Insert (0x532, V3)   ->  letters 2, 3, 5


                                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                /                       \
                                               /                         \
               0 1 2 3 4 5 6 7 8 9 a b c d e f                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                    /                         \                                             \
                   /                           \                                             \
                  /                             \                                             \
                 /                               \                                             \
0 1 2 3 4 5 6 7 8 9 a b c d e f           0 1 2 3 4 5 6 7 8 9 a b c d e f                 0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                            |                                                     |
                                                            V1                                                    V2
```

```
T = 4
Insert (0x9f2, V1)   ->  letters 2, f, 9
Insert (0xc8d, V2)   ->  letters d, 8, c
Insert (0x532, V3)   ->  letters 2, 3, 5


                                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                /                       \
                                               /                         \
               0 1 2 3 4 5 6 7 8 9 a b c d e f                             0 1 2 3 4 5 6 7 8 9 a b c d e f
                    /                         \                                             \
                   /                           \                                             \
                  /                             \                                             \
                 /                               \                                             \
0 1 2 3 4 5 6 7 8 9 a b c d e f           0 1 2 3 4 5 6 7 8 9 a b c d e f                 0 1 2 3 4 5 6 7 8 9 a b c d e f
          |                                                 |                                                     |
          V3                                                V1                                                    V2
```

Key space

- Let us denote
  - K: the set of all values a key can take
  - n: number of tuples in the associative array

- We say that the key space is “sparse” if $n << K$

- We call it “dense” otherwise

“Dense” key space

```
T = 4                     n = 4096
Insert (0x001, W_0)
Insert (0x002, W_2)
. . .
Insert (0xfff, W_4095)

                                              0 1 2 3 4 5 6 7 8 9 a b c d e f
                                                 /                       \
                                                /                         \
                      0 1 2 3 4 5 6 7 8 9 a b c d e f                      0 1 2 3 4 5 6 7 8 9 a b c d e f
                           /                       \                                   \
                          /                         \                                   \
                         /                           \                                   \
                        /                             \                                   \
0 1 2 3 4 5 6 7 8 9 a b c d e f           0 1 2 3 4 5 6 7 8 9 a b c d e f           0 1 2 3 4 5 6 7 8 9 a b c d e f
          |                                                 |                                                 |
          V3                                                V1                                                V2
```

- Insertion/deletion/lookup are $\simeq O(3) = O(\log_{16} 4096) = O(\log_{2^T} n)$
- but…
-
- … then why not use just a static array? (or equivalently choose T = 12)

“Sparse” key space

- Tries only make sense when the key space is sparse

  a static array of the size of the whole key space would be too big

- Complexity not dependent on number of entries
  - Depends on key size and T

- Memory overhead can be large

  worst case: every leaf node has a single tuple, $O(n \times 2^T)$

ASSOCIATIVE ARRAYS:

IMPLEMENTATIONS USING KEYS BITS

–

HASH TABLES

- Again, we denote
  - K: the set of all values a key can take
  - n: number of tuples in the associative array

- If we had a “dense” key space (n not much smaller than K)
  - then we would simply use a static array, indexed by keys

- Could we **map** K into something dense?
  - … and then use a static array

Hash function

- A hash function h is a **mapping** $h : K \to U$ where $U \subseteq \mathbb{N}$ and $|U| << |K|$
  (indeed K may not be a finite set, e.g. for arbitrary-sized keys)

- Since $|U| < |K|$, hash functions are necessarily surjective
  $\exists k_1 \neq k_2$ such that $h(k_1) = h(k_2)$

- Examples of (usually bad) hash functions:
  - take just the lower 8 bits of the key
  - $h : \mathbb{Z} \to \{0, \ldots, m-1\}, \qquad h(k) = k \bmod m$

Hash table

- A hash table is a static array of size $|U|$
- with an associated hash function $h : K \to U$.
- (k, v) tuples are stored in the static array at index $h(k)$
- Since h is surjective, we may have collisions

  (tuples with distinct keys stored at a same array index)

How to deal with collisions (1)

- Make the hash table
  - a static array of linked lists, or
  - a static array of dynamic arrays

- In case of collision, resort to $O(c)$ linear search

  (where c is the maximum number of collisions)
  - in the worst case, $c = n$

```
h(k) = k mod 16
Insert (0x9f2, V1)   ->  h(0x9f2) = 0x2
Insert (0xc8d, V2)   ->  h(0xc8d) = 0xd
Insert (0x532, V3)   ->  h(0x532) = 0x2


                                         0 1 2 3 4 5 6 7 8 9 a b c d e f
                                             |                       |
                                             |                       |
                                             V1                      V2
                                             V3
```

```
h(k) = k mod 16
Insert (0x9f2, V1)   ->  h(0x9f2) = 0x2
Insert (0xc8d, V2)   ->  h(0xc8d) = 0xd
Insert (0x532, V3)   ->  h(0x532) = 0x2
Lookup 0x4d2   ->  h(0x4d2) = 0x2


                        0 1 2 3 4 5 6 7 8 9 a b c d e f
                            |                       |
                            |                       |
                            V1                      V2
                            V3
```

```
h(k) = k mod 16
Insert (0x9f2, V1)   ->  h(0x9f2) = 0x2
Insert (0xc8d, V2)   ->  h(0xc8d) = 0xd
Insert (0x532, V3)   ->  h(0x532) = 0x2
Lookup 0x4d2   ->  h(0x4d2) = 0x2

                          *
                      0 1 2 3 4 5 6 7 8 9 a b c d e f
                          |                       |
                          |                       |
                          V1                      V2
                          V3
```

```
h(k) = k mod 16
Insert (0x9f2, V1)   ->  h(0x9f2) = 0x2
Insert (0xc8d, V2)   ->  h(0xc8d) = 0xd
Insert (0x532, V3)   ->  h(0x532) = 0x2
Lookup 0x4d2   ->  h(0x4d2) = 0x2   ->  not found

                                            *
                                        0 1 2 3 4 5 6 7 8 9 a b c d e f
                                            |                     |
                                            |                     |
                                            V1                    V2
                                            V3
```

How to deal with collisions (2): Open addressing

- Insertion of `(key, value)`:
  - Step 0: Compute index `i = h(key)`
  - Step 1: If `array[i]` is empty,
    - place `(key, value)` there, done.
  - Step 2: Otherwise,
    - let `i = (i + 1) mod |U|`,
    - go back to Step 1.

```
h(k) = k mod 16
Insert (0x9f2, V1)   ->  h(0x9f2) = 0x2
Insert (0xc8d, V2)   ->  h(0xc8d) = 0xd
Insert (0x532, V3)   ->  h(0x532) = 0x2
```

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
    | |                   |
    | |                   |
    V1V3                   V2
```

```
h(k) = k mod 16
Insert (0x9f2, V1)   ->  h(0x9f2) = 0x2
Insert (0xc8d, V2)   ->  h(0xc8d) = 0xd
Insert (0x532, V3)   ->  h(0x532) = 0x2
Lookup 0x4d2   ->  h(0x4d2) = 0x2


                          0 1 2 3 4 5 6 7 8 9 a b c d e f
                              | |                   |
                              | |                   |
                              V1V3                   V2
```

```
h(k) = k mod 16
Insert (0x9f2, V1)   ->  h(0x9f2) = 0x2
Insert (0xc8d, V2)   ->  h(0xc8d) = 0xd
Insert (0x532, V3)   ->  h(0x532) = 0x2
Lookup 0x4d2   ->  h(0x4d2) = 0x2

                                  *
                              0 1 2 3 4 5 6 7 8 9 a b c d e f
                                  | |                   |
                                  | |                   |
                                  V1V3                   V2
```

```
h(k) = k mod 16
Insert (0x9f2, V1)   ->  h(0x9f2) = 0x2
Insert (0xc8d, V2)   ->  h(0xc8d) = 0xd
Insert (0x532, V3)   ->  h(0x532) = 0x2
Lookup 0x4d2   ->  h(0x4d2) = 0x2

            *
0 1 2 3 4 5 6 7 8 9 a b c d e f
    | |                   |
    | |                   |
    V1V3                   V2
```

```
h(k) = k mod 16
Insert (0x9f2, V1)   ->  h(0x9f2) = 0x2
Insert (0xc8d, V2)   ->  h(0xc8d) = 0xd
Insert (0x532, V3)   ->  h(0x532) = 0x2
Lookup 0x4d2   ->  h(0x4d2) = 0x2

                                    *
                          0 1 2 3 4 5 6 7 8 9 a b c d e f
                              | |                   |
                              | |                   |
                              V1V3                   V2
```

```
h(k) = k mod 16
Insert (0x9f2, V1)   ->  h(0x9f2) = 0x2
Insert (0xc8d, V2)   ->  h(0xc8d) = 0xd
Insert (0x532, V3)   ->  h(0x532) = 0x2
Lookup 0x4d2   ->  h(0x4d2) = 0x2   ->  not found

                                     *
                         0 1 2 3 4 5 6 7 8 9 a b c d e f
                             | |                   |
                             | |                   |
                             V1V3                   V2
```

- Lookup for `key`:
  - Step 0: Compute index `i = h(key)`
  - Step 1: If `array[i]` matches `key`,
    - return `array[i]`.
  - Step 2: If `array[i]` is empty,
    - return not found.
  - Step 2: Otherwise,
    - let `i = (i + 1) mod |U|`,
    - go back to Step 1.

Probing

- Insertion of (k, v):
  - Step 0:
    - Compute index $h_0 = h(k)$
    - Let $j = 0$
  - Step 1: If $\texttt{array}[i(h_0, j)]$ is empty,
    - place (k, v) there, done.
  - Step 2: Otherwise,
    - let $j = j + 1$,
    - go back to Step 1.

- where $i(h_0, j)$ can be:
  - $i(h_0, j) = (h_0 + j) \bmod |U|$ as before
  - $i(h_0, j) = (h_0 + Kj) \bmod |U|$ for some constant K ("linear probing")
  - $i(h_0, j) = (h_0 + Kj + Lj^2) \bmod |U|$ for some K, L ("quadratic probing")

```
h(k) = k mod 16
Insert (0x9f2, V1)   ->  h(0x9f2) = 0x2
Insert (0xc8d, V2)   ->  h(0xc8d) = 0xd
Insert (0x532, V3)   ->  h(0x532) = 0x2


                          0 1 2 3 4 5 6 7 8 9 a b c d e f
                              | |                   |
                              | |                   |
                              V1V3                   V2
```

Good hash functions

- in practice, naive hash functions yield horrible collision rates

  (even for random keys!)

- good hash functions perform great on real (non-random) keys
  - they take a non-uniform distribution of keys over K
  - map it into a distribution over U that “looks” uniformly random

- Fowler–Noll–Vo (FNV), djb2, SipHash (lookup “non-cryptographic hash functions”)

- Such generic hash functions h_0 typically return 32-, 64- or 128-bit numbers.
  - we use index $h(k) = h_0(k) \bmod |U|$

Complexity of hash table operations

- performance depends on
  - density ($n/|U|$)
  - key distribution
  - hash function
  - probing method

- when density approaches 1,
  - increase $|U|$ (e.g. double it)
  - rebuild hash table ("rehashing")

In practice

- as long as collision rate is kept low
  - insert/delete/lookup are essentially $O(1)$

- first hash table access is typically a cache miss (at least L1)

- but with open addressing, in case of collisions, probing may not be

ASSOCIATIVE ARRAYS:

PERFORMANCE

- Between self-balancing trees, tries and hash tables, no clearly superior data structure.
- Data- and application-dependent.
- Try, benchmark

- Hash tables often perform better… when suitable:
  - when hashing is cheap
  - when we can ensure few collisions
  - when the order of magnitude of n known in advance
- Self-balancing trees are often more robust:
  - **much** better worst case non-amortized complexity (rehashing!)
- Tries can be faster when keys have a special structure
  - page table (virtual address translation)
  - network routing (IP addresses)
  - GPT-type tokenizers

Combinations are possible and commonly used

- Hash table as a static array of self-balancing trees
- Depth-K trie with self-balancing trees at leaf nodes
- …

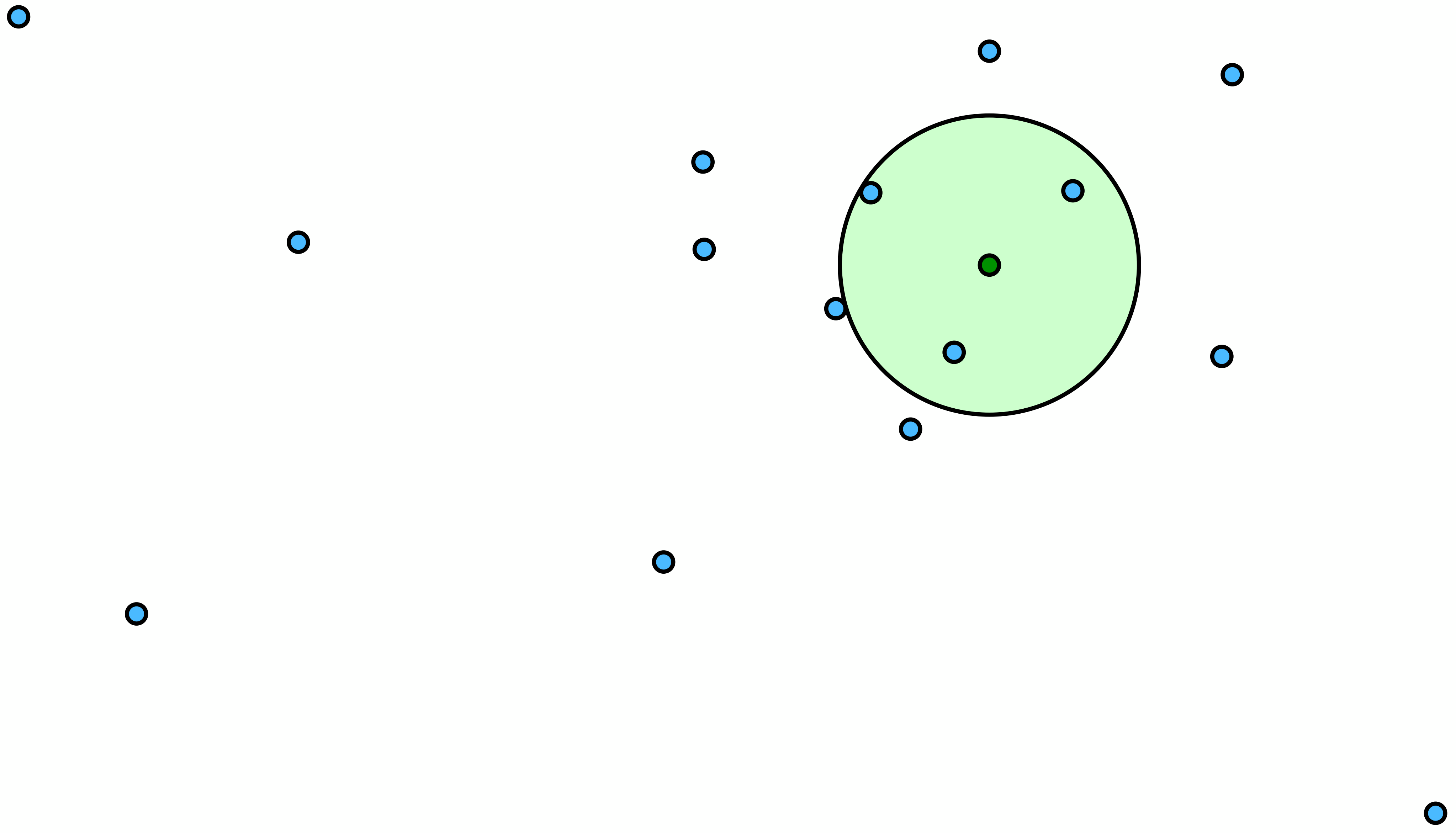
SPATIAL DATA STRUCTURES

Spatial data structures

- Spatial data structures store collections of vectors in $\mathbb{R}^m$
- they allow operations such as
  - insertion (add a vector $x \in \mathbb{R}^m$)
  - deletion (remove one vector)
  - find the vector closest to a given $y \in \mathbb{R}^m$
  - for every inserted vector, find its nearest neighbor
  - for every inserted vector, find its k nearest neighbors
  - for every inserted vector, find all other vectors within a distance d

The problem

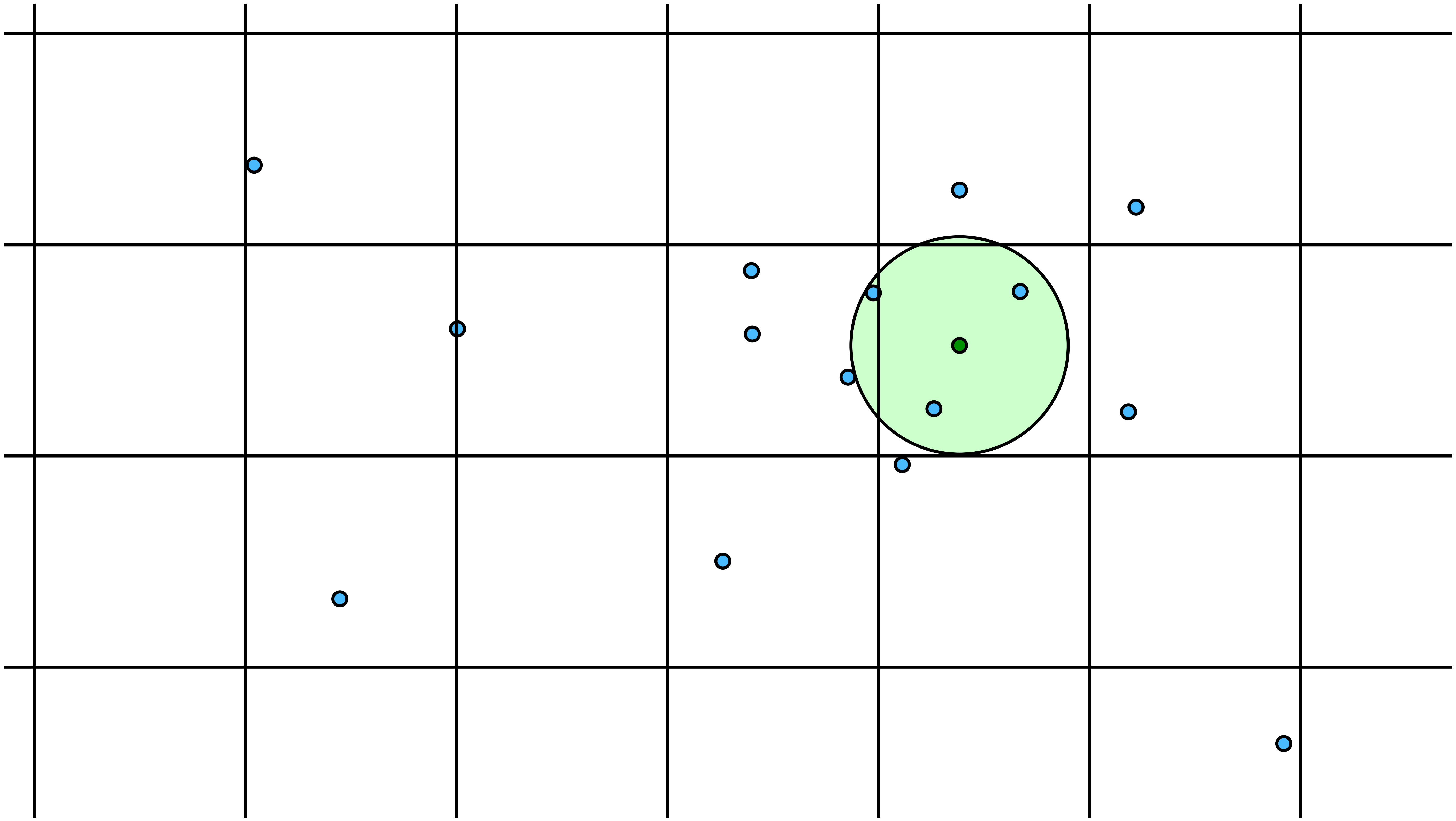
“for every inserted vector, find all other vectors within a distance d”



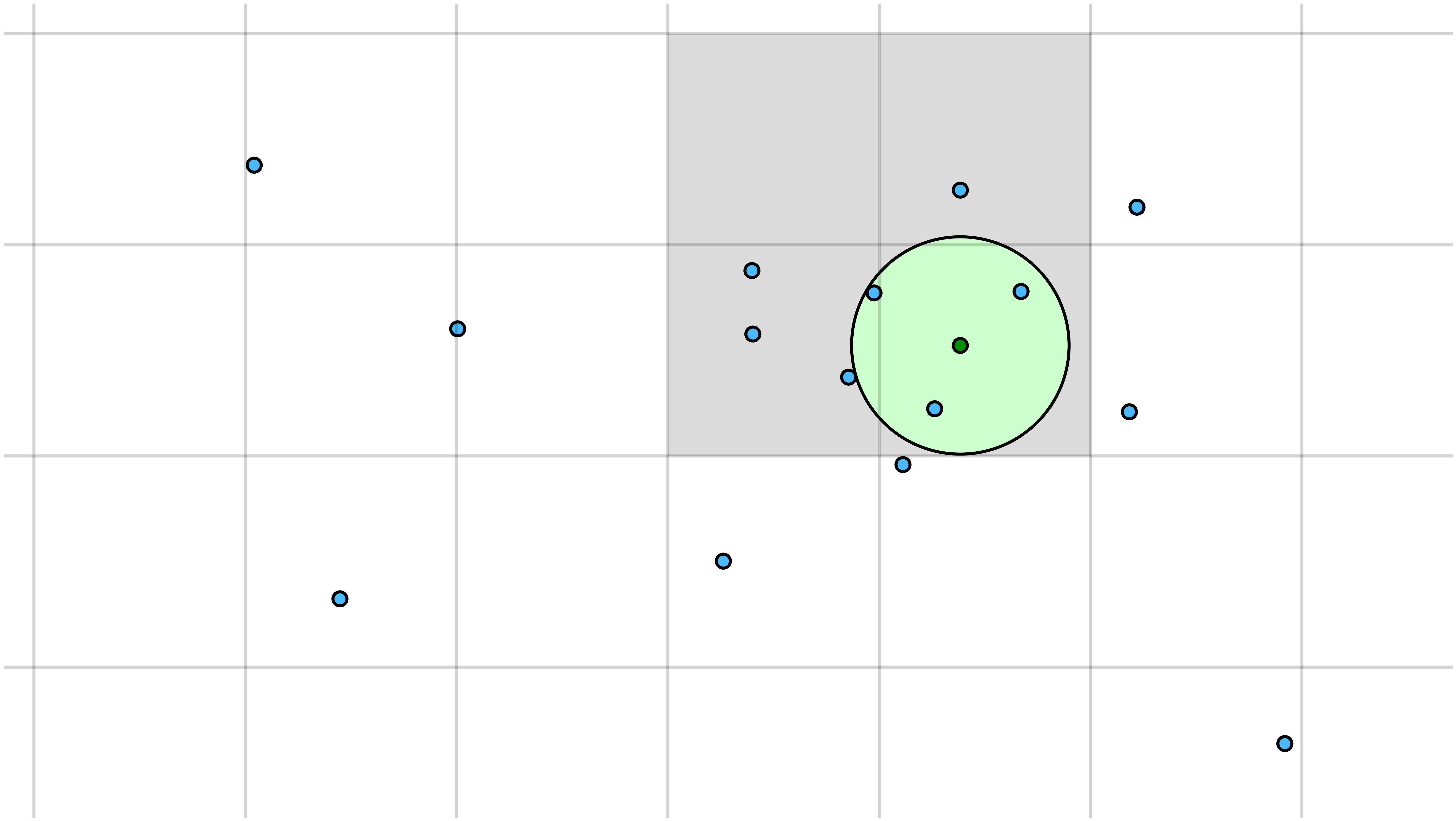
Naively, this problem has $O(n^2)$ complexity:

$$
\begin{aligned}
&R := \emptyset \\
&\text{For } i = 0, \ldots, n-1: \\
&\quad \text{For } j = i+1, \ldots, n-1: \\
&\qquad \text{If } ||x^i - x^j|| \leq d: \\
&\qquad\quad R := R \cup \{(i,j)\}
\end{aligned}
$$

Grids



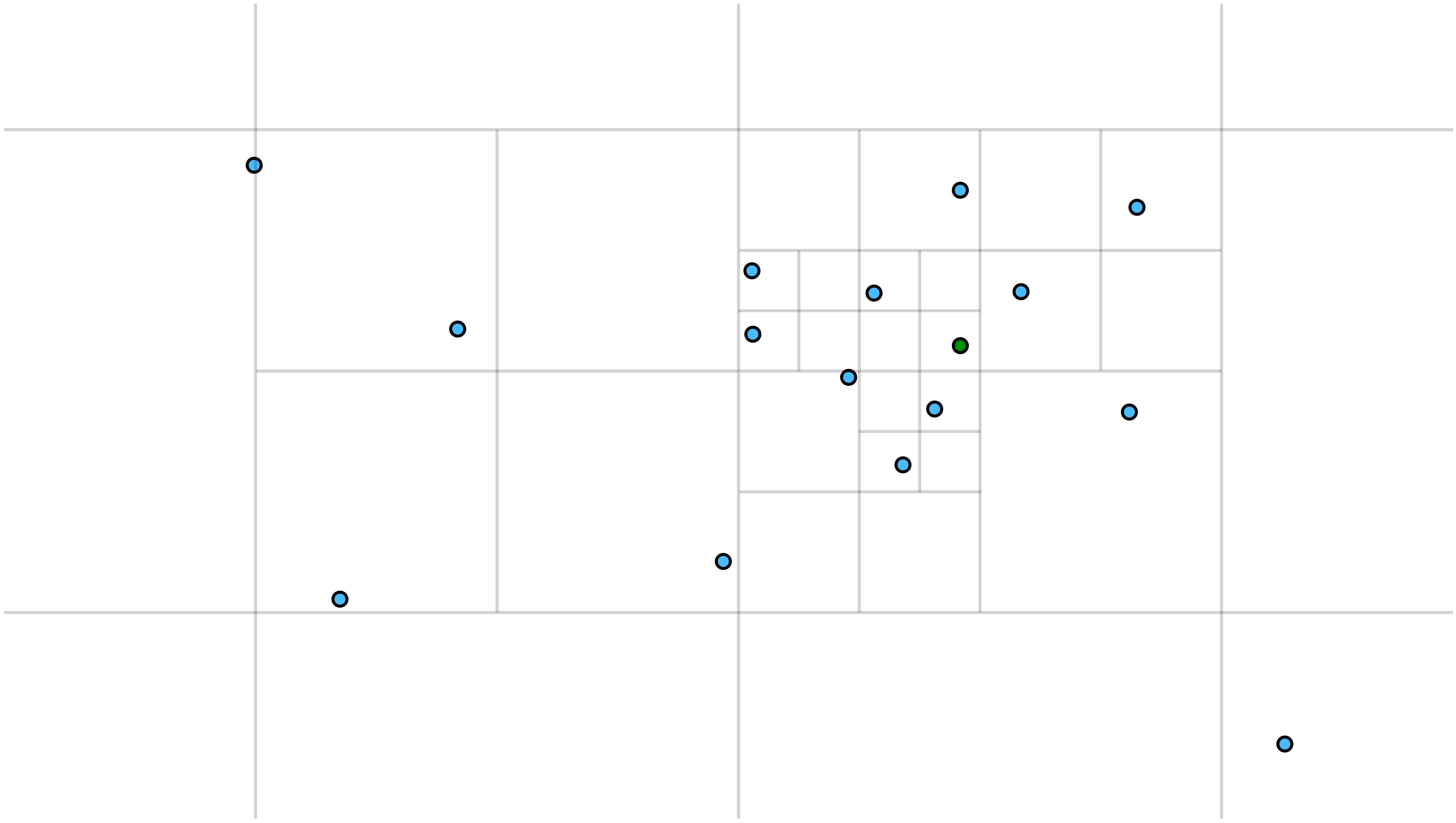
Grids



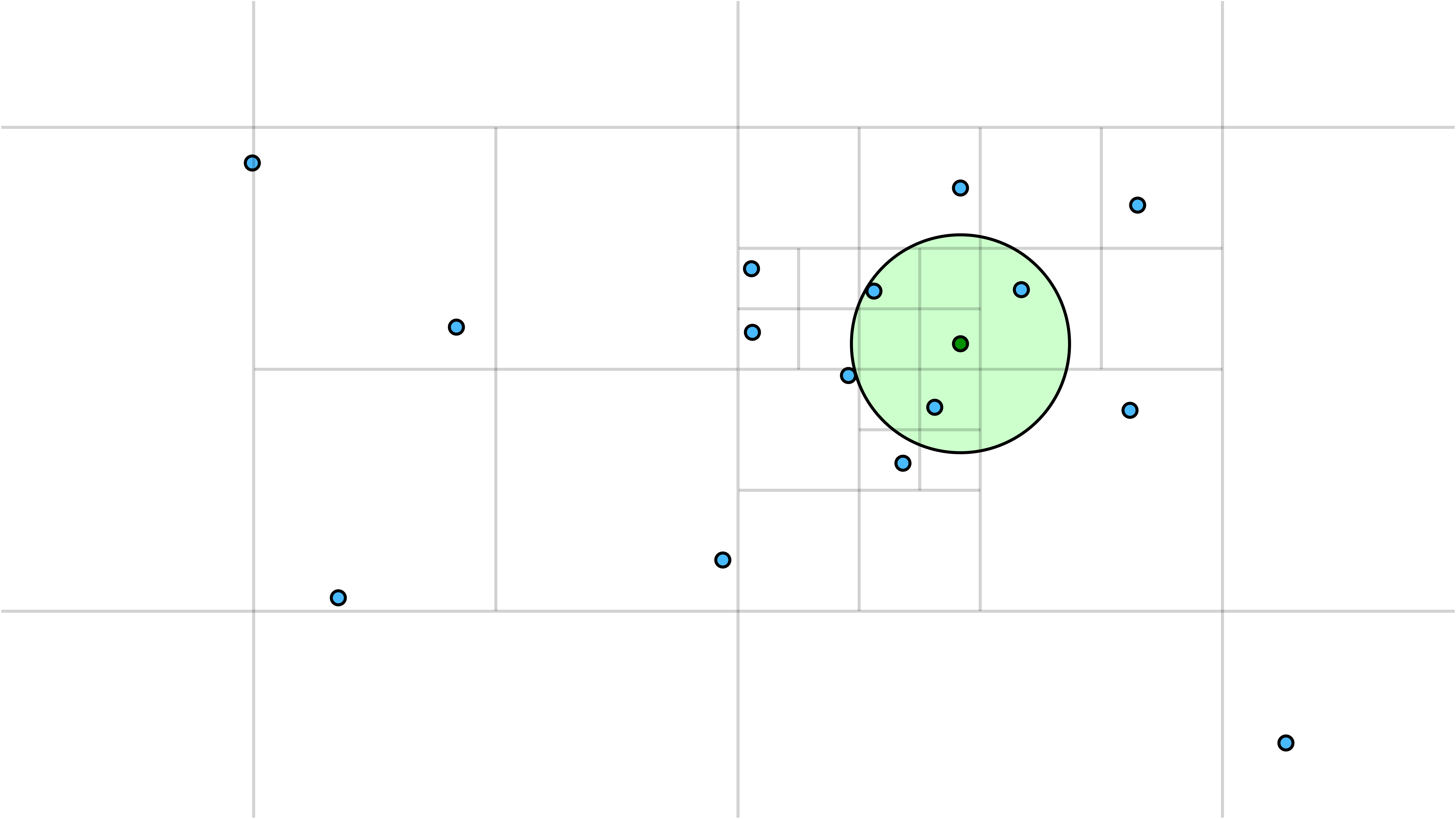
Grids

- Pros:
  - quadratic only within grid cells
- Cons:
  - need finite bounds $L \leq x_i \leq U$ for all x, for all i
  - fixed cell size
    - some may have too many xs
    - many may be empty

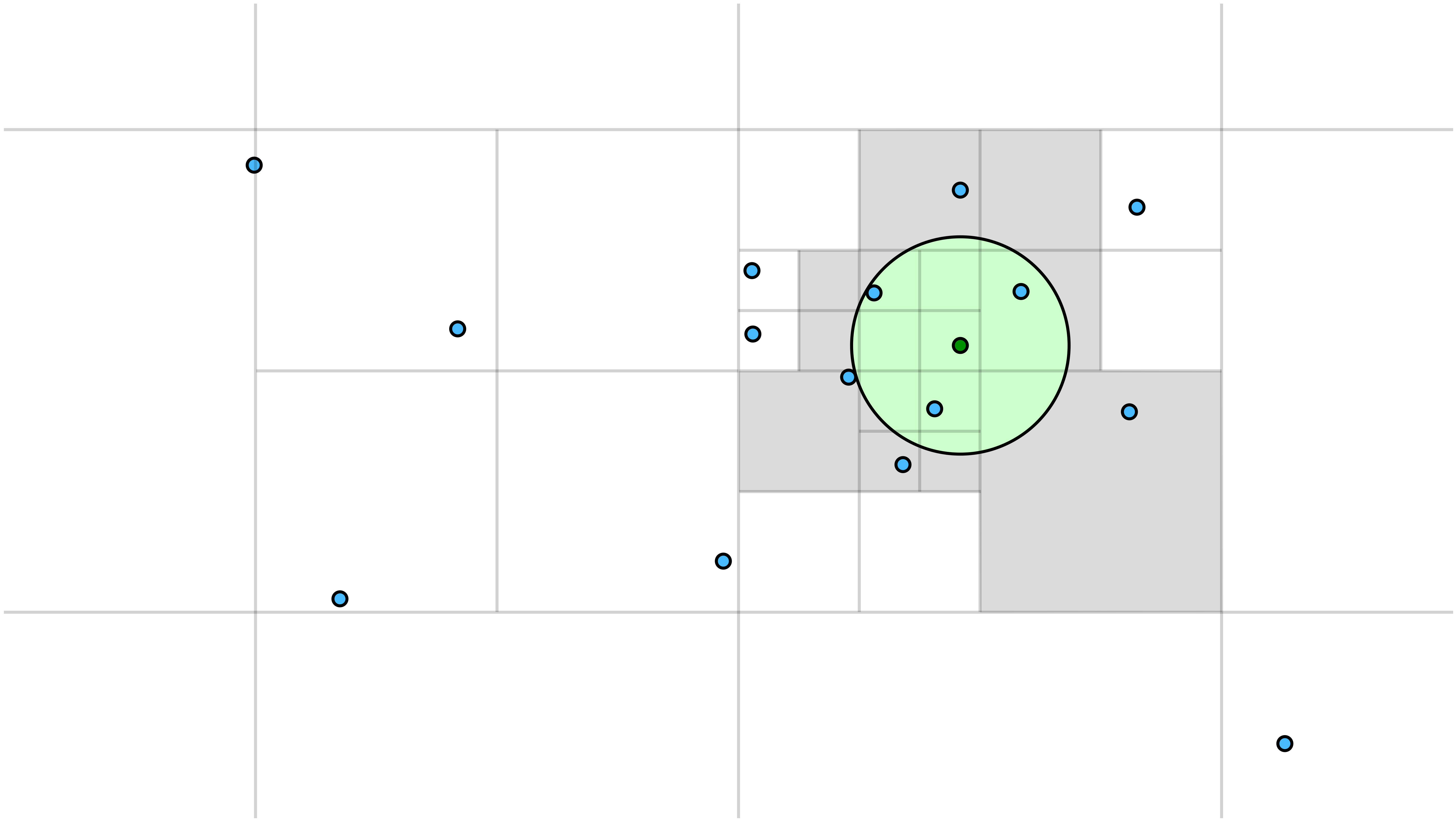
Quadtrees and octrees



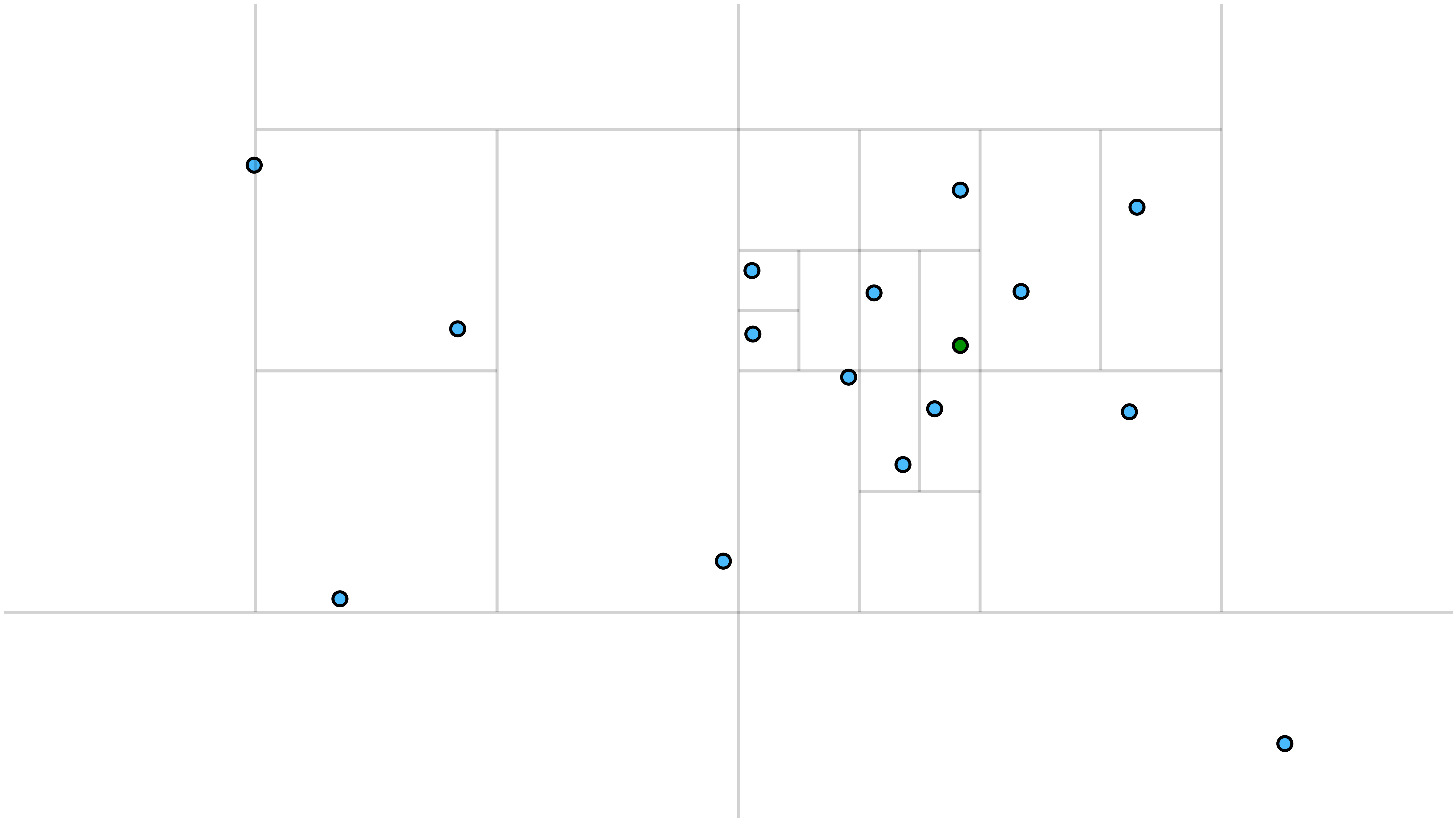
Quadtrees and octrees



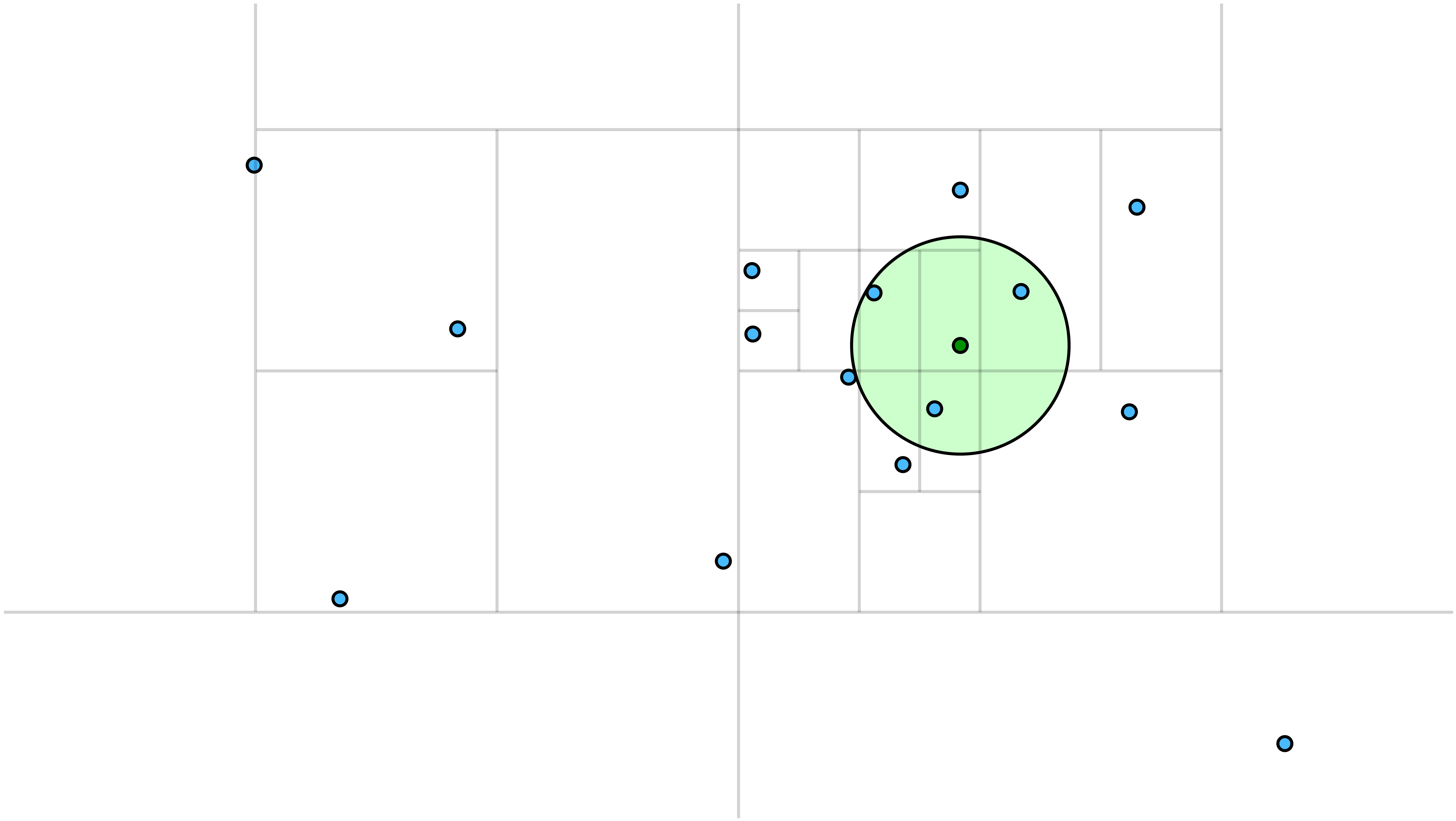
Quadtrees and octrees



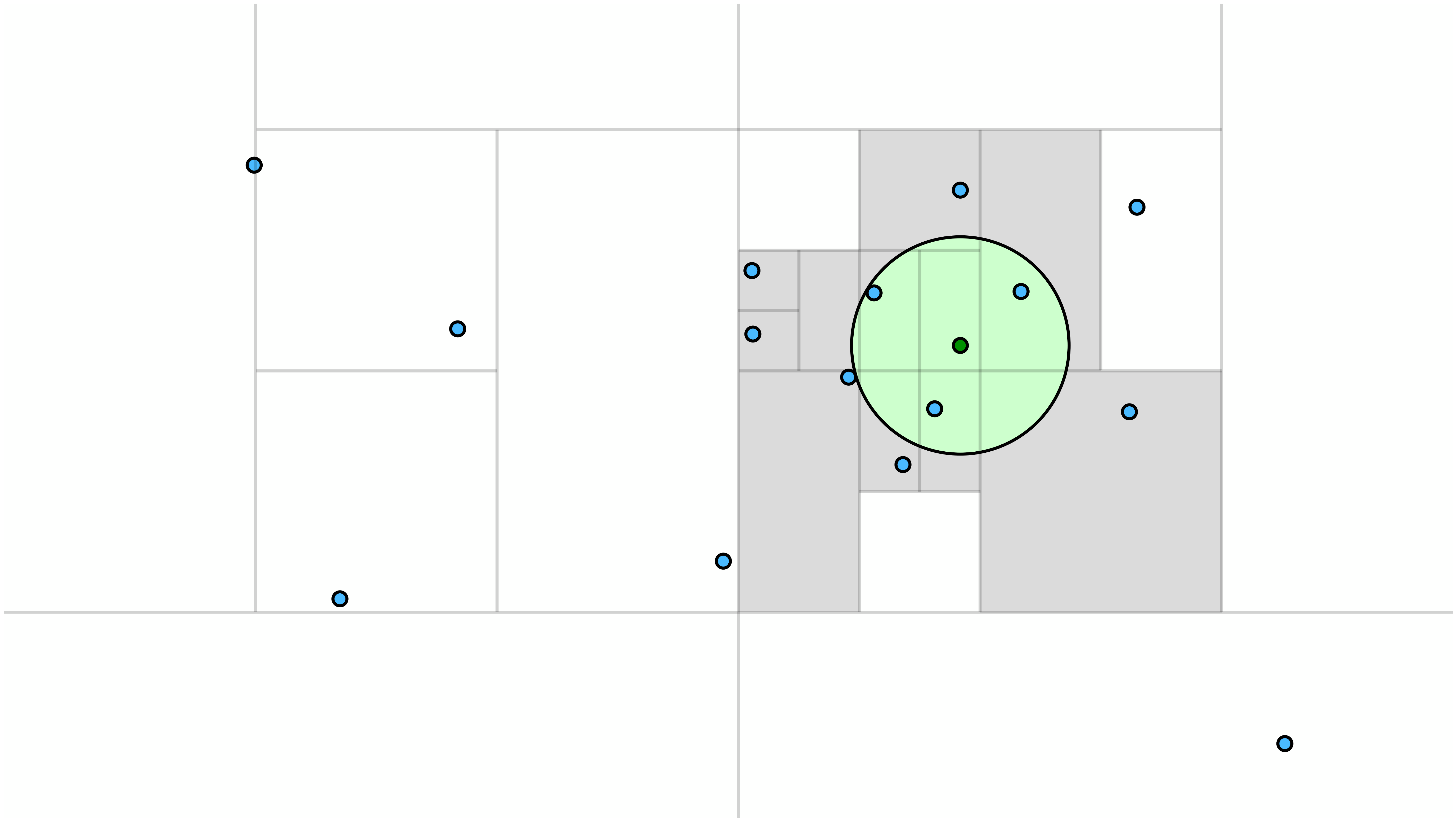
k-d trees



k-d trees



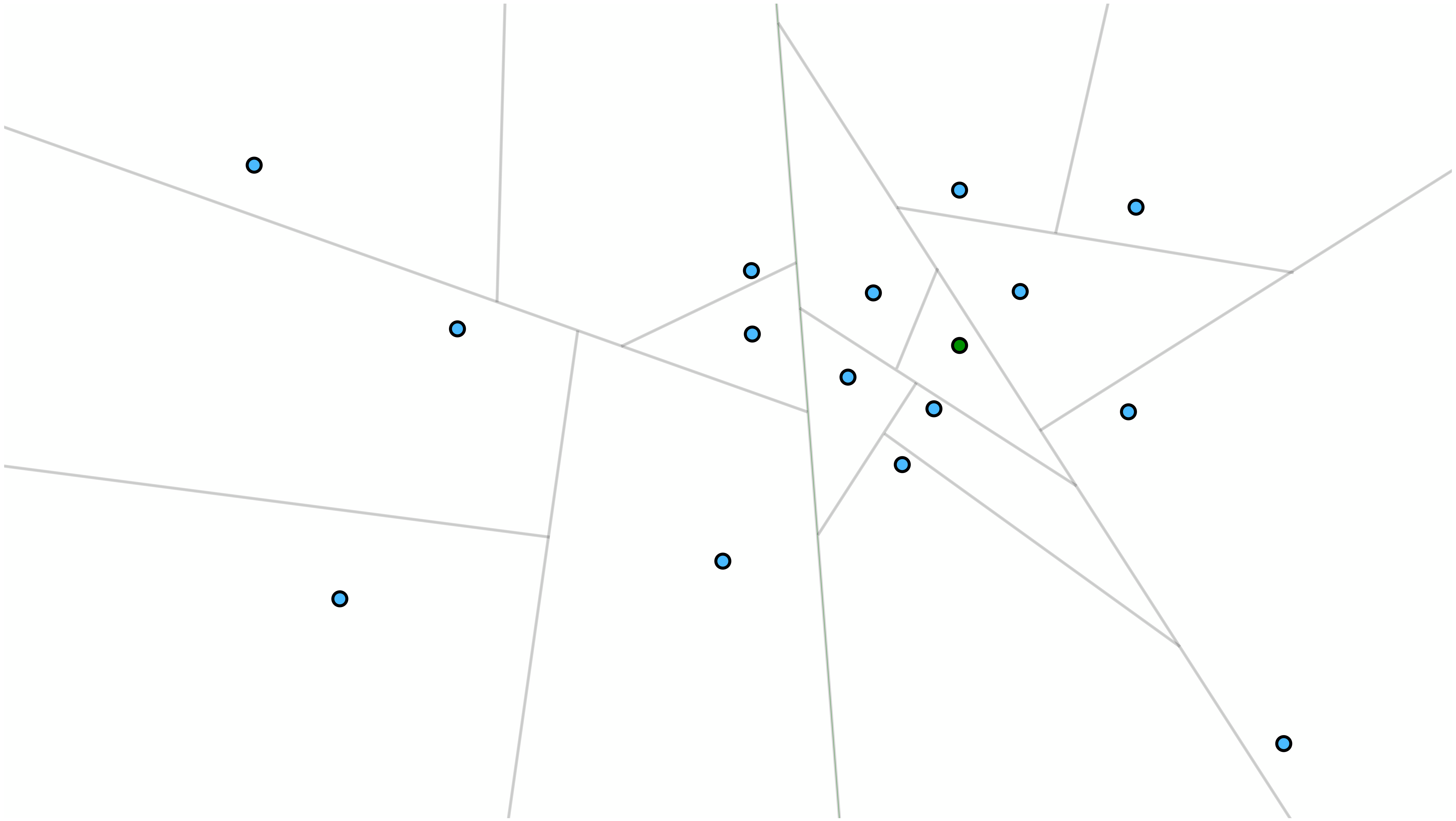
k-d trees



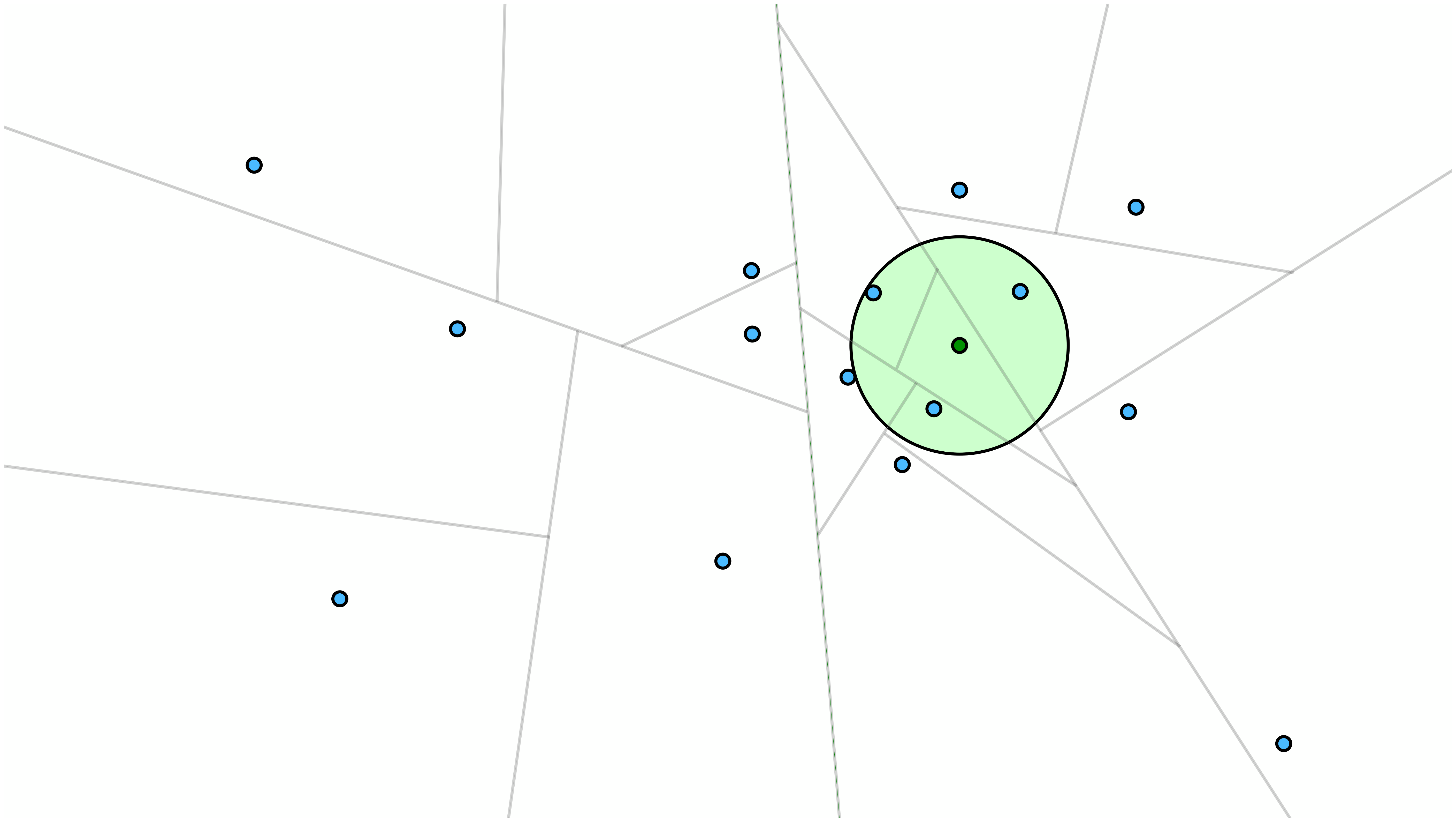
Quadtrees, octrees, k-d trees

- Pros:
  - no need for finite bounds $L \leq x_i \leq U$ for all x, for all i
  - variable cell size
- Limitations:
  - fixed cell shape (cubes / boxes)
  - poor fit for high-dimensional data:
    - as m grows
      - data size grows linearly
      - number of cells grows exponentially
        - even if all points are on a 2-dimensional hyperplane

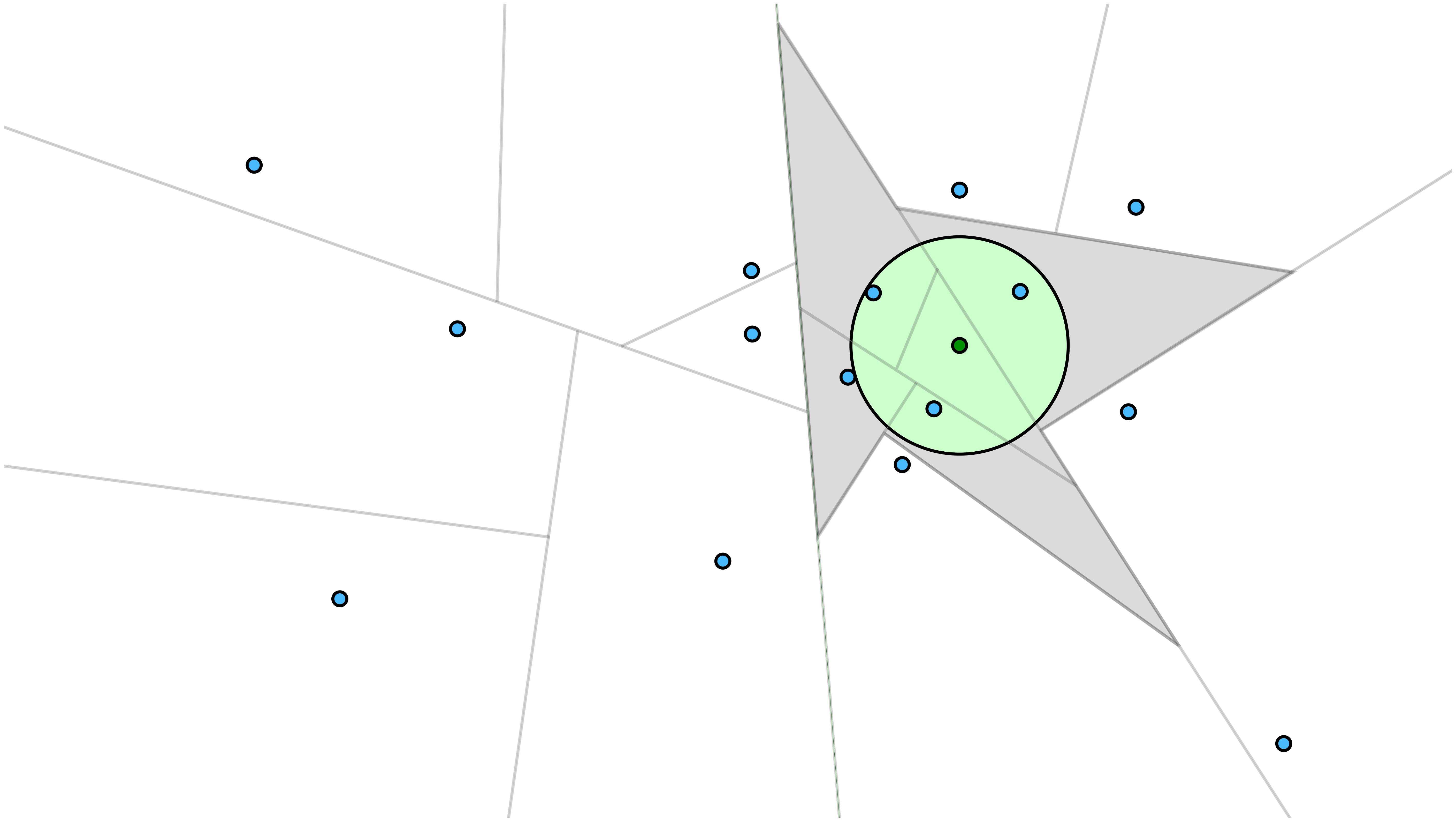
Binary space partitioning



Binary space partitioning



Binary space partitioning



Binary space partitioning

- Pros:
  - variable cell shape
- Cons:
  - separating hyperplane computation is costly
- Limitations:
  - not a good fit for high-dimensional data if, e.g. on a 2-dimensional curved manifold

Locality-sensitive hashing

- Design a function $h : \mathbb{R}^m \to \mathbb{R}$
- such that $||y - x||$ small $\Rightarrow$ $|h(y) - h(x)|$ small, with high probability
- Impossible in all generality
- Depends on data