



# Introduction to Cover Tree

Javen Qinfeng Shi

SML of NICTA

RSISE of ANU

2006 Oct. 15



# Outline

- ❖ Goal
- ❖ Related works
- ❖ Cover tree definition
- ❖ Optimized Implement in C++
- ❖ Complexity
- ❖ Further work

# Outline

- ❖ Goal
- ❖ Related works
- ❖ Cover tree definition
- ❖ Efficiently Implement in C++
- ❖ Advantages and disadvantages
- ❖ Further work

# Goal

- ❖ Speed up Nearest-neighbour search [John Langford etc. 2006]
  - ⌘ Preprocess a dataset  $S$  of  $n$  points in some metric space  $X$  so that given a query point  $p \in X$ , a point  $q \in S$  which minimises  $d(p, q)$  can be efficiently found
- ❖ Also Kmeans

# Outline

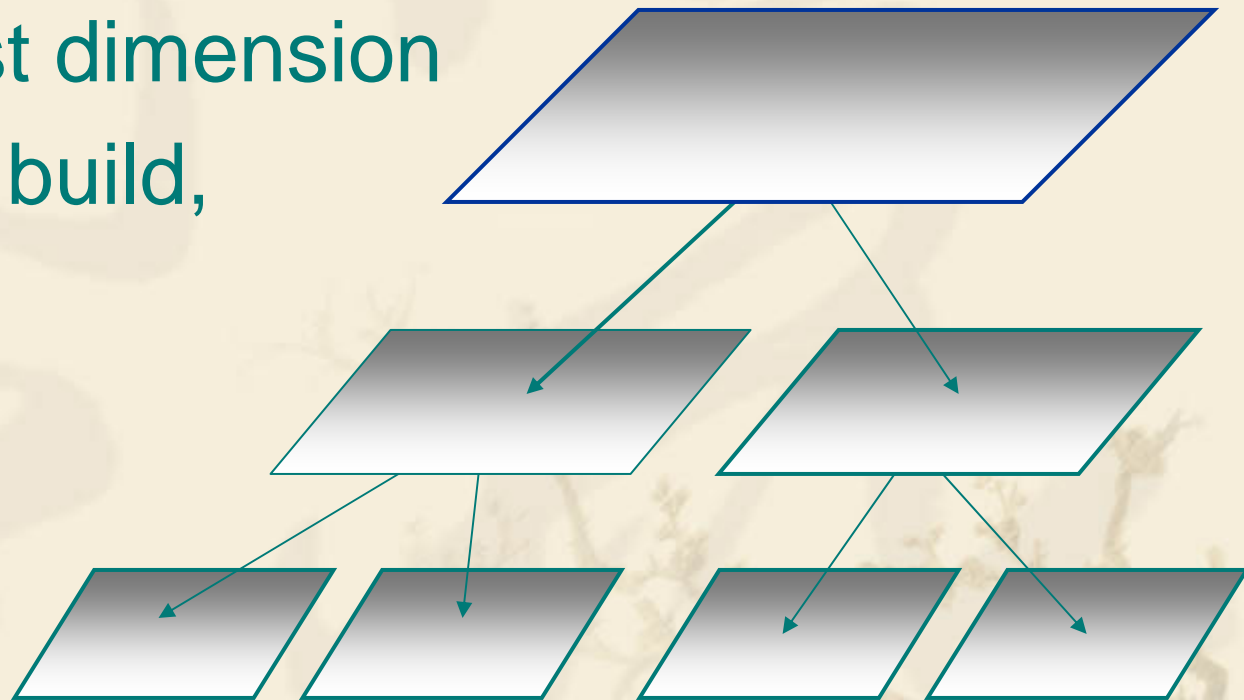
- ❖ Goal
- ❖ Related works
- ❖ Cover tree definition
- ❖ Example
- ❖ Optimized Implement in C++
- ❖ Advantages and disadvantages
- ❖ Further work

# Relevant works

- ❖ Kd tree [Friedman, Bentley & Finkel 1977]
- ❖ Ball tree family:
  - ❧ Ball tree [Omohundro, 1991]
  - ❧ Metric tree [Uhlmann, 1991]
  - ❧ Anchors metric tree [Moore 2000]
  - ❧ (Improved) Anchors metric tree [Charles 2003]
- ❖ Navigating net [R. Krauthgamer, J. Lee 2004] ?
  - Lots of discussions and proofs on its computational bound

# Kd tree

- ❖ Univariate axis-aligned splits
- ❖ Split on widest dimension
- ❖  $O(N \log N)$  to build,  
 $O(N)$  space



# Kd tree

```
class Node:
    pass

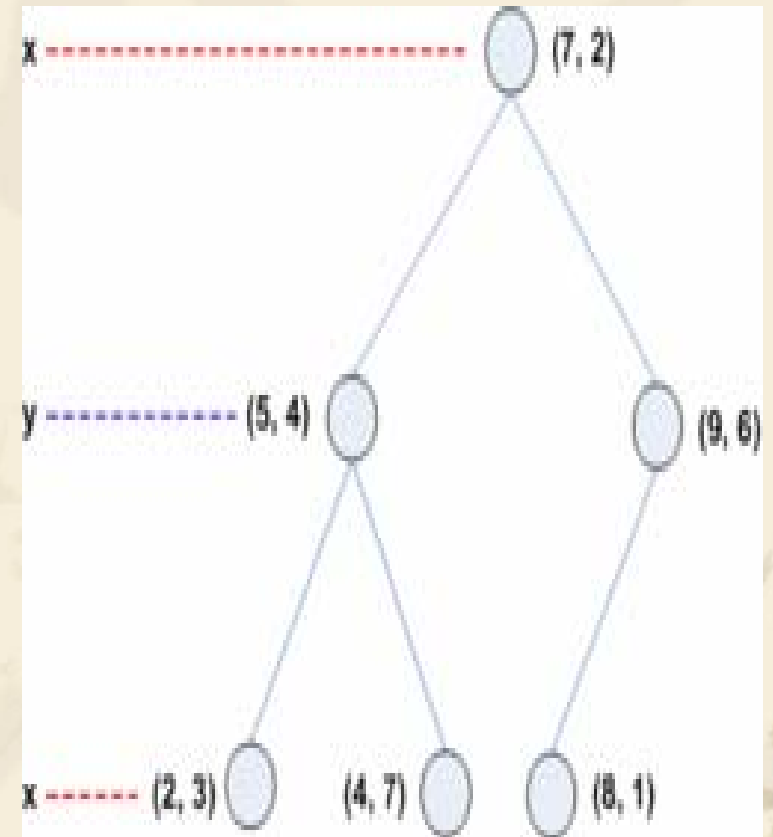
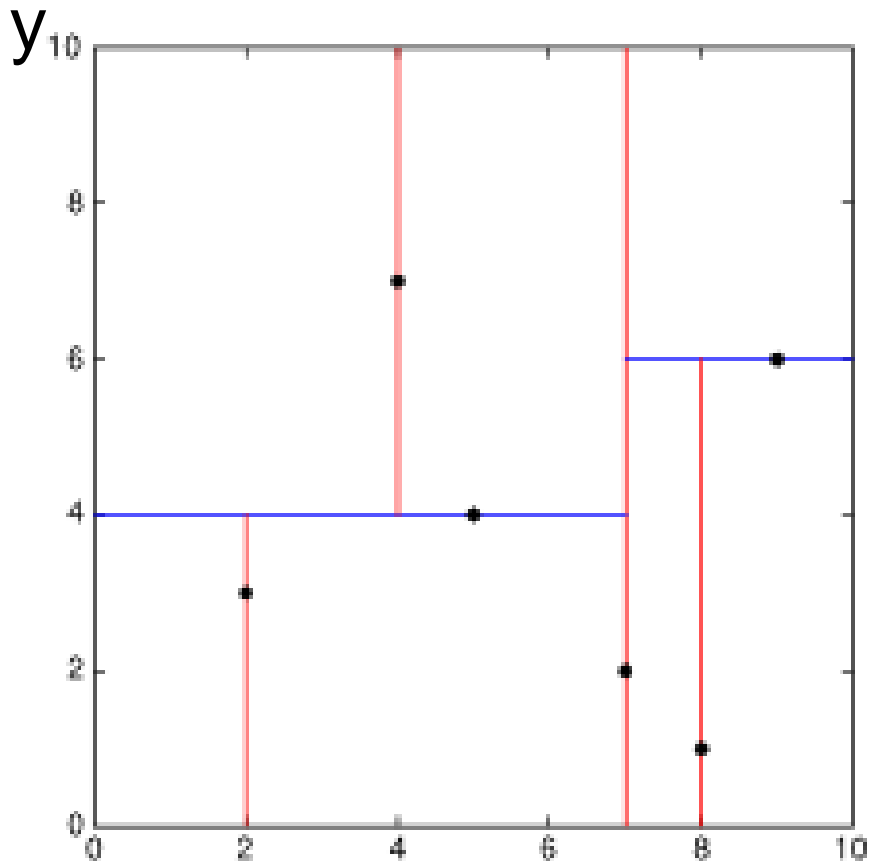
def kdtree(pointList, depth=0):
    if not pointList:
        return

    # Select axis based on depth so that axis cycles through all valid values
    k = len(pointList[0]) # Assumes all points have the same dimension
    axis = depth % k

    # Sort point list to select median
    pointList.sort(cmp=lambda x, y: cmp(x[axis], y[axis]))
    median = len(pointList)//2 # Choose median

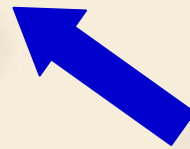
    # Create node and construct subtrees
    node = Node()
    node.location = pointList[median]
    node.leftChild = kdtree(pointList[:median], depth+1)
    node.rightChild = kdtree(pointList[median+1:], depth+1)
    return node
```

# Kd tree



x (axis)

# Ball tree/metric tree



A set of points in  $R^2$

[Uhlmann 1991], [Omohundro 1991]



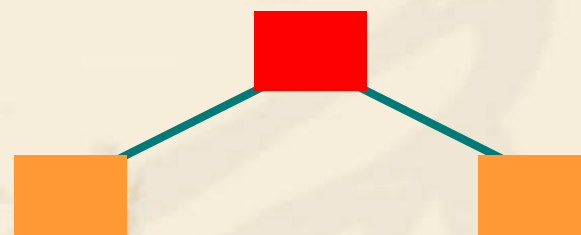
A ball-tree: level 1



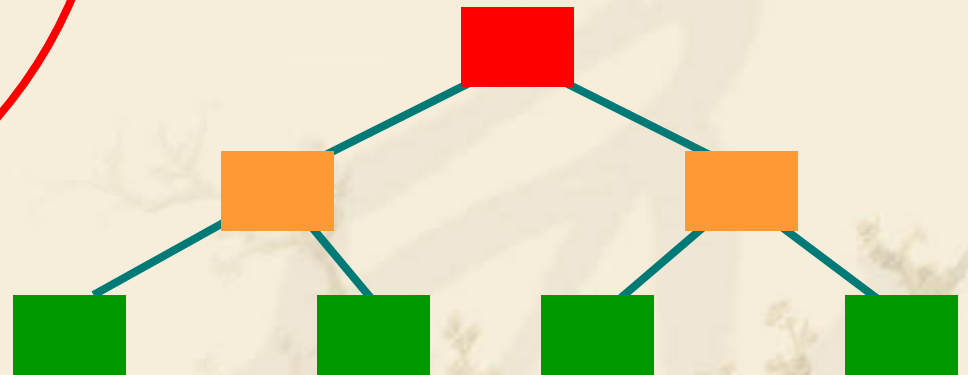
[Uhlmann 1991], [Omohundro 1991]



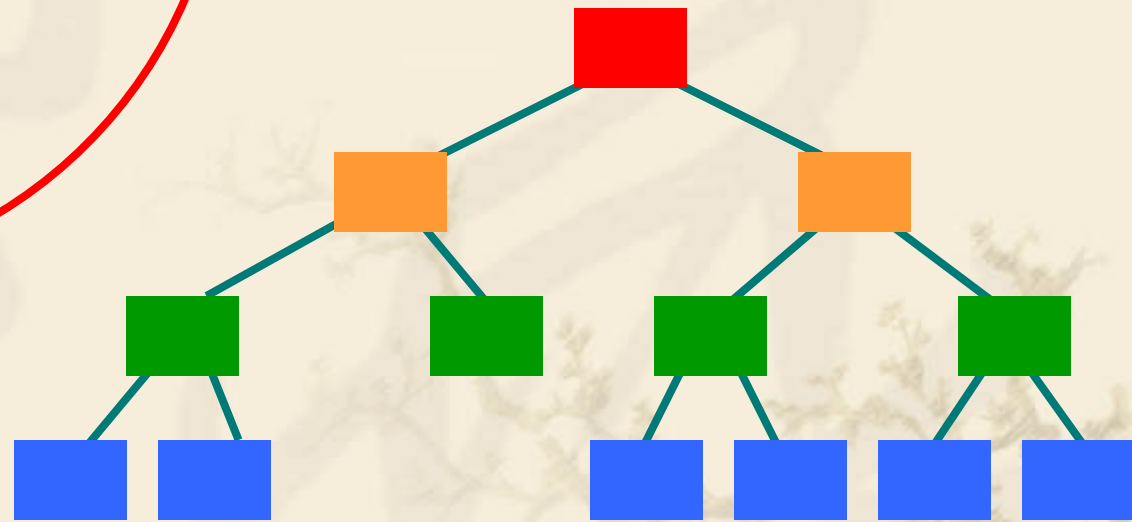
A ball-tree: level 2



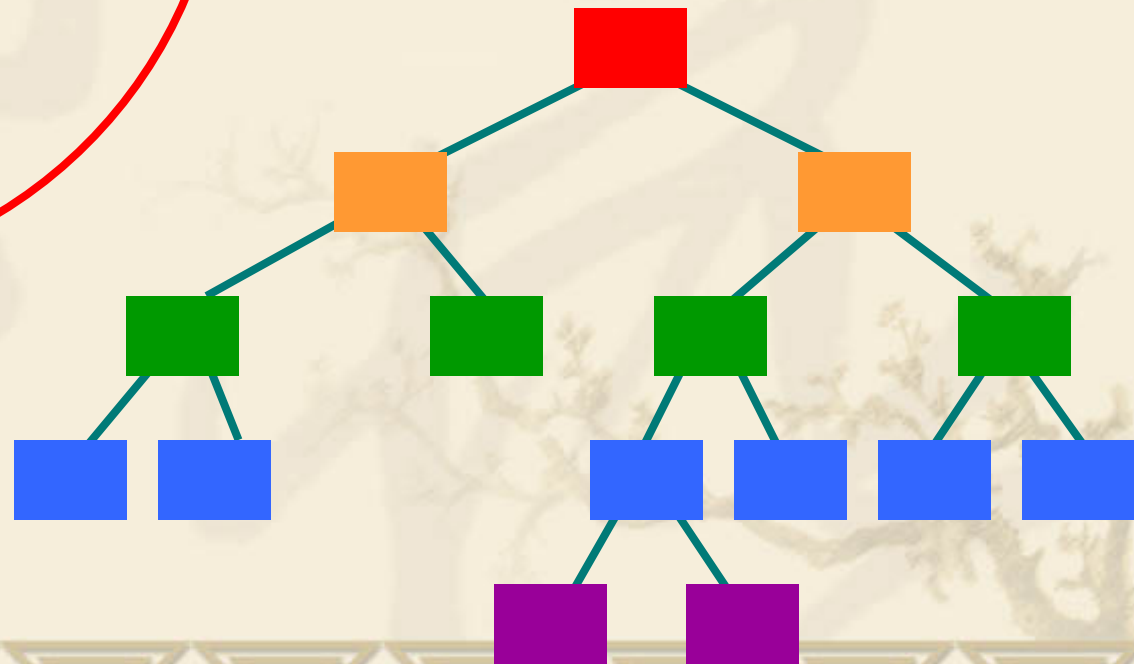
# A ball-tree: level 3



A ball-tree: level 4



A ball-tree: level 5



# Outline

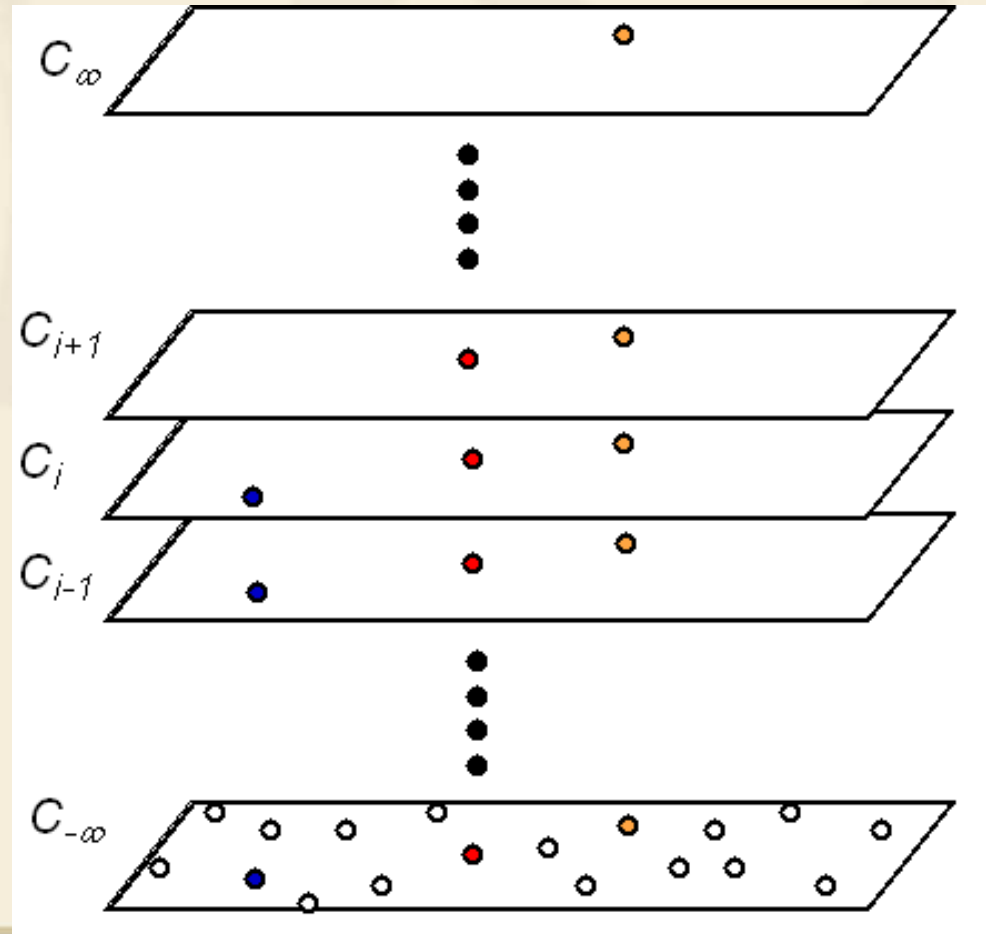
- ❖ Goal
- ❖ Related works
- ❖ Cover tree definition
  - ⌘ Basic definition
  - ⌘ How to build a cover tree
  - ⌘ Query a point/points
- ❖ Optimized Implement in C
- ❖ Advantages and disadvantages
- ❖ Further work

# Basic definition

- ❖ A cover tree  $T$  on a dataset  $S$  is a leveled tree where each level is indexed by an integer scale  $i$  which decreases as the tree is descended
- ❖  $C_i$  denotes the set of nodes at level  $i$
- ❖  $d(p, q)$  denotes the distance between points  $p$  and  $q$
- ❖ A valid tree satisfies the following properties
  - ⌘ **Nesting:**  $C_i \subset C_{i-1}$
  - ⌘ **Covering tree:** For every node  $p \in C_{i-1}$ , there exists a node  $q \in C_i$  satisfying  $d(p, q) \leq 2^i$  and exactly one such  $q$  is a parent of  $p$
  - ⌘ **Separation:** For all nodes  $p, q \in C_i$ ,  $d(p, q) > 2^i$

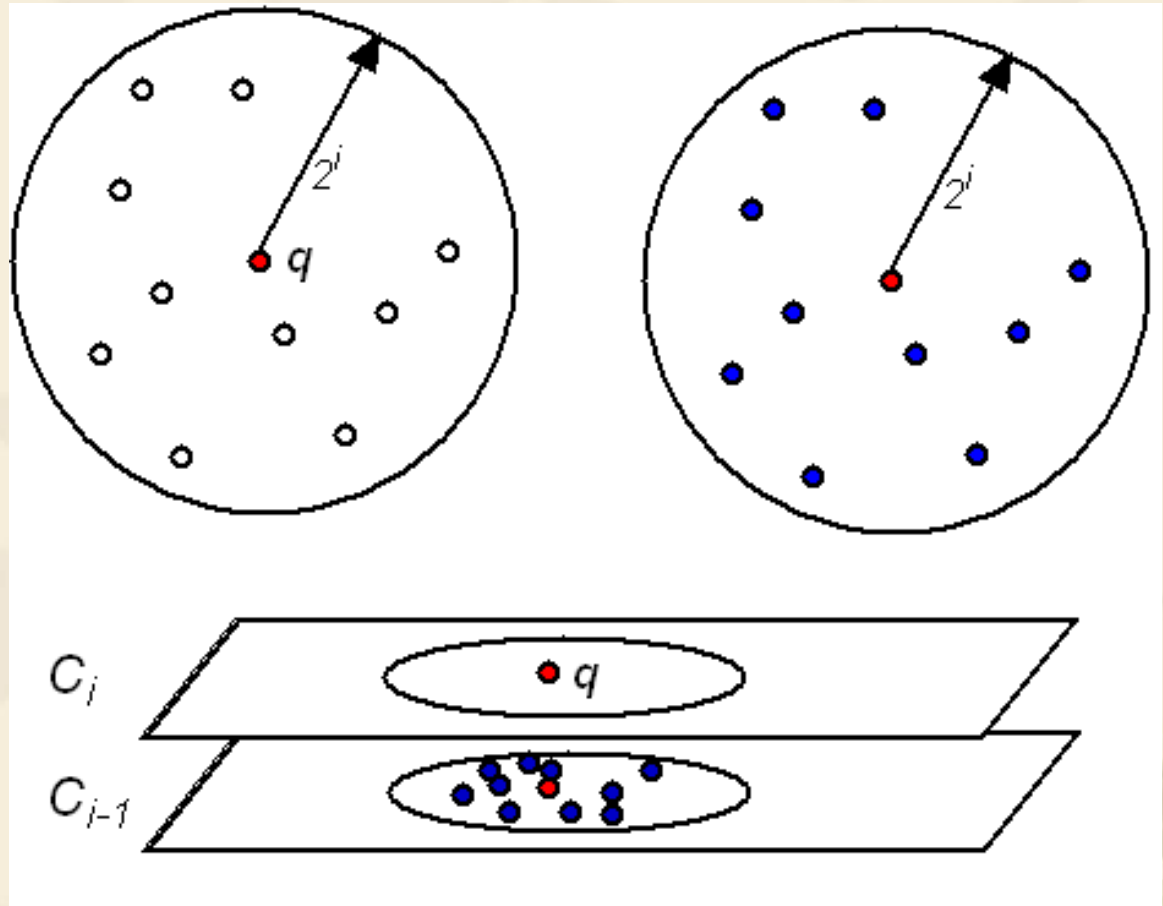
# Nesting

- ❖  $C_i \subset C_{i-1}$ 
  - ⌘ Each node in set  $C_i$  has a self-child
  - ⌘ All nodes in set  $C_i$  are also nodes in sets  $C_j$  where  $j < i$
  - ⌘ Set  $C_{-\infty}$  contains all the nodes in dataset  $S$



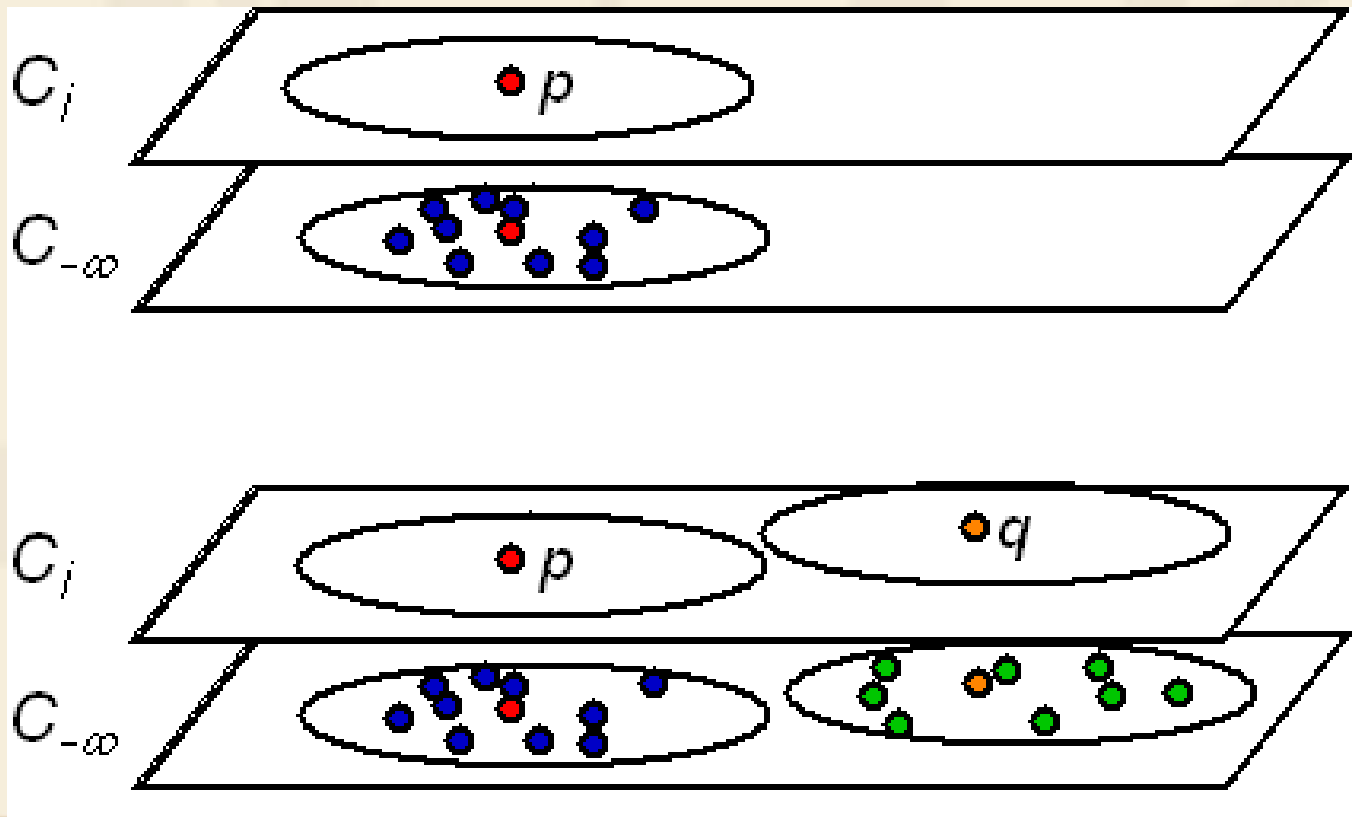
# Cover tree

- ❖ For every node  $p \in C_{i-1}$ , there exists a node  $q \in C_i$  satisfying  $d(p, q) \leq 2^i$  and exactly one such  $q$  is a parent of  $p$

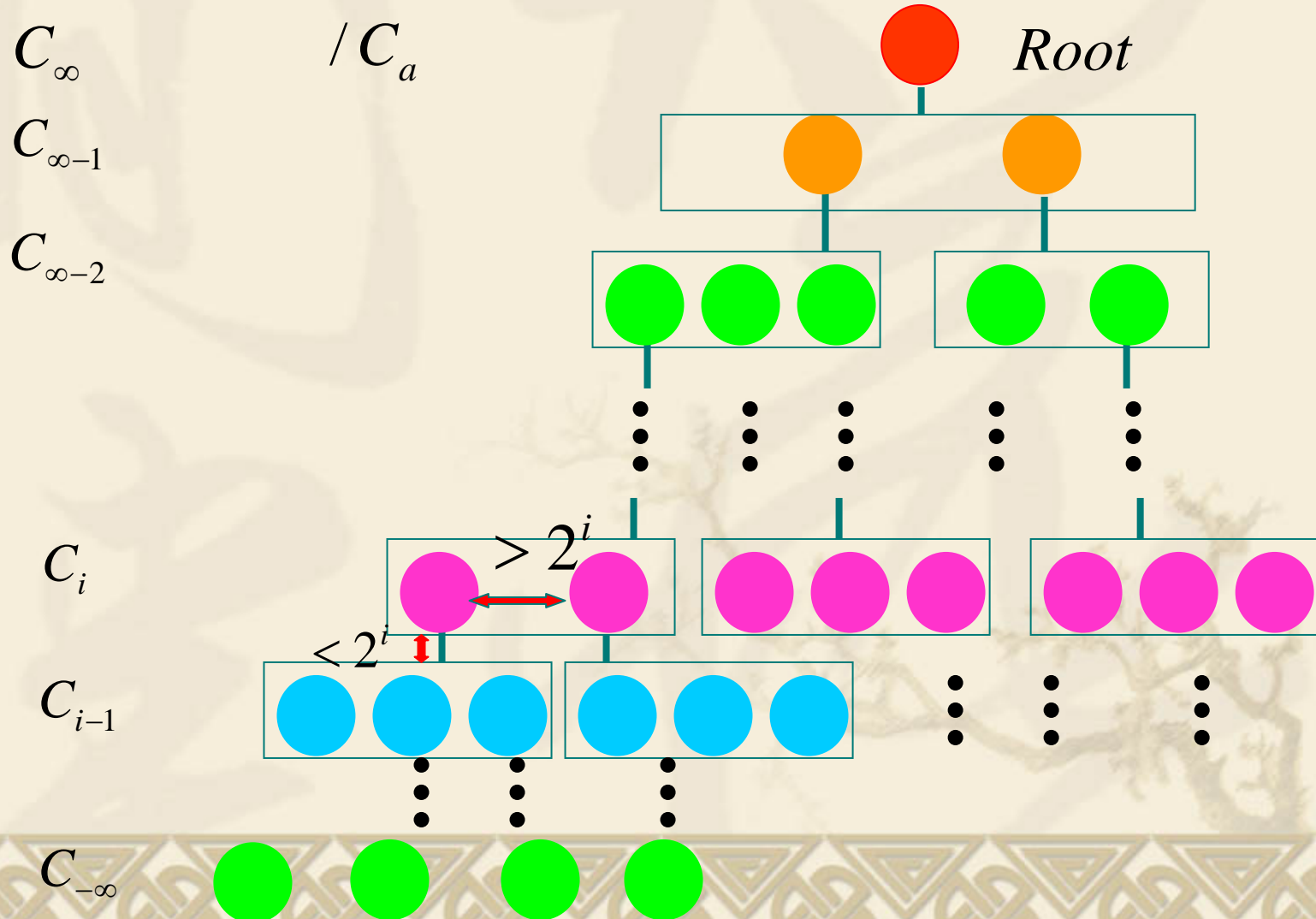


# Separation

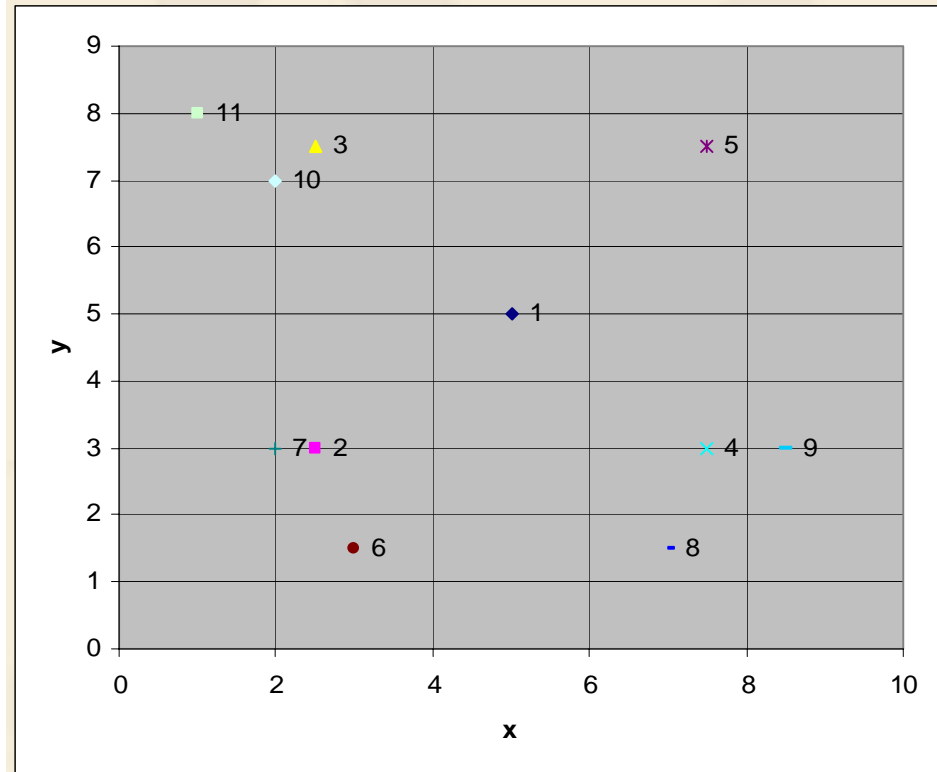
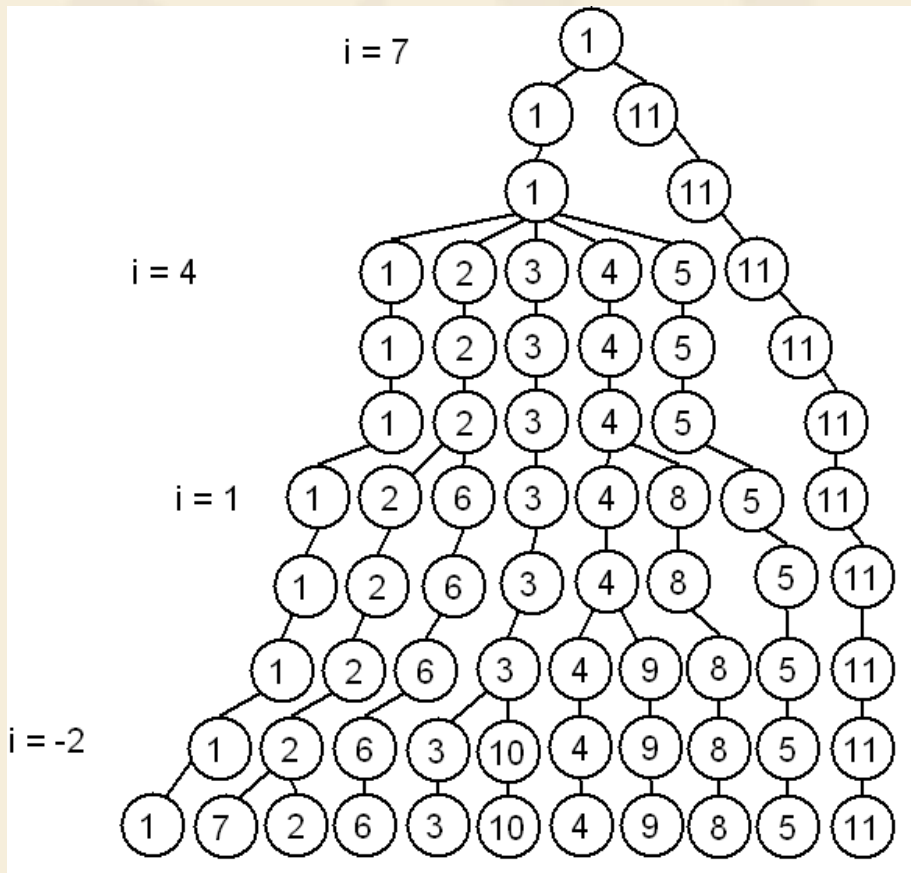
- ❖ For all nodes  $p, q \in C_i$   $d(p, q) > 2^i$



# Cover tree structure



# Cover tree structure

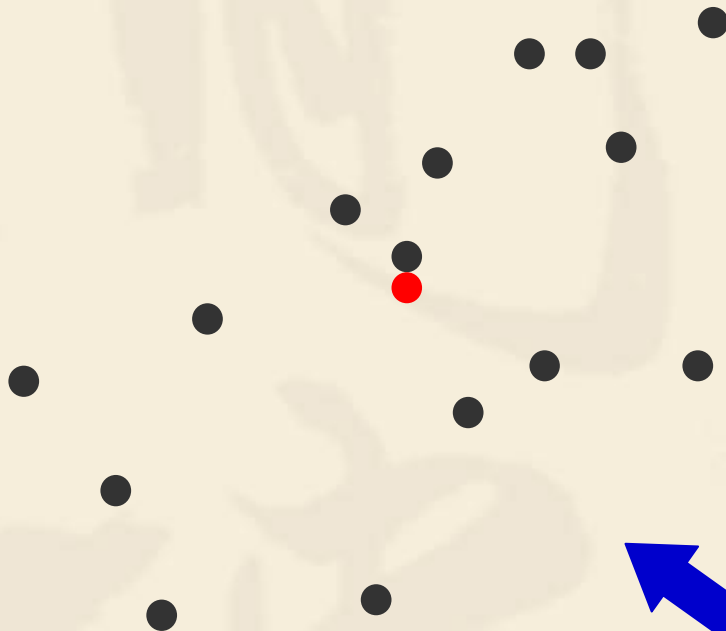


# How to build a Cover tree



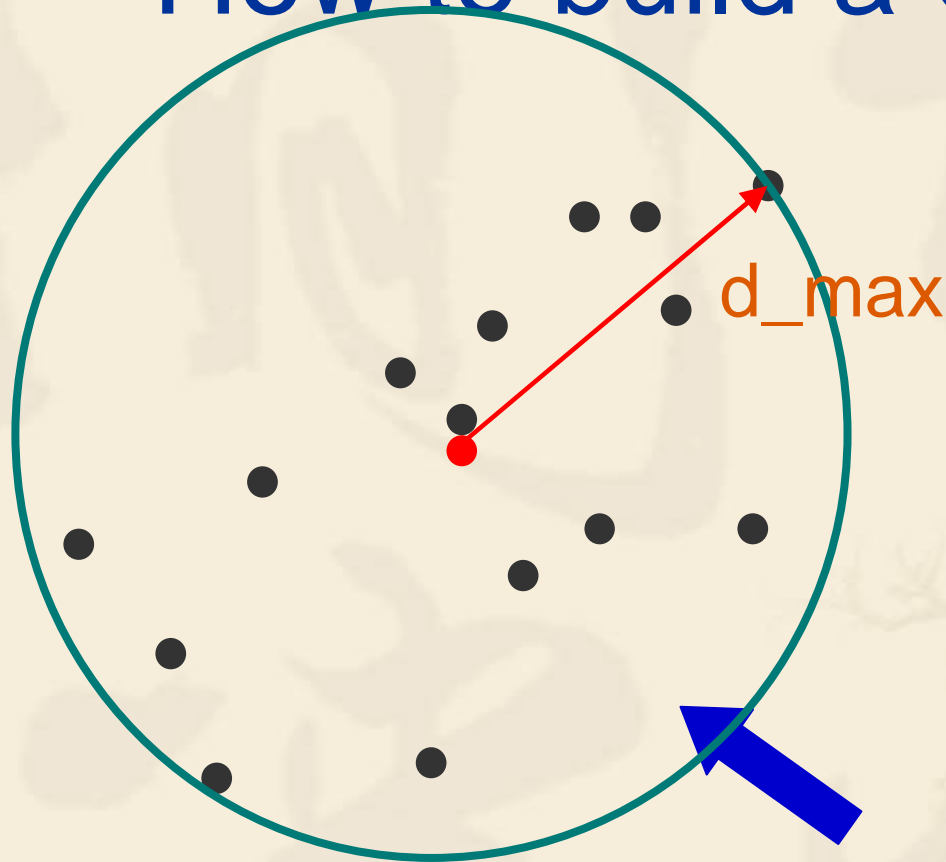
A set of points in  $\mathbb{R}^2$

# How to build a Cover tree



A set of points in  $R^2$

# How to build a Cover tree

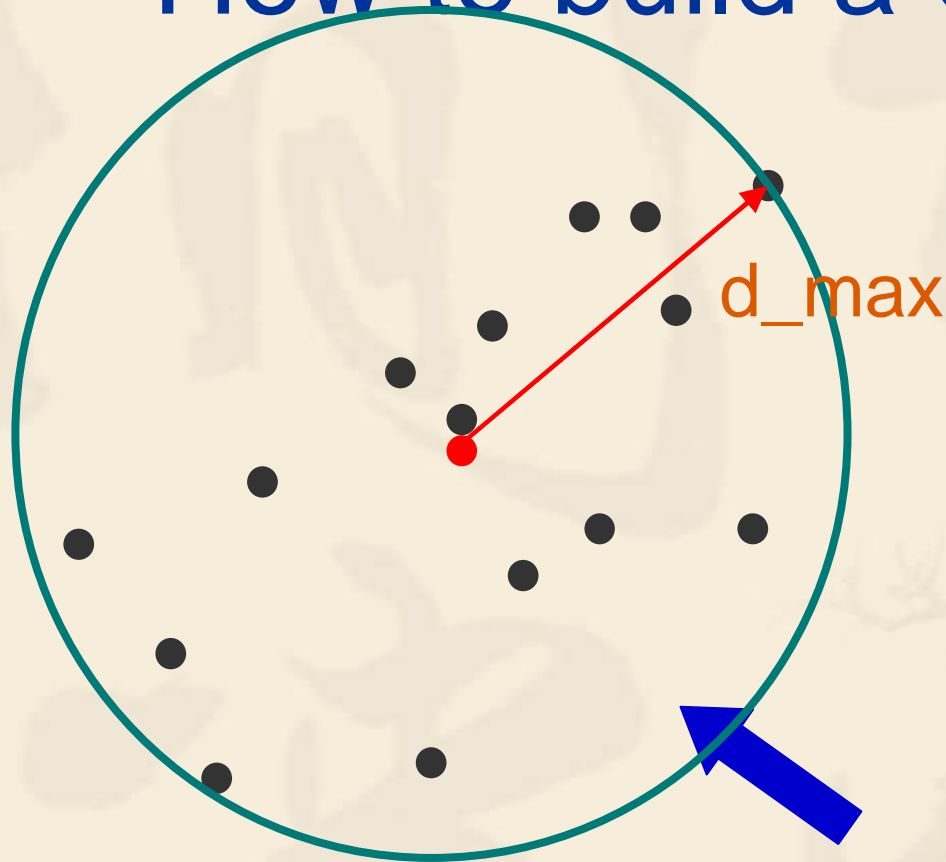


$C_a$  ● *Root*

$a=?$

A set of points in  $\mathbb{R}^2$

# How to build a Cover tree

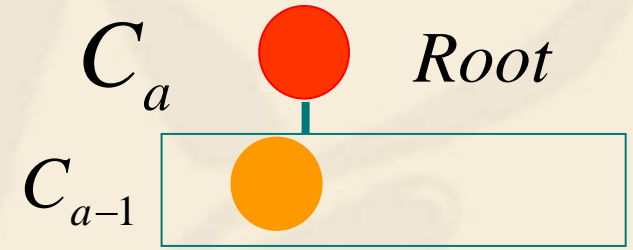
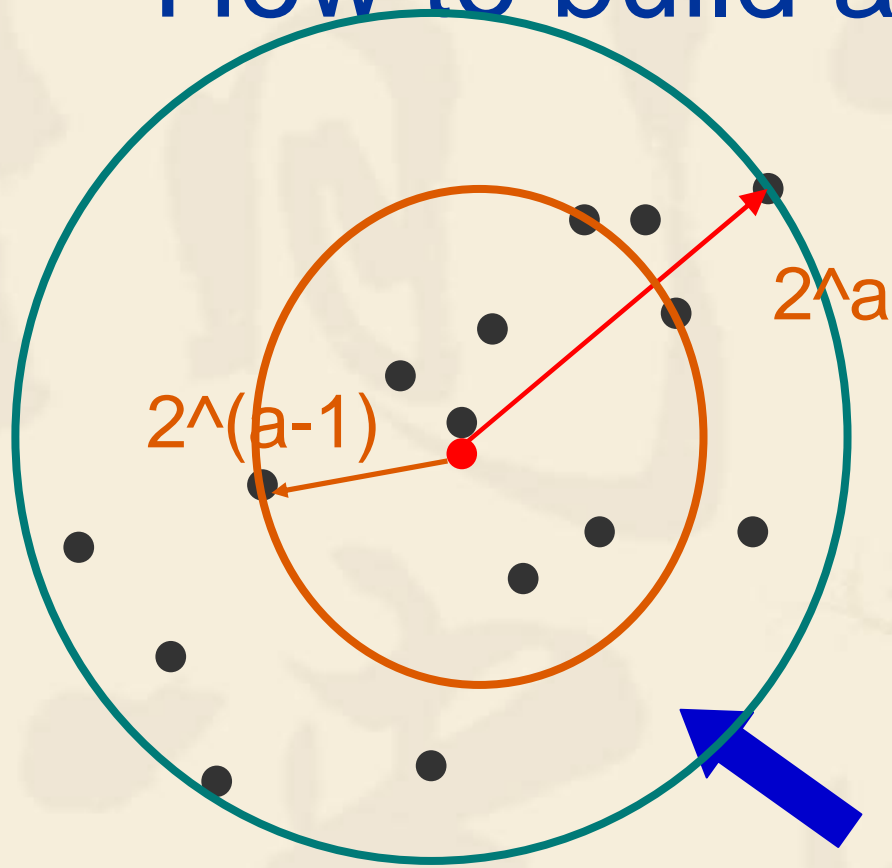


$C_a$  ● *Root*

$$a = \text{ceil}(\log_2(d_{\max}))$$

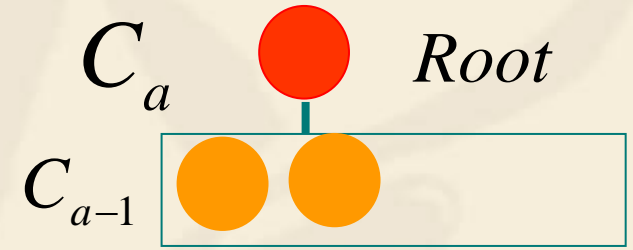
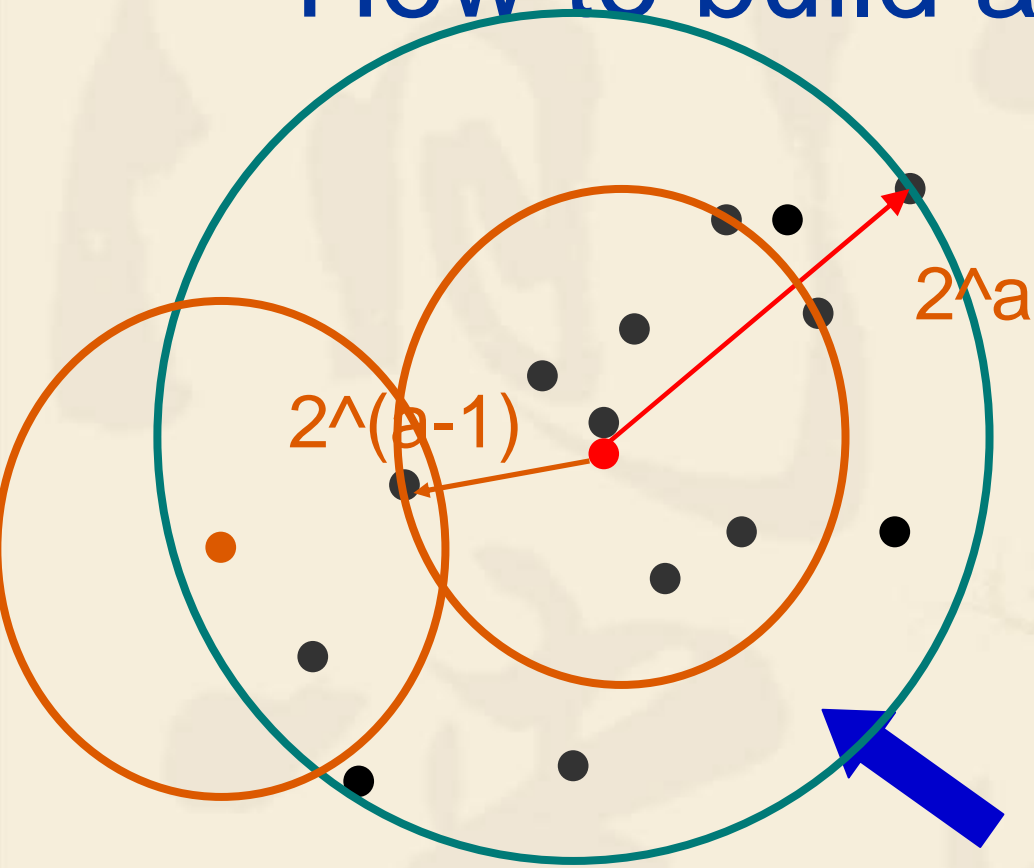
A set of points in  $\mathbb{R}^2$

# How to build a Cover tree



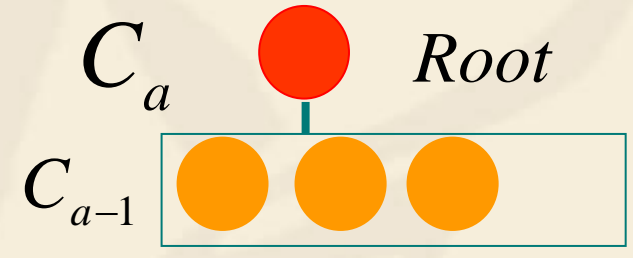
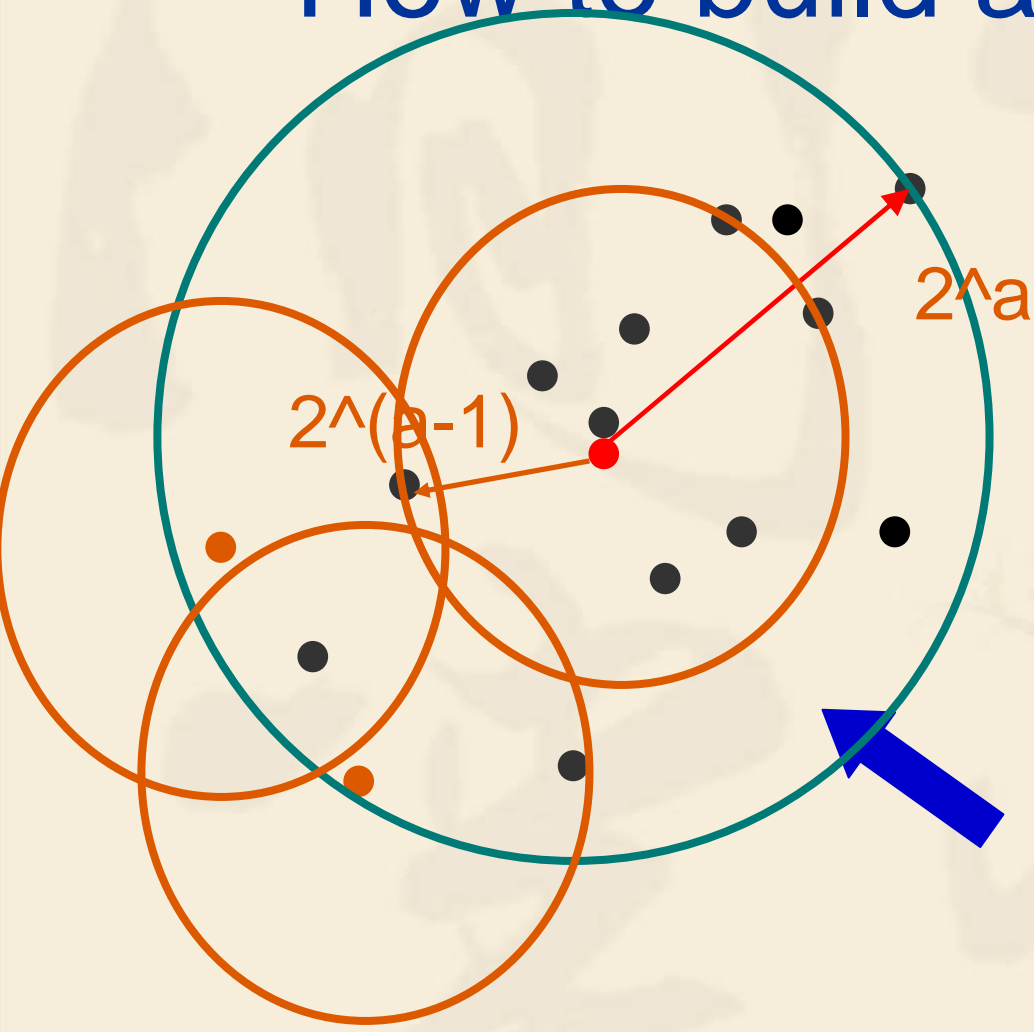
A set of points in  $\mathbb{R}^2$

# How to build a Cover tree



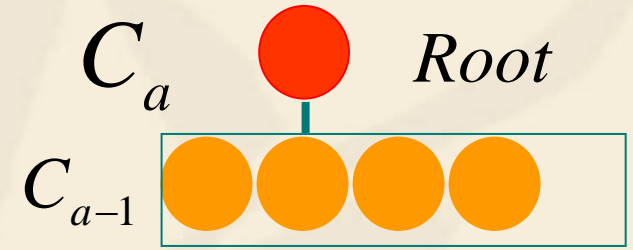
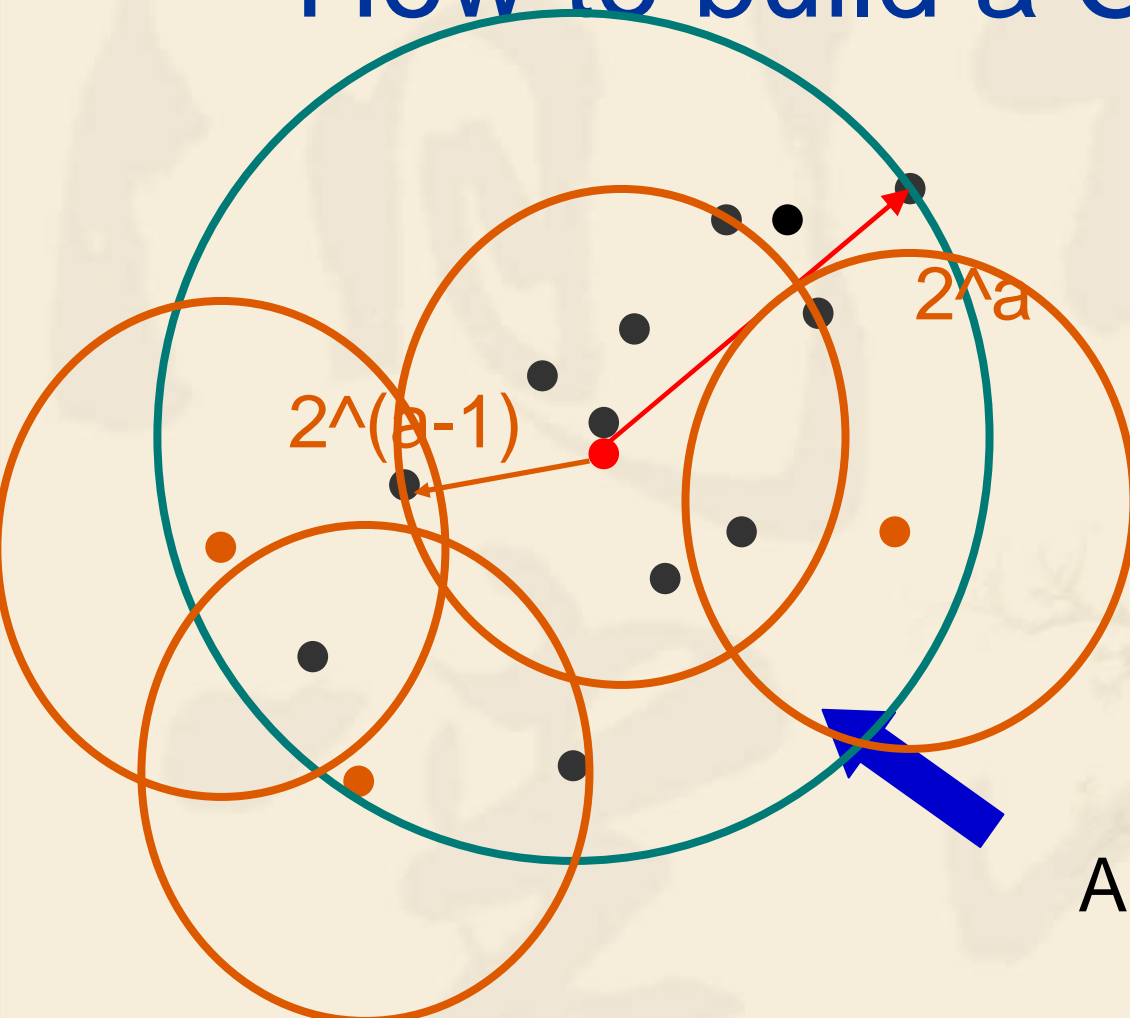
A set of points in  $\mathbb{R}^2$

# How to build a Cover tree



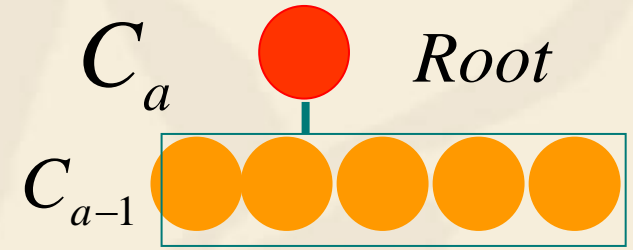
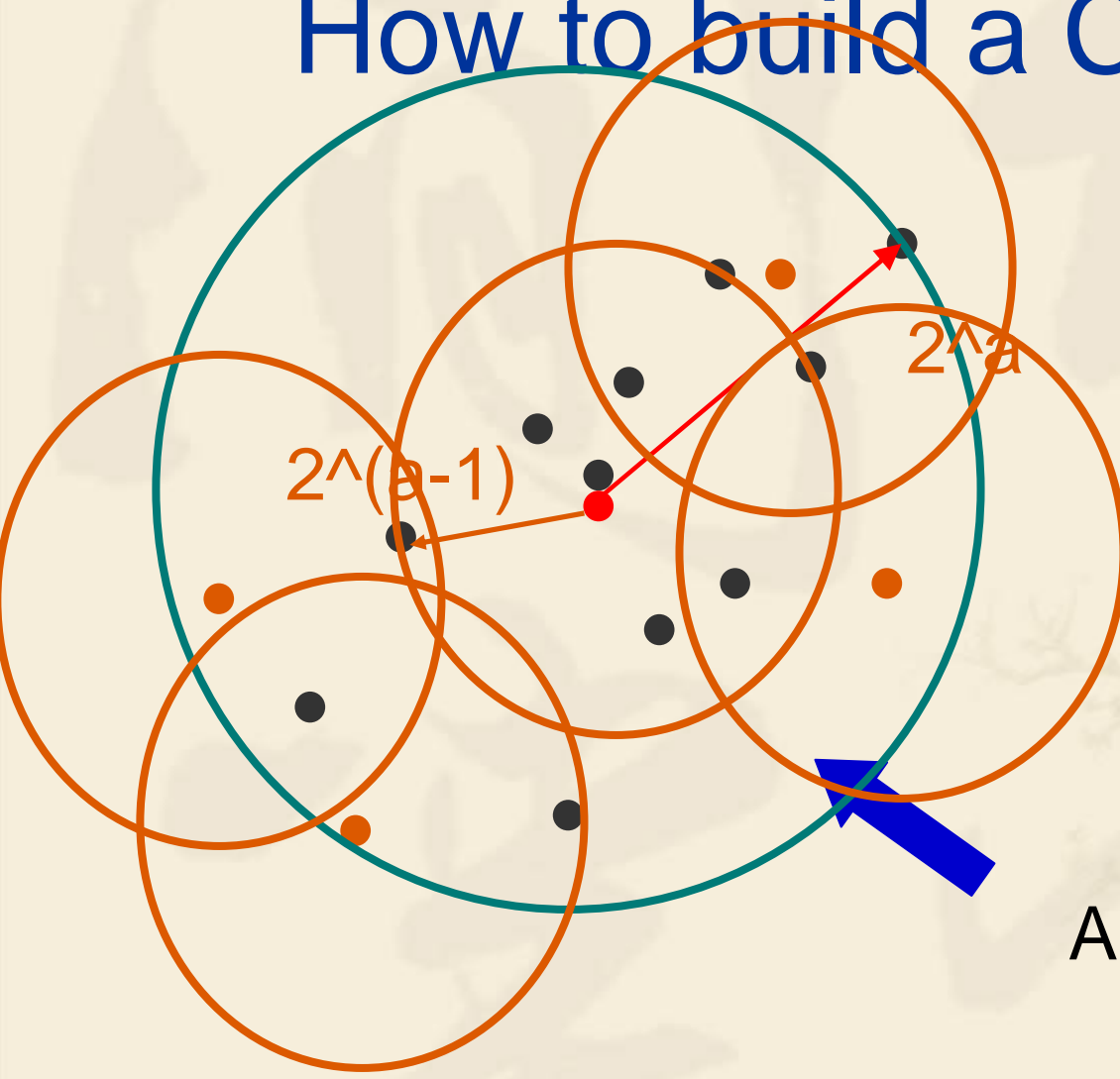
A set of points in  $\mathbb{R}^2$

# How to build a Cover tree



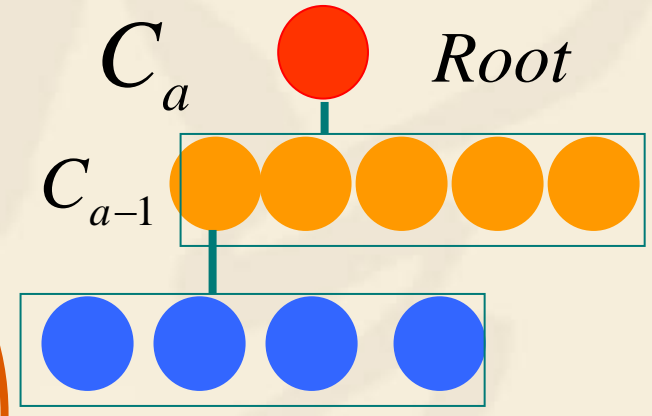
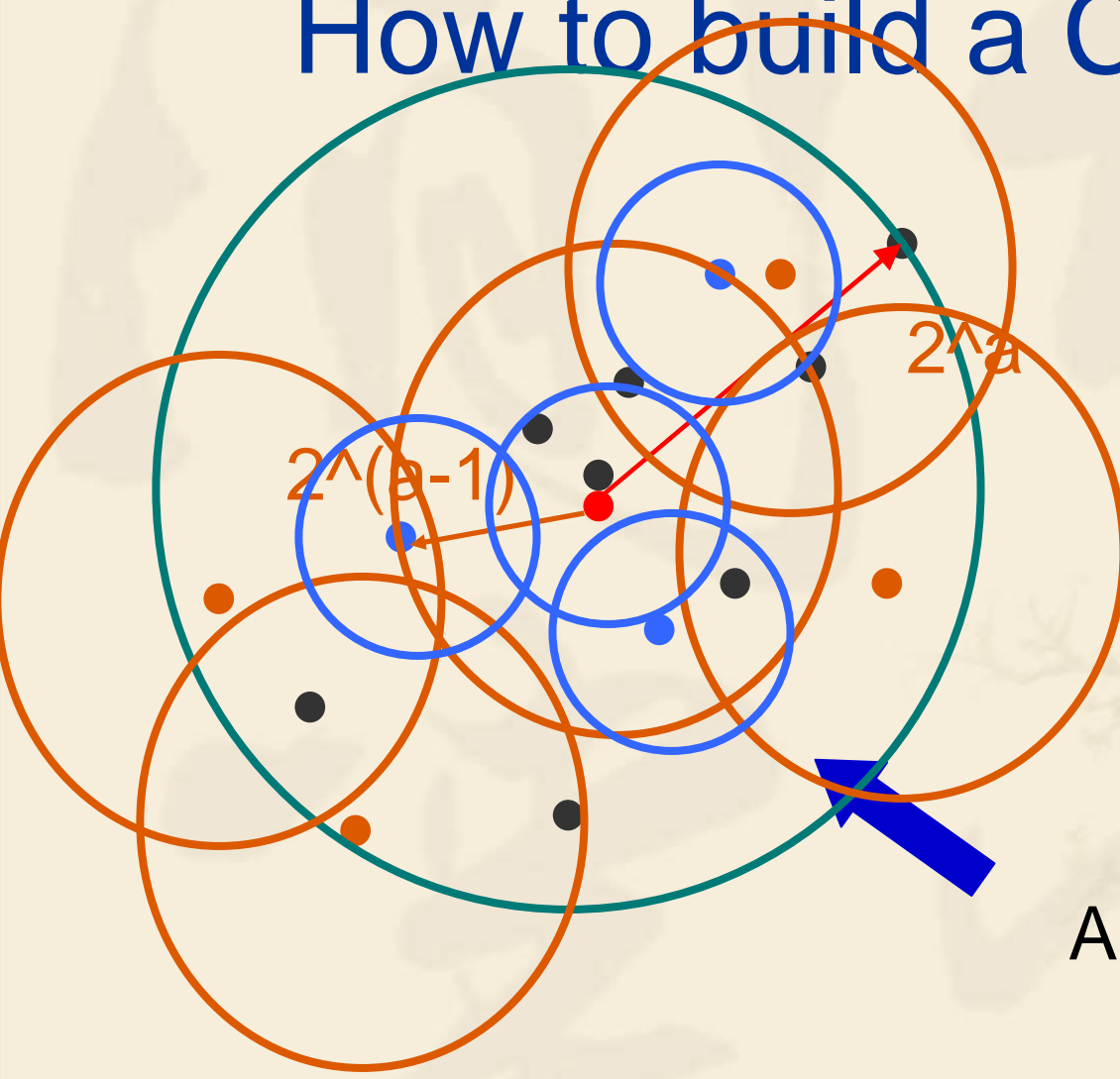
A set of points in  $\mathbb{R}^2$

# How to build a Cover tree



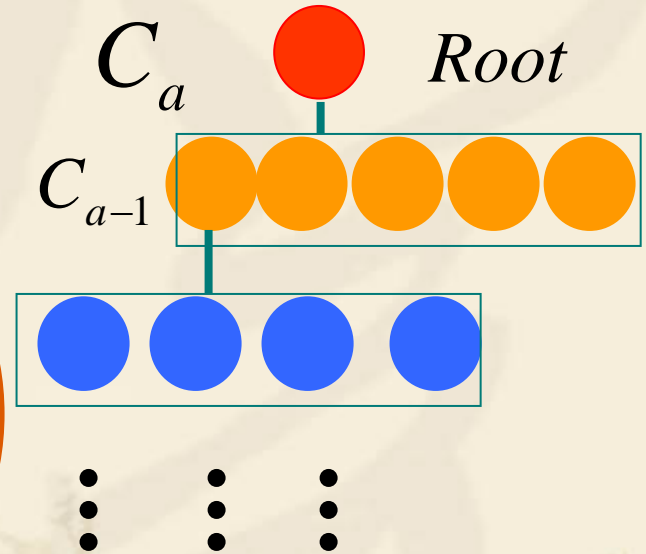
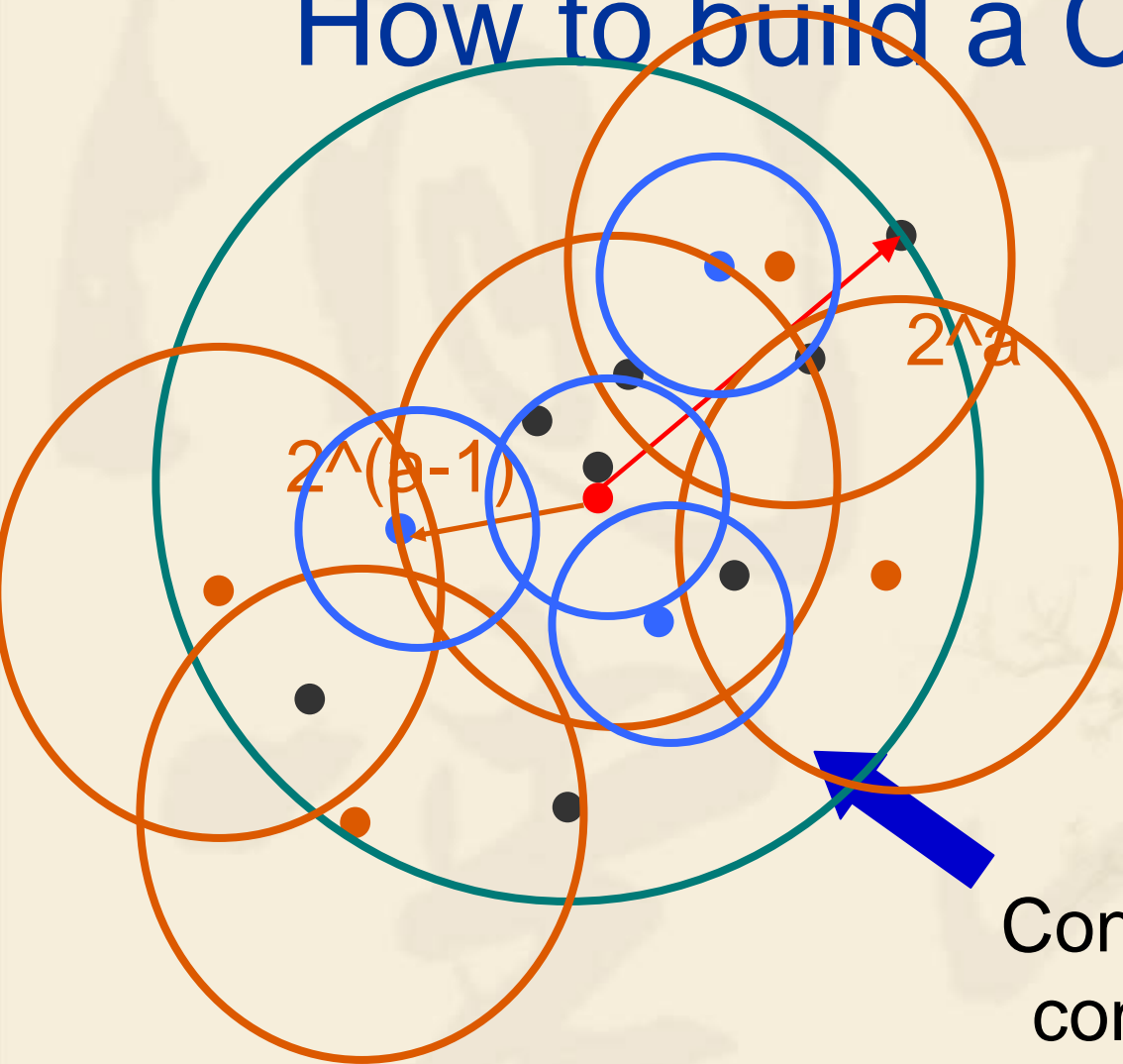
A set of points in  $\mathbb{R}^2$

# How to build a Cover tree



A set of points in  $\mathbb{R}^2$

# How to build a Cover tree



Continue till every circle contain only one point

# Query for a single point

set  $Q_\infty = C_\infty$

for  $i$  from  $\infty$  down to  $-\infty$

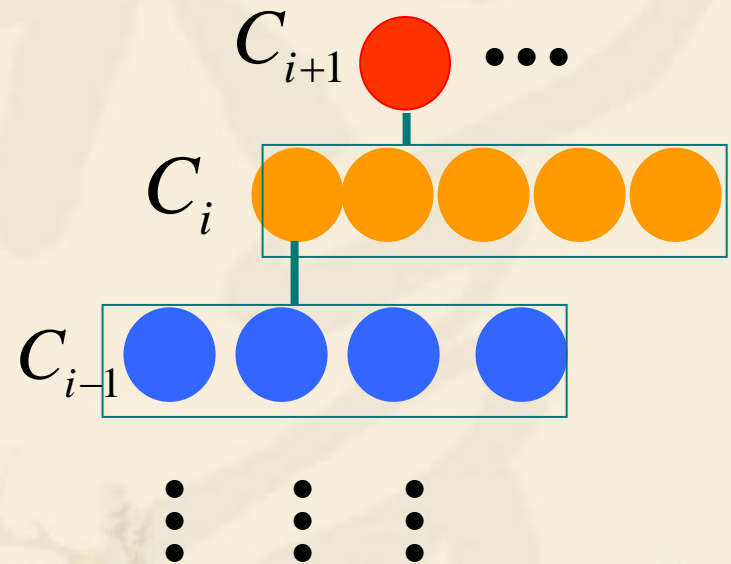
consider the set of children of  $Q_i$  :

$$set = \{Children(q) : q \in Q_i\}$$

form next cover set :

$$Q_{i-1} = \{q \in set : d(p, q) \leq d(p, set) + b\}$$

return  $\arg \min_{q \in Q_{-\infty}} d(p, q)$

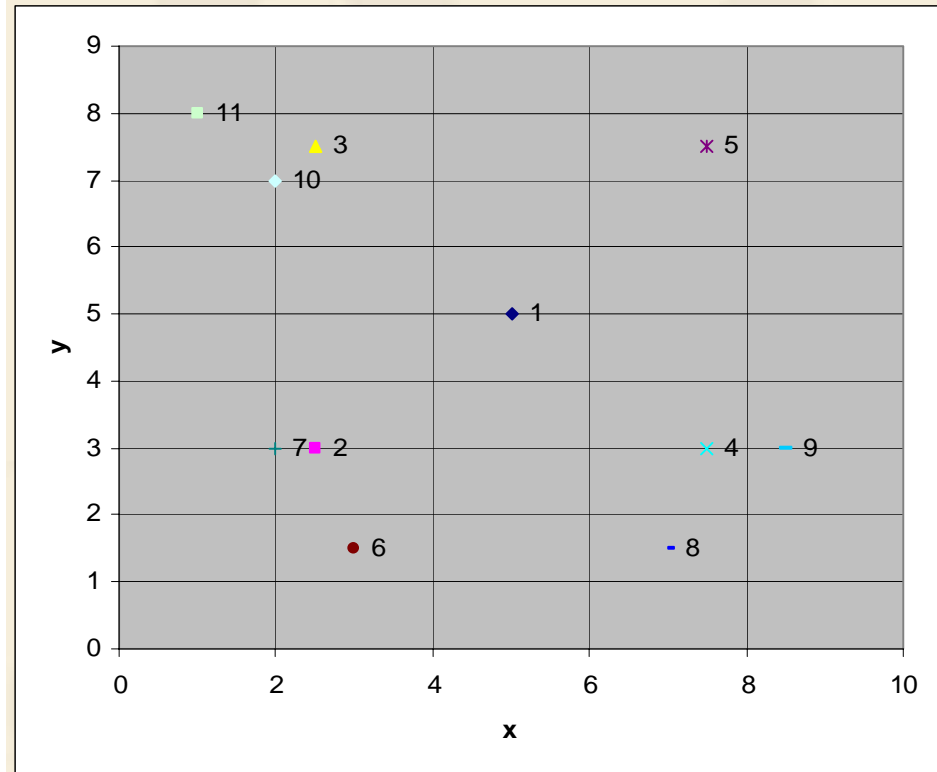
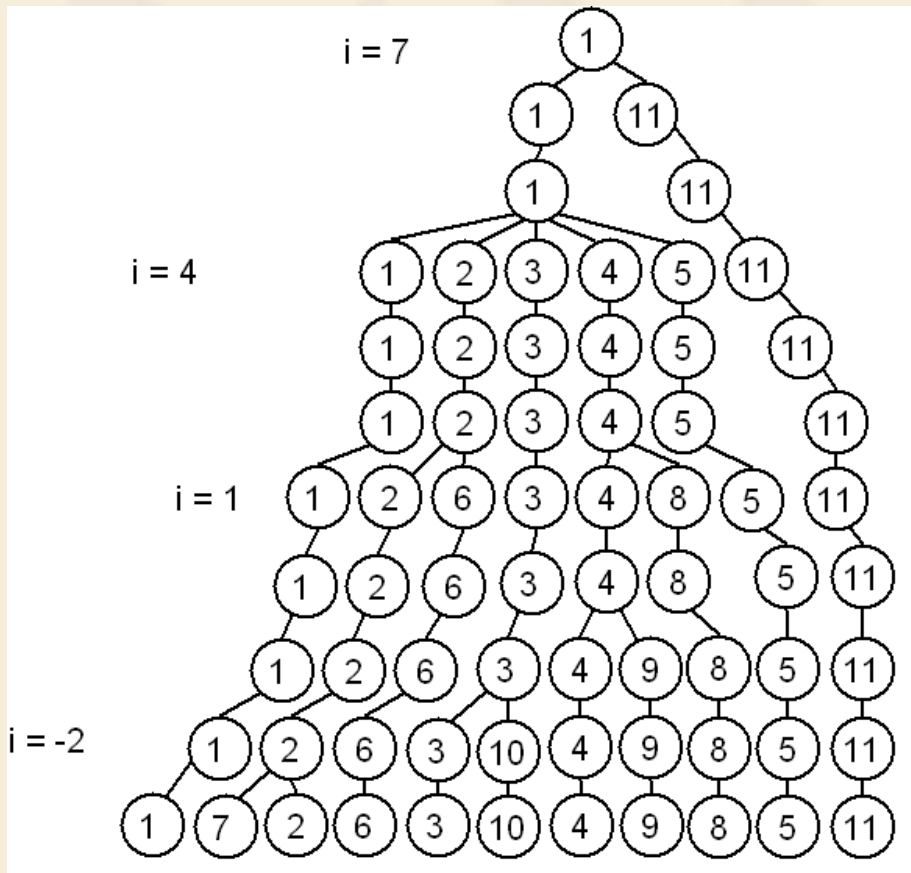


$b=?$

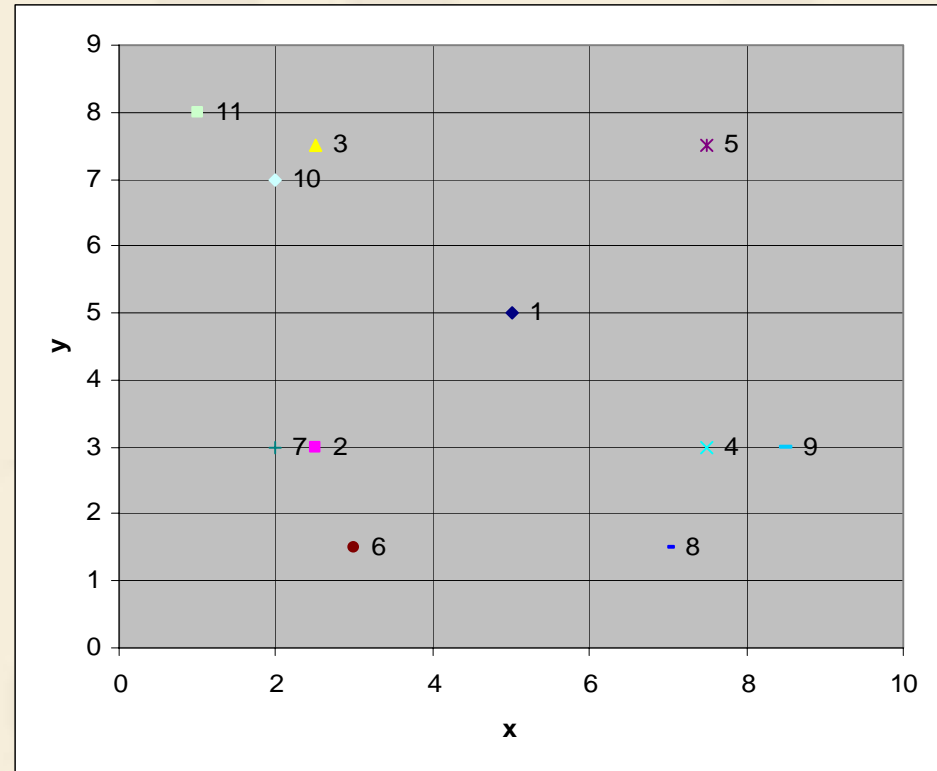
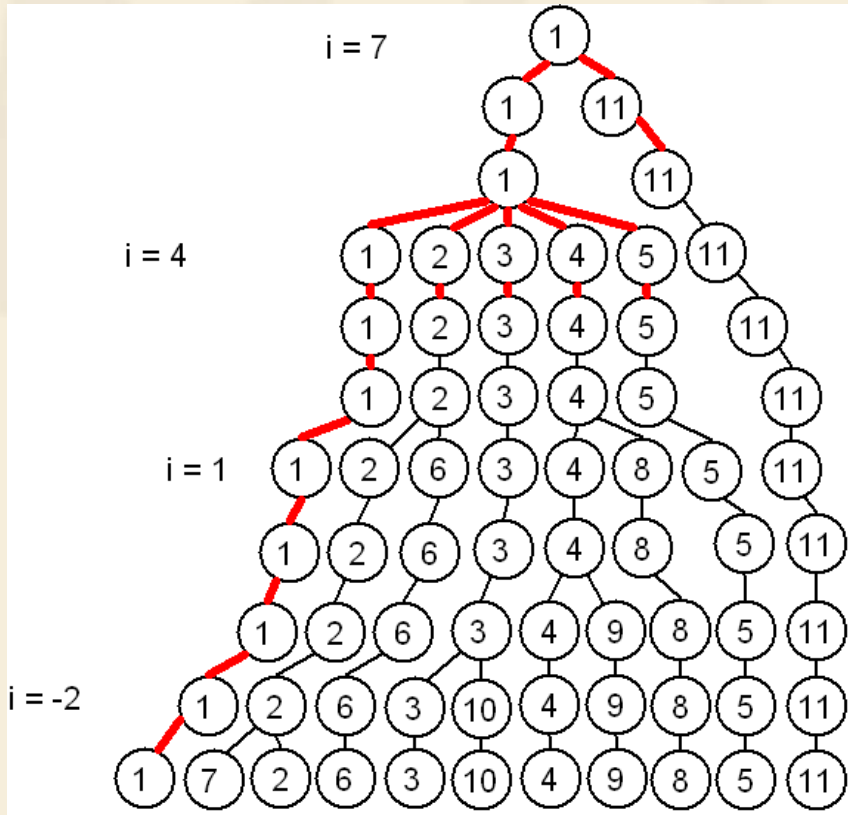
# Query for a single point

$$b = \sum_{-\infty}^{i-1} 2^k = 2^i$$

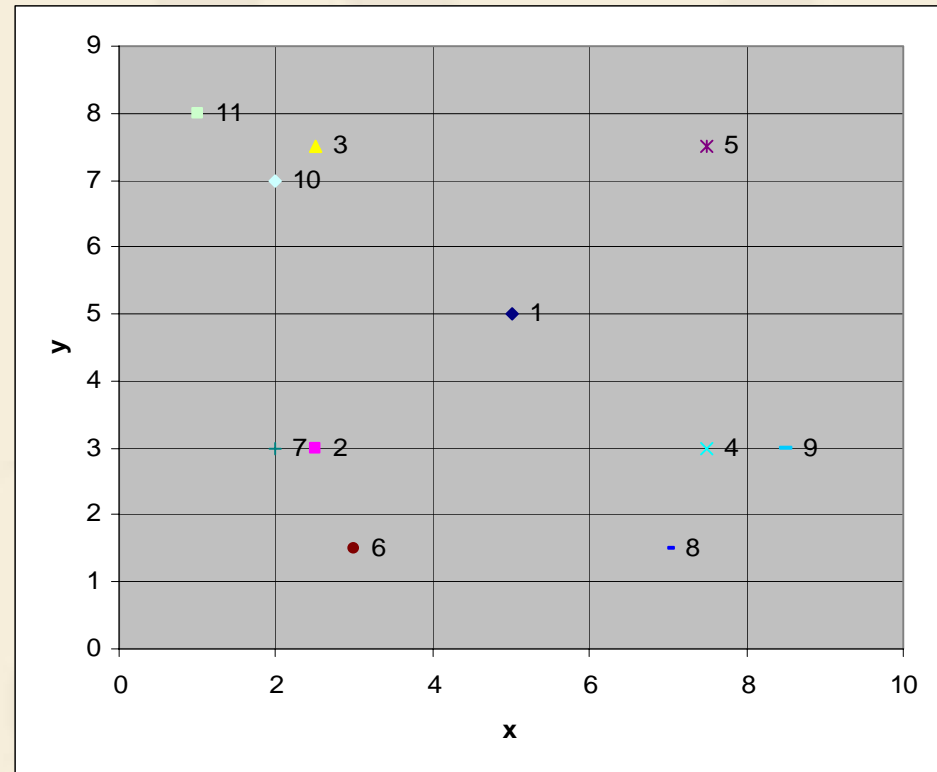
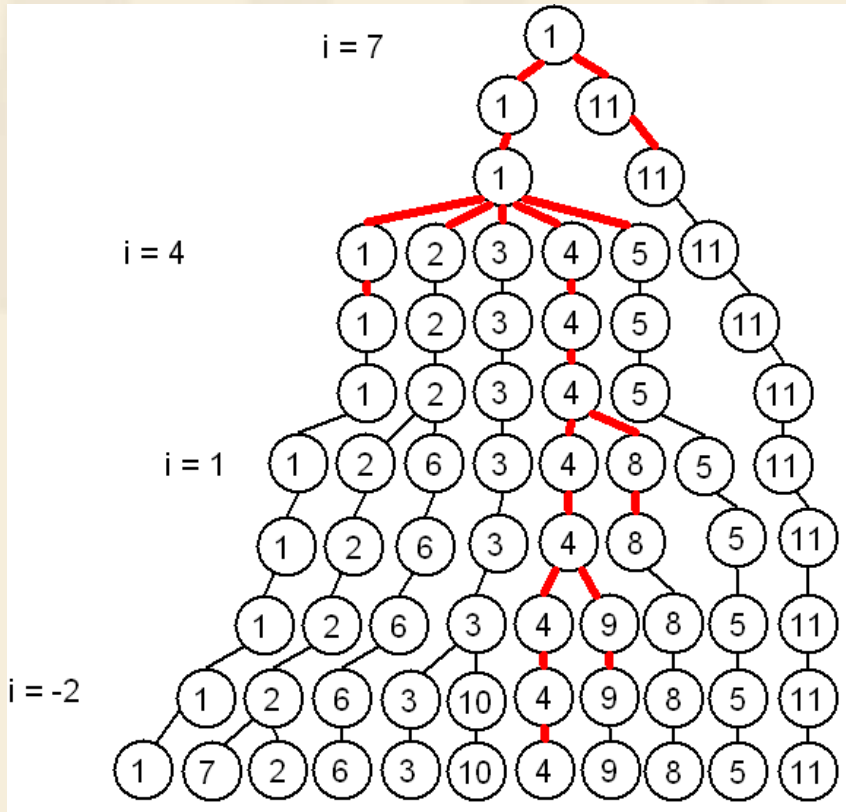
# Query Examples



# Query for Node 1



# Query for Node 4





# Query for many points

**Algorithm 5 Find-All-Nearest** (query cover tree  $p_j$ , cover set  $Q_i$ )

- (1) if  $i = -\infty$  then for each  $a \in L(p_j)$   
return  $\arg \min_{b \in Q_{-\infty}} d(a, b)$  as the nearest neighbor of  $a$ .
- (2) else
  - (a) if  $j < i$  then
    - (i) Set  $Q = \{ \text{Children}(q) : q \in Q_i \}$ .
    - (ii) Set  $Q_{i-1} = \{ q \in Q : d(p_j, q) \leq \min_{q \in Q} d(p_j, q) + 2^i + 2^{j+2} \}$
    - (iii) **Find-All-Nearest** ( $p_j, Q_{i-1}$ )
  - (b) else for each  $q_{j-1} \in \text{Children}(p_j)$   
**Find-All-Nearest** ( $q_{j-1}, Q_i$ )



$2^{j+1}$

# Optimized implement in C++

- ❖ Alignment memory (faster)
- ❖ exploit the four-stage fully pipelined floating-point adder (faster)
- ❖ Explicit cover tree (less space)
- ❖ Avoid declaring new variable (less space)
- ❖ Single-precision float constants (less space)
- ❖ Function pointers (instead of switch)
- ❖ Some differences from the paper

# Alignment memory

- ❖ pad data structure to make their sizes a multiple of a word, doubleword or quadword.
- ❖ Cpu can fetch the data (with multiple size of a word) from memory only one time, otherwise have to fetch twice.

# Alignment memory

```
struct {  
    char a[5]; \\ Smallest type size (1 byte * 5)  
    long k; \\ 4 bytes in this example  
    double x; \\ Largest type size (8 bytes)  
} baz;
```

A better approach is:

```
struct {  
    double x; \\ Largest type size (8 bytes)  
    long k; \\ 4 bytes in this example  
    char a[5]; \\ Smallest type size (1 byte * 5)  
    char pad[7]; \\ Make structure size a multiple of 8.  
} baz;
```

# Alignment memory

## ❖ Cover tree code:

```
if (p.index %8 > 0)
  for (int i = 8 - p.index %8; i > 0; i--)
    push(p,(float) 0.);
  float *new_p;
  posix_memalign((void **)&new_p, 16, p.index*sizeof(float));
  //new_p= (float*)malloc(p.index*sizeof(float));

  memcpy(new_p,p.elements,sizeof(float)*p.index);
```

## ❖ Result:

```
0.011000 1.000000 8.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.011000 1.000000 8.000000 0.000000 0.000000 0.000000 0.000000 0.000000

0.003000 2.500000 7.500000 0.000000 0.000000 0.000000 0.000000 0.000000
0.003000 2.500000 7.500000 0.000000 0.000000 0.000000 0.000000 0.000000
```

# exploit the four-stage fully pipelined floating-point adder

The original code:

```
double a[100], sum;
int i;
sum = 0.0f;
for (i = 0; i < 100; i++) {
    sum += a[i];
}
```

This version is faster, because the code implements four separate dependency chains instead of just one, which keeps the pipeline full. The `/fp:fast` compiler switch in Visual Studio 2005 can help do this sort of optimization automatically.

```
double a[100], sum1, sum2, sum3, sum4, sum;
int i;
sum1 = 0.0;
sum2 = 0.0;
sum3 = 0.0;
sum4 = 0.0;
for (i = 0; i < 100; i + 4) {
    sum1 += a[i];
    sum2 += a[i+1];
    sum3 += a[i+2];
    sum4 += a[i+3];
}
sum = (sum4 + sum3) + (sum1 + sum2);
```

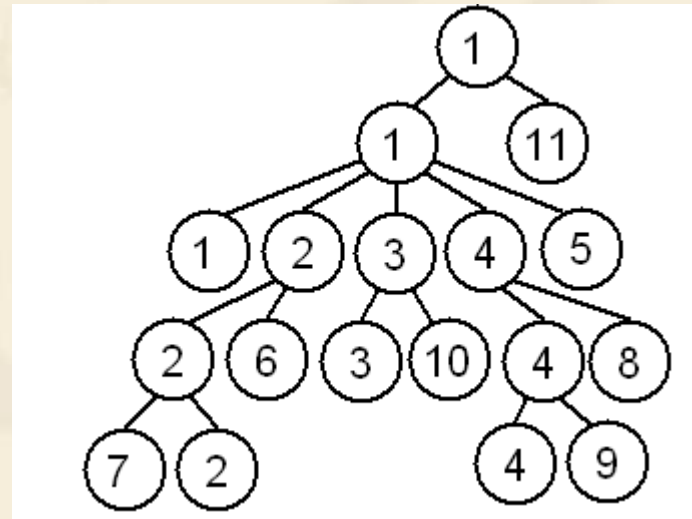
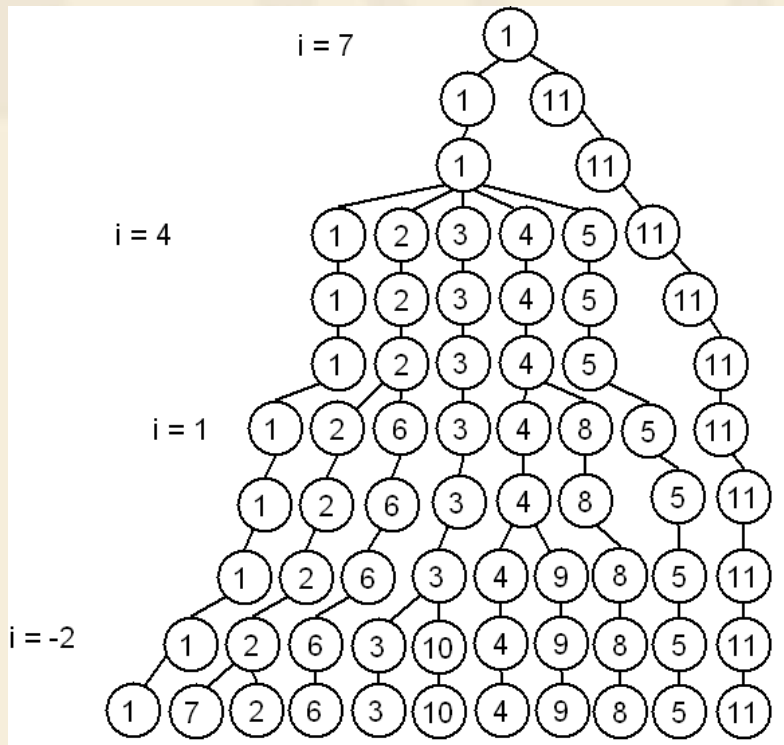
# exploit the four-stage fully pipelined floating-point adder

## ❖ Cover tree code:

```
//Assumption: points are a multiples of 8 long
float distance(vector p1, vector p2, float upper_bound)
{
    float sum = 0.;
    float *end = p1 + point_len;
    upper_bound *= upper_bound;
    for (float *batch_end = p1 + batch; batch_end <= end; batch_end += batch)
        {
            for (; p1 != batch_end; p1+=2, p2+=2)
                {
                    float d1 = *p1 - *p2;
                    float d2 = *(p1+1) - *(p2+1);
                    d1 *= d1;
                    d2 *= d2;
                    sum = sum + d1 + d2;
                }
            if (sum > upper_bound)
                return sqrt(sum);
        }
    for (; p1 != end; p1+=2, p2+=2)
        {
            float d1 = *p1 - *p2;
            float d2 = *(p1+1) - *(p2+1);
            d1 *= d1;
            d2 *= d2;
            sum = sum + d1 + d2;
        }
    return sqrt(sum);
}
```

# Explicit cover tree

- ❖ Theory is based on an implicit implementation, but tree is built with a condensed explicit implementation to preserve  $O(n)$  space bound



# Avoid declaring new variable

## ❖ Use the same variable

☞ point\_set here is the same one

```
template<class P>
void split(v_array<ds_node<P> >& point_set, v_array<ds_node<P> >& far_set, int max_scale)
{
    unsigned int new_index = 0;
    float fmax = dist_of_scale(max_scale);
    printf("fmax:%F",fmax);
    for (int i = 0; i < point_set.index; i++){
        if (point_set[i].dist.last() <= fmax) {
            point_set[new_index++] = point_set[i];
        }
        else
            push(far_set,point_set[i]);
    }
    point_set.index=new_index;
}
```

# Some differences from the paper

## ❖ Paper:

- ❧  $2^i$

- ❧ Scale  $i$  reduces (from top to bottom)

## ❖ Code:

- ❧  $1.3^i$

- ❧ Scale  $i$  increases (0~100)

- $n.\text{scale} = \text{top\_scale} - \text{cur\_scale};$

# Complexity

	Cover Tree	Nav. Net	[KR02]
Construction Space	$O(n)$	$c^{O(1)}n$	$c^{O(1)}n \ln n$
Construction Time	$O(c^6 n \ln n)$	$c^{O(1)}n \ln n$	$c^{O(1)}n \ln n$
Insertion/Removal	$O(c^6 \ln n)$	$c^{O(1)} \ln n$	$c^{O(1)} \ln n$
Query	$O(c^{12} \ln n)$	$c^{O(1)} \ln n$	$c^{O(1)} \ln n$
Batch Query	$O(c^{16}n)$	$c^{O(1)}n \ln n$	$c^{O(1)}n \ln n$

## ❖ Expansion constant $C$

smallest value  $c \geq 2$  such that  $|B_S(p, 2r)| \leq c|B_S(p, r)|$  for every  $p \in X$

# Advantages and disadvantages

## ❖ Advantage:

- ❧ Query many points at one time are faster (instead of query each by each)
- ❧ Less required space

## ❖ Disadvantage:

- ❧ Building cover tree is still time consuming.
- ❧ Expansion constant  $C$  (which is related to bound) is hard to be accurately determined
- ❧ Recursion might be slower than iteration, and may overflow the recursion stack in both C++ or python.

# Further work

- ❖ Provide python interface (swig, pyrex)
- ❖ Apply it into other classification methods (k-means, crf,...)to speed them up.



❖ Thanks for coming!

