Implementation:
Detri2

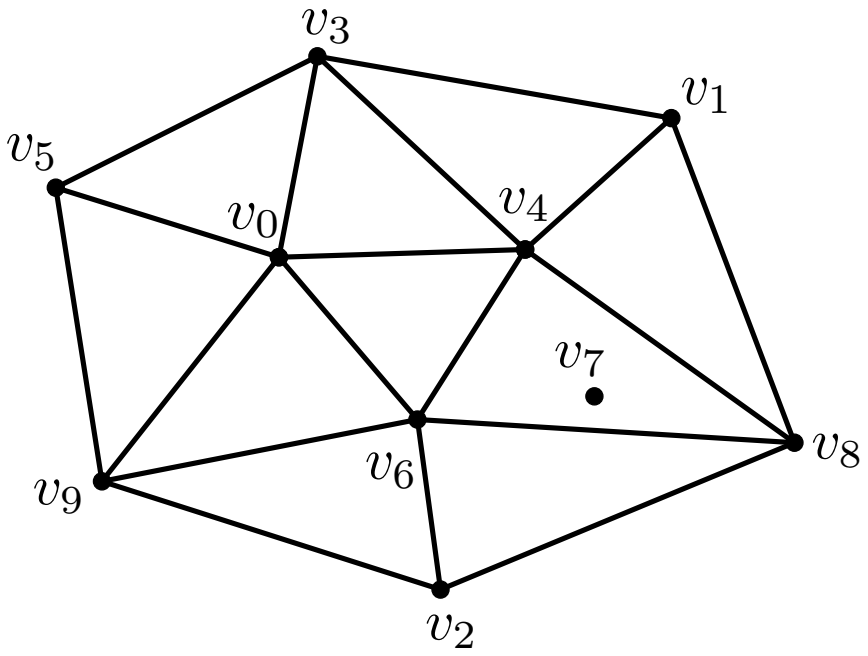
Mesh Data Structure

- Mesh data structures are needed to implement mesh generation algorithms.

- A mesh data structure allows iterating through the entities of a mesh, performing queries on incidence (like element-vertex), adjacency (element-element), and classification (in particular, boundary classification) of entities, and modifying the topology of the mesh.

Representation

- A 2d triangulation is a 2d simplical complex which consists of points, edges, and triangles.
- Detri2 only stores the points and triangles



Triangle based DS: for triangle meshes

```
class Point{
  float x;
  float y;
  float z;
}
```

```
class Triangle{
  Triangle t1, t2, t3;
  Vertex v1, v2, v3;
}
 class Vertex{
  Triangle  root;
  Point p;
}
```

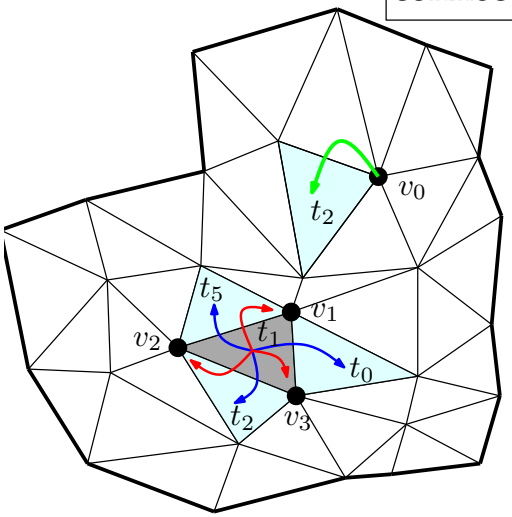
connectivity

for each triangle, store:

- 3 references to neighboring faces
- 3 references to incident vertices

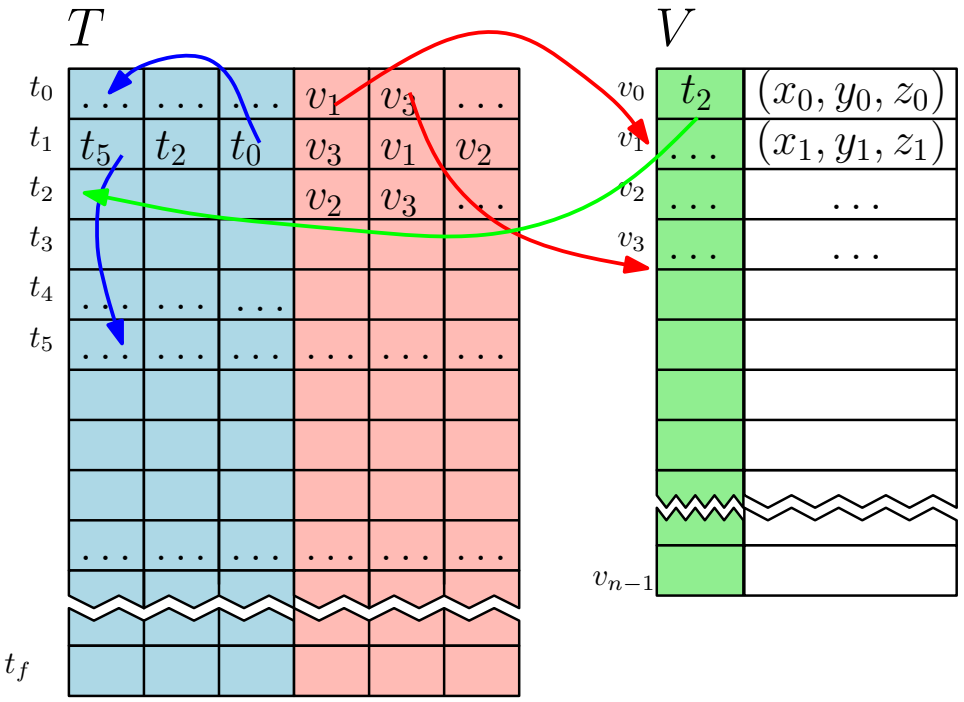
for each vertex, store:

- 1 reference to an incident face



$$(3+3) \times f + n = 6 \times 2n + n = 13n$$

Size (number of references)

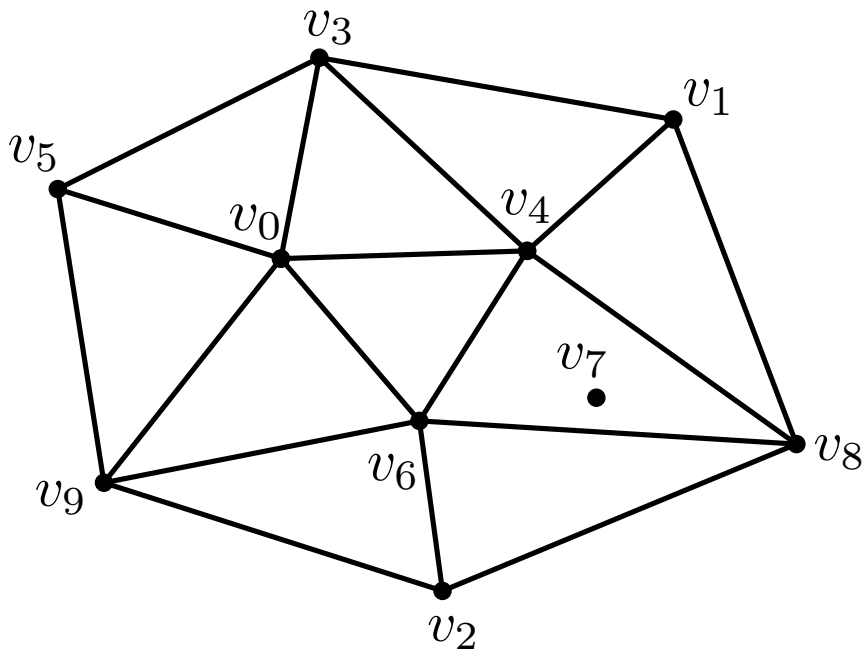


Detri2's data structure

- Vertex
- Triangle
- TriEdge
- Arraypool
- Triangulation

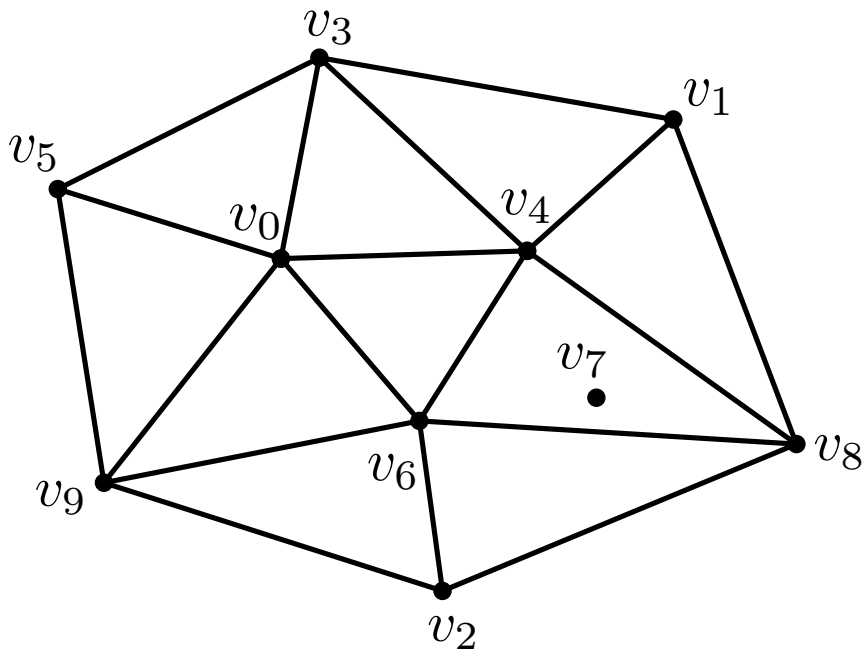
Vertex

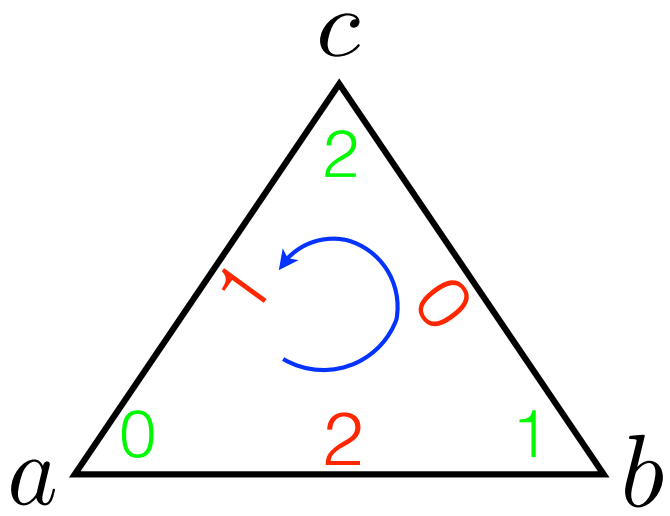
```
class Vertex
{
 public:
  REAL  crd[3]; // x, y, w (weight)
  int   idx;    // Its index (0 or 1-based).
  int   tag;    // Boundary marker.
  int   flags;  // Vertex type, infect, etc.
  REAL  val;    // A scalar (nodel) value.
  Triangle* adj;// Adjacent (or dual) triangle.
  Triangle* on; // Segment containing this vertex.

  void init() {
    crd[0] = crd[1] = crd[2] = 0.0;
    idx = 0; tag = 0; flags = 0;
    val = 0.0;
    adj = on = (Triangle *) 0;
  }

  Vertex() {init();}

  bool is_infected();
  void set_infect();
  void clear_infect();
  int  get_vrttype();
  void set_vrttype(int);
  int  get_ver();
  void set_ver(int val);
  TriEdge get_triorg();
};
```



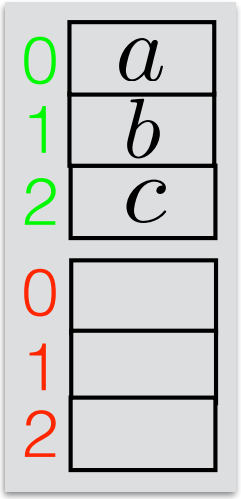
Triangle

```
class Triangle
{
 public:
  Vertex*   vrt[3];
  Triangle* nei[3];
  int       flags;  // Encode, hullflag, infect,
  int       tag;    // Boundary marker.
  int       idx;
  Vertex*   dual_vrt; // the dual vertex

  void init() {
    nei[0] = nei[1] = nei[2] = (Triangle *) 0;
    vrt[0] = vrt[1] = vrt[2] = (Vertex *) 0;
    flags = 0; tag = 0; idx = 0;
    dual_vrt = NULL;
  }

  Triangle() { init();}

  int  get_ver(int i);
  void set_ver(int i, int val);
  bool is_hulltri();
  void set_hullflag();
  void clear_hullflag();
  void set_infect();
  bool is_infected();
  void clear_infect();
  void set_exterior();
  bool is_exterior();
  void clear_exterior();
};
```



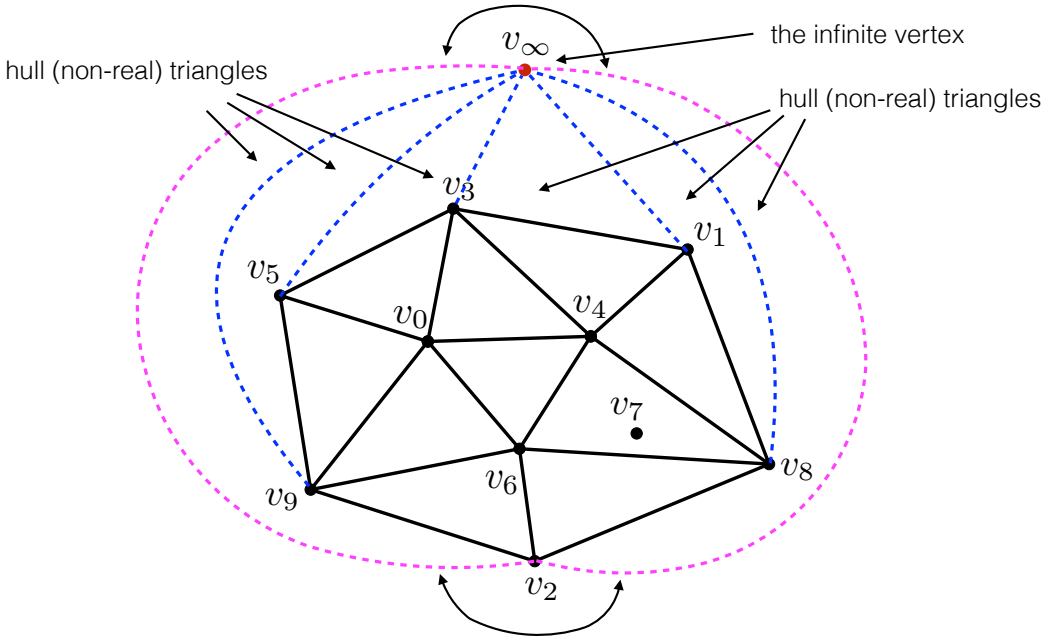



```
class Triangle
{
 public:
  Vertex*   vrt[3];
  Triangle* nei[3];
  int       flags;  // Encode,
  int       tag;    // Boundary
  int       idx;
  Vertex*   dual_vrt; // the du
```



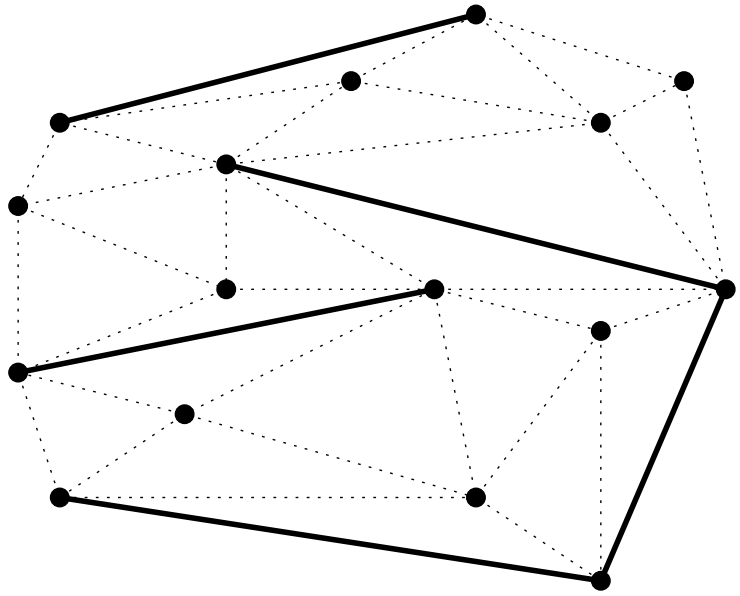
Infinite vertex and hull triangles

- it is convenient to think that the complement of the convex hull in the plane is also a part of this triangulation.
- Detri2 adds to any triangulation a fictitious vertex, called infinite vertex, with the convention that every edge on the convex hull forms an infinite triangle with this vertex.



Constrained edges

- The edges of a triangulation are only implicitly represented through the adjacency relations of triangles.
- A constrained edge is a special one which needs to be classified.
- The status (constrained or not constrained) of an edge is stored in both incident triangles by using only one bit in each of its memory.



```
// Triangle(t)->segment(s)
bool is_segment();
void set_segment();
void clear_segment();
```

Ordered Triangles

Let $t_{\mathbf{abc}}$ be a triangle with vertices $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$. There are three even permutations of the three vertices of a triangle, which are $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, $\{\mathbf{b}, \mathbf{c}, \mathbf{a}\}$, $\{\mathbf{c}, \mathbf{a}, \mathbf{b}\}$, respectively. We call each permutation an *ordered triangle*, and denoted them by $t_{\mathbf{a},\mathbf{b},\mathbf{c}}$, $t_{\mathbf{b},\mathbf{c},\mathbf{a}}$, and $t_{\mathbf{c},\mathbf{a},\mathbf{b}}$.

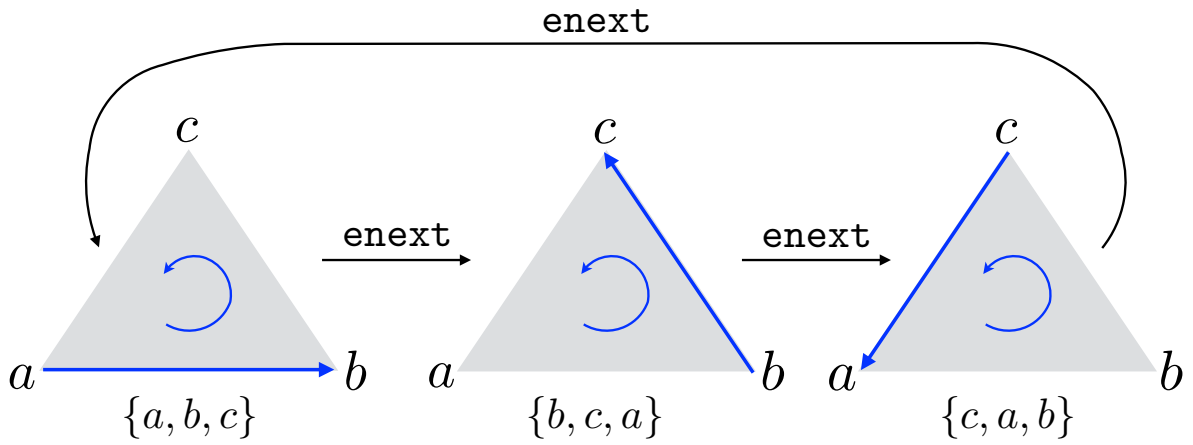
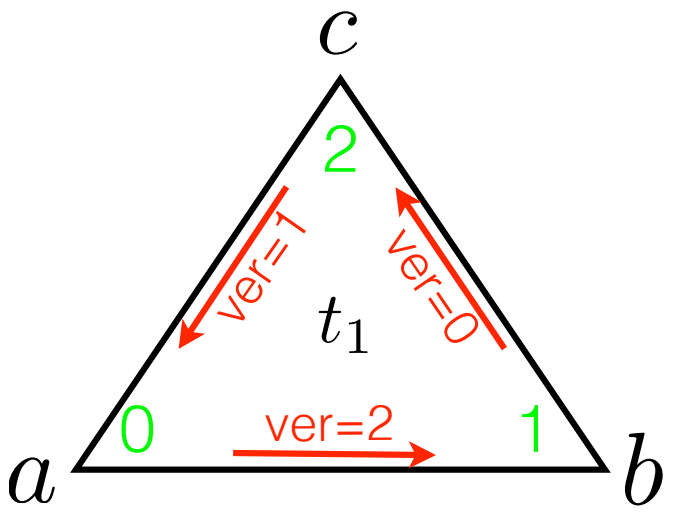


Figure 6.1: The three ordered triangles and the `enext` operation on them.

TriEdge

```
class TriEdge
{
 public:
  Triangle* tri;
  int       ver; // = 0,1,2

  TriEdge() {tri = (Triangle *) 0; ver = 0;}
  TriEdge(Triangle* _t, int _v) {tri = _t; ver = _v;}

  // Primitives
  Vertex* org();
  Vertex* dest();
  Vertex* apex();
  void set_vertices(Vertex *pa, Vertex *pb, Vertex *pc);

  TriEdge enext();
  TriEdge enext2();
  TriEdge sym();
  void enextself();
  void enext2self();
  void symself();
  void bond2(const TriEdge &Enei); // Both direction
  void dissolve();

  // Triangle(t)->segment(s)
  bool is_segment();
  void set_segment();
  void clear_segment();

  void set_edge_infect();
  bool is_edge_infected();
  void clear_edge_infect();
};
```



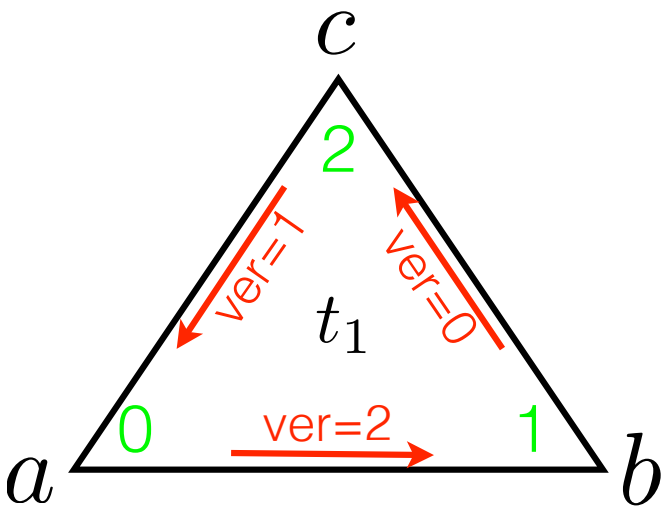
Primitives

```
class TriEdge
{
 public:
  Triangle* tri;
  int       ver; // = 0,1,2

  TriEdge() {tri = (Triangle *) 0; ver = 0;}
  TriEdge(Triangle* _t, int _v) {tri = _t; ver = _v;}

  // Primitives
  Vertex* org();
  Vertex* dest();
  Vertex* apex();
  void set_vertices(Vertex *pa, Vertex *pb, Vertex *pc);

  TriEdge enext();
  TriEdge enext2();
  TriEdge sym();
  void enextself();
  void enext2self();
  void symself();
  void bond2(const TriEdge &Enei); // Both direction
  void dissolve();

  // Triangle(t)->segment(s)
  bool is_segment();
  void set_segment();
  void clear_segment();

  void set_edge_infect();
  bool is_edge_infected();
  void clear_edge_infect();
};
```

- Navigation and updating triangulation are through a set of primitives (defined by functions) operating on the triangles and vertices.
- Three basic navigation primitive functions are highlighted.

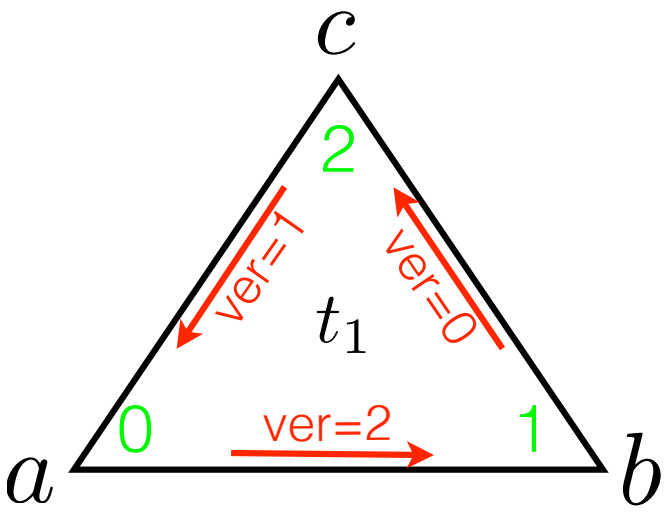
TriEdge::org()

- Returns the starting vertex of this TriEdge



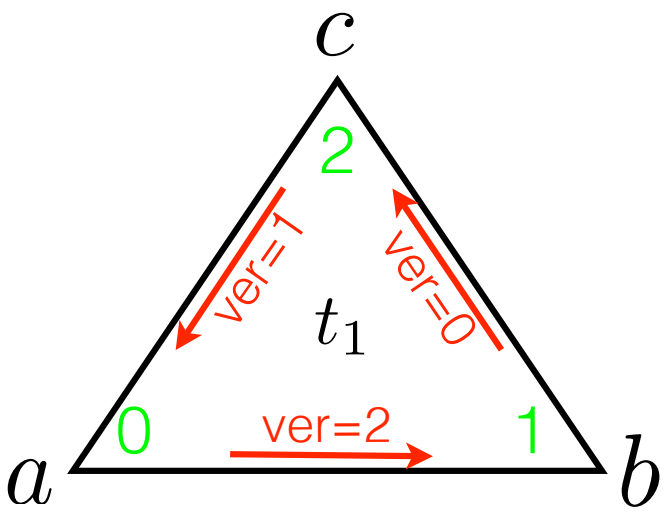
TriEdge::enext()

- Returns the next TriEdge of this one.



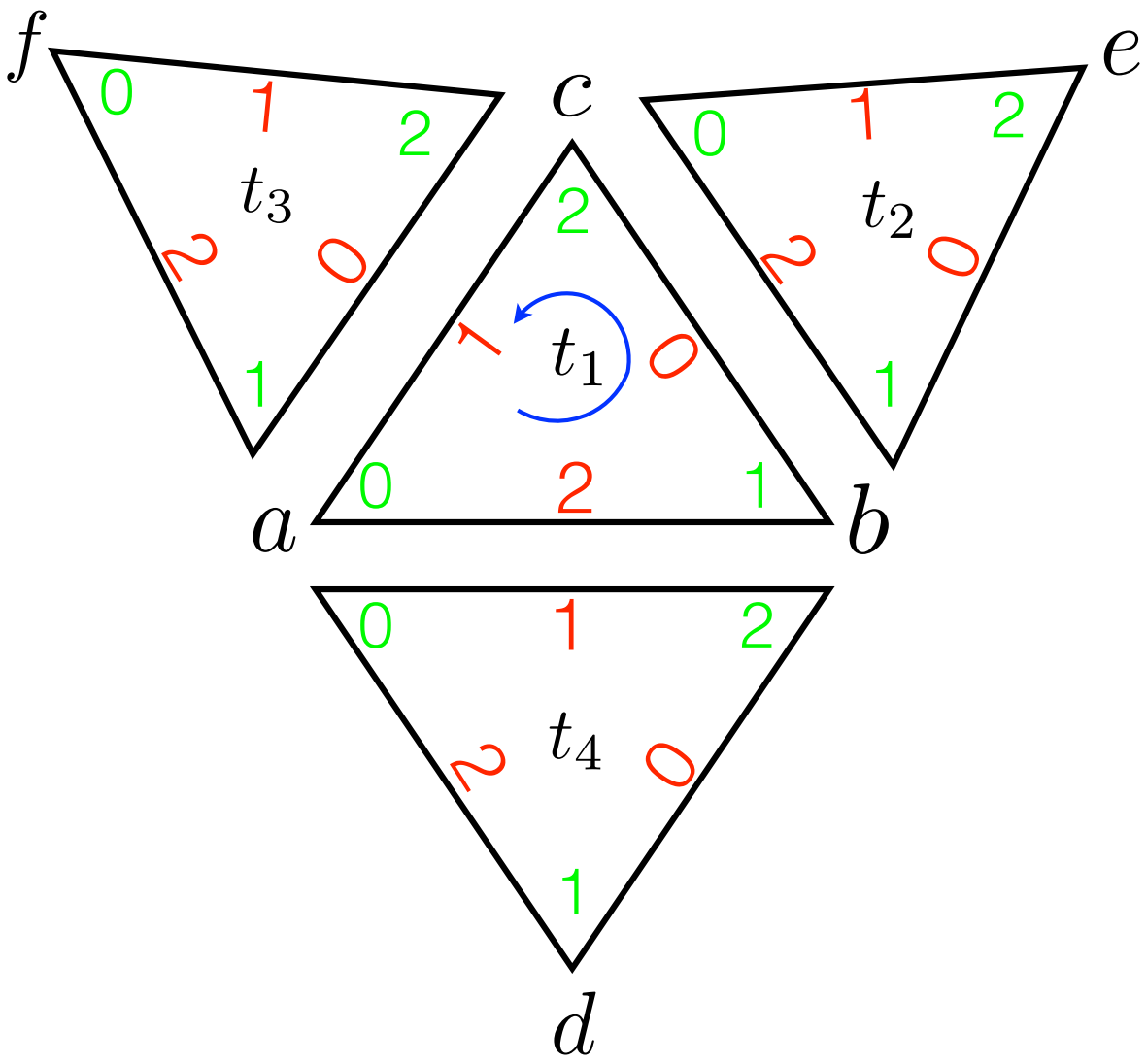
- We can use the three basic primitive functions to traverse and visit any simplex of the triangulation. For examples,

$$
\begin{aligned}
a &:= (t_1, 0).\texttt{org}(); \\
b &:= (t_1, 0).\texttt{org}(\texttt{enext}()); \\
c &:= (t_1, 0).\texttt{org}(\texttt{enext}(\texttt{enext}()));
\end{aligned}
$$



TriEdge::esym()

- Returns the adjacent TriEdge (which belongs to another triangle) of this one.



- The org() and enext() primitives can be implemented by using one lookup table,

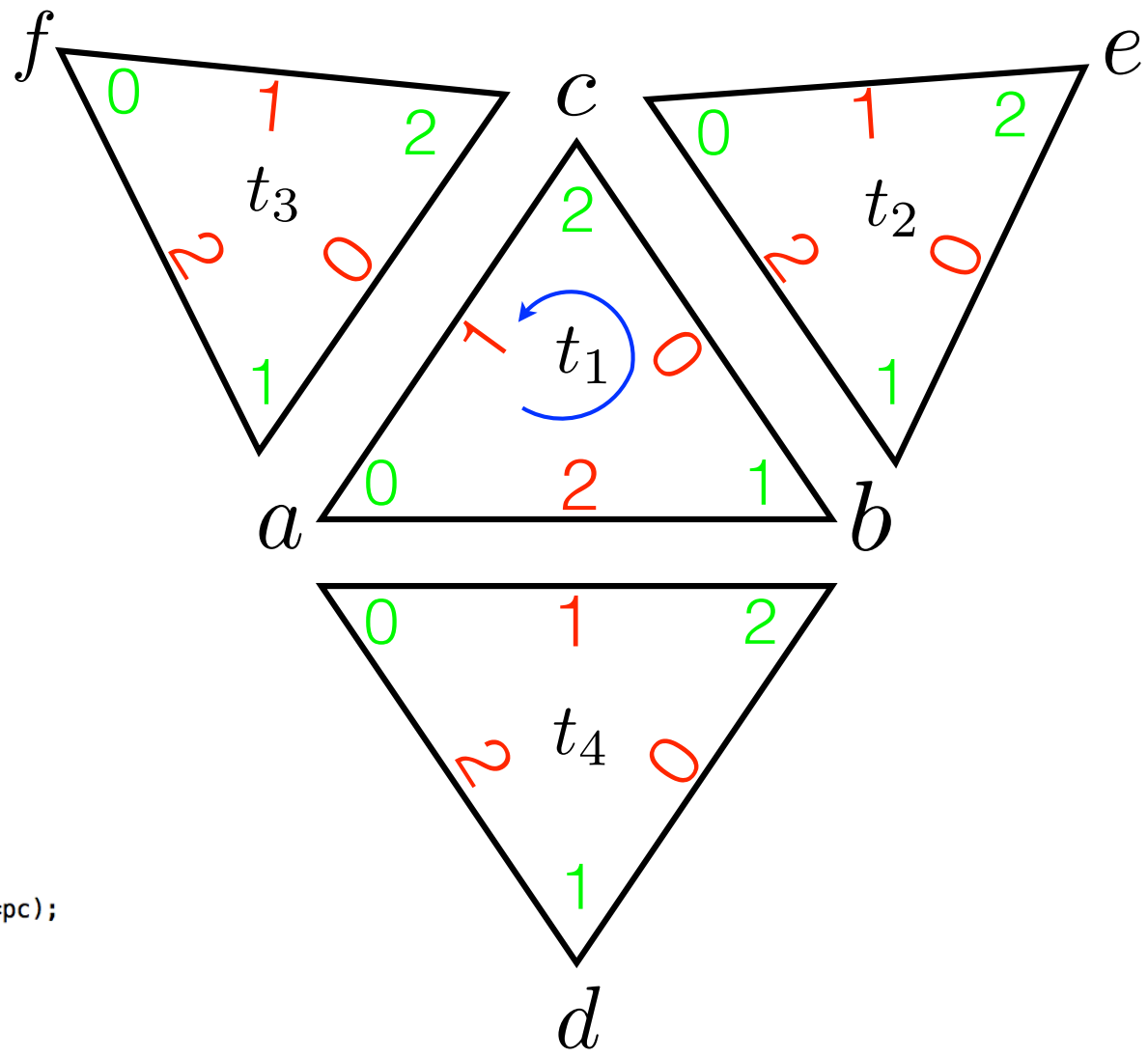
```
105  // Initialize tables
106  static int _vo[3] = {1, 2, 0};

110  Vertex* TriEdge::org() {return tri->vrt[_vo[ver]];}

122  TriEdge TriEdge::enext() {return TriEdge(tri, _vo[ver]);}
```

- The esym() operator will visit the neighbour information stored in a Triangle. Since Detri2 saved both the pointers and the edge indices of its neighbour triangles, the esym() operator can simply extract these information and return the TriEdge.

```
127  TriEdge TriEdge::sym() {return TriEdge(tri->nei[ver], tri->get_ver(ver));}
```

- The mesh update (change mesh connectivity) primitives

```
class TriEdge
{
public:
  Triang*       tri;
  unsigned char ver; // = 0,1,2

  TriEdge() {tri = NULL; ver = 0;}
  TriEdge(Triang* _t, int _v) {tri = _t; ver = _v;}

  bool is_connected();
  void connect(const TriEdge& te);

  TriEdge enext();
  TriEdge eprev();
  TriEdge esym();
  TriEdge enext_esym();
  TriEdge eprev_esym(); // ccw rotate
  TriEdge esym_enext(); //  cw rotate
  TriEdge esym_eprev();

  bool is_edge_infected();
  void set_edge_infect();
  void clear_edge_infect();

  bool is_segment();
  void set_segment();
  void clear_segment();
  Triang *get_segment();

  Vertex* org();
  Vertex* dest();
  Vertex* apex();
  void set_vertices(Vertex *pa, Vertex *pb, Vertex *pc);

  void print(); // debug
}; // class TriEdge
```

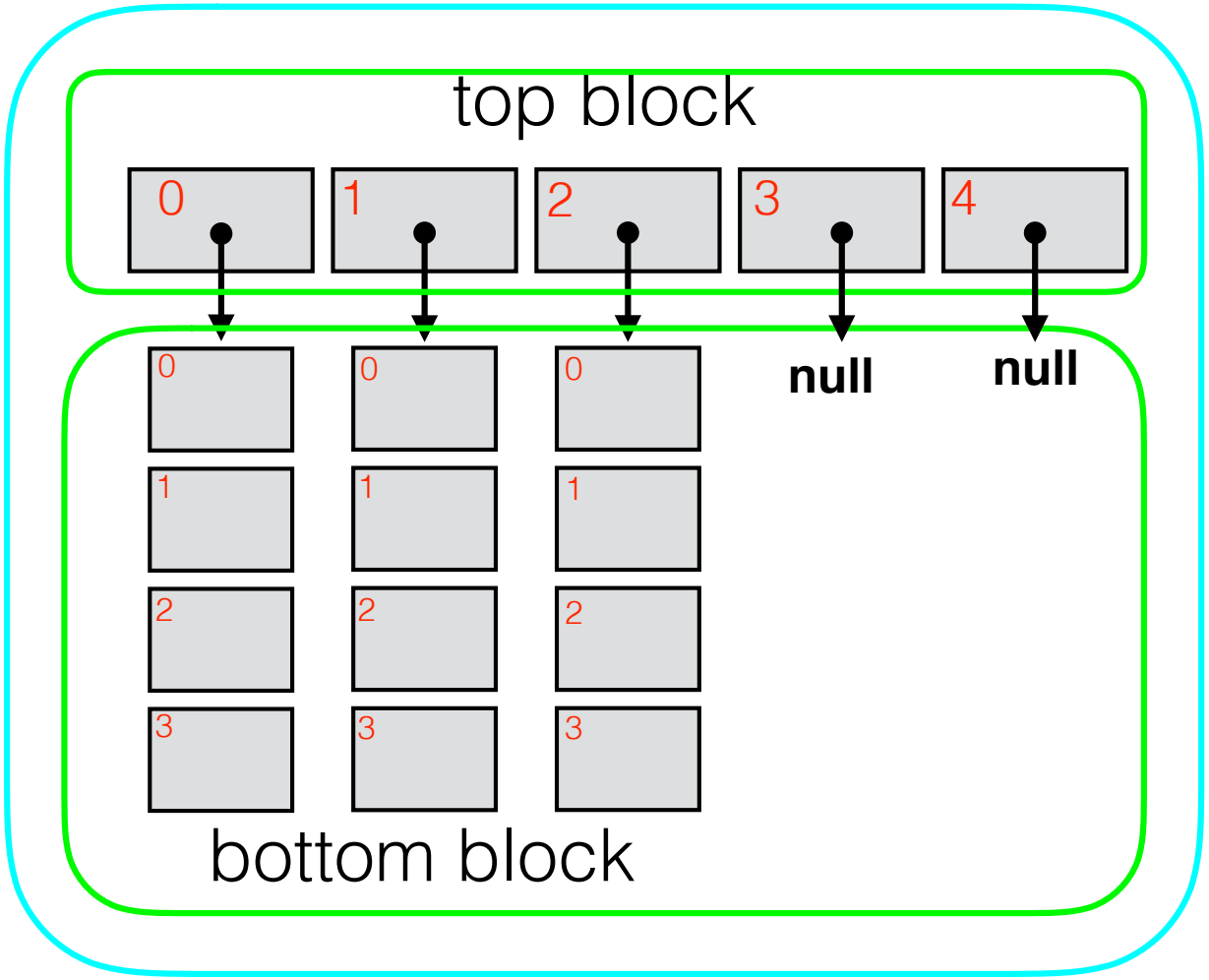


Resizable arrays

- For the construction and manipulation of the triangulation, we need to constantly modify and update the data structure while maintaining the combinatorial validity of the simplical complex properties. The number of triangles and vertices are continuously changing.

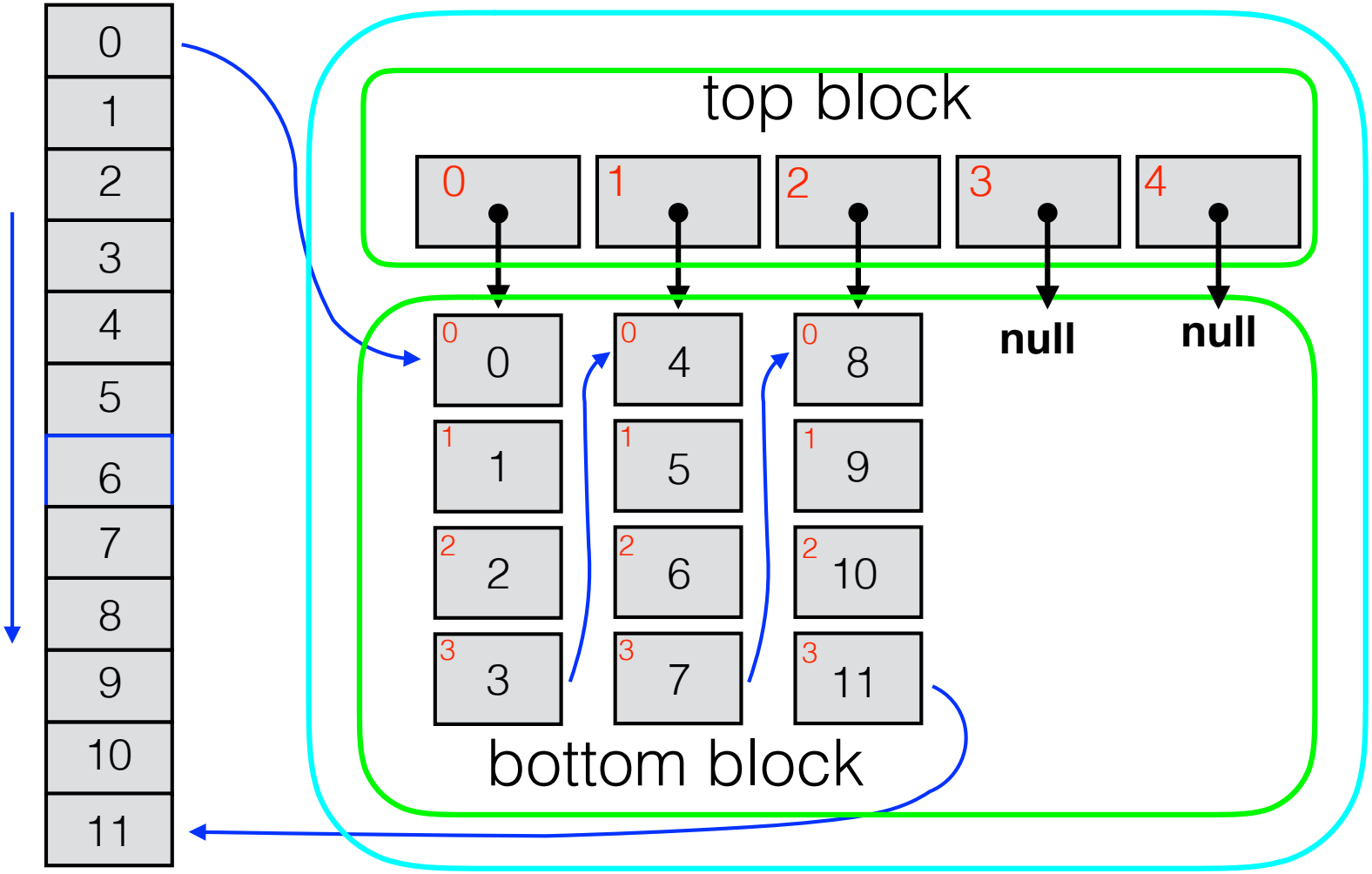
- Detri2 uses a ``resizable array" structure (developed by J. Shewchuk) to store them. This data structure allows you to access objects quickly by number (index), without limiting how many objects you can allocate.

- This data structure consists of two blocks. The top block is a single array of pointers (*void**) to the bottom block which is a list of arrays, each containing the same fixed number of objects



- The data are stored in the arrays of the bottom block.

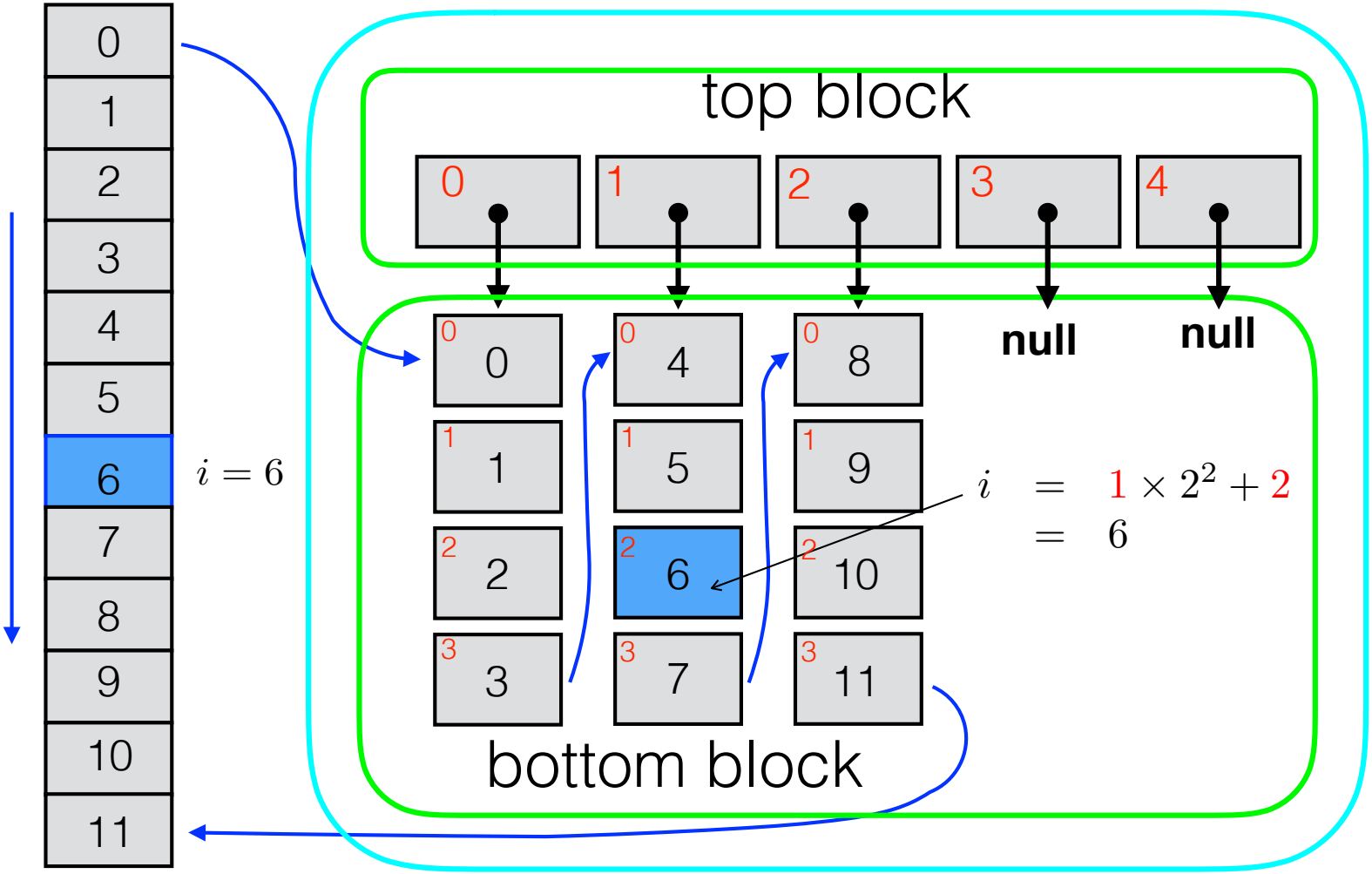
- Like a static linear array, each array index (>= 0) addresses a particular objects in a particular array of the bottom block.



- It is assumed that the length l of the arrays in the bottom block is a power of 2. We can split an index i >= 0 into a couple(m, n), where m >= 0, n < l, such that:

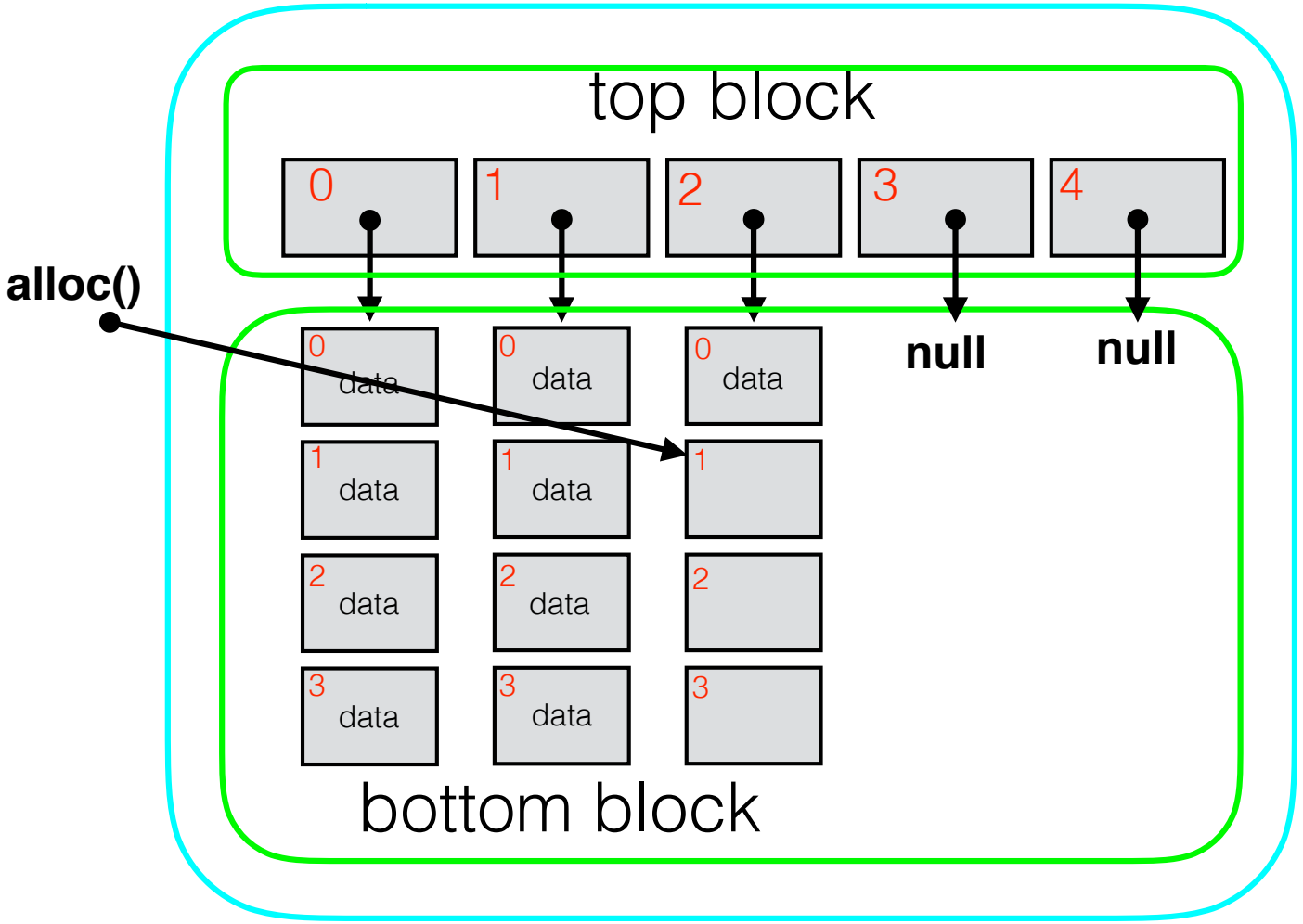
$$i = m \times 2^k + n$$

- where m locates a pointer in the top block, this points to an array in the second block which contains the i-th object which is addressed by the next index n within the array.



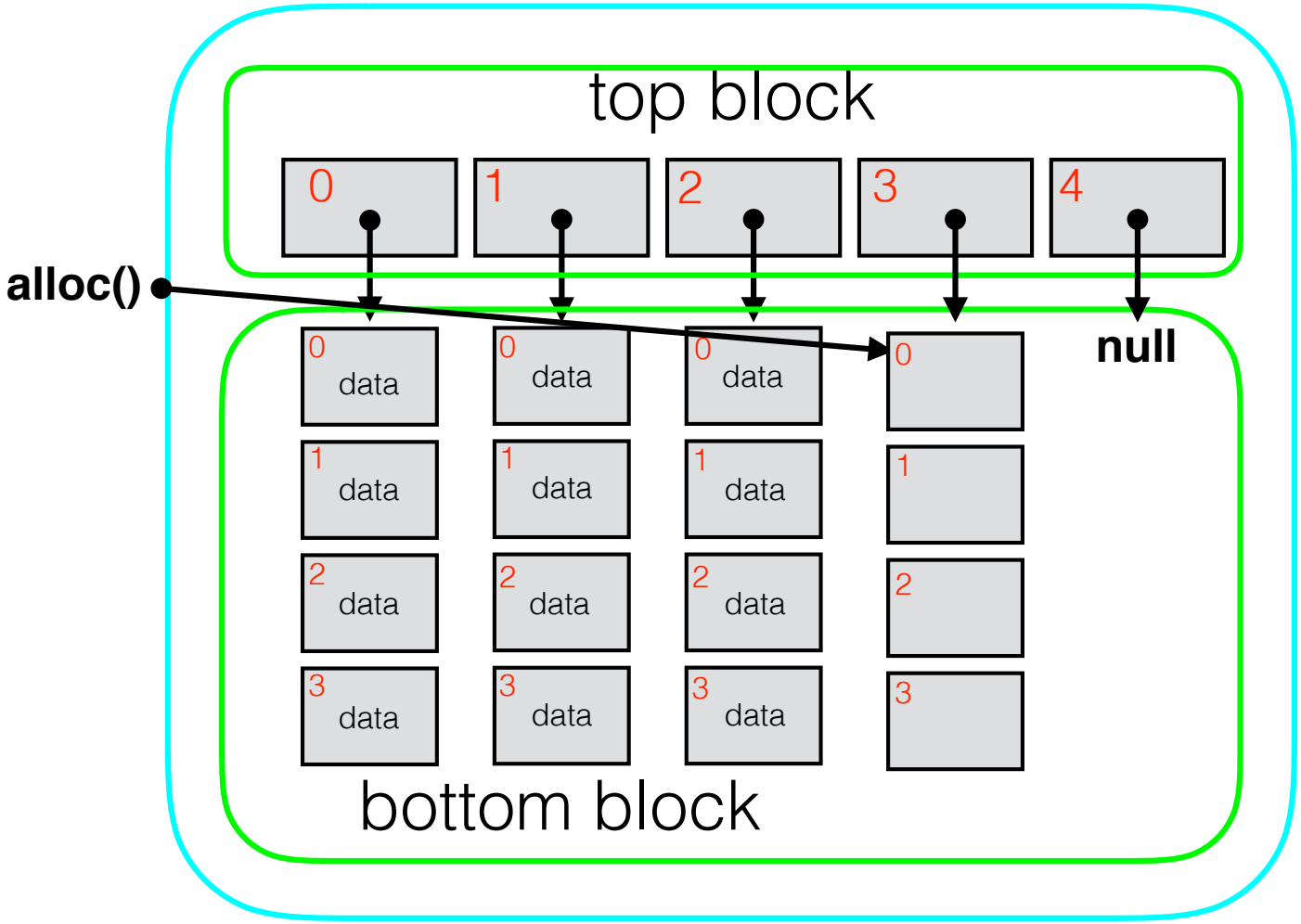
Memory allocation

- Define a function **alloc()** to return an empty slot.



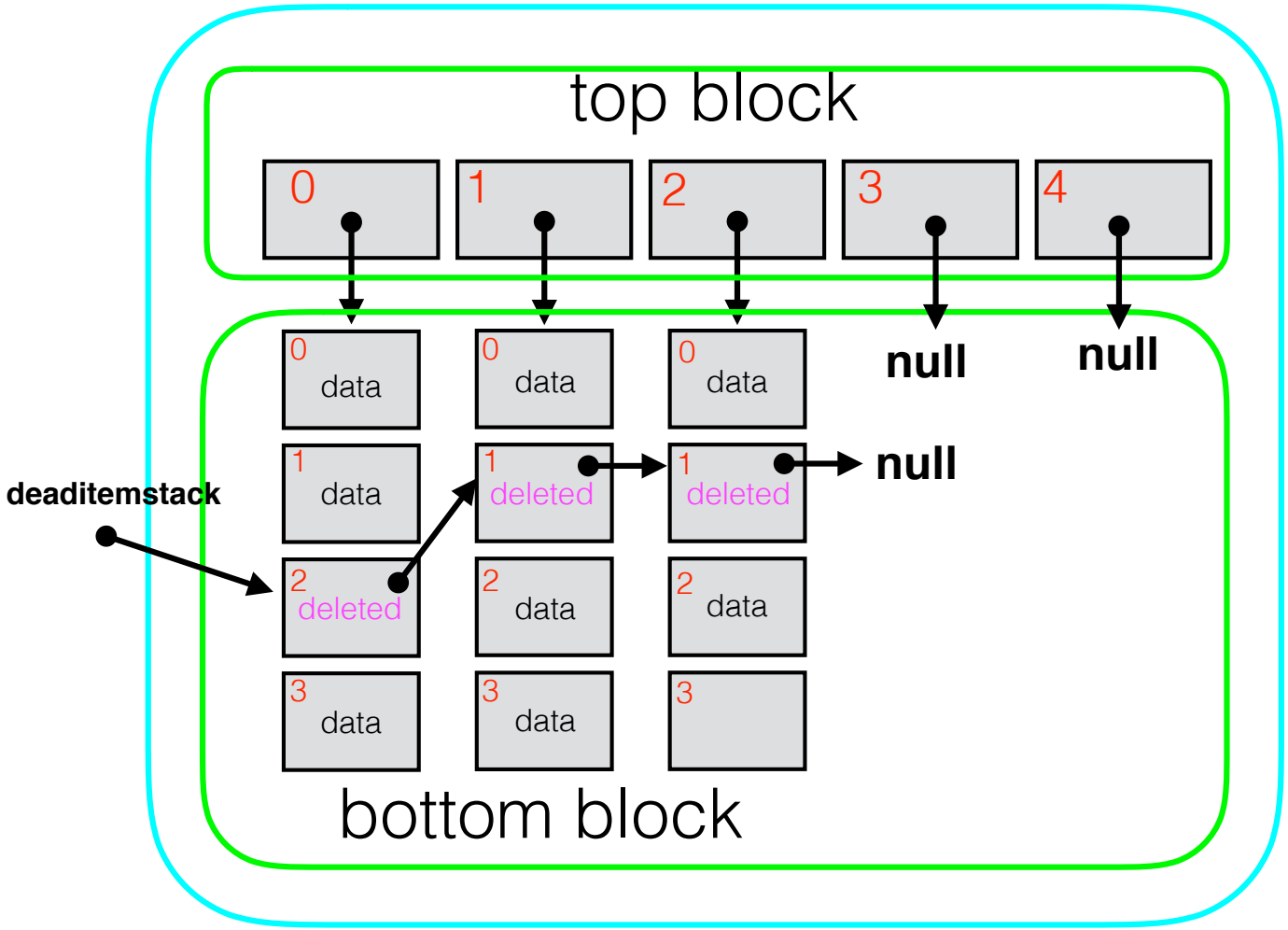
Memory allocation

- If there is no empty slot available, automatically allocate a new array.



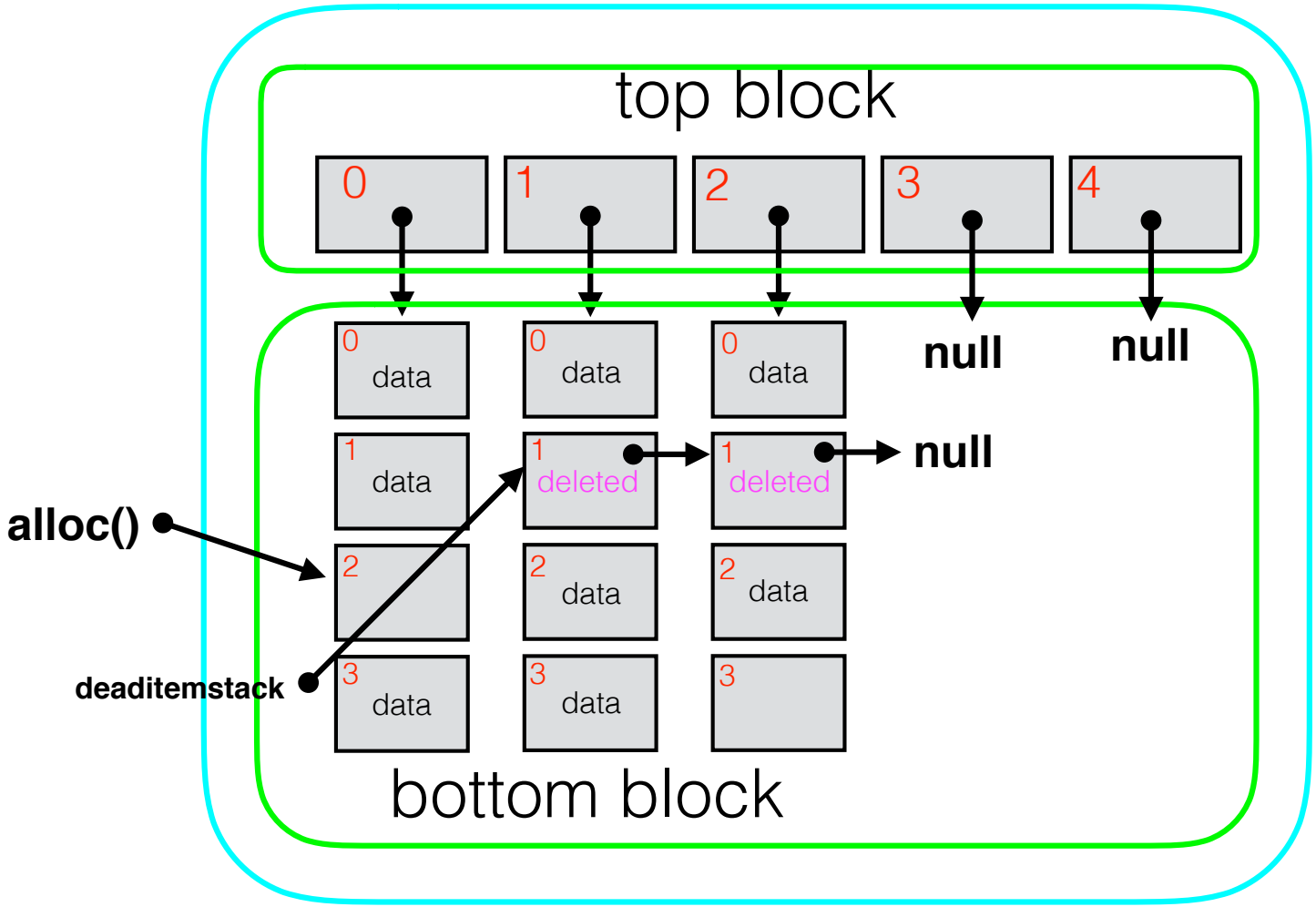
Memory recycling

- Unused slots are stacked (**deaditemstack**) and will be returned by **alloc()** first.



Memory recycling

- Unused slots are stacked (**deaditemstack**) and will be returned by **alloc()** first.



arraypool

```
// A dynamic (resizable) array
class arraypool
{
 public:
  int objectbytes;            // size of an object of the array
  int objectsperblock;        // bottom-array length (must be 2^k)
  int log2objectsperblock;    // k
  int objectsperblockmark;    // 2^k - 1
  int toparraylen;            // top-array length (> 0)
  char **toparray;            // the top-array
  long objects, used_items;   // actual and allocated objects
  unsigned long totalmemory; // memory used
  void *deaditemstack;        // a stack of dead elements (can be re-used)

  void restart();
  void poolinit(int sizeofobject, int log2objperblk);
  char* getblock(int objectindex);
  void* alloc();
  void dealloc(void *dyingitem);
  void* lookup(int objectindex);
  void* operator[](int index); // fast lookup

  arraypool();
  arraypool(int sizeofobject, int log2objperblk);
  ~arraypool();
};
```

A Basic Example

```
#include <stdio.h>
#include "arraypool.h"

class mydata
{
  public:
    int data1;
    double data2;
};

int main()
{
  arraypool Ary(sizeof(mydata), 3); // length = 2^3

  // Save some data into arraypool.
  // Here we save the sum of the Harmonic series: H_n = \sum_{i=1}^n 1/i
  mydata *pData;

  for (int i = 0; i < 10; i++) {
    pData = (mydata *) Ary.alloc();
    pData->data1 = i+1;
    pData->data2 = 0.0; // initialise
    for (int j = 1; j <= i; j++) {
      pData->data2 += (1.0 / (double)j);
    }
  }

  // Show the saved data of arraypool
  for (int i = 0; i < Ary.objects; i++) {
    mydata *ptr = (mydata *) Ary[i];
    printf("%d: %d,%f\n", i, ptr->data1, ptr->data2);
  }

  return 0;
}
```

make your own choice to identify deleted elements

```
class Vertex
{
 public:
  REAL        crd[3]; // x, y, h (height = x^2+y^2 - w).
  REAL        mtr[3]; // a metric tensor (a11,a21,a22), or function values (f,fx,fy)
  REAL        val;    // a nodal value (used as nodal mesh size or density)
  REAL        wei;    // vertex weight (used by Detri2)
  int         idx;    // Its index (0 or 1-based).
  int         tag;    // Boundary marker.
  char        typ;    // Vertex type.
  char        flags;  // flags of infected, etc.
  TriEdge     adj;    // Adjacent (or dual) triangle.

  bool is_deleted() {return typ == DEADVERTEX;}
  void set_deleted() {typ = DEADVERTEX;}

  void print(); // debug
}; // class Vertex

class Triang
{
 public:
  Vertex*  vrt[3];
  TriEdge  nei[3];
  REAL     cct[3]; // circumcenter and radius (dual and weight)
  REAL     val;    // a scalar value (also used as mesh size)
  int      flags;  // Encode, hullflag, infect, etc.
  int      tag;    // Boundary marker.
  int      idx;    // Index of its dual (Voronoi) vertex

  bool is_deleted() {return nei[0].ver == UNDEFINED;}
  void set_deleted() {nei[0].ver = UNDEFINED;}

  void print(int deatil = 0); // debug
};
```

An Example

```
void Triangulation::save_to_ucd(int meshidx, int reindex)
{

  idx = 1;
  count = 0;
  for (i = 0; count < tris->objects; i++) {
    Triangle* tri = tris->get(i);
    if (!tris->is_deleted(tri)) {
      // ignore a hull triangle.
      if (!tri->is_hulltri()) {
        fprintf(outfile, "%d 0 tri %d %d %d\n", idx,
                tri->vrt[0]->idx,
                tri->vrt[1]->idx,
                tri->vrt[2]->idx);
        idx++;
      }
      count++;
    }
  }
  fprintf(outfile, "\n");

}
```

Triangulation

```
class Triangulation
{
 public:
  // Input (in_) points, triangles, segments, subdomains.
  Vertex* in_vrts;
  Vertex* in_sdms;

  // Triangulation (tr_) elements
  arraypool* tr_steiners; // Steiner vertices
  arraypool* tr_segs;
  arraypool* tr_tris;
  Vertex*    tr_infvrt;   // The infinite vertex
```

Triangulation

```
// Flips (flips.cpp)
int  flip13(Vertex *pt, TriEdge *tt);
int  flip31(TriEdge *tt, Vertex **ppt);
int  flip22(TriEdge *tt);
int  flip24(Vertex *pt, TriEdge *tt);
int  flip42(TriEdge *tt, Vertex **ppt);
int  flip_check(TriEdge *te);
int  flip(TriEdge tt[4], Vertex **ppt, int& fflag, arraypool* fqueue);
int  first_triangle(Vertex *pa, Vertex *pb, Vertex *pc);
bool regular_test(Vertex* pa, Vertex* pb, Vertex* pc, Vertex* pd);
int  lawson_flip(Vertex *pt, int hullflag, arraypool *fqueue);

// (Weighted) Delaunay triangulation (delaunay.cpp)
int  locate_point(Vertex *pt, TriEdge &E, int encflag, int liftflag = 0);
int  sort_vertices(Vertex* vrtarray, int, Vertex**& permutarray);
int  first_tri(Vertex **ptlist, int ptnum);
int  incremental_delaunay();
```

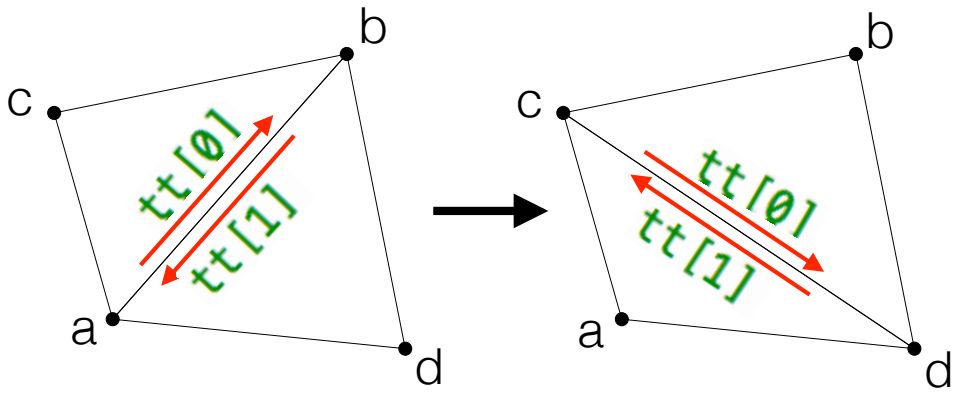
Flips

```
class Triangulation
{
 public:

  // Flips
  void flip13(Vertex *pt, TriEdge *tt);
  void flip31(TriEdge *tt);
  void flip22(TriEdge *tt);
  int flip(TriEdge *tt, int &flipflag, int check_only);
  int lawson_flip(Vertex *pt);
  int insert_point(Vertex *pt, TriEdge *tt, int loc);
  int remove_point(Vertex *pt, TriEdge *tt);
```

An Example: flip22

```
int Triangulation::flip22(TriEdge *tt)
{
  TriEdge nn[4];

  Vertex *pa = tt[0].org();
  Vertex *pb = tt[0].dest();
  Vertex *pc = tt[0].apex();
  Vertex *pd = tt[1].apex();

  nn[0] = (tt[0].enext()).esym(); // [b,c]
  nn[1] = (tt[0].eprev()).esym(); // [c,a]
  nn[2] = (tt[1].enext()).esym(); // [a,d]
  nn[3] = (tt[1].eprev()).esym(); // [d,b]

  tt[0].tri->init();
  tt[1].tri->init();

  tt[0].set_vertices(pc, pd, pb);
  tt[1].set_vertices(pd, pc, pa);

  tt[0].connect(tt[1]);

  tt[0] = tt[0].eprev(); // [b,c,d]
  tt[1] = tt[1].enext(); // [c,a,d]
  tt[2] = tt[1].enext(); // [a,d,c]
  tt[3] = tt[0].eprev(); // [d,b,c]

  pa->adj = tt[2];
  pb->adj = tt[0];
  pc->adj = tt[1];
  pd->adj = tt[3];

  return 1;
}
```

```
// On input,
//   tt[0] is [a,b,c], where [a,b] to be flipped.
//   tt[1] is [b,a,d]
// On output:
//   tt[0] is [b,c,d],
//   tt[1] is [c,a,d],
//   tt[2] is [a,d,c].
//   tt[3] is [d,b,c].
```



Incremental Delaunay

```
class Triangulation
{
 public:


  // (Weighted) Delaunay triangulation (delaunay.cpp)
  int  locate_point(Vertex *pt, TriEdge &E, int encflag, int liftflag = 0);
  int  sort_vertices(Vertex* vrtarray, int, Vertex**& permutarray);
  int  first_tri(Vertex **ptlist, int ptnum);
  int  incremental_delaunay();
```

Point Sorting

```
class Triangulation
{
 public:

   void randomly_permute(Vertex **permutarray, int arraysize);
   void sweepline_sort(Vertex **sortarray, int arraysize, double vx, double vy);
   void hilbert_init (int n);
   int  hilbert_split(Vertex **sortarray, int arraysize, int gc0, int gc1,
                      double, double, double, double, FILE*);
   void hilbert_sort (Vertex **sortarray, int arraysize, int e, int d,
                      double bxmin, double bxmax, double bymin, double bymax,
                      int depth, int order, arraypool *hvertices, FILE*);
   void brio_multiscale_sort(Vertex **sortarray, int arraysize,
                             int threshold, double ratio, int *depth, int order);
```

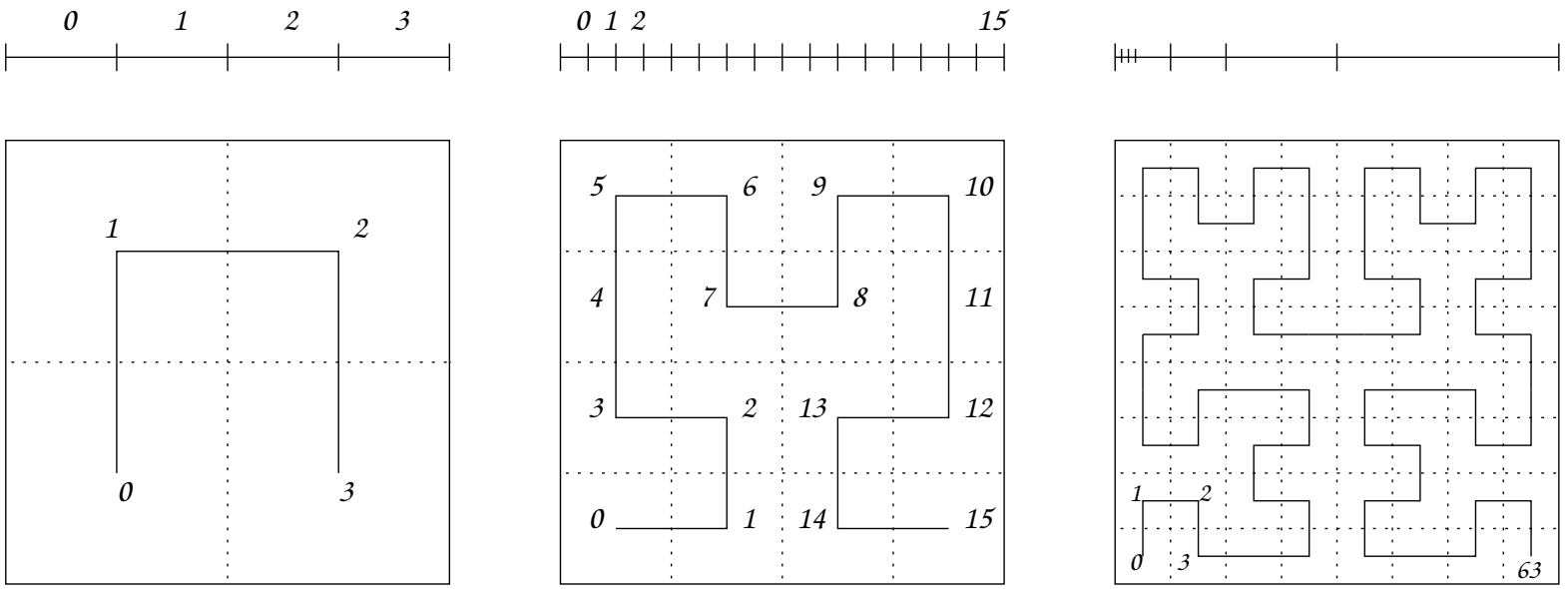
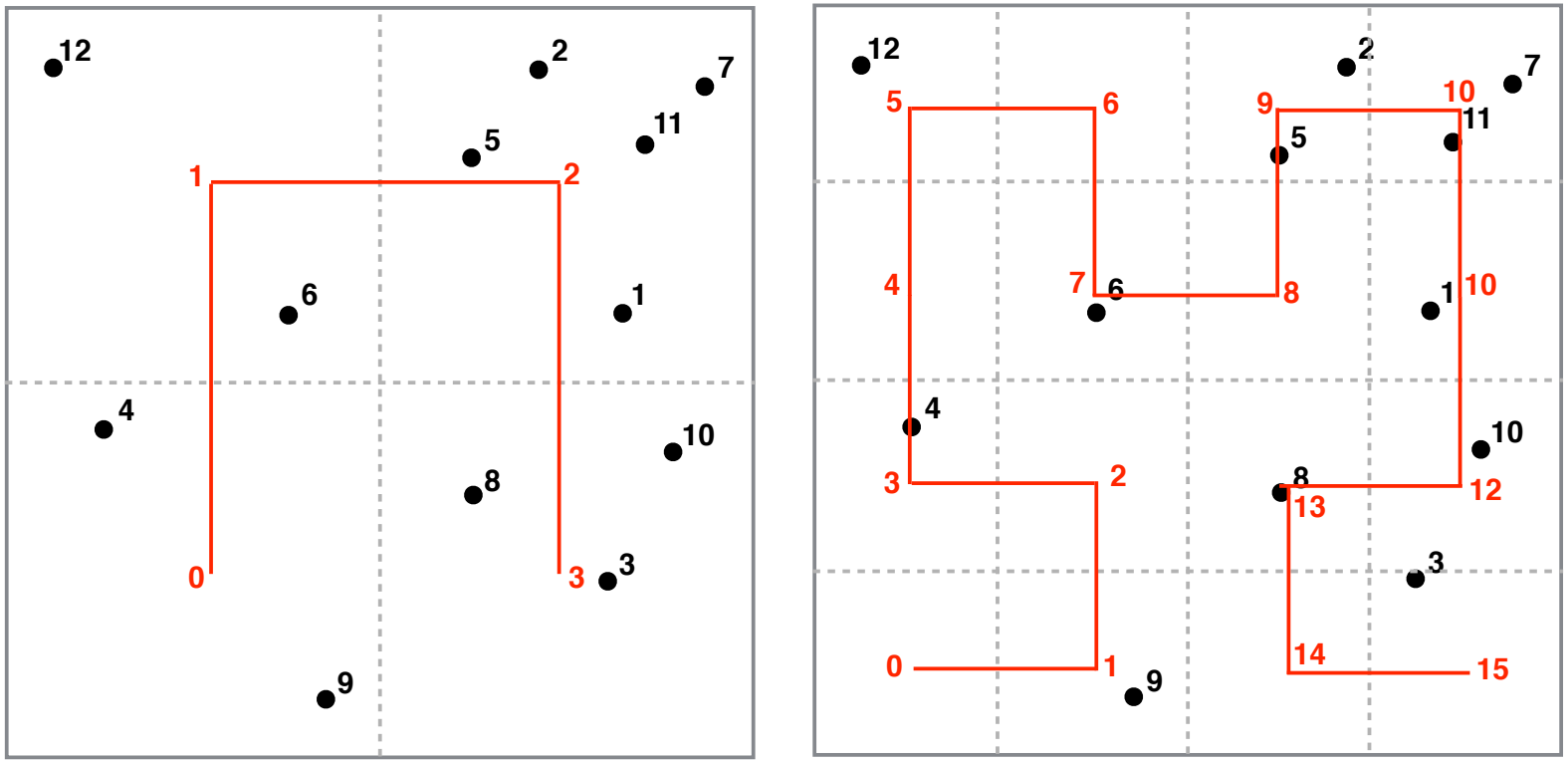


FIGURE 6. The first three iterations of the Hilbert curve in the plane (Figures from [**?**]).



00	01	11	10
0	1	2	3
{4,9}	{6,12}	{1,2,5,7,11}	{3,8,10}

← Grey code
← Hilbert indices
← sorted points

0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	9		4		12		6		{2,5}	{7,11}	1	10	8		3

FIGURE 7. An example of Hilbert sort on a set of 12 points in the plane.