# Header file

The header file contains a few structs and some of them are meant to be used as linked lists. Struct **cell\_search\_list\_s** is a singly linked list as it contains one pointer pointing to the same struct. Structs **cell\_search\_data\_s** and **cell\_search\_bucket\_s** are doubly linked lists. Pointer to these structs are often defined in the following pattern:

typedef struct struct\_name \*struct\_name\_p;

This allows to declare pointers easily without the asterisk:

struct struct\_name\_p ptr;

This also make easier to delcare pointers to pointers, which have been used in the source file:

struct struct\_name\_p \*ptr\_ptr;

The struct cell\_search\_t uses conditional compilation:

#ifdef CELL\_DISCRETE\_SIZES

double \*cell\_size;

#else

double cell\_size;

#endif

This tells the compiler to compile the struct with a *double* pointer if **CELL\_DISCRETE\_SIZES** has been defined (which it hasn’t), otherwise compile with just a normal *double* variable. If the *double* pointer was intended for dynamic memory allocation then I think used in this way is very unsafe because we do not have a safe way to calculate the amount of memory allocated in that pointer and this could easily lead to segmentation faults. The variable “int dimension;” might refer to the size of the *double* pointer.

# Source File

Overall, the source performs some operations on hash table chained with linked lists.

1. void cell\_search\_free(cell\_search\_p \*cell);

This function is used to free memory inside the celll\_search\_t struct. The function argument is a pointer to a pointer of type cell\_search\_t. Essentially the above declaration translates to:

void cell\_search\_free(cell\_search\_t \*\*cell);

The line:

if(!cell || !\*cell)

return;

Checks whether the pointer to the pointer or the pointer (derefenced pointer to pointer) is NULL. Without this if statement, the later part of the code can cause segmentation faults.

Next we have this while loop:

while((\*cell)->buckets)

{

bucket=(\*cell)->buckets;

(\*cell)->buckets=(\*cell)->buckets->next;

SAFE\_FREE(bucket->contents);

SAFE\_FREE(bucket);

}

This scans through all the nodes in the linked list **(\*cell)->bucket** and frees the memory from the **bucket->contents** pointer and then deletes the whole node. In this way the whole linked list is deleted from memory. “SAFE\_FREE” might be a macro, which I suppose would be:

#define SAFE\_FREE(ptr) if(ptr!=NULL)free(ptr)

The next while loop performs a similar task on **(\*cell)->entries** but in this loop we have this for loop:

for(i=0;i<entities->size;++i)

SAFE\_FREE(entities->entity[i].domain);

The next two lines frees a couple of other pointers inside the (\*cell) object:

free\_generic\_hash\_table(&(\*cell)->lookup);

SAFE\_FREE((\*cell)->tmp\_index);

And finally the **(\*cell)** pointer is deleted. After that statement, the pointer cell will point to NULL.

In a few words, the function deletes an entire list.

2. cell\_search\_p cell\_search\_create(int dimension,double cell\_size,int bucket\_size,int guess);

Just by looking at the return type and function argument we can guess that this function creates a struct type of “**cell\_search\_t**” and returns a pointer to it.

The object is created with the statement:

cell\_search\_p new=SAFE\_CALLOC(1,cell\_search\_t);

This function uses conditional compilation inside its definition:

#ifdef CELL\_DISCRETE\_SIZES

int i;

new->cell\_size=SAFE\_CALLOC(dimension,double);

for(i=0;i<dimension;++i)

new->cell\_size[i]=cell\_size;

#else

new->cell\_size=cell\_size;

#endif

This essentially means that if **CELL\_DISCRETE\_SIZES** has been defined then, the variable “**size**” is a pointer and memory allocation is needed. Otherwise, “**size**” is just a normal *double* variable and needs to be initialized as usual.

3. static cell\_search\_bucket\_p create\_cell\_bucket(cell\_search\_p cell);

This function creates a struct of type **cell\_search\_bucket\_s** and returns a pointer to it. The static keyword means that this function is used only in this file. This function essentially creates a new node for a linked list of type **cell\_search\_bucket\_t**.

4. static cell\_search\_data\_p create\_cell\_data(cell\_search\_p cell);

Similar to the function above, this one creates a new node for a struct of type “**cell\_search\_data\_t**”. The function simply allocates memory for a new node and initialize the member data.

5. static void local\_add\_cell\_aabb(cell\_search\_p cell,

double \*lower,double \*upper,

cell\_search\_data\_item\_p entity,int level);

The function checks whether the given cell’s dimension has reached a certain amount. In that case, an item is looked up in the hash table contained in the **cell\_search\_t** struct. From the code it appears that if the item (struct of type **generic\_hash\_data\_t**) has not been found in the hash table then the item is created and initialized.

This particular line is very important, because it appears very often in other functions:

bucket=(cell\_search\_bucket\_p)found->data.pointer;

From this, it seems clear that if the item has been found, the pointer to the bucket containing that item is stored inside the content of the **found** pointer. Hence, the above statement retrieves the bucket where the item was stored.

The item is then added to the hash table. If the bucket is full then the item is put in the overflow bucket. Conditional compilation is used in this function. The variable min and max are defined after taking into consideration whether the macro **CELL\_DISCRETE\_SIZE** has been defined or not.

The function keeps doing these tasks in a recursive way until this if statement returns true:

if (level == cell->dimension){

....

}

6. void cell\_search\_add\_point(cell\_search\_p cell,double \*point,long id);

The function adds a new node to the **cell->entries** list and initializes some of its member data. Conditional compilation is used to define the key of the item in the hash table. The rest of the code performs similar tasks described in function number 5. So if the destination bucket of the item has reached full capacity then the item is moved to the overflow bucket. One particular line is the use of the following macro:

entity->domain=SAFE\_MEMDUP(point,cell->dimension,double);

Possibly this is a macro using memory reallocation, or at least, something to do with memory management.

7. static void local\_find\_cell\_aabb(cell\_search\_p cell,

double \*lower,double \*upper,int level,int mid,

cell\_search\_list\_p \*search);

The function looks for an entry in a hash table. If the entry has been found then the it iteratively loops over a list in the following way:

bucket, bucket->overflow, bucket->overflow->overflow, bucket->overflow->overflow->overflow

However, I don’t think that each bucket has its own overflow bucket and that in turns has another overflow bucket within it and so on. Instead, it is more reasonable to assume that every bucket shares a common overflow bucket and that the *while(bucket)* loop will just iterate over the bucket and the overflow bucket. Essentially, the while loop will iterate in the following way:

bucket, bucket->overflow, bucket returns null pointer so the program exits from the while loop.

Inside the while loop, if the following *if statement* is verified:

if(data->op\_number!=SI\_op\_number){...}

Then a new **cell\_search\_list\_t** is created and copied to the content of the **search** pointer (argument of this function):

cell\_search\_list\_p new;

data->op\_number=SI\_op\_number;

new=SAFE\_MALLOC(sizeof(cell\_search\_list\_t));

new->entry=data;

new->next=\*search;

\*search=new;

This is clearly a very common algorithm to add a new node in a linked list.

If the entry has not been found in the hash table, then the operations described above is performed in a slightly different way.

It’s unclear what the segment of code after the directives does, but it is definetely a recursive function. The function exits from the recursive loop if this *if statement* returns *null*:

if(found){...}

8. void cell\_search\_add\_aabb(cell\_search\_p cell,double \*lower,double \*upper,long id);

The function seems to add a new node to the list **cell->entries->entities.** Some of the content in this node is copied into the **\*lower** and **\*upper** pointers, which are then passed as parameters to a call of function 5.

9. static void local\_find\_cell\_point\_in\_aabb(cell\_search\_p cell,

double \*lower,double \*upper,int level,int mid,

cell\_search\_list\_p \*search);

The function is identical to function number 7, except for this *if statement*:

if(mid==level){...}

This is the condition to exit the recursion for this function

10-11. cell\_search\_list\_p cell\_search\_aabb(cell\_search\_p cell,double \*lower,double \*upper);

cell\_search\_list\_p cell\_search\_point\_in\_aabb(cell\_search\_p cell,double \*lower,double \*upper);

The two functions initially perform the same task:

cell\_search\_list\_p found=(cell\_search\_list\_p)0;

++SI\_op\_number;

And then call in function 7 and 9 respectively. Now it is clear why function 7 and 9 are declared static, because they were meant to be called within function 10 and 11. The variable SI\_op\_number seems to track the number of times either function 10 or 11 is called.

12. void cell\_search\_free\_list(cell\_search\_list\_p \*list);

The argument of the function seems like a list of lists. The function deletes every list (nodes of the main list) and then deletes the main list.

13. cell\_search\_list\_p cell\_search\_point(cell\_search\_p cell,double \*point);

Like function 7 and 9, this function looks up an item in the hash table contained inside **cell** and then scans through the bucket containing the item. Unlike function 7, some extra operations are performed to calculate a value for the **closest\_dist** variable. The actual meaning of that section of code is unclear without any documentation. If the variable **closest\_dist** is within a certain limit then the function returns a pointer of type struct **cell\_search\_list\_t** that is not null as memory has been allocated for it. Otherwise it returns a null pointer.

14. static void local\_find\_cell\_point\_tol(cell\_search\_p cell,

double \*point,double \*lower,double \*upper,int level,

double \*closest\_dist,

cell\_search\_data\_item\_p \*closest);

Another static function so it will be called inside another function. Again, this function performs the same operations of function 7 with slight changes:

cell\_search\_data\_item\_p data=bucket->contents[i];

dist=0.0;

for(j=0;j<cell->dimension;++j)

dist+=(data->domain[j]-point[j])\*(data->domain[j]-point[j]);

if(!\*closest || dist<\*closest\_dist)

{

\*closest\_dist=dist;

\*closest=data;

}

And the meaning of this section of code is unclear.

15. cell\_search\_list\_p cell\_search\_point\_tol(cell\_search\_p cell,

double \*point,double tolerance);

The function allocates memory for 2 arrays, **\*upper** and **\*lower** and calculates the values of each element. The purpose of these 2 arrays is to pass them as parameters to function 14. If this condition is met:

if(closest && sqrt(closest\_dist) < tolerance){...}

Then the function will return a pointer to a struct of type **cell\_search\_list\_t** that is not null.