

Programming assignment – Sparse Vector

Traditionally vectors are implemented using an array data structure (static or dynamic). In this assignment you will use linked-list data structure to obtain memory-efficient implementation of vectors. The assignment should give you an additional practice with pointers and dynamical memory allocation. The goal is to implement several functions that manipulate nodes in a linked list. This functions include the traditional list operations like inserting, removing, searching, as well as several vector-related functions – scalar product and addition.

Here is a structure to represent a linked list node

```
struct ElementNode {
    int    data;
    int    pos;
    struct ElementNode *next;
};
```

Integer **data** is the value which is stored in the node. Compared to the traditional node ours has an additional field – **pos**, which represents element's index in the vector. Since the nodes are to be used in a singly linked list, we have a pointer to the next element **next**.

Opaque pointers

First of all, if you were using regular C-style pointers, you'd put the above definition of **ElementNode** in the beginning of **spvector.h** and your functions would look like:

```
int scalar_product(const ElementNode *p_e1, const ElementNode *p_e2);
```

For this assignment you are required to use opaque pointers – here is a roadmap to convert existing code that uses pointers to opaque pointers. I'll also explain why regular pointers are bad.

1. move struct **ElementNode** definition into **spvector.c** (thus it is hidden from the client)
2. in **spvector.h** add

```
typedef struct ElementNode* ElementNode_handle;
typedef const struct ElementNode* ConstElementNode_handle;
```

(see source code for explanation) note that when compiler reads the above line, it DOES NOT know the definition of **ElementNode** – **spvector.c** has not been seen yet (make sure you understand why). Therefore **ElementNode** is an incomplete type. The rule in C/C++ is that pointers to incomplete types are legal (but dereferencing of those pointers is not)
3. rewrite all functions to use **ElementNode_handle** instead of **ElementNode*** and **ConstElementNode_handle*** instead of **ConstElementNode****. Note that inside **spvector.c** you may use **ElementNode** type and **ElementNode** structure fields, since by that time compiler will have full information about **ElementNode** structure. Note that inside **spvector.h** you may NOT use **ElementNode*** and **ElementNode_handle** interchangeably.

Example: this is legal in **spvector.c**, but not **spvector.h** and **driver.c**:

```
ElementNode p = ...;
p.pos = 5;
```

So what?

As we mentioned, since **->next** is illegal in the driver, client cannot modify keys and linked list is safe. To make modification of **next** value possible, implement **setNext(...)** function. Notice how much similar opaque pointers to private data in a C++ class!

Opaque pointers are more than just safety. They also provide encapsulation. Here is a typical example: in a big project we have a struct **ElementNode** in a file **spvector.h** – so its definition is visible to everyone. **ElementNode** uses **next** and **pos** names for its fields. Since **next** and **pos** names are visible to the client (like public data in a C++ class) other files may use them (and will do that, even though methods like **getNext()** and **getpos()** are implemented). Then the maintainer of **spvector.h** decides to change the names to **Next** and **Pos** (or may be change their types or eliminate them completely). Now every file that uses **spvector.h** is broken and has to be updated, which may be a lot of work. Opaque pointers help to avoid this kind of problems by **forcing** the clients to use available getters and setters **getNext()** and **getpos()** instead of directly accessing the data. So now the change will proceed like this:

- implementor changes `next` to `Next`
- updates `getNext()` from
`... getNext() return next;`
to
`... getNext() return Next;`
- recompiles the project

that's all, no other code is effected since everyone was using `getNext()` instead of `next`.

back to sparse vectors

These nodes may be used to represent a very long vector of integers most of which are 0's. Example vector (0,0,0,1,0,0,0,0,0,0,0,2) is a vector with 1 at position 3 and 2 at position 11 (counting from 0). When stored in an array, the array size should be at least 12 (which gives $12 * 4 = 48$ bytes). When stored as a list, the vector looks like (1,3)->(2,11), where the first number in the pair is the value and second is position. So that the total amount of memory used is 2 nodes, which is $2 * (4 + 4 + 4) = 24$ bytes only.

The idea is to save storage space by never storing 0's. Make sure that all functions that modify the vector test for 0 before actually writing into it. Note that insert function may be used to overwrite a value, that is `insert_element(list,pos,val)`

- creates a new node if no node at position `pos` exists
- just modifies the data if a node at position `pos` exists
- deletes a node at position `pos` exists if `val=0`

This type of vector is usually known as *sparse vector*.

List representation has only one drawback – it doesn't remember the length of the original vector, e.g. both (0,0,0,1,2,0,0,0) and (0,0,0,1,2,0,0,0,0,0,0,0,0,0,0) have the same list representation (1,3)->(2,4). Functions that require size information will have an additional parameter – dimension of the vector, see `printf_elements`.

Functions to be implemented are insert (ordered by position), delete at position.

Because lists with positions may be viewed as vectors, there are additional functions that may be useful:

- `get` the value at the given position – similar to index operator.
- vector addition
- scalar multiplication

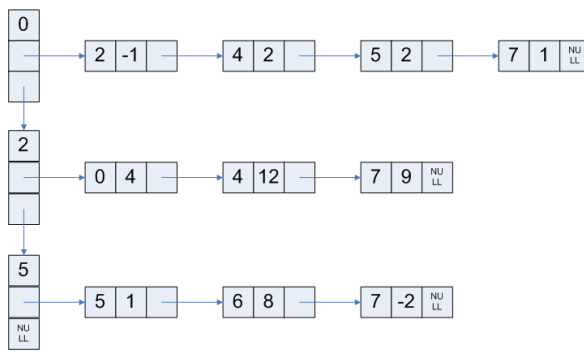
Extra credit part 30 points Almost every library that implements a matrix-like type does it by reusing a vector type. We can do the same by introducing an additional type `RowNode`:

```
struct RowNode {
    int pos;
    struct ElementNode *elements;
    struct RowNode *next;
};
```

This type is basically yet another node for a singly linked list of rows (vectors) which allows several lists of the `ElementNode` type to be tied together. For example, if we have a matrix with only 3 non-zero rows:

```
0'th row (0, 0,-1, 0,  2, 2, 0, 1)
2'nd row (4, 0, 0, 0, 12, 0, 0, 9)
5'th row (0, 0, 0, 0, 0,  1, 8,-2)
```

graphical representation for the above structure is



and as a matrix (I used `printf_rows(p_r, "%4i",8);`)

```

(0, 0,-1, 0,  2, 2, 0, 1)
(0, 0, 0, 0,  0, 0, 0, 0)
(4, 0, 0, 0, 12, 0, 0, 9)
(0, 0, 0, 0,  0, 0, 0, 0)
(0, 0, 0, 0,  0, 0, 0, 0)
(0, 0, 0, 0,  0, 0, 0, 0)
(0, 0, 0, 0,  0, 1, 8,-2)
(0, 0, 0, 0,  0, 0, 0, 0)
(0, 0, 0, 0,  0, 0, 0, 0)

```

where rows 1,3,4,6,7 are all-zero rows. Row 6,7 are added to make the structure look like 8x8 matrix (see the second parameter of `printf_rows`).

Extra credit (20 pts): implement `determinant` function – see bottom of `matrix.h`. If you choose not to implement it – provide an empty implementation like:

```

int determinant(const RowNode *p_r,int dim) {
    /* print args to get rid of warnings */
    printf("%p\n",(void*)p_r);
    printf("%i\n",dim);
    return 0;
}

```

Notes: The header file contains functions that need to be implemented. Several function will call other function you’ve implemented, so the total amount of work is not that big.

Criteria:

- For this assignment, you are not allowed to use any **for** loops. All of the list traversals must use **while** or **do...while** loops with pointer tests like `pList==NULL`. Do not use non-pointer tests – this is the same as having a **for** loop. Note that I had to use **for** loops in `printf...` functions. If you feel that you need a **for** loop for the extra credit part – talk to me first.
- Simpler implementation of `determinant` function uses recursion.
- Compilation (see Makefile in the project folder), you may use it as `make gcc0`, `make bcc0`, `make msc0` from bash and MS command prompt if the latter has Cygwin in the PATH variable).
- you should also test with Borland CodeGuard, compile with `make bcc_cg` and run the tests normally (you DO NOT add 100 to the test number). After each test check the content of the file `bcc_cg.cgl`. If it’s empty, everything is OK. Otherwise there are problems, content of the log-file (`cgl` = code-guard log) should help you to track it down. See ”memory errors” on the class web-site for some common mistakes. Check out `bcc_cg.cgi` – CodeGuard initialization/configuration for this project (you may modify it if you know what you are doing, I’ll use the provided configuration for grading),

[Options]

```

CodeGuard=yes
Stats=no           #display run-time statistics
MessageBox=yes     #pop-up window in the case of an error
DebugInfo=yes      #very useful

```

```
Append=no                #append or overwrite log file
Repeats=yes              #???
Srcpath=                 #I think it defaults to "current folder", so OK
IgnoredModules=          #we may add STL here
ErrorCaption=Error
ErrorText=There are errors in the program. A log file will be created.
OutputDebugString=no
StackFillFrequency=2
ResourceLeakReport=yes
MaxCGLerrors=65535
LimitCGLerrors=no
```

To submit

You must submit electronically through submission page a single zip-file containing:

- implementation file `spvector.c`
- header file `spvector.h`, you should only modify the comments – make them doxygen-style. Do not modify function signatures – otherwise my driver may not compile.
- `index.chm` - doxygen-compiled help file