Basic Data Structures from a Low Level Viewpoint

Oliver-Matis Lill

May 16, 2017
Session topics

The covered data structures are:

1. Lists
2. Vectors
3. Deques

To force a deeper understanding of these structures, you will have to code them in the C language.

We will be covering some of the important concepts of the C language.
The C language

- C is a relatively simple language
- The vast majority of what C allows is close to what the machine itself does
- As such it doesn’t allow for highly abstractive features like Object Oriented Programming and doesn’t have non-trivial structures like Vectors, Sets and Maps built-in
- The flip side of this is that the language should be much easier to learn than C++ or Java
- For a more complete picture, we will go over how memory is used in C
C Variable Mechanics

- Each variable is associated with a fixed memory segment from the start of its lifespan (as opposed to Java)

  ```
  int a=1521;
  ```

- The assignment operator copies data into the memory segment of the variable (as opposed to Java)

  ```
  int b=a;
  ```
C Pointers

- A pointer is just a variable that stores some memory address:

  ![Diagram showing memory addresses and pointers]

- You can dereference pointers with the * sign and use the dereferenced pointer as a variable to, for example, modify it:

  ```c
  int p = &a;
  *p = 130;
  ```

- The type specifier (here `int`) only helps the compiler know how to use the pointer, otherwise all pointers are the same, just variables that store memory addresses.
Pointers can change the address they point to:

You can use addition and subtraction to move the pointer (useful for iterating over arrays):

In fact you can perform all sorts of arithmetic on pointers. I recommend you spend some time learning it and playing with it.
In C, the array elements always occupy consecutive memory locations:

In multidimensional arrays the data is consecutive as well and the ordering is fairly intuitive:

With C arrays, their size must be known on compile time, so unless n is constant you can’t declare an array like `int a[n];`
C Dynamic Memory

- You can use malloc to allocate a new chunk of memory. The allocated size can be calculated on runtime.

```c
int *p = malloc(6 * sizeof(int));
```

- You can use free to deallocate previously allocated memory. That memory can be reused (even by other programs).

```c
free(p);
```

- malloc stores extra data on allocation size, so free will always know the size to deallocate.

```c
int *q = malloc(4 * sizeof(int));
```
A list is a group of objects connected into a "chain". It might look like this:

- It’s advantage is that it allows you to insert to and delete from anywhere in $O(1)$
- It’s disadvantage is that access by index is $O(n)$
- Additionally the elements are dispersed in memory, eliminating vectorization and cache locality opportunities and making it much slower than an array in practice
Example

```c
void insert(struct List* list, struct Node* pos, int toAdd) {
    struct Node* cur = malloc(sizeof(struct Node));
    cur->prev = pos; cur->value = toAdd;
    if(cur->prev == 0) {
        cur->next = list->first;
        list->first = cur;
    } else {
        cur->next = cur->prev->next;
        cur->prev->next = cur;
    }
    if(cur->next == 0)
        list->last = cur;
    else
        cur->next->prev = cur;
}
```
Vectors are basically resizeable arrays

Vectors have a capacity. They reserve some extra memory where new elements are added.

1. A new memory block with (2x) larger capacity is allocated
2. The old data is copied over into the new block
3. The old block is deallocated

Once capacity is exceeded the following is done:

\[ \text{append(4)} \quad (3, 7) \rightarrow (3, 7, 4) \quad \text{Size} = 3 \quad \text{Capacity} = 5 \]

\[ \text{append(7)} \quad (3, 7, 4, 6) \rightarrow (3, 7, 4, 6, 7) \]
- $O(1)$ append operation (insertion to the last position)
- Reallocation takes $O(n)$ time, however if vector is empty and $m$ append operations are performed, then the time complexity of all reallocations is: 
  \[ O(2^0) + O(2^1) + O(2^2) + \ldots + O(2^{\lfloor \log_2(m) \rfloor}) \in O(m) \]
- The advantage is that the data is consecutive, so in practice it's about as fast as an array with the benefit of resizeability
- The disadvantage is that it can have significant memory overhead (up to 3x during reallocation), however usually it doesn’t matter
- Note that reallocation invalidates all pointers to data in the vector, so be careful!
Example

```c
void append(struct Vector* vector, int toAdd) {
    if(vector->size == vector->capacity) {
        int* newData = malloc(2*vector->capacity*sizeof(int));
        memcpy(newData, vector->data, vector->size * sizeof(int));
        free(vector->data);
        vector->data = newData;
        vector->capacity *= 2;
    }
    vector->data[vector->size++] = toAdd;
}
```
Deques

- Deques are resizeable sequences like vectors, but internally they behave differently.
- Deques store elements in (reasonably large) chunks and when capacity is exceeded they simply add a new chunk.
- The data is more consecutive than in a list, but less than in a vector.
- As a result in practice it’s much faster than a list, but somewhat slower than a vector.
- The main benefit over a vector is that no reallocation is performed, so pointers to its data don’t get invalidated.