“Memset” in $O(1)$

DONG Yuxuan <https://dyx.name>

11 Nov 2023 (+0800)

Introduction

In many cases we need an array with initialized elements, e.g., `a[i] = 1`, or more dynamic, `a[i] = i`. The initialization uses $O(n)$ time for n elements. If there are only few elements will be accessed, the initialization becomes the bottleneck of performance. Exercise 1.9 of [2] describes a data structure maintaining a list of elements with $O(1)$-time random access as arrays, but also with

- $O(1)$ initialization and reset time;
- $O(n)$ extra space;
- not relying on memory-initialization values of the system.

All the above complexities are for the worst case. According to [2], the data structure can at least be traced back to exercise 2.12 of [1].

A C-like pseudo language is used to describe the data structure. Formally, the data structure has the following operations.

- `void init(void);`
- `int get(int i, int d);`
- `void set(int i, int v);`

The `init()` function initializes or resets the data structure. The `set(i, v)` function sets the value of the `i`-th element to `v`. If `set()` has been called on an element, the element is called *dirty*. The `get(i, d)` function returns the `i`-th element: if the element is not dirty, the default value `d` is returned.

Implementation

The data structure maintains 3 arrays and an integer variable.

```
#define N 1024

int data[N];
int dirty[N];   /* indexes of dirty elements */
int ndirty;     /* size of dirty[] */
int invert[N];  /* inverted indexes of dirty[] */
```

The `data[]` array stores the elements: if the `i`-th element is dirty, `data[i]` contains its value, otherwise `data[i]` is undefined.

The `dirty[]` array stores indexes of dirty elements and `ndirty` indicates the size of `dirty[]`.

The `invert[]` array is the "inverted index" of `dirty[]`: `invert[i]` represents the index of `data[i]` in `dirty[]` if the `i`-th element is dirty, otherwise `invert[i]` is undefined.

The `init()` function just sets `ndirty` to zero.

```
void init(void)
{
        ndirty = 0;
}
```

If `invert[i]` is not smaller than `ndirty`, its value is definitely undefined, thus the element is not dirty. If `invert[i]` is smaller than `ndirty`, the element may be dirty, but there is still a chance that `invert[i]` happens to be a value smaller than `ndirty`. Fortunately, as `invert[i]` is smaller than `ndirty`, `dirty[invert[i]]` is defined. Thus we check if `dirty[invert[i]]` equals to `i`. If this is true, the element is dirty, otherwise it's not.

The `get()` and `set()` functions use the above check to determine the action.

```
int get(int i, int d)
{
        int j;

        j = invert[i];
        if (j < ndirty && dirty[j] == i)
                return data[i];
        return d;
}

void set(int i, int v)
{
        int j;

        data[i] = v;
        j = invert[i];
        if (j < ndirty && dirty[j] == i)
                return;
        dirty[ndirty] = i;
        invert[i] = ndirty++;
}
```

Application

The general sampling problem has a de-facto algorithm: run the Fisher-Yates shuffle [3:5.3] but terminate after the first k elements are determined. The following code samples k integers from the array `a[]` with n elements.

```
for (i = 0; i < k; i++) {
        j = i + rand() % (n - i);
        printf("%d\n", a[j]);
        a[j] = a[i];
}
```

Unlike the standard Fisher-Yates, I don't actually swap `a[i]` and `a[j]` here because `a[i]` will never be accssed again.

Sometimes we don't want to sample from an arbitrary array but from the first n natural numbers: $0, 1, \ldots, n-1$. This can be done by initializing the array to natural numbers then runs the above sampling algorithm.

```
for (i = 0; i < n; i++)
        a[i] = i;
```

This uses $O(n) + O(k) = O(n)$ time. If we use the data structure introduced in this text, it could be decreased to $O(k)$.

```
init();
for (i = 0; i < k; i++) {
        j = i + rand() % (n - i);
        printf("%d\n", get(j, j));
        set(j, get(i, i));
}
```

Remarks

The application to sampling is served as my solution to exercise 1.4 of [2].

The natural solution to initializing $O(n)$ elements in $O(1)$ is setting up an `isdirty[]` array. If the `i`-th element is dirty, set `isdirty[i]` to 1, otherwise 0. This solution relies on the memory-initialization values of the system to be zero and it needs $O(n)$ time for reset.

These downsides are not that important because (1) setting a restriction on not relying on memory-initialization values is more an intellectual game than practicability; (2) reset is a rare demand.

However, the data structure described in this text is a tremendous example of how simply a seemed impossible task can be solved algorithmically.

The former text said it uses a C-like pseudo language but you may notice that it's actually the real C programming language. The reason this text can't claim it's real C is that accessing uninitialized objects in C is an undefined behavior. It may not behave correctly with some compilers.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

[2] Jon Bentley. 1999. *Programming Pearls (2nd Ed.)*. Addison-Wesley.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms Third Edition*. Mit Press.