CS100 Recitation 6

Dynamically Expanding Storage

GKxx

March 28, 2022

Motivation

- Store a compile-time-determined amount of data
- Store a runtime-determined amount of data
- Store an unknown amount of data

Motivation

- Store a compile-time-determined amount of data: Built-in arrays.
- Store a runtime-determined amount of data: Allocate memory on heap (`malloc`, `free`, etc.).
- Store an unknown amount of data?

Motivation

- Store a compile-time-determined amount of data:
  Built-in arrays.
- Store a runtime-determined amount of data:
  Allocate memory on heap (`malloc`, `free`, etc.).
- Store an unknown amount of data?
  - Suppose we want to create a list by appending n elements one-by-one, as in `Python`...

Motivation

- Store a compile-time-determined amount of data:
  Built-in arrays.
- Store a runtime-determined amount of data:
  Allocate memory on heap (`malloc`, `free`, etc.).
- Store an unknown amount of data?
  - Suppose we want to create a list by appending n elements one-by-one, as in `Python`...
  - We need some kind of storage that can dynamically grow.

What can we do?

- We can allocate a specific number of bytes of memory on heap.
- We **cannot** specify the exact location of the memory allocated. (Why?)

A Basic Idea

Suppose we have stored n elements in some contiguous memory `p[0], ..., p[n-1]`. When the $(n+1)$-th element x comes...

- We cannot force the system to allocate the space at `p[n]`.

A Basic Idea

Suppose we have stored n elements in some contiguous memory `p[0], ..., p[n-1]`. When the $(n+1)$-th element x comes...

- We cannot force the system to allocate the space at `p[n]`.
- Naive idea:
  1. Allocate another block of memory `q[0], ..., q[n]` that can contain $n+1$ elements.
  2. Copy the original n elements to the new place.
  3. Place x at `q[n]`.

A Basic Idea

Suppose we have stored n elements in some contiguous memory `p[0], ..., p[n-1]`. When the $(n+1)$-th element x comes...

- We cannot force the system to allocate the space at `p[n]`.
- Naive idea:
  1. Allocate another block of memory `q[0], ..., q[n]` that can contain $n+1$ elements.
  2. Copy the original n elements to the new place.
  3. Place x at `q[n]`.
  4. Are we done?

A Basic Idea

Suppose we have stored n elements in some contiguous memory `p[0], ..., p[n-1]` **(dynamically allocated)**. When the $(n+1)$-th element x comes...

- We cannot force the system to allocate the space at `p[n]`.
- Naive idea:
  1. Allocate another block of memory `q[0], ..., q[n]` that can contain $n+1$ elements.
  2. Copy the original n elements to the new place.
  3. Place x at `q[n]`.
  4. `free(p)!`

A Basic Idea

```
int *new_data
    = (int *)malloc(sizeof(int) * (n + 1));
for (int i = 0; i < n; ++i)
  new_data[i] = data[i];
new_data[n] = x;
free(data);
data = new_data;
```

A Basic Idea

```c
int *new_data
    = (int *)malloc(sizeof(int) * (n + 1));
for (int i = 0; i < n; ++i)
  new_data[i] = data[i];
new_data[n] = x;
free(data);
data = new_data;
```

Question

How many times of copying will happen if we append n elements one-by-one?

Reduce Copying

The number of times of copying that will happen is

$$\sum_{i=1}^{n}(i-1)=\frac{n(n-1)}{2},$$

which is quadratic in n. (Time complexity: $O(n^2)$)

Reduce Copying

The number of times of copying that will happen is

$$\sum_{i=1}^{n}(i-1)=\frac{n(n-1)}{2},$$

which is quadratic in n. (Time complexity: $O(n^2)$)

- What if we allocate more space each time?

Reduce Copying

The number of times of copying that will happen is

$$\sum_{i=1}^{n}(i-1)=\frac{n(n-1)}{2},$$

which is quadratic in n. (Time complexity: $O(n^2)$)

- What if we allocate more space each time?
- If we allocate space for $2n$ elements, we don't need to copy anything when appending the $(n+1)$-th, $(n+2)$-th, ..., $2n$-th elements.

Reduce Copying

The number of times of copying that will happen is

$$\sum_{i=1}^{n}(i-1)=\frac{n(n-1)}{2},$$

which is quadratic in n. (Time complexity: $O(n^2)$)

- What if we allocate more space each time?
- If we allocate space for $2n$ elements, we don't need to copy anything when appending the $(n+1)$-th, $(n+2)$-th, ..., $2n$-th elements.
  - $2n$ and n are not so different for computers. Don't worry!

A Better Way

If we append $n = 2^m$ elements one-by-one, the number of times of copying is

$$\sum_{i=0}^{m-1} 2^i = 2^m - 1 = n - 1,$$

which is linear in n.

- This idea is adopted in the C++ `vector` library.

Question

Can we do better than linear time?