Lectures: Data Structures

Programming Systems and Tools

State University of New York at Binghamton

April 11, 2024

BINGHAMTON UNIVERSITY | THOMAS J. WATSON COLLEGE OF ENGINEERING AND APPLIED SCIENCE

Vectors

Motivation

Suppose we have a mutable data set with the following requirements:

- Varies in size from 10,000 to 10,000,000 elements
- Requires random access

Can we use a typical C array?

Vectors: Dynamic Arrays

- Dynamic arrays are resized as needed using a predefined algorithm
- Vectors are an *abstract data type*(ADT) implementing the dynamic array
- Functions just like a standard array, except it automatically grows
- Exist in other languages:
  - `std::vector` in C++,
  - `java.util.ArrayList<E>` in Java,
  - `list()` in Python

Vectors: Dynamic Arrays

Pros

- Allocates memory only as needed
- Allows dynamically-sized arrays
- Resizing penalty can be *amortized*

Cons

- Memory and performance intensive
- Still wastes space, particularly at larger array sizes
- Members must be a single type (not a "bag")

Definition

We will implement struct Vector using this definition:

```
typedef uint64_t Data;
typedef struct Vector {
  size_t size;      // index of last element in the array
  size_t capacity; // the total number of elements that can be stored
  Data *array;      // pointer to the storage space
} Vector;

Vector *newVector(size_t size);
void deleteVector(Vector *v);
int printVector(const void *v);

Data *at(Vector *v, size_t index);
size_t capacity();
void clear(Vector *v);
Data *find(Vector *v, Data d);
Data *insert(Vector *v, Data d, size_t idx);
Vector *resize(Vector *v, size_t count);
Data *remove(Vector *v, size_t index);
```

Initializing Vectors

```
Vector *newVector(size_t size);
```

```
1 v ← (Vector *) malloc(sizeof(Vector))
2 v.capacity ← size
3 v.size ← 0
4 v.array ← (Data *) malloc(sizeof(Data) * v.capacity)
5 Use memset to zero-initialize the array storage
6 return v
```

Inserting into Vector

```
Data *insert(Vector *v, Data d, size_t idx);
```

- There is enough storage space for the index:
  1. v.array[index] $\leftarrow$ d
  2. if $index > v.len$ v.len $\leftarrow$ index
- There is not enough storage space for the index:
  1. Resize the array using `resize()`
  2. v.array[index] $\leftarrow$ d
  3. if $index > v.len$, v.len $\leftarrow$ index

Resizing the Vector

```
Vector *resize(Vector *v, size_t count);
```

We will eventually need to grow our dynamic array's storage space. Two strategies:

- Every time we add, we *incrementally* expand the array (e.g. if size was 10, we resize to 11, 12, 13, etc.).
    - Keeps array exactly large enough for current needs
    - Growth operations require heap reallocation on every insertion ... no amortization!
- If length equals the size, we *geometrically* increase the size to the next power of two (e.g. if size was 10, we resize to 16, 32, 64, etc.)
    - Array will outsize the current needs, particularly at large array sizes.
    - Growth operations can be amortized

Resizing the Vector

What happens if the size argument is less than v.size?

- Allow the resize, but some data is lost
- Disallow the resize, return a pointer to the original Vector
- Allow `realloc()` default behavior:
  - The contents will be unchanged in the range from the start of the re- gion up to the minimum of the old and new sizes
  - From the minimum size (new capacity) to the maximum size (old capacity), the data is not copied

Resizing the Vector

```
Vector *resize(Vector *v, size_t count);
```

1. v.array ←`realloc(v.array, v.size * 2 );`
   - If successful (v.array is not NULL):
     1. temp ←v.capacity
     2. v.capacity ←v.capacity * 2
     3. Zero initialize from temp to v.capacity using `memset`
   - If not successful (i.e. `realloc()` fails):
     1. Return a void pointer. It is the caller's responsibility to check this condition.

Reallocation in the C Standard Library

```c
#include <stdio.h>
void *realloc( void *ptr, size_t new_size );
```

`realloc()` reallocates the given area of memory.

- It must be previously allocated by malloc(), calloc() or realloc() and not yet freed with a call to free() or realloc().
- Reallocation happens by one of two strategies:
  - Expanding or contracting the existing area pointed to by ptr, if possible. The contents of the area remain unchanged up to the lesser of the new and old sizes. If the area is expanded, the contents of the new part of the array are undefined.
  - Allocating a new memory block of size new_size bytes, copying memory area with size equal the lesser of the new and the old sizes, and freeing the old block.
- If there is not enough memory, the old memory block is not freed and null pointer is returned.

Experiment: Using realloc()

1. Create a file **realloc.c**
2. Allocate a 6 byte-wide character buffer on the heap using `malloc()`.
3. Use `strcpy()` to copy the string “Hello” into the buffer
4. Print the address of the buffer, and the string in the buffer.
5. Use `realloc()` to change the size of the original buffer to 14 bytes.
6. Use `strcat()` to append the string “ World!” to the original buffer.
7. Again, print the address of the buffer and the string in the buffer.
8. Call `realloc()` a second time, this time with size 0. Do not call free.
9. Compile the program and run it in Valgrind. Is there a memory leak?

Reallocation in the C Standard Library

```
#include <stdio.h>  // printf
#include <stdlib.h> // malloc, realloc,free
#include <string.h> // strcpy,strcat
int main () {
  char *str;

  /* Initial memory allocation */
  str = (char *) malloc(6);
  strcpy(str, "Hello");
  printf("String = %s,  Address = %p \n", str, str);

  /* Reallocating memory */
  str = (char *) realloc(str, 14);
  strcat(str, " World!");
  printf("String = %s,  Address = %p \n", str, str);

  /* Undefined freeing */
  str = (char *) realloc(str, 0);
  //free(str);
  return(0);
}
```

Reading from a Vector

```
Data *at(Vector *v, size_t index);
```

Reading from vectors is just like using the indexing operator on arrays, except:

- Instead of returning the actual value, we return a pointer; this allows us to have a failure state...
- We must check bounds: if the index is larger than the bounds of the vector, return NULL
- Is it a problem to have an index less than the capacity, but larger than the size?
  - → *No!* . . . We're not causing a buffer overread (the memory's allocated, right?)
  - → *Yes!* . . . We're violating the intent of the data structure
  - → **We'll consider it a problem and return a nullptr.**

Removing from a Vector

```
Data *remove(Vector *v, size_t index);
```

1. Actual deletion by deallocation of that part of vector memory
   - Expensive and must reallocate space carefully
   - What if item deleted is not last item?
2. Set v.array[index] to an "empty" value
   - Most efficient option
   - Doesn't cause problems with memory management or external references
   - Need to reserve an "empty" value ... no "None" value in C
3. Remove the item and move items at higher indicies up to size-1.
   - Most accurately models behavior of bounds checked buffers
   - Could cause expensive iteration to move items leftward!
   - Dangerous to mutate storage while also iterating across it!

Removing from a Vector

What if another variable outside the struct has a reference for an index into the array?

- Vector cannot stop unauthorized access into the array
- Unauthorized access might not cause `.size` or `.capacity` to be properly updated

Clearing a Vector

```
void clear(Vector *v);
```

Clearing the vector is just iterating across the entire vector, and removing the value at each index:

1. For index i in range 0 ... v.size - 1: v.array[i] $\leftarrow 0$
2. v.size $= 0$

- Alternative option: memset from index 0 to size . . . Much faster!
- Should we resize the array also? **We do not. Why?**
  - Relies on memory allocator not causing a failure
  - Expensive to call realloc()
  - Don't quietly change state on the user.