CS 261

Dynamic Arrays
Introduction
by
Tim Budd
Ron Metoyer
Sinisa Todorovic
Arrays

• Core data structure
• Example:

```c
double arrayBag[100];

struct Students{
    int count;
    char initials[2];
};
```
Arrays -- Positives and Negatives

• Positives:
  – Simple
  – Each element accessible in $O(1)$

• Negatives:
  – Size must be fixed when created
  – What happens when the program later requires more space?
Dynamic Arrays

- in Java called: Vector, ArrayList

- Our goal: Hide memory management details behind a simple API

- Each element is still accessible in O(1)

- But a dynamic array can change capacity
Size and Capacity

- **Data**: The area used to store data.
- **Size**: The number of elements currently occupied by data.
- **Capacity**: The total number of elements available for storage.
- **Unused Elements**: The elements that are not in use.
Size and Capacity

• Size:
  – Current number of elements
  – Managed by an internal data value

• Capacity:
  – number of elements that a dynamic array can hold before it must resize
Dynamic Array

data = 
size = 10
-cap = 16

Size (= size)

Capacity (= cap)
Adding an Element

- Increment the size

- Put the new value at the end of the dynamic array
Adding an Element

• What happens when size == capacity?

• Must:
  – reallocate new space
  – copy all data values to the new space
  – hide these details from the user
Reallocate and Copy

Before reallocation:

\[
\begin{align*}
data &= \begin{bmatrix} 8 \end{bmatrix} \\
size &= 8 \\
cap &= 8
\end{align*}
\]

After reallocation:

\[
\begin{align*}
data &= \begin{bmatrix} 8 \end{bmatrix} \\
size &= 8 \\
cap &= 16
\end{align*}
\]

Must allocate new (larger) array and copy valid data elements.
Reallocate and Copy

Before reallocation:

- data = [ ]
- size = 8
- cap = 8

After reallocation:

- data = [ ]
- size = 8
- cap = 16

DO NOT forget to free up the old array
Adding an Element to Middle

• May also require reallocation
  – When?

• Will ALWAYS require elements to be moved up to make space
Adding an Element to Middle

Loop from the end while copying data!

Add at $\text{id}_x \rightarrow$

Before

After
Adding an Element to Middle

- Complexity?
- \( O(n) \) in the worst case
Removing an Element

• Requires “sliding over” to delete the value
Removing an Element

• Remove also requires loop. This time should it be from the end?

Before

Remove \textbf{idx} \rightarrow

After
Removing an Element

- Complexity?
- $O(n)$ worst case
Interface View of Dynamic Arrays
General Purpose Dynamic Array

- Define TYPE as symbolic preprocessor constant. Default double.

- Requires recompiling the source code if new types are needed
#ifndef _DyArray_H
#define _DyArray_H

#define TYPE double
#define TYPE_SIZE sizeof(TYPE)

#define LT(a, b) (a < b)
#define EQ(a, b) (a == b)
...

/* Rest of dynarr.h on next slide */

#endif
interface (continued)

```c
struct dyArr {
    TYPE * data; /* Pointer to data array */
    int size;    /* Number of elements */
    int capacity; /* Capacity of array */
};

/* Rest of dynarr.h on next slide */
```
/* function prototypes */

void initDynArr (struct dyArr *da, int cap);

void freeDynArr (struct dyArr *da);

void addDynArr (struct dyArr *da, TYPE val);

TYPE getDynArr (struct dyArr *da, int idx);

void putDynArr (struct dyArr *da, int idx, TYPE val);

int sizeDynArr (struct dyArr *da);

void _dyArrDoubleCapacity (struct dyArray * da);
Implementation View of Dynamic Arrays
void initDynArr (struct dyArr *da, int cap) {
    assert (cap >= 0);
    da->capacity = cap;
    da->size = 0;
    da->data = (TYPE *) malloc(da->capacity * sizeof(TYPE));
    assert (da->data != 0);
}
freeDynArr -- Clean-up

void freeDynArr (struct dyArr * da)
{
    assert (da != 0);
    free (da->data); /*free entire array*/
    da->capacity = 0;
    da->size = 0;
}
int sizeDynArr (struct dyArr * da){
    return da->size;
}

Size
Get the Value at a Given Position

```c
TYPE getDynArr (struct dyArr *da, int idx);
{
    assert((sizeDynArr(da) > idx) && (idx >= 0));
    return da->data[idx];
}
```
Add a New Element

```c
void addDynArr (struct dyArr * da, TYPE val){
    if (da->size >= da->capacity)
        _dyArrDoubleCapacity(da);
    da->data[da->size] = val;
    da->size++;
}
```
Double the Capacity

Before reallocation:

- data = [8]
- size = 8
- cap = 8

Must: allocate space
copy data
free old space

After reallocation:

- data = [8]
- size = 8
- cap = 16
void _dyArrDoubleCapacity (struct dyArray * da) {
    TYPE * oldbuffer = da->data;
    int oldsize = da->size;
    int i;
    initDynArr (da, 2 * da->capacity);
    for (i = 0; i < oldsize; i++)
        da->data[i] = oldbuffer[i];
    da->size = oldsize;
    free(oldbuffer);
}
Next Class

How to implement

– Stack

– Bag

by using Dynamic Array