CSCI203

Week 3 – Lecture B

Some Tricks With Arrays

- Virtual initialization of arrays
- Compact string storage

Array Initialization...

...And How To Avoid It.

Virtual Initialization.

- Often, we need to know whether an array element contains valid data.
- When we create an array it is usually filled with random garbage (whatever the memory used last contained).
  - **`data: array[1..length] of stuff`**
- A standard technique is to initialize arrays to some special value, the *sentinel* value, when they are created:
  - ```
    for i = 1 to length
     data[i] = sentinel
    rof
    ```
- We can now read data into some of the array:
  - ```
    for i = 1 to n
     read j, k
     data[j]=k
    rof
    ```

Virtual Initialization

- With this done, we can look at an array element to determine whether it contains valid data.
  - ```
    if data[i] == sentinel then
       data[i] is not initialized
    else
       data[i] contains valid data
    fi
    ```
- The problem with this approach is the cost of initialization.
    ```

Virtual Initialization

- If we are going to use pretty much all of the array, the cost of initializing it is acceptable as we are doing work which will be of value later in the program.
- If we are only going to access a few of the elements of the array the story is different.
- In the worst case, the cost of initializing the array, O(*length*), can dominate the cost of using the array, O(*n*)
- NOTE: in this example *length* is the size of the array, *n* is the number of elements we actually use.

The Problem

- What we need to do is find a method to determine if an array element is valid without initializing all of the array.
- Is there some way we can determine whether an array element contains valid data *without* initializing every element?
- Perhaps we could keep track of which elements contain valid data by doing something only when data is first stored into a given element.
- This would give us an initialization mechanism that required O(*n*) operations.

A Simple Solution

- Our first approach is to use a second array, valid[], which is the same size as our data[] array to track which entries in data[] contain valid values.
  - ```
    data: array[1..length] of stuff
    valid: array[1..length] of boolean
    ```
- Our data storage loop becomes:
  - ```
    for i = 1 to n
       read j, k
       data[j]=k
       valid[j]=true
    rof
    ```

A Simple Solution

- We can now use the values in valid[] to determine whether an entry in data[] contains good data or not:
  - ```
    if valid[i] == true then
       data[i] contains valid data
    else
       data[i] is not initialized
    fi
    ```
    ```

An Example

- We start with both arrays containing unknown values:
  - data[]

?	?	?	?	?	?	?	?	?	?

  - valid[]

?	?	?	?	?	?	?	?	?	?

- Now we set data[4] to some valid value:
  - data[]

?	?	?	42	?	?	?	?	?	?

  - valid[]

?	?	?	true	?	?	?	?	?	?

- Because valid[4]==true we know the contents of data[4] contains valid data.
- Anything wrong with this?

A Simple (WRONG) Solution

- What value did we use as the sentinel?
- Just as the data[] array contains random values when it is allocated, so does the valid[] array.
- This means that some entries in valid[] may have a value equal to **`true` by chance...**
- **... before** we store any values into data[].
- This is a BAD THING™
- Even if we use a different sentinel this can still happen.

This is harder than it looks...

... It might be time for a pause.

A More Complex Solution

- Clearly, to get this fixed, we need to try harder.
- Let us examine a more complicated approach.
- This time we will use two arrays to keep track of whether an element of data[] contains a valid value.
  - **`data: array[1..length] of stuff`**
    `forward: array[1..length] of integer`
    `backward: array[1..length] of integer`
- We also need a counter to keep track of how many elements of data[] contain valid data.
  - **`valid_count: integer`**

A More Complex Solution

- Our data storage loop now looks like this:
  - ```
    valid_count=0
    for i = 1 to n
       read j, k
       valid_count=valid_count+1
       data[j]=k
       forward[valid_count]=j
       backward[j]=valid_count
    rof
    ```
    ```

A More Complex Solution

- We can determine whether an element of data[] is valid as follows:
  - **if backward[i]>0 and backward[i]<=valid_count and forward[backward[i]]==i then**
    data[i] contains valid data
    else
    data[i] is not initialized
    fi

An Example

- We start with all three arrays containing random data:
  - valid_count: 0

data	?	?	?	?	?	?	?	?	?	?
forward	?	?	?	?	?	?	?	?	?	?
backward	?	?	?	?	?	?	?	?	?	?

- Now we set data[4] to some valid value
  - valid_count: 1

data[]	?	?	?	42	?	?	?	?	?	?
forward[]	4	?	?	?	?	?	?	?	?	?
backward[]	?	?	?	1	?	?	?	?	?	?

Correct?

- If we look at the value of backward[4] we see it is equal to 1,which is > 0 and <= valid_count.
- Also, forward[backward[4]]=forward[1]=4
- Can this happen without element 4 being initialized?
- If we want to check some other element, say data[7], can we conclude, incorrectly that it has been initialized?
- Let us look at this in more detail.

Correct?

- In order to pass the first part of our test for validity, backward[7] must be <=valid_count.
- In this case, this means that backward[7] must equal 1.
- However, forward[backward[1]] is equal to 4, not 7.
- Therefore data[7] does not contain valid data.
- Our data tracking is now correct.

Compact String Storage...

...Without Compression.

Compact String Storage

- Let us assume that we need to store a large number, N, of text strings.
- Let us also assume that the strings vary significantly in length, with a maximum length of L:
- How can we store these string so that:
  - We use a minimum amount of storage;
  - We can access any string quickly and efficiently;
  - We avoid the overhead of dynamic memory.
- Let us look at some alternatives.

Option 1

- Our first option might be to use an array of strings:
  - text: array[1..N] of string
- Minimum amount of storage used?
  - Yes – each string is exactly as long as we need.
- Access any string quickly and efficiently?
  - Yes.
- Avoids dynamic memory?
  - No – strings are dynamic.

Option 2

- How about a doubly dimensioned array of characters.
  - text: array[1..N,1..L] of character
- Minimum amount of storage used?
  - No – we use N×L characters which can be far more than we really need.
  - What if all of the strings are 1 character long except for one which is 1,000,000 characters long?
- Access any string quickly and efficiently?
  - Yes.
- Avoids dynamic memory?
  - Yes.

Option 3

- Our next approach is a bit more complex and involves three arrays:
  - text[], a character array which stores the strings packed tightly together;
  - start[], an integer array which stores the starting position of each string in text[];
  - length[], an integer array which contains the length of each string in text[].
- To do this we need to know the average length of the words in the set of strings.
  - Let us call this, average string length, A.

An Example

- Let us assume that we want to store each word of a text file in this way.
  - "The quick brown fox jumps over the extraordinarily lazy dog"
  - The text contain 10 words so we will set N to 10
  - The average length is 5 so we will set A to 5.
- Our data structure will look like this:
  - ```
    text: array[1..50] of character
    start: array[1..10] of integer
    length: array[1..10] of integer
    ```
    ```

An Example

- The stored data looks like this:
  - text:

T	h	e	q	u	i	c	k	b	r
o	w	n	f	o	x	j	u	m	p
s	o	v	e	r	t	h	e	e	x
t	r	a	o	r	d	i	n	a	r
i	l	y	l	a	z	y	d	o	g

  - start:

1	4	9	14	17	22	26	29	44	48

  - length:

3	5	5	3	5	4	3	15	4	3

Option 3

- Let us evaluate this option:
- Minimum amount of storage used?
  - Nearly – we use N×A characters plus 2xN integers which is a bit more than the minimum required.
- Access any string quickly and efficiently?
  - Yes. The i^{th} entry is text[j..k] where
  - j=start[i]
  - k=start[i]+length[i]-1
- Avoids dynamic memory?
  - Yes.

Option 3a

- If we store the end position of each string rather than its length we obtain a slight gain in accessing the strings:
- Text[] and start[] remain unchanged but we replace length[] with an integer array end[].
- In our example
  - end[]

3	8	13	16	21	25	28	43	47	50

- This uses no more memory and we can now access the i^{th} string as text[start[i]..end[i]], a slight improvement

Option 3b

- If we are observant we will see that end[i] = start[i+1]-1.
- We can use this fact to save a bit more memory:
- Eliminate the length[] or end[] arrays entirely and make the start array longer by one element:
  - The last element of start[] now contains the position where the next string would start if there was one.
  - Start[]

1	4	9	14	17	22	26	29	44	48	51

- The i^{th} string is now text[start[i]..start[i+1]-1]

String Pool

- The various data structures shown in option 3 are collectively known as *string pools*.
- Each provides some small advantages and disadvantages but, broadly speaking, they are all about as efficient as each other.
- For the storage of large numbers of strings which vary widely in length, the string pool provides a very attractive alternative.