Week 03 - Lecture 2 Slides

Dr. Yeganeh Bahoo

Created: 2023-09-19 Tue 23:13

Table of Contents

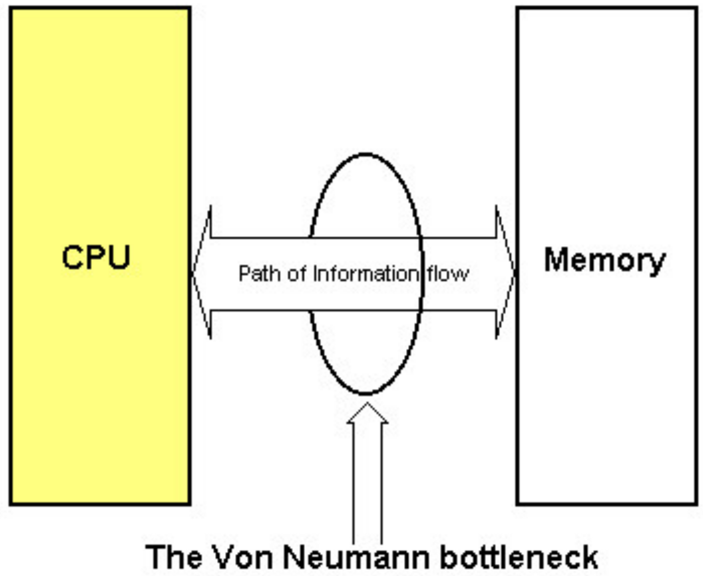
Lecture 2: Essential Data Structures

Learning objectives: By the end of this lecture you should be able to

1. Describe the computational constraints imposed by the von Neumann's architecture with respect to access to data
2. Describe the basic differences between contiguous and linked data structures
3. Write programs that use structs, static and dynamic one-dimentional arrays

The von Neumann's bottleneck

- Your computer's Central Processing Unit (CPU) only has access to data stored in registers
  - **numbers** (intergers, float, and characters)
  - **pointers** (memory addresses) – the number of bits in a register defines the maximum memory address (e.g., 32-bit architecture – 4G bytes of addressable memory)
- But the bulk of the data is stored in memory

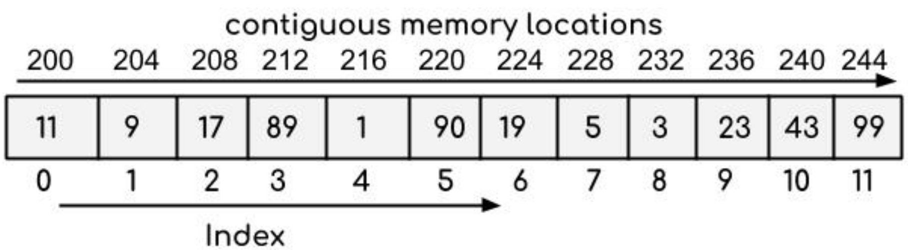


Two ways to store data in memory

- **Contiguous structure**: occupies a single chunk of memory and its contents are stored in adjacent memory blocks
- **Linked structure**: doesn't occupy a contiguous block of memory

---

Contiguous structure

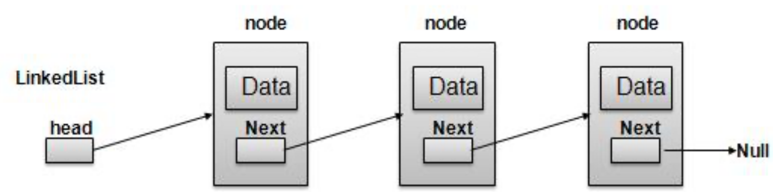


Enables more efficient data access

Takes constant time to access an element (address of the first + offset)

Ex: arrays and structs

Linked structure



Data access is less efficient

Cost (in time) depends on size of structure

Ex: lists, trees, graphs

Contiguous structures: The *tuple* abstract data structure

- All programming languages use such abstraction, calling it "record" or "object"
- In lisp, this abstraction is called a **structure**

A **structure** is a structured data type with named *slots*, each holding an primitive object or another structure

We define a structure using lisp's defstruct macro.

Example: let's define a tuple/structure representing information about a movie

```
CL-USER> (defstruct movie
           title  director year type)
  MOVIE
```

Once you define a structure, Lisp automatically creates the following tools for you:

- a constructor, named `MAKE-`$structureName$, for creating instances of the structure
- accessors, named $structureName$-$slotName$, for accessing the structure's slots

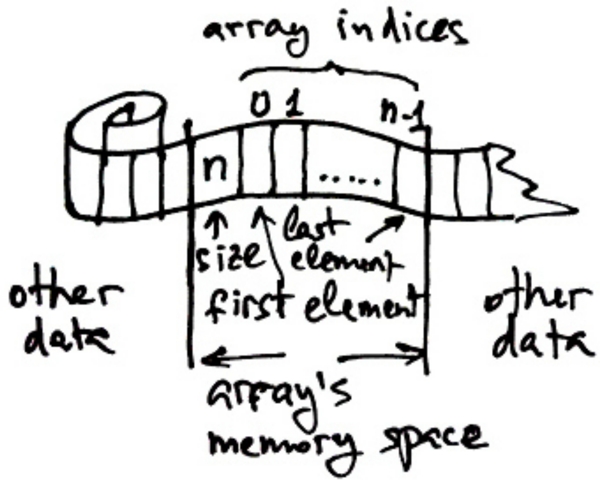
```
CL-USER> (make-movie)
#S(MOVIE :TITLE NIL :DIRECTOR NIL :YEAR NIL :TYPE NIL)
CL-USER>  (let ((movie (make-movie :title "Blade Runner" :director "Ridley Scott")))
            (setf (movie-year movie) 1982)
            (values (movie-title movie)
                    (movie-director movie)
                    (movie-year movie)))
"Blade Runner"
"Ridley Scott"
1982
```

Homework

Try the following on the REPL or on a lisp program file:

- Define a structure that represents a computer science course.
- Create a structure, bind it to a variable, change the contents of its slots.

Arrays



Alongside structs, it is the most basic data structure and favoured choice for implementing algorithms

```
CL-USER> #(3 4 5 6)
#(3 4 5 6)
CL-USER> (make-array 3)
#(0 0 0)
CL-USER> (make-array 3 :initial-element 1/2)
#(1/2 1/2 1/2)
```

Arrays: example

Let's prototype a movie database using a tuple and array.

```
(defstruct movie
  title  director year type)

(defparameter *size* 10) ;; Size of the db

(defvar *db*)

(setf *db* (make-array *size*  :initial-element nil))

(defun add-movie (m)
  "Adds a movie to the database and returns true. Otherwise, returns NIL.
   Notice: adds duplicates!!!"
  (dotimes (i *size*)
    (unless (aref *db* i)
      (setf (aref *db* i) m)
      (return t))))


CL-USER> (add-movie (make-movie :title "Blade Runner"))
T
CL-USER> (aref *db* 0)
#S(MOVIE :TITLE "Blade Runner" :DIRECTOR NIL :YEAR NIL :TYPE NIL)
```

```
CL-USER> (add-movie (make-movie :title "The Big Lebowsky"))
T
CL-USER> *db*
#(#S(MOVIE :TITLE "Blade Runner" :DIRECTOR NIL :YEAR NIL :TYPE NIL)
  #S(MOVIE :TITLE "The Big Lebowsky" :DIRECTOR NIL :YEAR NIL :TYPE NIL) NIL NIL
  NIL NIL NIL NIL NIL NIL)
```

Exercise

Complete function IN-DB? that returns the movie's tuple if the movie title is in the database; otherwise it returns NIL (false).

```
(defun in-db? (title)
  "Returns true if movie title is in the database; otherwise returns NIL"
  (dotimes (i *size*)
    (when (and (typep (aref *db* i) 'movie)  ; verifies the i-th item is a movie tuple
               (equal (movie-title ...) ...))
      (return ...))))
```

Notice: if the element is not found, the DOTIMES will return NIL. Because there is no *result-form* in its preamble.

```
CL-USER> *db*
#(#S(MOVIE :TITLE "Blade Runner" :DIRECTOR NIL :YEAR NIL :TYPE NIL))
CL-USER> (in-db? "Parasite")
NIL
CL-USER> (in-db? "Blade Runner")
#S(MOVIE :TITLE "Blade Runner" :DIRECTOR NIL :YEAR NIL :TYPE NIL)
```

Solution

```
(defun in-db? (title)
  "Returns true if movie title is in the database; otherwise returns NIL"
  (dotimes (i *size*)
    (when (and (typep (aref *db* i) 'movie)
               (equal (movie-title (aref *db* i)) title))
      (return (aref *db* i)))))
```

Array operations: algorithmic complexity

- **Memory space**: is the minimum possible; some meta-data for array size
- **Access to *i*-th element**: $O(1)$, as it is an offset from the first element.
- **Search** can be done in $O(\log n)$ using binary search
- **Insertion/Deletion**: this is the major drawback, because arrays are static structures; require $O(n)$ time

  Despite this drawback, arrays should be the default choice for most algorithms, specially if you know the size of the result.

  - But what if you don't and still want to use arrays?
    - Most programming languages provide some form of "dynamic arrays"

When you don't know in advance the amount of storage you'll need

- Dynamic vector: vector size is automatically expanded proportionally to its current size

```
(let ((vec (make-array 0 :fill-pointer t  ; set initially to 0
                         :adjustable t)))
  (dotimes (i 10)
    (vector-push-extend i vec)
    (describe vec)))
```

Notice: `(vector-push-extend i vec)` stores `i` in `(aref vec fill-pointer)` then updates `fill-pointer` by incrementing it by one, i.e., `(incf fill-pointer)`

Try the above function at your REPL.

Below is a digest of DESCRIBE's output showing the array elements and its actual (expanded) size.

```
#(0)
Fill-pointer: 1
Size: 1
#(0 1)
Fill-pointer: 2
Size: 2
#(0 1 2)
Fill-pointer: 3
Size 4
...
#(0 1 2 3 4 5)
Fill-pointer: 6
Size: 8
...
#(0 1 2 3 4 5 6 7 8 9)
Fill-pointer: 10
Size: 16
```