Data Structures – Week #1

Introduction

Goals

- We will learn methods of how to
  - (explicit goal) organize or **structure large amounts of data in the main memory (MM)** considering efficiency; i.e,
    - ***memory space*** **and**
    - ***execution time***
  - (implicit goal) gain additional experience on
    - what data structures to use for solving what kind of problems
    - programming

Goals continued…1

- **Explicit Goal**
  - We look for answers to the following question:

"*How do we store data in MM* such that

1. ***execution time*** grows as *slow* as possible with the growing size of input data, and
2. data uses up *minimum* ***memory space***?"

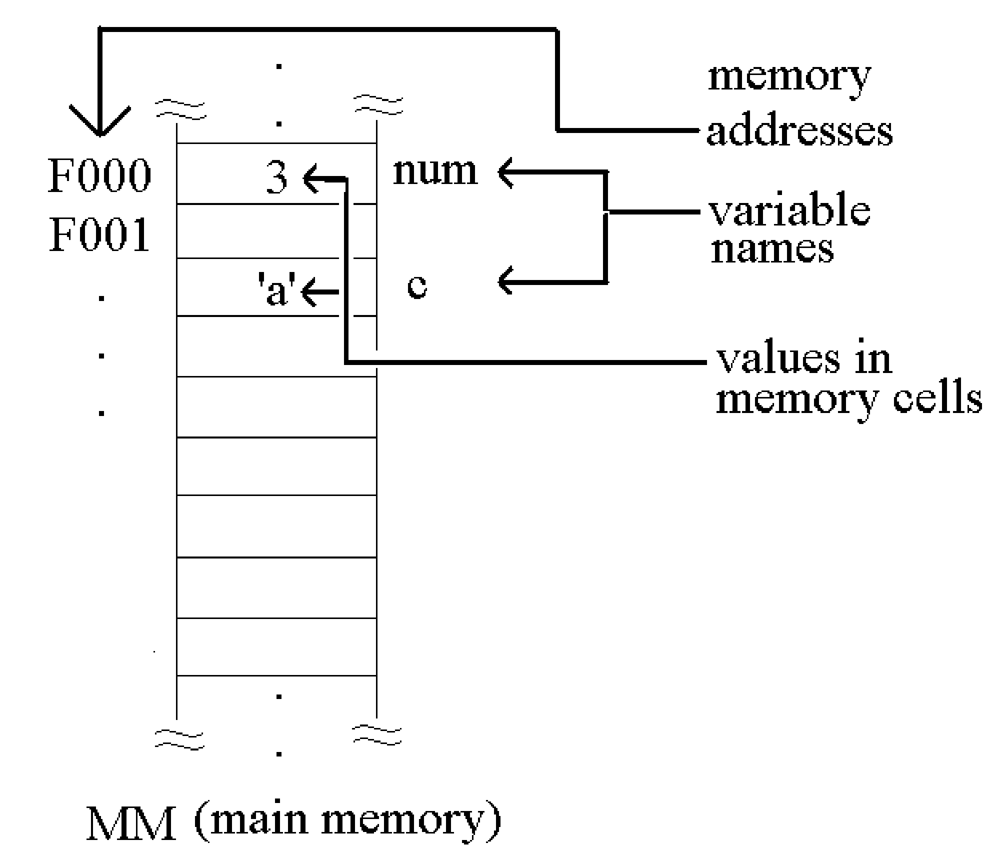
Goals continued…2

- As a tool to calculate the execution time of algorithms, we will learn the basic principles of **algorithm analysis**.
- To efficiently structure data in MM, we will thoroughly discuss the
  - ***static***, (arrays)
  - ***dynamic*** (structures using pointers)

  ways of *memory allocations*, two fundemantal

  implementation tools for data structures.

Representation of Main Memory



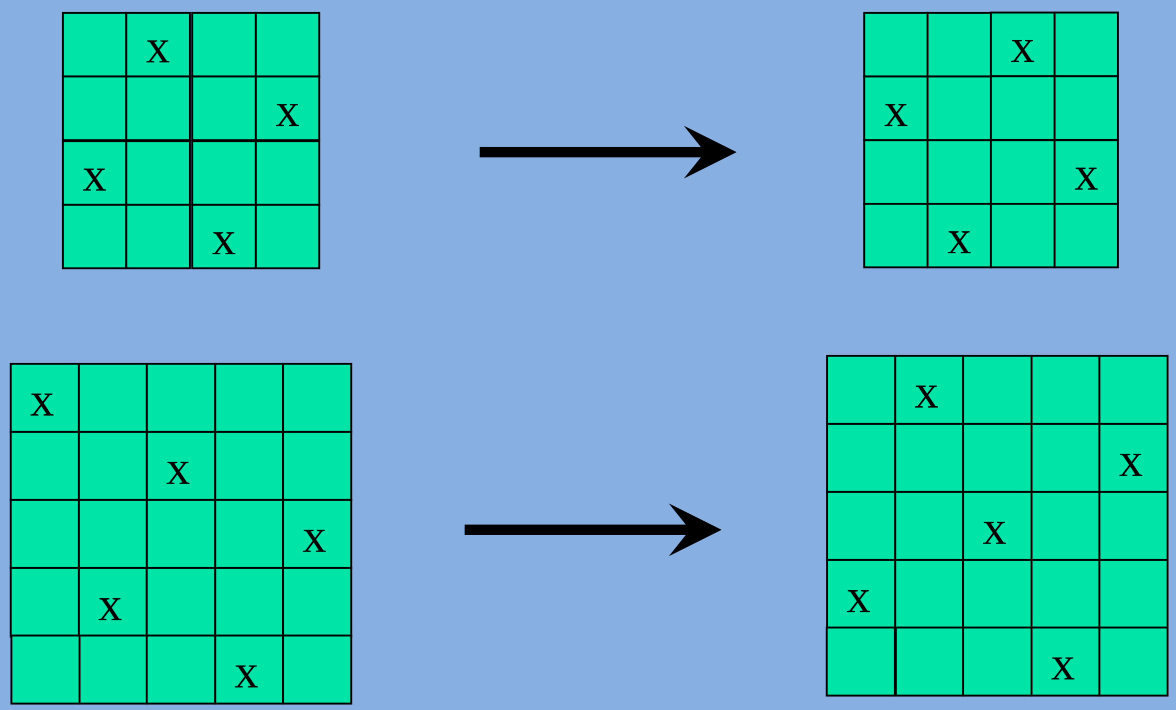
Examples for efficient vs. inefficient data structures

- 8-Queen problem
  - 1D array vs. 2D array representation results in saving memory space
  - Search for proper spot (square) using horse moves save time over square-by-square search

- Fibonacci series: A lookup table avoids redundant recursive calls and saves time

Examples for efficient vs. inefficient data structures

8-Queen problem (4-queen and 5-queen versions)



Examples for efficient vs. inefficient data structures

8-Queen problem (4-q and 5-q versions)

	x		
			x
x			
		x	

```
int a[4][4];
....
a[0][1]=1;
a[1][3]=1;
a[2][0]=1;
a[3][2]=1;
```

inefficient: more memory space (16 bytes for 4-q version) required

x				
		x		
				x
	x			
			x	

```
int a[5];
....
a[0]=0;
a[1]=2;
a[2]=4;
a[3]=1;
a[4]=3;
```

efficient: less memory space (5 bytes for 5-q version) required

Math Review

- Exponents

$$x^a x^b = x^{a+b}; \qquad \frac{x^a}{x^b} = x^{a-b}; \qquad (x^a)^b = x^{ab};$$

- Logarithms

$$y = x^a \Leftrightarrow \log_x y = a, \quad y > 0; \qquad \log_x y = \frac{\log_z y}{\log_z x}, \quad z > 0;$$

$$\log xy = \log x + \log y; \quad \log \frac{1}{x} = -\log x; \quad \log x^a = a \log x$$

Math Review

- Arithmetic Series: Series where the variable of summation is the base.

$$\sum_{i=1}^{k+1} i = \frac{k(k+1)}{2} + k + 1 = \frac{(k+1)(k+2)}{2};$$

$$\frac{k(k+1)}{2} + \frac{2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$$

Math Review

- Geometric Series: Series at which the variable of summation is the exponent.

$$\sum_{i=0}^{n} a^i = \frac{1-a^{n+1}}{1-a}, \quad 0<a<1; \quad \sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}, \quad a \in N^+ - \{1\};$$

$$\lim_{n\to\infty} \sum_{i=0}^{n} a^i = \frac{1}{1-a}, \quad 0<a<1;$$

$$s = \lim_{n\to\infty} \sum_{i=0}^{n} a^i = 1 + a + a^2 + a^3 + a^4 + \ldots = \frac{1}{1-a};$$

$$as = \lim_{n\to\infty} a\sum_{i=0}^{n} a^i = a + a^2 + a^3 + a^4 + \ldots = \frac{a}{1-a};$$

$$\Rightarrow s - as = s(1-a) = 1$$

Math Review

- Geometric Series…cont'd
- An example to using above formulas to calculate another geometric series

$$s = \sum_{i=1}^{\infty} \frac{i}{2^i};$$

$$s = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{i}{2^i} + \cdots$$

$$2s = 1 + \frac{2}{2} + \frac{3}{2^2} + \frac{4}{2^3} + \cdots + \frac{i}{2^{i-1}} + \cdots$$

$$s = 2s - s = 1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^i} + \cdots$$

$$s = \sum_{i=0}^{\infty} \frac{1}{2^i} = 2;$$

Math Review

- Proofs
  - Proof by Induction
    - Steps
      1. Prove the base case ($k=1$)
      2. Assume hypothesis holds for $k=n$
      3. Prove hypothesis for $k=n+1$
  - Proof by counterexample
    - Prove the hypothesis wrong by an example
  - Proof by contradiction ($A \Rightarrow B \Leftrightarrow \sim B \Rightarrow \sim A$)
    - Assume hypothesis is wrong,
    - Try to prove this
    - See the contradictory result

Math Review

- Proof examples (Proofs… cont'd)
- Proof by Induction
  - Hypothesis

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

  - Steps
    1. Prove true for n=1: $\sum_{i=1}^{1} i = 1$
    2. Assume true for n=k: $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$
    3. Prove true for n=k+1:

$$\sum_{i=1}^{k+1} i = \frac{k(k+1)}{2} + k + 1 = \frac{(k+1)(k+2)}{2};$$

$$\frac{k(k+1)}{2} + \frac{2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$$

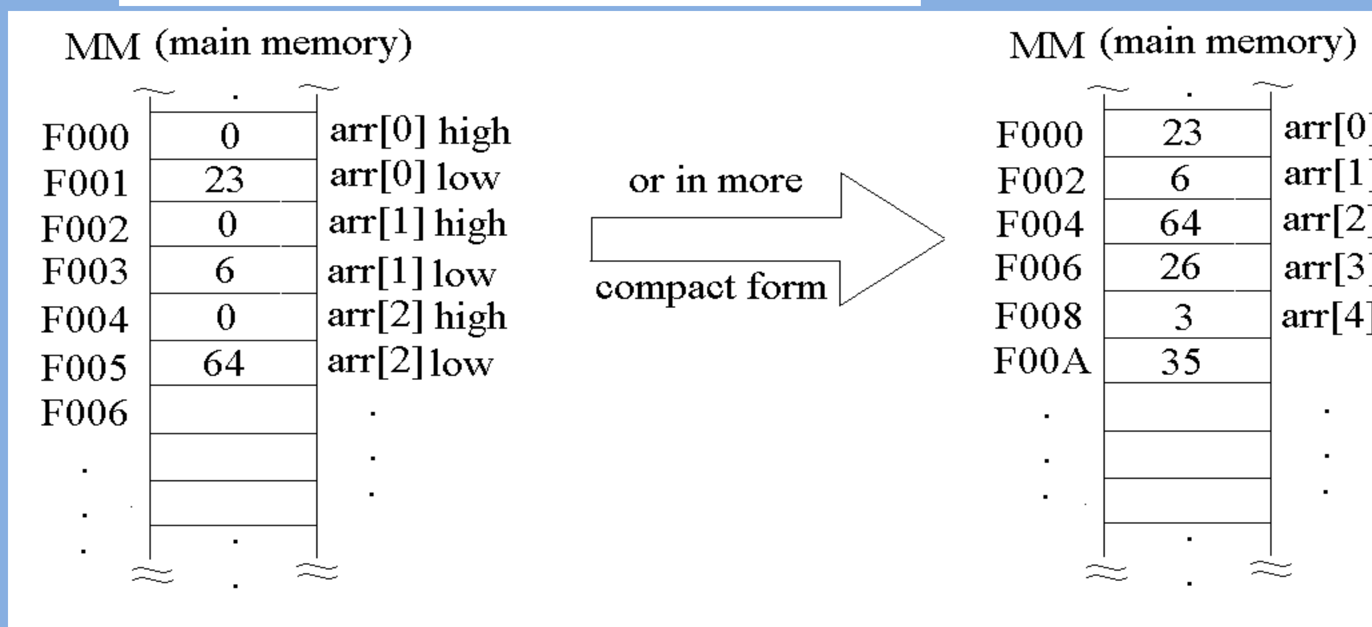
Arrays

- Static data structures that
  - represent **contiguous** memory locations holding **data of the same type**
  - provide ***direct access*** to data they hold
  - have a ***constant size*** determined up front (at the beginning of) the run time

Arrays… cont'd

- An integer array example in C
- int arr[12]; //12 integers

Index →	0	1	2	3	4	5	6	7	8	9	10	11	
Value	23	6	64	26	3	35	8	56	39	48	41	12	arr



Multidimensional Arrays

- To represent data with multiple dimensions, multidimensional array may be employed.
- Multidimensional arrays are structures specified with
  - the data value, and
  - as many indices as the dimensions of array
- Example:
  - `int arr2D[r][c];`

Multidimensional Arrays

$$\begin{bmatrix} m[0][0] & m[0][1] & m[0][2] & \cdots & m[0][c-1] \\ m[1][0] & m[1][1] & m[1][2] & \cdots & m[1][c-1] \\ m[2][0] & m[2][1] & m[2][2] & \cdots & m[2][c-1] \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ m[r-1][0] & m[r-1][1] & m[r-1][2] & & m[r-1][c-1] \end{bmatrix}$$

- ***m*: a two dimensional (2D) array with *r* rows and *c* columns**
- **Row-major representation: 2D array is implemented row-by-row.**
- **Column-major representation: 2D array is implemented column-first.**
- **In row-major rep., *m[i][j]* is the entry of the above matrix *m* at i+1st row and j+1st column. "i" and "j" are row and column indices, respectively.**
- **How many elements?** $n = r*c$ **elements**

Row-major Implementation

- ***Question***: How can we store the matrix in a 1D array in a ***row-major*** fashion or how can we map the 2D array *m* to a 1D array *a*?

Row-major Implementation

- Question: How can we store the matrix in a 1D array in a row-major fashion or how can we map the 2D array *m* to a 1D array *a*?

l-1 elements before the matrix representation

elements after the matrix representation

a	**...**	***m[0][0]***	**...**	***m[0][c-1]***	**...**	***m[r-1][0]***	**...**	***m[r-1][c-1]***	**...**
index: k $\longrightarrow$		***k=l***		***k=l+c-1***		***k=l+(r-1)c+0***		***k=l+(r-1)c+c-1***	

in general what is ***k*** in terms of ***l, i, j*** and ***c*** so we know ***m[i][j]=a[k]?***

Row-major Implementation

- Question: How can we store the matrix in a 1D array in a row-major fashion or how can we map the 2D array *m* to a 1D array *a*?

l elements

a	...	**m[0][0]**	...	**m[0][c-1]**	...	**m[r-1][0]**	...	**m[r-1][c-1]**	...

index: k $\longrightarrow$ ***k=l*** ***k=l+c-1*** ***k=l+(r-1)c+0*** ***k=l+(r-1)c+c-1***

In general**,** ***m[i][j]*** is placed at ***a[k]*** where ***k=l+ic+j*****.**

Hence, ***m[i][j] = a[l+ic+j]*****.**

Implementation Details of Arrays

1. *Array names are pointers* that point to the first byte of the first element of the array.
   a) double vect[row_limit];// vect is a pointer!!!
2. Arrays may be efficiently passed to functions using their ***name*** and their ***size*** where
   a) the name specifies the beginning address of the array
   b) the size states the bounds of the index values.
3. Arrays can only be copied element by element.

Implementation Details… cont’d

```
#define maxrow …;
#define maxcol …;
…
int main()
{
int minirow;
double min;
double probability_matrix[maxrow][maxcol];
… ;   //probability matrix initialized!!!
min=minrow(probability_matrix,maxrow,maxcol,&minirow);
…
return 0;
}
```

Implementation Details… cont'd

```
double minrow(double darr[][maxcol], int xpos, int ypos, int *ind)
{// finds minimum of sum of rows of the matrix and returns the sum
   // and the row index with minimum sum.
    double mn;
    …
    mn=<a large number>;
    for (i=0; i<=xpos; i++) {
        sum=0;
        for (j=0; j<=ypos; j++)
            sum+=darr[i][j];
        if (mn > sum) { mn=sum; *ind=i; }   // call by reference!!!
    }
    return mn;
}
```

Records (Structures)

- As opposed to **arrays** in which we keep data of the same type, we keep related data of various types in a **record**.
- Records are used to encapsulate (keep together) related data.
- Records are composite, and hence, user-defined data types.
- In C, records are formed using the reserved word "struct."

Struct

- We declare as an example a student record called “stdType”.
- We declare first the data types required for individual fields of the record stdType, and then the record stdType itself.

Struct… Example

```
enum genderType = {female, male}; // enumerated type declared…
typedef enum genderType genderType; // name of enumerated type shortened…
struct instrType {
…                                              //left for you as exercise!!!
}

typedef struct instrType instrType;
struct classType {// fields (attributes in OOP) of a course declared…
char classCode[8];
char className[60];
instrType instructor;
struct classtype *clsptr;
}
typedef struct classType classType; // name of structure shortened…
```

Struct… Example continues

```
struct stdType {
      char id[8];                              //key
                  //personal info
      char name[15];
      char surname[25];
      genderType gender;                             //enumerated type
      …
                  //student info
      classType current_classes[10]; //...or     class_type *cur_clsptr
      classType classes_taken[50];   //...or     class_type *taken_clsptr
      float grade;
      unsigned int credits_earned;
      …
                  //next record's first byte's address
      struct stdType *sptr;              //address of next student record
}
```

Memory Issues

- Arrays can be used within records.
  - Ex: classType current_classes[10]; // from previous slide
- Each element of an array can be a record.
  - stdType students[1000];
- Using an array of classType for keeping taken classes wastes memory space (Why?)
  - Any alternatives?
- How will we keep student records in MM?
  - In an array?
  - Advantages?
  - Disadvantages?

Array Representation

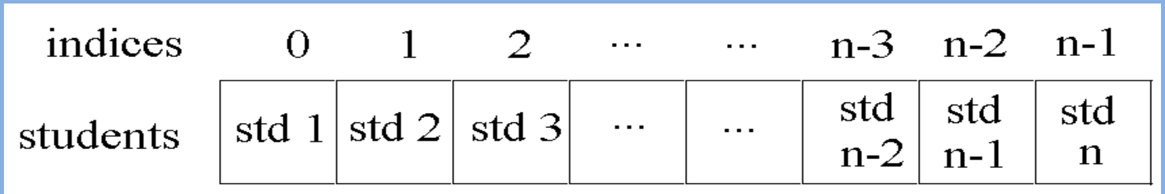
Advantages

1. Direct access (i.e., faster execution)

Disadvantages

1. Not suitable for changing number of student records
   - The higher the extent of memory waste the smaller the number of student records required to store than that at the initial case.
   - The (constant) size of array requires extension which is impossible for static arrays in case the number exceeds the bounds of the array.

The other alternative is **pointers** that provide **dynamic memory allocation**



Array Representation

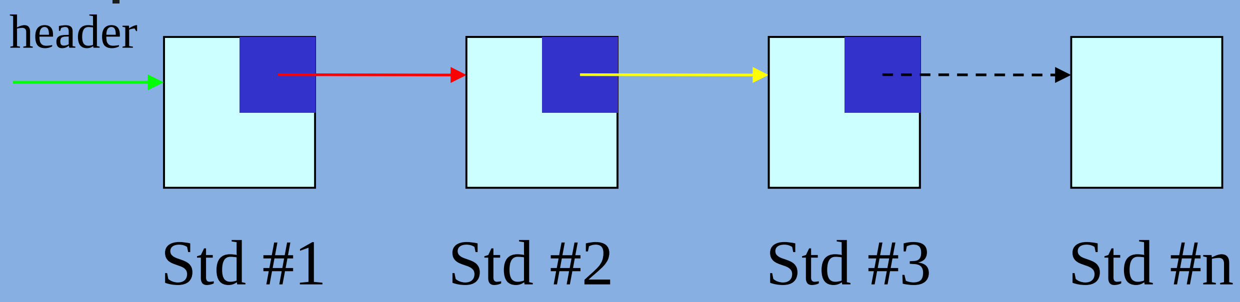
Pointers

- Pointers are variables that hold memory addresses.

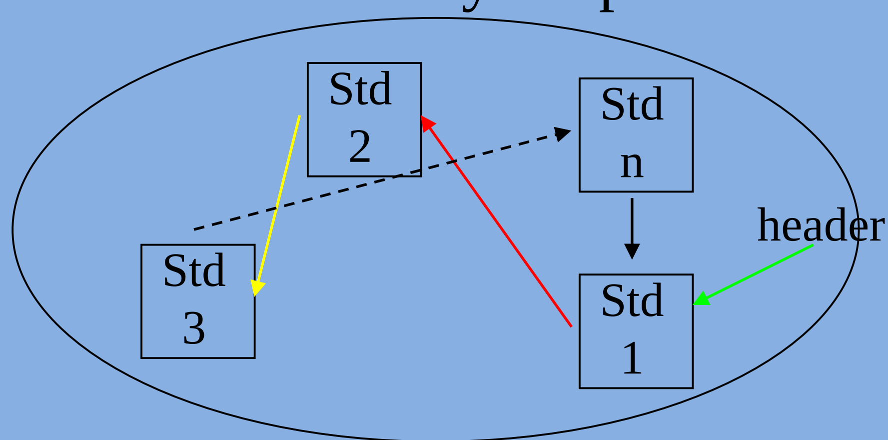
- Declaration of a pointer is based on the type of data of which the pointer holds the memory address.

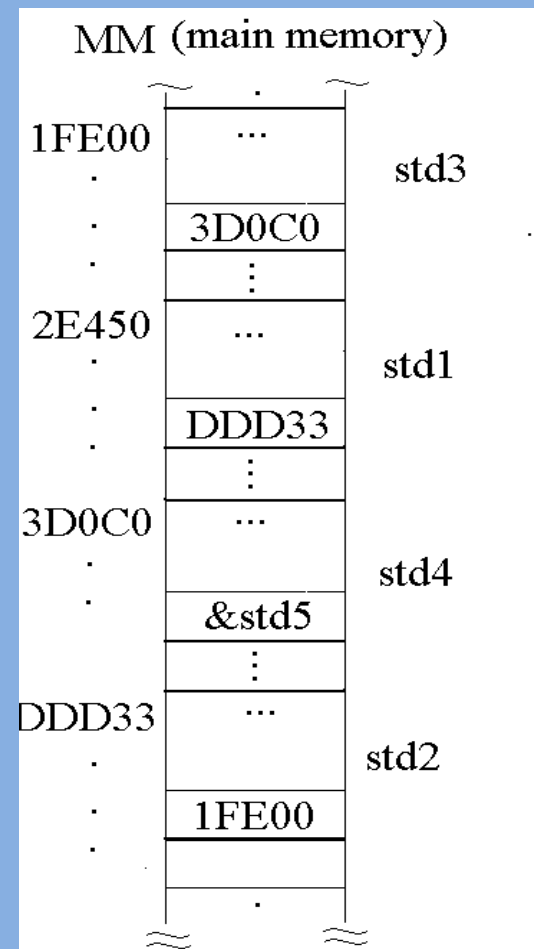
  - Ex: stdtype *stdptr;

Linked List Representation



Memory heap

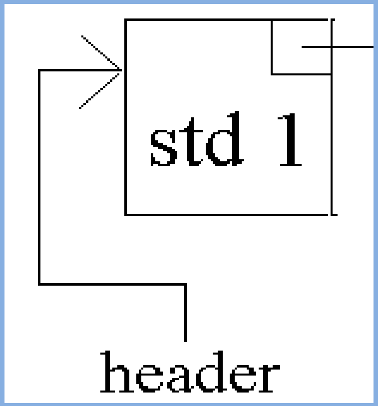


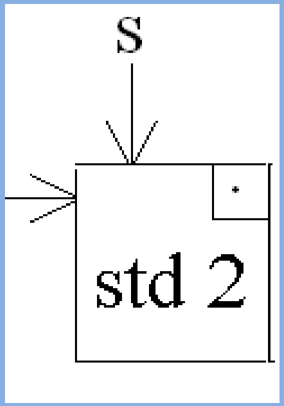


Value of header=**2E450**

Dynamic Memory Allocation

header=(*stdtype) malloc(sizeof(stdtype));
//Copy the info of first student to node pointed to by header
s =(*stdtype) malloc(sizeof(stdtype));
//Copy info of second student to node pointed to by header
Header->sptr=s;
…





Arrays vs. Pointers

- Static data structures
- Represented by an index and associated value
- Consecutive memory cells
- Direct access (+)
- Constant size (-)
- Memory not released during runtime (-)

- Dynamic data structures
- Represented by a record of information and address of next node
- Randomly located in heap (cause for need to keep address of next node)
- Sequential access (-)
- Flexible size (+)
- Memory space allocatable and releasable during runtime (+)