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
Explicit Goal

We look for answers to the following question:

“How do we store data in MM such that

• execution time grows as slow as possible with the growing size of input data, and
• 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** (pointers)
ways of **memory allocations**, two fundamental Implementation tools for data structures.
Representation of Main Memory

MM (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

\[ \begin{array}{cccc}
  x & x & x \\
  x & x & x & x \\
\end{array} \] → \[ \begin{array}{cccc}
  x & x & x \\
  x & x & x \\
\end{array} \]

\[ \begin{array}{cccc}
  x & x & x & x \\
  x & x & x & x \\
\end{array} \] → \[ \begin{array}{cccc}
  x & x & x \\
  x & x & x \\
\end{array} \]

\[ \begin{array}{cccc}
  x & x & x & x \\
  x & x & x & x \\
\end{array} \] → \[ \begin{array}{cccc}
  x & x & x \\
  x & x & x \\
\end{array} \]
Examples for efficient vs. inefficient data structures

8-Queen problem

- Inefficient:
  - More memory space required

- Efficient:
  - Less memory space required

```c
int a[4][4];
...
int a[5];
...
```
Math Review

- **Exponents**

\[ x^a x^b = x^{a+b}; \quad \frac{x^a}{x^b} = x^{a-b}; \quad (x^a)^b = x^{ab}; \]

- **Logarithms**

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

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

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

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

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

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

\[\Rightarrow s - as = s(1 - a) = 1\]
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
    - Prove the base case \(k=1\)
    - Assume hypothesis holds for \(k=n\)
    - Prove hypothesis for \(k=n+1\)

- Proof by counterexample
  - Prove the hypothesis wrong by an example

- Proof by contradiction \(A \Rightarrow B \iff \sim B \Rightarrow \sim A\)
  - 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:
  2. Assume true for n=k:
  3. Prove true for n=k+1:

\[
\sum_{i=1}^{1} i = 1
\]

\[
\sum_{i=1}^{k} i = \frac{k(k + 1)}{2}
\]

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

\[
\frac{k(k + 1)}{2} + \frac{2(k + 1)}{2} = \frac{(k + 1)(k + 2)}{2}
\]
Arrays

- Static data structures that
  - represent **contiguous** memory locations holding **data of 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`

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>23</td>
<td>6</td>
<td>64</td>
<td>26</td>
<td>3</td>
<td>35</td>
<td>8</td>
<td>56</td>
<td>39</td>
<td>48</td>
<td>41</td>
<td>12</td>
</tr>
<tr>
<td>MM (main memory)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[0] high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[0] low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[1] high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[1] low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[2] high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[2] low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F001</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F002</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F003</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F004</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F005</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

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

- \( m \): 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 \( i+1 \)\(^{th} \) row and \( j+1 \)\(^{th} \) column. “i” and “j” are row and column indices, respectively.
- How many elements? \( n = r\times c \) 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 \( a \)?

In general, \( m[i][j] \) is placed at \( a[k] \) where \( k=l+ic+j \).
Implementation Details of Arrays

- 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!!!
- Arrays may be efficiently passed to functions using their name and their size where
  - the name specifies the beginning address of the array
  - the size states the bounds of the index values.

3. Arrays can only be copied element by element.
Implementation Details… cont’d

```c
#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 arr[][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 min;
    ...
    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

- 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.
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 stdType {
    char id[8];                    //key
                      //personal info
    char name[15];
    char surname[25];
    genderType gender;            //enumerated type
    ...

                      //student info
    classType current_classes[10]; //...or    classType *cur_clsptr
    classType classes_taken[50];   //...or    classType *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
• Direct access (i.e., faster execution)

Disadvantages
• 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**

<table>
<thead>
<tr>
<th>indices</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>...</th>
<th>n-3</th>
<th>n-2</th>
<th>n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>students</td>
<td>std 1</td>
<td>std 2</td>
<td>std 3</td>
<td>...</td>
<td>...</td>
<td>std n-2</td>
<td>std n-1</td>
<td>std n</td>
</tr>
</tbody>
</table>

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

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;
...

![Diagram showing memory allocation and node connections]
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 (+)