Introduction to Data Structures

Subhabrata Samajder

IIIT, Delhi

Winter Semester,
17^{th} March, 2023

Full History Recurrences (Cont.)

A Not So Simple Full-history Recurrence Relation

$$T(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} T(i), \ \text{(for } n \geq 2); \ T(1) = 0.$$

A Not So Simple Full-history Recurrence Relation

$$T(n) = n - 1 + \frac{2}{n} \sum_{i=1}^{n-1} T(i), \text{ (for } n \geq 2); \ T(1) = 0.$$

- **Idea:** Use the shifting and canceling terms technique, s.t., most of the $T(i)$ terms gets canceled out.

A Not So Simple Full-history Recurrence Relation (Cont.)

Multiplying both sides by n, we get:

$$\begin{aligned} nT(n) &= n(n-1) + 2\sum_{i=1}^{n-1} T(i), \\ (n+1)T(n+1) &= n(n+1) + 2\sum_{i=1}^{n} T(i). \end{aligned}$$

A Not So Simple Full-history Recurrence Relation (Cont.)

Multiplying both sides by n, we get:

$$\begin{aligned} nT(n) &= n(n-1) + 2\sum_{i=1}^{n-1} T(i), \\ (n+1)T(n+1) &= n(n+1) + 2\sum_{i=1}^{n} T(i). \end{aligned}$$

Therefore,

$$(n+1)T(n+1) - nT(n) \quad = \quad n(n+1) - n(n-1) + 2T(n)$$

A Not So Simple Full-history Recurrence Relation (Cont.)

Multiplying both sides by n, we get:

$$\begin{aligned} nT(n) &= n(n-1) + 2\sum_{i=1}^{n-1} T(i), \\ (n+1)T(n+1) &= n(n+1) + 2\sum_{i=1}^{n} T(i). \end{aligned}$$

Therefore,

$$\begin{aligned} (n+1)T(n+1) - nT(n) &= n(n+1) - n(n-1) + 2T(n) \\ &= 2n + 2T(n) \end{aligned}$$

A Not So Simple Full-history Recurrence Relation (Cont.)

Multiplying both sides by n, we get:

$$\begin{aligned} nT(n) &= n(n-1) + 2\sum_{i=1}^{n-1} T(i), \\ (n+1)T(n+1) &= n(n+1) + 2\sum_{i=1}^{n} T(i). \end{aligned}$$

Therefore,

$$\begin{aligned} (n+1)T(n+1) - nT(n) &= n(n+1) - n(n-1) + 2T(n) \\ &= 2n + 2T(n) \\ \Rightarrow \quad T(n+1) &= \frac{n+2}{n+1}T(n) + \frac{2n}{n+1} \end{aligned}$$

A Not So Simple Full-history Recurrence Relation (Cont.)

Multiplying both sides by n, we get:

$$\begin{aligned} nT(n) &= n(n-1) + 2\sum_{i=1}^{n-1} T(i), \\ (n+1)T(n+1) &= n(n+1) + 2\sum_{i=1}^{n} T(i). \end{aligned}$$

Therefore,

$$\begin{aligned} (n+1)T(n+1) - nT(n) &= n(n+1) - n(n-1) + 2T(n) \\ &= 2n + 2T(n) \\ \Rightarrow \quad T(n+1) &= \frac{n+2}{n+1}T(n) + \frac{2n}{n+1} \\ &\leq \frac{n+2}{n+1}T(n) + 2 \quad \text{(A close approx.)} \end{aligned}$$

A Not So Simple Full-history Recurrence Relation (Cont.)

Expanding, we get

$$T(n) \leq 2 + \frac{n+1}{n}\left[2 + \frac{n}{n-1}\left[2 + \frac{n-1}{n-2}\left[\cdots \frac{4}{3}\right]\right]\right]$$

A Not So Simple Full-history Recurrence Relation (Cont.)

Expanding, we get

$$
\begin{aligned}
T(n) &\leq 2 + \frac{n+1}{n}\left[2 + \frac{n}{n-1}\left[2 + \frac{n-1}{n-2}\left[\cdots \frac{4}{3}\right]\right]\right] \\
&= 2\left[1 + \frac{n+1}{n} + \frac{n+1}{n}\cdot\frac{n}{n-1} + \frac{n+1}{n}\cdot\frac{n}{n-1}\cdot\frac{n-1}{n-2} + \right. \\
&\qquad \left. \cdots + \frac{n+1}{n}\cdot\frac{n}{n-1}\cdot\frac{n-1}{n-2}\cdots\frac{4}{3}\right]
\end{aligned}
$$

A Not So Simple Full-history Recurrence Relation (Cont.)

Expanding, we get

$$
\begin{aligned}
T(n) &\leq 2 + \frac{n+1}{n}\left[2 + \frac{n}{n-1}\left[2 + \frac{n-1}{n-2}\left[\cdots \frac{4}{3}\right]\right]\right] \\
&= 2\left[1 + \frac{n+1}{n} + \frac{n+1}{n} \cdot \frac{n}{n-1} + \frac{n+1}{n} \cdot \frac{n}{n-1} \cdot \frac{n-1}{n-2} + \right. \\
&\qquad \left. \cdots + \frac{n+1}{n} \cdot \frac{n}{n-1} \cdot \frac{n-1}{n-2} \cdots \frac{4}{3}\right] \\
&= 2\left[1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \frac{n+1}{n-2} + \cdots + \frac{n+1}{3}\right]
\end{aligned}
$$

A Not So Simple Full-history Recurrence Relation (Cont.)

Expanding, we get

$$\begin{aligned}
T(n) &\le 2 + \frac{n+1}{n}\left[2 + \frac{n}{n-1}\left[2 + \frac{n-1}{n-2}\left[\cdots\frac{4}{3}\right]\right]\right] \\
&= 2\left[1 + \frac{n+1}{n} + \frac{n+1}{n}\cdot\frac{n}{n-1} + \frac{n+1}{n}\cdot\frac{n}{n-1}\cdot\frac{n-1}{n-2} + \right. \\
&\quad \left. \cdots + \frac{n+1}{n}\cdot\frac{n}{n-1}\cdot\frac{n-1}{n-2}\cdots\frac{4}{3}\right] \\
&= 2\left[1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \frac{n+1}{n-2} + \cdots + \frac{n+1}{3}\right] \\
&= 2(n+1)\left[\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{3}\right]
\end{aligned}$$

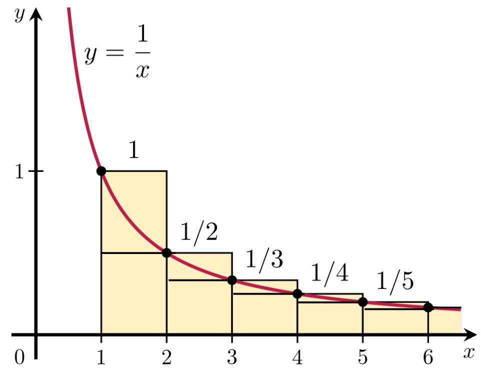
A Not So Simple Full-history Recurrence Relation (Cont.)

Expanding, we get

$$
\begin{aligned}
T(n) &\le 2 + \frac{n+1}{n}\left[2 + \frac{n}{n-1}\left[2 + \frac{n-1}{n-2}\left[\cdots \frac{4}{3}\right]\right]\right] \\
&= 2\left[1 + \frac{n+1}{n} + \frac{n+1}{n}\cdot\frac{n}{n-1} + \frac{n+1}{n}\cdot\frac{n}{n-1}\cdot\frac{n-1}{n-2} + \right. \\
&\quad \left. \cdots + \frac{n+1}{n}\cdot\frac{n}{n-1}\cdot\frac{n-1}{n-2}\cdots\frac{4}{3}\right] \\
&= 2\left[1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \frac{n+1}{n-2} + \cdots + \frac{n+1}{3}\right] \\
&= 2(n+1)\left[\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{3}\right] \\
&= 2(n+1)(H(n+1) - 1.5);
\end{aligned}
$$

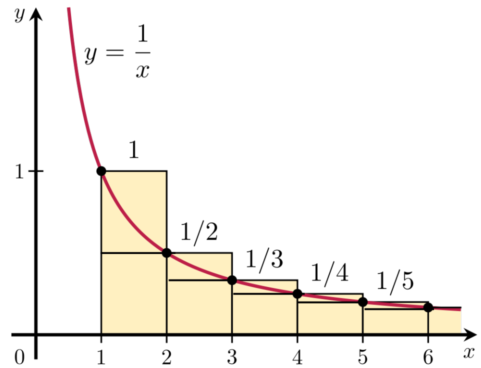
where $H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$ is the Harmonic series.

Harmonic Series Approximation



$$H(n) = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \quad > \quad \int_1^{n+1} \frac{dx}{x} = \ln(n+1) \quad \text{and}$$

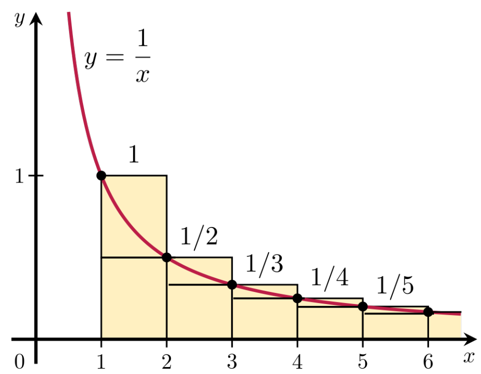
Harmonic Series Approximation



$$H(n) = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \quad > \quad \int_1^{n+1} \frac{dx}{x} = \ln(n+1) \quad \text{and}$$

$$H(n) = 1 + \left(\frac{1}{2} + \cdots + \frac{1}{n}\right) \quad < \quad 1 + \int_1^{n} \frac{dx}{x} = 1 + \ln(n).$$

Harmonic Series Approximation



$$H(n) = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \quad > \quad \int_1^{n+1} \frac{dx}{x} = \ln(n+1) \quad \text{and}$$

$$H(n) = 1 + \left(\frac{1}{2} + \cdots + \frac{1}{n}\right) \quad < \quad 1 + \int_1^{n} \frac{dx}{x} = 1 + \ln(n).$$

Combining: $\ln(n+1) < H(n) < 1 + \ln(n)$.

Harmonic Series Approximation (Cont.)

Let $\delta_n = H(n) - \ln n$.

Harmonic Series Approximation (Cont.)

Let $\delta_n = H(n) - \ln n$.
Then,

$$\ln(1 + 1/n) < H(n) - \ln n < 1, \text{ and}$$

Harmonic Series Approximation (Cont.)

Let $\delta_n = H(n) - \ln n$.
Then,

$$\ln(1 + 1/n) < H(n) - \ln n < 1, \text{ and}$$

$$\delta_n - \delta_{n+1} = (H(n) - \ln n) - (H(n+1) - \ln(n+1))$$

Harmonic Series Approximation (Cont.)

Let $\delta_n = H(n) - \ln n$.
Then,

$$\ln(1 + 1/n) < H(n) - \ln n < 1, \text{ and}$$

$$\begin{aligned} \delta_n - \delta_{n+1} &= (H(n) - \ln n) - (H(n+1) - \ln(n+1)) \\ &= \ln(n+1) - \ln n - \frac{1}{n+1} \end{aligned}$$

Harmonic Series Approximation (Cont.)

Let $\delta_n = H(n) - \ln n$.
Then,

$$\ln(1 + 1/n) < H(n) - \ln n < 1, \text{ and}$$

$$\begin{aligned}
\delta_n - \delta_{n+1} &= (H(n) - \ln n) - (H(n+1) - \ln(n+1)) \\
&= \ln(n+1) - \ln n - \frac{1}{n+1} \\
&= \int_n^{n+1} \frac{dx}{x} - \frac{1}{n+1} > 0 \quad \text{(See the picture!)},
\end{aligned}$$

i.e., δ_n is monotone decreasing.

Harmonic Series Approximation (Cont.)

Let $\delta_n = H(n) - \ln n$.
Then,

$$\ln(1 + 1/n) < H(n) - \ln n < 1, \text{ and}$$

$$\begin{aligned}
\delta_n - \delta_{n+1} &= (H(n) - \ln n) - (H(n+1) - \ln(n+1)) \\
&= \ln(n+1) - \ln n - \frac{1}{n+1} \\
&= \int_n^{n+1} \frac{dx}{x} - \frac{1}{n+1} > 0 \quad \text{(See the picture!)},
\end{aligned}$$

i.e., δ_n is monotone decreasing. Therefore δ_n converges and let

$$\gamma = \lim_{n\to\infty} \delta_n = \lim_{n\to\infty} (H(n) - \ln n).$$

Harmonic Series Approximation (Cont.)

Let $\delta_n = H(n) - \ln n$.
Then,

$$\ln(1 + 1/n) < H(n) - \ln n < 1, \text{ and}$$

$$\begin{aligned}
\delta_n - \delta_{n+1} &= (H(n) - \ln n) - (H(n+1) - \ln(n+1)) \\
&= \ln(n+1) - \ln n - \frac{1}{n+1} \\
&= \int_n^{n+1} \frac{dx}{x} - \frac{1}{n+1} > 0 \quad \text{(See the picture!)},
\end{aligned}$$

i.e., δ_n is monotone decreasing. Therefore δ_n converges and let

$$\gamma = \lim_{n\to\infty} \delta_n = \lim_{n\to\infty} (H(n) - \ln n).$$

$\gamma \approx 0.5772$ is called the **Euler constant** (Euler, 1735).

Harmonic Series Approximation (Cont.)

Let $\delta_n = H(n) - \ln n$.
Then,

$$\ln(1+1/n) < H(n) - \ln n < 1, \text{ and}$$

$$\begin{aligned}
\delta_n - \delta_{n+1} &= (H(n) - \ln n) - (H(n+1) - \ln(n+1)) \\
&= \ln(n+1) - \ln n - \frac{1}{n+1} \\
&= \int_n^{n+1} \frac{dx}{x} - \frac{1}{n+1} > 0 \quad \text{(See the picture!)},
\end{aligned}$$

i.e., δ_n is monotone decreasing. Therefore δ_n converges and let

$$\gamma = \lim_{n\to\infty} \delta_n = \lim_{n\to\infty} (H(n) - \ln n).$$

$\gamma \approx 0.5772$ is called the **Euler constant** (Euler, 1735).
$\therefore\ H(n) \approx \ln n + \gamma$.

A Not So Simple Full-history Recurrence Relation (Cont.)

Using the Harmonic series approximation, we get

$$T(n) \leq 2(n+1)(H(n+1) - 1.5)$$

A Not So Simple Full-history Recurrence Relation (Cont.)

Using the Harmonic series approximation, we get

$$\begin{aligned} T(n) &\leq 2(n+1)(H(n+1) - 1.5) \\ &\approx 2(n+1)(\ln n + \gamma - 1.5) + \mathcal{O}(1) \end{aligned}$$

A Not So Simple Full-history Recurrence Relation (Cont.)

Using the Harmonic series approximation, we get

$$\begin{aligned} T(n) &\leq 2(n+1)(H(n+1) - 1.5) \\ &\approx 2(n+1)(\ln n + \gamma - 1.5) + \mathcal{O}(1) \\ &= \mathcal{O}(n \log n). \end{aligned}$$

A Not So Simple Full-history Recurrence Relation (Cont.)

Using the Harmonic series approximation, we get

$$\begin{aligned} T(n) &\leq 2(n+1)(H(n+1) - 1.5) \\ &\approx 2(n+1)(\ln n + \gamma - 1.5) + \mathcal{O}(1) \\ &= \mathcal{O}(n \log n). \end{aligned}$$

$\therefore$ **Quicksort is indeed quick on the average!!**

Few Remarks

- In practice, quicksort is very fast, so it well deserves it's name.

Few Remarks

- In practice, quicksort is very fast, so it well deserves it's name.
- **Reason:**
  - Many elements are compared against the same pivot element.
  - $\therefore$ the pivot can be stored in a register.
  - It is an in-place algorithm.

Few Remarks

- In practice, quicksort is very fast, so it well deserves it's name.
- **Reason:**
  - Many elements are compared against the same pivot element.
  - $\therefore$ the pivot can be stored in a register.
  - It is an in-place algorithm.
- **Improvement:** By choosing the base of the induction wisely.

Few Remarks

- In practice, quicksort is very fast, so it well deserves it's name.
- **Reason:**
  - Many elements are compared against the same pivot element.
  - $\therefore$ the pivot can be stored in a register.
  - It is an in-place algorithm.
- **Improvement:** By choosing the base of the induction wisely.
  - The idea is to start the induction not always from 1.
  - Use, simple sorting techniques, like insertion sort or selection sort for small sequences.
  - **Note:** The efficiency of quicksort shows only for large sequences.
  - Define the base case for quicksort to be of size larger than 1 (10 to 20 is a good size).
  - Handle the base case by insertion sort or selection sort.

Introduction to Data Structures

Data Structures

Data: Collection of raw facts.

Data Structures

Data: Collection of raw facts.

Definition

A data structure is a data organization, management, and storage format that enables efficient access and modification.

More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

Data Structures

Data: Collection of raw facts.

Definition

A data structure is a data organization, management, and storage format that enables efficient access and modification.

More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

- They are the building blocks of computer algorithms.
- Design of an algorithm must be based on a thorough understanding of data structure techniques and costs.

Abstract Data Type (ADT)

- It is a useful notion in the study of data structures.
- Normally, when we write a program, we have to specify the data type (e.g., integers, reals, characters).
  - Also called primitive data structures.
  - Directly operated upon by the machine-level instructions.

Abstract Data Type (ADT)

- It is a useful notion in the study of data structures.
- Normally, when we write a program, we have to specify the data type (e.g., integers, reals, characters).
- But, in some cases, the exact data type is not important.

  Example:

  - **Frequent Operations:** Insertions and deletions.
  - **Condition:** First-in first-out (FIFO).

Abstract Data Type (ADT)

- It is a useful notion in the study of data structures.
- Normally, when we write a program, we have to specify the data type (e.g., integers, reals, characters).
- But, in some cases, the exact data type is not important.

  Example:
  - **Frequent Operations:** Insertions and deletions.
  - **Condition:** First-in first-out (FIFO).
  - **Data Structure:** Queue.
- It is more convenient and more general to design algorithms for these operations without specifying the data type of the items.

Abstract Data Type (ADT)

- The most important part is the list of operations that we want to support.

Abstract Data Type (ADT)

- The most important part is the list of operations that we want to support.
- **Another Example:** A queue where items have priorities = priority queue.

Abstract Data Type (ADT)

- The most important part is the list of operations that we want to support.
- **Another Example:** A queue where items have priorities = priority queue.
- Make the design more general by
  - Concentrating on the operational nature of a data structure,
  - and not on the precise implementation for a particular problem.

Abstract Data Type (ADT)

- The most important part is the list of operations that we want to support.
- **Another Example:** A queue where items have priorities = priority queue.
- Make the design more general by
  - Concentrating on the operational nature of a data structure,
  - and not on the precise implementation for a particular problem.

Example: The techniques for implementing a priority queue are for the most part independent of the exact data type.

Abstract Data Type (ADT)

- The most important part is the list of operations that we want to support.
- **Another Example:** A queue where items have priorities = priority queue.
- Make the design more general by
  - Concentrating on the operational nature of a data structure,
  - and not on the precise implementation for a particular problem.

  Example: The techniques for implementing a priority queue are for the most part independent of the exact data type.
- If our needs matches the definition of an ADT, we use it.

Abstract Data Type (ADT)

- The most important part is the list of operations that we want to support.
- **Another Example:** A queue where items have priorities = priority queue.
- Make the design more general by
  - Concentrating on the operational nature of a data structure,
  - and not on the precise implementation for a particular problem.

  Example: The techniques for implementing a priority queue are for the most part independent of the exact data type.
- If our needs matches the definition of an ADT, we use it.
- Make the algorithm-design process more modular.

Abstract Data Type: Definition

Definition

An abstract data type (ADT) is a mathematical model for data types, where a data type is defined by its behavior (semantics) from the point of view of a *user* of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

In contrast a Data structures requires

- concrete representations of data, and
- are of the point of view of an *implementer* and not a *user*.

ADT Example: Integers

Integers are an ADT, defined by the

- Values: $\ldots, -2, -1, 0, 1, 2, \ldots$, and by the
- Operations: '$+$', '-', '$\cdot$', '$/$', '$<$', etc.
- Behavior:
  - Obeying various axioms (associativity and commutativity of addition etc.).
  - Preconditions on operations (cannot divide by zero).
  - Must be independent of how the integers are represented.
- Representation:
  - Typically represented in 2's complement.
  - Can also be binary-coded decimal or in 1' complement.
- User:
  - Abstracted from the concrete choice of representation.
  - Simply use it as data types.

Elementary Data Structures

- Sets are also fundamental to computer science.
- Dynamic Sets: Sets manipulated by algorithms can *grow*, *shrink*, or *change over time*.
- Algorithms may require several different types of operations to be performed on sets.
- Dictionary: A dynamic set that supports insertion, deletion, and membership testing.
- The best way to implement a dynamic set depends upon the operations that must be supported.

Elements

- Elements: Generic name for an unspecified data type.

  Example: Integer, a set of integers, a text file, etc.

Elements

- Elements: Generic name for an unspecified data type.

  Example: Integer, a set of integers, a text file, etc.

- **Assumptions:**
  1. Elements can be compared for equality.
  2. Elements are taken from a **totally ordered set**.
  3. Elements can be copied.

Elements

- Elements: Generic name for an unspecified data type.

  Example: Integer, a set of integers, a text file, etc.

- **Assumptions:**
  1. Elements can be compared for equality.
  2. Elements are taken from a **totally ordered set**.
  3. Elements can be copied.

- **Assumption:** All these operations take unit amount of time.

Keys and Satallite Data

Keys:

- An identifying field of the objects in a dynamic set.
- If all the keys are different, then we think of the dynamic set as being a set of key values.

Satellite Data:

- Data carried around in other object fields but are otherwise not part of the implementation.

Operations on a Dynamic Set

- Can be grouped into two categories:
  - **Queries:** Which simply return information about the set.
  - **Modifying Operations:** Which change the set.

Modifying Operations

$\text{INSERT}(S, x)$: Augments the set S with the element pointed to by x.

$\text{DELETE}(S, x)$: Given a pointer to an element x in the set S, removes x from S.

Note: These operations uses a pointer to x and not a key value.

Queries

$\textsc{Search}(S, k)$: Given a set S and a key value k, returns

- a pointer x to an element in S such that $key[x] = k$, or
- **nil** if no such element belongs to S.

$\textsc{Minimum}(S)$: Returns a pointer to the element with smallest key.

$\textsc{Maximum}(S)$: Returns a pointer to the element with largest key.

$\textsc{Successor}(S, x)$: Given an element x, returns

- a pointer to the next larger element in S, or
- **nil** if x is the maximum element.

$\textsc{Predecessor}(S, x)$: Given an element x, returns

- a pointer to the next smaller element in S, or
- **nil** if x is the minimum element.

ADT: Arrays

Arrays

- Row of elements of the same type.
- The size of an array is the number of elements in that array.
- The size must be fixed.
- $\therefore$ all the elements are of the same type.
- Thus amount of memory is known a priori.
- Every element of an array can be accessed in constant time.

Arrays

- Row of elements of the same type.
- The size of an array is the number of elements in that array.
- The size must be fixed.
- $\therefore$ all the elements are of the same type.
- Thus amount of memory is known a priori.
- Every element of an array can be accessed in constant time.
- **Drawbacks:**
  - Cannot be used to store elements of different types (or sizes).
  - The size of an array cannot be changed dynamically.

ADT: Records

Records

- Similar to arrays, except elements can be of different types.
- A record is thus a list of elements of different types.
- The exact combination of types is fixed.
- $\therefore$ the storage size of a record is known in advance.
- Each element in a record can be accessed in constant time.

Records

- Similar to arrays, except elements can be of different types.
- A record is thus a list of elements of different types.
- The exact combination of types is fixed.
- $\therefore$ the storage size of a record is known in advance.
- Each element in a record can be accessed in constant time.
  - This is accomplished by keeping an array.
  - Element are then accessed by consulting the array.
  - The exact program that maintains the array is created automatically by the compiler.

Records: An Example

record example1
Begin
Int1 : integer;
Int2 : integer;
Ar1 : array[1 . . . 20] of integer;
Ar2 : array[1 . . . 20] of integer;
Ar3 : array[1 . . . 20] of integer;
Int3 : integer;
Int4 : integer;
Int5 : integer;
Int6 : integer;
Name1 : array[1 . . . 11] of character;
Name2 : array[1 . . . 12] of character;

End

- The array contains starting relative locations of all the elements.
- Thus 'Int6' starts at byte number 261 $(= 2{\cdot}4+3{\cdot}20{\cdot}4+3{\cdot}4+1)$
- Like arrays, the storage for a record is always consecutive.

Records: An Example

record example1
Begin
  Int1 : integer;
  Int2 : integer;
  Ar1 : array[1 . . . 20] of integer;
  Ar2 : array[1 . . . 20] of integer;
  Ar3 : array[1 . . . 20] of integer;
  Int3 : integer;
  Int4 : integer;
  Int5 : integer;
  Int6 : integer;
  Name1 : array[1 . . . 11] of character;
  Name2 : array[1 . . . 12] of character;
End

- The array contains starting relative locations of all the elements.
- Thus 'Int6' starts at byte number 261 $(= 2{\cdot}4+3{\cdot}20{\cdot}4+3{\cdot}4+1)$
- Like arrays, the storage for a record is always consecutive.
- **Drawback:** It is not possible to add elements dynamically.

Records in C

```
struct Record {
   int Int1;
   int Int2;
   int Ar1[20];
   int Ar2[20];
   int Ar3[20];
   int Int3;
   int Int4;
   int Int5;
   int Int6;
   char Name1[11];
   char Name2[12];
} Example1;
```

Records in C

```
struct Record {
   int Int1;
   int Int2;
   int Ar1[20];
   int Ar2[20];
   int Ar3[20];
   int Int3;
   int Int4;
   int Int5;
   int Int6;
   char Name1[11];
   char Name2[12];
} Example1;
```

sizeof(Example1)?

Books Consulted

1. Chapter 10 of *Introduction to Algorithms* by Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein.

Thank You for your kind attention!

Questions!!