Introduction to Data Structures

Subhabrata Samajder

IIIT, Delhi

Summer Semester,

12^{th} May, 2022

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: '+', '-', '·', '/', '<', 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

INSERT(S, x): Augments the set S with the element pointed to by x.

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

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.

MINIMUM(S): Returns a pointer to the element with smallest key.

MAXIMUM(S): Returns a pointer to the element with largest key.

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.

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!!