Chapter 1 : Formats and Editions of Data structures with abstract data types and Pascal [calendrierdelasci

As pointed out by Appleby in her comparative review of data structure books [1], the texts on this subject with which most of us are familiar are aimed at the Curriculum '78 course CS7, a sophomore or junior level course.

Usage[edit] Data structures serve as the basis for abstract data types ADT. The data structure implements the physical form of the data type. For example, relational databases commonly use B-tree indexes for data retrieval, [6] while compiler implementations usually use hash tables to look up identifiers. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Data structures can be used to organize the storage and retrieval of information stored in both main memory and secondary memory. Thus, the array and record data structures are based on computing the addresses of data items with arithmetic operations , while the linked data structures are based on storing addresses of data items within the structure itself. Many data structures use both principles, sometimes combined in non-trivial ways as in XOR linking. The efficiency of a data structure cannot be analyzed separately from those operations. This observation motivates the theoretical concept of an abstract data type , a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations including their space and time cost. List of data structures There are numerous types of data structures, generally built upon simpler primitive data types: Elements are accessed using an integer index to specify which element is required. Typical implementations allocate contiguous memory words for the elements of arrays but this is not always a necessity. Arrays may be fixed-length or resizable. A linked list also just called list is a linear collection of data elements of any type, called nodes, where each node has itself a value, and points to the next node in the linked list. The principal advantage of a linked list over an array, is that values can always be efficiently inserted and removed without relocating the rest of the list. Certain other operations, such as random access to a certain element, are however slower on lists than on arrays. A record also called tuple or struct is an aggregate data structure. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called fields or members. A union is a data structure that specifies which of a number of permitted primitive types may be stored in its instances, e. Contrast with a record , which could be defined to contain a float and an integer; whereas in a union, there is only one value at a time. Enough space is allocated to contain the widest member datatype. A tagged union also called variant , variant record, discriminated union, or disjoint union contains an additional field indicating its current type, for enhanced type safety. An object is a data structure that contains data fields, like a record does, as well as various methods which operate on the data contents. An object is an in-memory instance of a class from a taxonomy. In the context of object-oriented programming , records are known as plain old data structures to distinguish them from objects. Language support[edit] Most assembly languages and some low-level languages, such as BCPL Basic Combined Programming Language , lack built-in support for data structures. On the other hand, many high-level programming languages and some higher-level assembly languages, such as MASM , have special syntax or other built-in support for certain data structures, such as records and arrays. For example, the C a direct descendant of BCPL and Pascal languages support structs and records, respectively, in addition to vectors one-dimensional arrays and multi-dimensional arrays. Modern languages usually come with standard libraries that implement the most common data structures. Modern languages also generally support modular programming , the separation between the interface of a library module and its implementation. Some provide opaque data types that allow clients to hide implementation details. Many known data structures have concurrent versions which allow multiple computing threads to access a single concrete instance of a data structure simultaneously.

Chapter 2 : PASCAL Programming: Abstract Data Structures and Files

Computer program design can be made much easier by organizing information into abstract data structures (ADS). For example, one can model a table that has three columns and an indeterminate number of rows, in terms of an array with two dimensions: (1) a large number of rows, and (2) three columns.

It also presents working definitions for the fundamental but somewhat slippery terms " data item " and " data structure ". We begin with the basic elements on which data structures are built. A type is a collection of values. For example, the Boolean type consists of the values true and false. The integers also form a type. An integer is a simple type because its values contain no subparts. A bank account record will typically contain several pieces of information such as name, address, account number, and account balance. Such a record is an example of an aggregate type or composite type. A data item is a piece of information or a record whose value is drawn from a type. A data item is said to be a member of a type. A data type is a type together with a collection of operations to manipulate the type. For example, an integer variable is a member of the integer data type. Addition is an example of an operation on the integer data type. A distinction should be made between the logical concept of a data type and its physical implementation in a computer program. For example, there are two traditional implementations for the list data type: The list data type can therefore be implemented using a linked list or an array. For example, a list might be used to help implement a graph data structure. As another example, the term "array" could refer either to a data type or an implementation. By this meaning, an array is a physical data structure. However, array can also mean a logical data type composed of a typically homogeneous collection of data items, with each data item identified by an index number. It is possible to implement arrays in many different ways besides as a block of contiguous memory locations. The sparse matrix refers to a large, two-dimensional array that stores only a relatively few non-zero values. This is often implemented with a linked structure, or possibly using a hash table. But it could be implemented with an interface that uses traditional row and column indices, thus appearing to the user in the same way that it would if it had been implemented as a block of contiguous memory locations. An abstract data type ADT is the specification of a data type within some language, independent of an implementation. The interface for the ADT is defined in terms of a type and a set of operations on that type. The behavior of each operation is determined by its inputs and outputs. An ADT does not specify how the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as encapsulation. A data structure is the implementation for an ADT. In an object-oriented language, an ADT and its implementation together make up a class. Each operation associated with the ADT is implemented by a member function or method. The variables that define the space required by a data item are referred to as data members. An object is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program. The related term file structure often refers to the organization of data on peripheral storage, such as a disk drive or CD. The int variable type is a physical representation of the abstract integer. The int variable type, along with the operations that act on an int variable, form an ADT. Unfortunately, the int implementation is not completely true to the abstract integer, as there are limitations on the range of values an int variable can store. If these limitations prove unacceptable, then some other representation for the ADT "integer" must be devised, and a new implementation must be used for the associated operations. Insert a new integer at a particular position in the list. Return True if the list is empty. Return the number of integers currently in the list. Retrieve the integer at a particular position in the list. Delete the integer at a particular position in the list. From this description, the input and output of each operation should be clear, but the implementation for lists has not been specified. One application that makes use of some ADT might use particular member functions of that ADT more than a second application, or the two applications might have different time requirements for the various operations. These differences in the requirements of applications are the reason why a given ADT might be supported by more than one implementation. Both support efficient insertion and deletion of records, and both support exact-match queries. However, hashing is more efficient than the B-tree for exact-match queries. On the other hand, the B-tree can perform range queries efficiently, while hashing is hopelessly inefficient for range queries. Thus, if the database application limits searches to exact-match queries, hashing is preferred. On the other hand, if the application requires support for range queries, the B-tree is preferred. Despite these performance issues, both implementations solve versions of the same problem: The concept of an ADT can help us to focus on key issues even in non-computing applications. On nearly all passenger cars, you steer by turning the steering wheel, accelerate by pushing the gas pedal, and brake by pushing the brake pedal. This design for cars can be viewed as an ADT with operations "steer", "accelerate", and "brake". Two cars might implement these operations in radically different ways, say with different types of engine, or front- versus rear-wheel drive. Yet, most drivers can operate many different cars because the ADT presents a uniform method of operation that does not require the driver to understand the specifics of any particular engine or drive design. These differences are deliberately hidden. The concept of an ADT is one instance of an important principle that must be understood by any successful computer scientist: A central theme of computer science is complexity and techniques for handling it. Humans deal with complexity by assigning a label to an assembly of objects or concepts and then manipulating the label in place of the assembly. Cognitive psychologists call such a label a metaphor. A particular label might be related to other pieces of information or other labels. This collection can in turn be given a label, forming a hierarchy of concepts and labels. This hierarchy of labels allows us to focus on important issues while ignoring unnecessary details. These and other labels are gathered together under the label "computer". Because even the smallest home computers today have millions of components, some form of abstraction is necessary to comprehend how a computer operates. Consider how you might go about the process of designing a complex computer program that implements and manipulates an ADT. The ADT is implemented in one part of the program by a particular data structure. Without this ability to simplify your thinking about a complex program, you would have no hope of understanding or implementing it. Typically, records on disk in such a program are accessed through a buffer pool rather than directly. Variable length records might use a memory manager to find an appropriate location within the disk file to place the record. Multiple index structures will typically be used to support access to a collection of records using multiple search keys. Thus, we have a chain of classes, each with its own responsibilities and access privileges. A database query from a user is implemented by searching an index structure. This index requests access to the record by means of a request to the buffer pool. If a record is being inserted or deleted, such a request goes through the memory manager, which in turn interacts with the buffer pool to gain access to the disk file. A program such as this is far too complex for nearly any human programmer to keep all of the details in their head at once. The only way to design and implement such a program is through proper use of abstraction and metaphors. In object-oriented programming, such abstraction is handled using classes. Data types have both a logical form and a physical form. The definition of the data type in terms of an ADT is its logical form. The implementation of the data type as a data structure is its physical form. Sometimes you might see the term concrete implementation, but the word concrete is redundant. The figure below illustrates this relationship between logical and physical forms for data types. When you implement an ADT, you are dealing with the physical form of the associated data type. Some sections of this book focus on physical implementations for a given data structure. Other sections use the logical ADT for the data structure in the context of a higher-level task. The relationship between data items, abstract data types, and data structures. The ADT defines the logical form of the data type. The data structure implements the physical form of the data type. Typically, these programmers want to use the ADT as a component in another application. The interface becomes a form of communication between the two programmers. The logical form of the list is defined by the public functions, their inputs, and their outputs that define the class. This might be all that you know about the list class implementation, and this should be all you need to know. Within the class, a variety of physical implementations for lists is possible.

Chapter 3 : Array data type - Wikipedia

Note: Citations are based on reference standards. However, formatting rules can vary widely between applications and fields of interest or study. The specific requirements or preferences of your reviewing publisher, classroom teacher, institution or organization should be applied.

This should be marked as answer. The examples are spot on. ADT may be defined as a set of data values and associated operations that are precisely specified independent of any particular implementation. Thus an Abstract Data Type is an organized collection of information and a set of operations used to manage that information. The set of operations defines the interface of the ADT. Since, in ADT, the data values and operations are defined with mathematical precision, rather than as an implementation in a computer language, we may reason about effects of the operations, relations to other abstract data types whether a program implements the data type etc. One of the simplest abstract data type is the stack data type for which functions might be provided to create an empty stack, to push values onto a stack and to pop values from a stack. The basic difference between abstract data type ADT and concrete data type is that the latter allow us to look at the concrete representation, whereas the former hide the representation from us. A pure ADT is one where all operations are pure functions. This means that operations have no side effects. In particular, they do not modify or update there input arguments. They just use these arguments to generate output, which are fresh values of ADT or of other types. Most concrete types are pure. For example, no operation on integers actually modifies an integer. This operation would be considered impure and the whole ADT would then be impure also. We know that an Abstract Data Type is a data type that satisfies the following two conditions: The representation, or definition, of the type and the operations are contained in a single syntactic unit. A user defined Abstract Data Type should provide: A type definition that allows program units to declare variables of the type, but hides the representation of these variables. A set of operations for manipulating objects of the type. An example of a user defined abstract data type is structure. Structure is one such example. A structure is an aggregate of different parts, where each part is of some existing type. Variables of this type may be created in a similar way to variables of a built in type. AB b; Data Structures: The following are the characteristic features of data structures: It contains component data items, which may be atomic or another data structure still a domain. A set of operations on one or more of the component items. Defines rules as to how components relates to each other and to the structure as a whole assertions. A data structure may be static or dynamic. A static data structure has a fixed size. This meaning is different from the meaning of static modifier. A dynamic data structure grows and shrinks at execution time as required by its contents. A dynamic data structure is implemented using links. Data structures may further be categorized into linear data structures and non-linear data structures. In linear data structures every component has a unique predecessor and successor, except first and last elements, whereas in case of non-linear data structures, no such restriction is there as elements may be arranged in any desired fashion restricted by the way we use to represent such types.

Chapter 4 : Abstract data type - Wikipedia

Data Structures With Abstract Data Types and Pascal by Daniel F. Stubbs, Neil W. Webre and a great selection of similar Used, New and Collectible Books available now at calendrierdelascience.com

Each of these ADTs may be defined in many ways and variants, not necessarily equivalent. For example, an abstract stack may or may not have a count operation that tells how many items have been pushed and not yet popped. This choice makes a difference not only for its clients but also for the implementation. Abstract graphical data type An extension of ADT for computer graphics was proposed in Opaque data type Implementing an ADT means providing one procedure or function for each abstract operation. Usually there are many ways to implement the same ADT, using several different concrete data structures. Thus, for example, an abstract stack can be implemented by a linked list or by an array. In order to prevent clients from depending on the implementation, an ADT is often packaged as an opaque data type in one or more modules , whose interface contains only the signature number and types of the parameters and results of the operations. The implementation of the moduleâ€"namely, the bodies of the procedures and the concrete data structure usedâ€"can then be hidden from most clients of the module. This makes it possible to change the implementation without affecting the clients. If the implementation is exposed, it is known instead as a transparent data type. When implementing an ADT, each instance in imperative-style definitions or each state in functional-style definitions is usually represented by a handle of some sort. When a class is used as a type, it is an abstract type that refers to a hidden representation. In this model an ADT is typically implemented as a class , and each instance of the ADT is usually an object of that class. However, such an approach does not easily encapsulate multiple representational variants found in an ADT. It also can undermine the extensibility of object-oriented programs. In a pure object-oriented program that uses interfaces as types, types refer to behaviors not representations. Imperative-style interface[edit] An imperative-style interface might be: The implementation may be arbitrarily inefficient, since the formal definition of the ADT, above, does not specify how much space the stack may use, nor how long each operation should take. In practice the formal definition should specify that the space is proportional to the number of items pushed and not yet popped; and that every one of the operations above must finish in a constant amount of time, independently of that number. To comply with these additional specifications, the implementation could use a linked list, or an array with dynamic resizing together with two integers an item count and the array size. Functional-style interface[edit] Functional-style ADT definitions are more appropriate for functional programming languages, and vice versa. However, one can provide a functional-style interface even in an imperative language like C. Built-in abstract data types[edit] The specification of some programming languages is intentionally vague about the representation of certain built-in data types, defining only the operations that can be done on them. Therefore, those types can be viewed as "built-in ADTs". Examples are the arrays in many scripting languages, such as Awk , Lua , and Perl , which can be regarded as an implementation of the abstract list.

Chapter 5 : Pascal Plus Data Structures, Algorithms, and Advanced Programming by Nell B. Dale

Data structure modeling is a process to creatively extract and abstractly represent a real-world problem by data models based on constraints of given computing resources.

History[edit] Stacks entered the computer science literature in , when Alan M. Turing used the terms "bury" and "unbury" as a means of calling and returning from subroutines. Klaus Samelson and Friedrich L. Bauer of Technical University Munich proposed the idea in and filed a patent in , [4] and in March Bauer received the Computer Pioneer Award for the invention of the stack principle. When a plate is removed from the stack, the one below it pops up to become the new top. Non-essential operations[edit] In many implementations, a stack has more operations than "push" and "pop". An example is "top of stack", or "peek", which observes the top-most element without removing it from the stack. An underflow condition can occur in the "stack top" operation if the stack is empty, the same as "pop". Also, implementations often have a function which just returns whether the stack is empty. Implementation[edit] A stack can be easily implemented either through an array or a linked list. What identifies the data structure as a stack in either case is not the implementation but the interface: The following will demonstrate both implementations, using pseudocode. Array[edit] An array can be used to implement a bounded stack, as follows. The first element usually at the zero offset is the bottom, resulting in array[0] being the first element pushed onto the stack and the last element popped off. The program must keep track of the size length of the stack, using a variable top that records the number of items pushed so far, therefore pointing to the place in the array where the next element is to be inserted assuming a zero-based index convention. Thus, the stack itself can be effectively implemented as a three-element structure: The size of the stack is simply the size of the dynamic array, which is a very efficient implementation of a stack since adding items to or removing items from the end of a dynamic array requires amortized O 1 time. Linked list[edit] Another option for implementing stacks is to use a singly linked list. A stack is then a pointer to the "head" of the list, with perhaps a counter to keep track of the size of the list: Some languages, notably those in the Forth family including PostScript , are designed around language-defined stacks that are directly visible to and manipulated by the programmer. PHP has an SplStack class. Following is an example program in Java language, using that class. Basic architecture of a stack[edit] A typical stack, storing local data and call information for nested procedure calls not necessarily nested procedures. This stack grows downward from its origin. The stack pointer points to the current topmost datum on the stack. A push operation decrements the pointer and copies the data to the stack; a pop operation copies data from the stack and then increments the pointer. Each procedure called in the program stores procedure return information in yellow and local data in other colors by pushing them onto the stack. This type of stack implementation is extremely common, but it is vulnerable to buffer overflow attacks see the text. A typical stack is an area of computer memory with a fixed origin and a variable size. Initially the size of the stack is zero. A stack pointer, usually in the form of a hardware register, points to the most recently referenced location on the stack; when the stack has a size of zero, the stack pointer points to the origin of the stack. The two operations applicable to all stacks are: There are many variations on the basic principle of stack operations. Every stack has a fixed location in memory at which it begins. As data items are added to the stack, the stack pointer is displaced to indicate the current extent of the stack, which expands away from the origin. Stack pointers may point to the origin of a stack or to a limited range of addresses either above or below the origin depending on the direction in which the stack grows ; however, the stack pointer cannot cross the origin of the stack. In other words, if the origin of the stack is at address and the stack grows downwards towards addresses , , and so on , the stack pointer must never be incremented beyond to , , etc. If a pop operation on the stack causes the stack pointer to move past the origin of the stack, a stack underflow occurs. If a push operation causes the stack pointer to increment or decrement beyond the maximum extent of the stack, a stack overflow occurs. Some environments that rely heavily on stacks may provide additional operations, for example: This is also called top operation in many articles. Many variants of this operation are possible, with the most common being called left rotate and right rotate. Stacks are often visualized growing from the bottom up like real-world stacks. They may also be visualized growing from left to right, so that "topmost" becomes "rightmost", or even growing from top to bottom. The important feature is that the bottom of the stack is in a fixed position. The illustration in this section is an example of a top-to-bottom growth visualization: A right rotate will move the first element to the third position, the second to the first and the third to the second. Here are two equivalent visualizations of this process: The top and bottom terminology are used irrespective of whether the stack actually grows towards lower memory addresses or towards higher memory addresses. Pushing an item on to the stack adjusts the stack pointer by the size of the item either decrementing or incrementing, depending on the direction in which the stack grows in memory , pointing it to the next cell, and copies the new top item to the stack area. Depending again on the exact implementation, at the end of a push operation, the stack pointer may point to the next unused location in the stack, or it may point to the topmost item in the stack. If the stack points to the current topmost item, the stack pointer will be updated before a new item is pushed onto the stack; if it points to the next available location in the stack, it will be updated after the new item is pushed onto the stack. Popping the stack is simply the inverse of pushing. The topmost item in the stack is removed and the stack pointer is updated, in the opposite order of that used in the push operation. Stack in main memory[edit] Many CISC -type CPU designs, including the x86 , Z80 and , have a dedicated register for use as the call stack stack pointer with dedicated call, return, push, and pop instructions that implicitly update the dedicated register, thus increasing code density. Some CISC processors, like the PDP and the , also have special addressing modes for implementation of stacks , typically with a semi-dedicated stack pointer as well such as A7 in the In contrast, most RISC CPU designs do not have dedicated stack instructions and therefore most if not all registers may be used as stack pointers as needed. Stack in registers or dedicated memory[edit] Main article: Stack machine The x87 floating point architecture is an example of a set of registers organised as a stack where direct access to individual registers relative the current top is also possible. As with stack-based machines in general, having the top-of-stack as an implicit argument allows for a small machine code footprint with a good usage of bus bandwidth and code caches , but it also prevents some types of optimizations possible on processors permitting random access to the register file for all two or three operands. A stack structure also makes superscalar implementations with register renaming for speculative execution somewhat more complex to implement, although it is still feasible, as exemplified by modern x87 implementations. Sun SPARC , AMD Am , and Intel i are all examples of architectures using register windows within a register-stack as another strategy to avoid the use of slow main memory for function arguments and return values. There are also a number of small microprocessors that implements a stack directly in hardware and some microcontrollers have a fixed-depth stack that is not directly accessible. Many stack-based microprocessors were used to implement the programming language Forth at the microcode level. Stacks were also used as a basis of a number of mainframes and mini computers. Such machines were called stack machines , the most famous being the Burroughs B Applications of stacks[edit] Expression evaluation and syntax parsing[edit] Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations and conversion from one form to another may be accomplished using a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. Most programming languages are context-free languages , allowing them to be parsed with stack based machines. Backtracking Another important application of stacks is backtracking. Consider a simple example of finding the correct path in a maze. There are a series of points, from the starting point to the destination. We start from one point. To reach the final destination, there are several paths. Suppose we choose a random path. After following a certain path, we realise that the path we have chosen is wrong. So we need to find a way by which we can return to the beginning of that path. This can be done with the use of stacks. With the help of stacks, we remember the point where we have reached. This is done by pushing that point into the stack. In case we end up on the wrong path, we can pop the last point from the stack and thus return to the last point and continue our quest to find the right path. This is called backtracking. The prototypical example of a backtracking algorithm is depth-first search , which finds all vertices of a graph that can be reached from a specified starting vertex. Other applications of backtracking involve searching through spaces that represent potential solutions to an optimization problem. Branch and bound is a technique for performing such backtracking searches without exhaustively searching all of the potential solutions in such a space. Compile time memory management[edit] Main articles: Stack-based memory allocation and Stack machine A number of programming languages are stack-oriented , meaning they define most basic operations adding two numbers, printing a character as taking their arguments from the stack, and placing any return values back on the stack. For example, PostScript has a return stack and an operand stack, and also has a graphics state stack and a dictionary stack. Many virtual machines are also stack-oriented, including the p-code machine and the Java Virtual Machine. The functions follow a runtime protocol between caller and callee to save arguments and return value on the stack. Stacks are an important way of supporting nested or recursive function calls. Some programming languages use the stack to store data that is local to a procedure. Space for local data items is allocated from the stack when the procedure is entered, and is deallocated when the procedure exits. The C programming language is typically implemented in this way. Using the same stack for both data and procedure calls has important security implications see below of which a programmer must be aware in order to avoid introducing serious security bugs into a program. Efficient algorithms[edit] Several algorithms use a stack separate from the usual function call stack of most programming languages as the principle data structure with which they organize their information. Graham scan , an algorithm for the convex hull of a two-dimensional system of points. A convex hull of a subset of the input is maintained in a stack, which is used to find and remove concavities in the boundary when a new point is added to the hull.

Chapter 6 : Daniel F. Stubbs (Author of Data Structures With Abstract Data Types And Pascal)

Neurons or nerve cells - Structure function and types of neurons - Human Anatomy - 3D Biology - VÃ¬deo Dailymotion.

Rutishauser however although describing how a compiler for his language should be built, did not implement one. Assembly languages and low-level languages like BCPL [3] generally have no syntactic support for arrays. Abstract arrays[edit] An array data structure can be mathematically modeled as an abstract data structure an abstract array with two operations get A, I: The first axiom means that each element behaves like a variable. The second axiom means that elements with distinct indices behave as disjoint variables, so that storing a value in one element does not affect the value of any other element. These axioms do not place any constraints on the set of valid index tuples I, therefore this abstract model can be used for triangular matrices and other oddly-shaped arrays. This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. May Learn how and when to remove this template message In order to effectively implement variables of such types as array structures with indexing done by pointer arithmetic , many languages restrict the indices to integer data types or other types that can be interpreted as integers, such as bytes and enumerated types , and require that all elements have the same data type and storage size. Most of those languages also restrict each index to a finite interval of integers, that remains fixed throughout the lifetime of the array variable. In some compiled languages, in fact, the index ranges may have to be known at compile time. On the other hand, some programming languages provide more liberal array types, that allow indexing by arbitrary values, such as floating-point numbers , strings , objects , references , etc.. Such index values cannot be restricted to an interval, much less a fixed interval. So, these languages usually allow arbitrary new elements to be created at any time. This choice precludes the implementation of array types as array data structures. That is, those languages use array-like syntax to implement a more general associative array semantics, and must therefore be implemented by a hash table or some other search data structure. May Multi-dimensional arrays[edit] The number of indices needed to specify an element is called the dimension, dimensionality, or rank of the array type. This nomenclature conflicts with the concept of dimension in linear algebra, [5] where it is the number of elements. Thus, an array of numbers with 5 rows and 4 columns, hence 20 elements, is said to have dimension 2 in computing contexts, but represents a matrix with dimension 4-by-5 or 20 in mathematics. Also, the computer science meaning of "rank" is similar to its meaning in tensor algebra but not to the linear algebra concept of rank of a matrix. Many languages support only one-dimensional arrays. In those languages, a multi-dimensional array is typically represented by an Iliffe vector , a one-dimensional array of references to arrays of one dimension less. A two-dimensional array, in particular, would be implemented as a vector of pointers to its rows. Thus an element in row i and column j of an array A would be accessed by double indexing A[i][j] in typical notation. This way of emulating multi-dimensional arrays allows the creation of jagged arrays , where each row may have a different size â€” or, in general, where the valid range of each index depends on the values of all preceding indices. Early languages used parentheses, e. A[i,j] or A[i][j], as in Algol 60 and Pascal to distinguish from the use of parentheses for function calls. Index types[edit] Array data types are most often implemented as array structures: In some languages, however, array data types have the semantics of associative arrays, with indices of arbitrary type and dynamic element creation. Bounds checking[edit] Some languages like Pascal and Modula perform bounds checking on every access, raising an exception or aborting the program when any index is out of its valid range. Compilers may allow these checks to be turned off to trade safety for speed. Good compilers may also analyze the program to determine the range of possible values that the index may have, and this analysis may lead to bounds-checking elimination. Index origin[edit] Some languages, such as C, provide only zero-based array types, for which the minimum valid value for any index is 0. This choice is convenient for array implementation and address computations. With a language such as C, a pointer to the interior of any array can be defined that will symbolically act as a pseudo-array that accommodates negative indices. This works only because C does not check an index against bounds when used. Other languages provide only one-based array types, where each index starts at 1; this is the traditional convention in mathematics for matrices and mathematical sequences. A few languages, such as Pascal and Lua, support n-based array types, whose minimum legal indices are chosen by the programmer. The relative merits of each choice have been the subject of heated debate. Zero-based indexing has a natural advantage to one-based indexing in avoiding off-by-one or fencepost errors. In many languages such as C , one should specify the number of elements contained in the array; whereas in others such as Pascal and Visual Basic. NET one should specify the numeric value of the index of the last element. Needless to say, this distinction is immaterial in languages where the indices start at 1. Array algebra[edit] Some programming languages support array programming , where operations and functions defined for certain data types are implicitly extended to arrays of elements of those types. Languages providing array programming capabilities have proliferated since the innovations in this area of APL. They are a core facility in newer languages, such as Julia and recent versions of Fortran. These capabilities are also provided via standard extension libraries for other general purpose programming languages such as the widely used NumPy library for Python. String types and arrays[edit] Many languages provide a built-in string data type, with specialized notation " string literals " to build values of that type. In some languages such as C , a string is just an array of characters, or is handled in much the same way. Other languages, like Pascal , may provide vastly different operations for strings and arrays. Array index range queries[edit] Some programming languages provide operations that return the size number of elements of a vector, or, more generally, range of each index of an array. Elements of a newly created array may have undefined values as in C , or may be defined to have a specific "default" value such as 0 or a null pointer as in Java. Vectors can be queried for their size and can be resized. Slower operations like inserting an element in the middle are also supported. Slicing[edit] An array slicing operation takes a subset of the elements of an array-typed entity value or variable and then assembles them as another array-typed entity, possibly with other indices. If array types are implemented as array structures, many useful slicing operations such as selecting a sub-array, swapping indices, or reversing the direction of the indices can be performed very efficiently by manipulating the dope vector of the structure. The possible slicings depend on the implementation details: On the other hand, other slicing operations are possible when array types are implemented in other ways. Resizing[edit] Some languages allow dynamic arrays also called resizable, growable, or extensible: For one-dimensional arrays, this facility may be provided as an operation "append A,x " that increases the size of the array A by one and then sets the value of the last element to x. Other array types such as Pascal strings provide a concatenation operator, which can be used together with slicing to achieve that effect and more. In some languages, assigning a value to an element of an array automatically extends the array, if necessary, to include that element. Resizable arrays are conceptually similar to lists , and the two concepts are synonymous in some languages. An extensible array can be implemented as a fixed-size array, with a counter that records how many elements are actually in use. The append operation merely increments the counter; until the whole array is used, when the append operation may be defined to fail. This is an implementation of a dynamic array with a fixed capacity, as in the string type of Pascal. Alternatively, the append operation may re-allocate the underlying array with a larger size, and copy the old elements to the new area.

Chapter 7 : Data structure - Wikipedia

This books (Data Structures with Abstract Data Types and PASCAL [DOWNLOAD]) Made by Daniel Stubbs About Books none To Download Please Click https://cbo Slideshare uses cookies to improve functionality and performance, and to provide you with relevant advertising.

Schmalz Computer program design can be made much easier by organizing information into abstract data structures ADS. For example, one can model a table that has three columns and an indeterminate number of rows, in terms of an array with two dimensions: A key feature of modern computer programs is the ability to manipulate ADS using procedures or methods that are predefined by the programmer or software designer. This requires that data structures be specified carefully, with forethought, and in detail. This section is organized as follows: Arrays and their Manipulation 5. Arrays and their Manipulation. We begin with several observations about the use of arrays in computer programs. In the early days of computer programming, machines were dedicated to the task of computing tables of artillery trajectories WWII and tables of accounting and business inventory information early s. Thus, numerically intensive computing machines were designed to handle linear or two-dimensional arrays. An array is a data structured whose domain is a finite subset of Euclidean n-space Rn. Arrays in PASCAL are assigned the datatype of the elements that they contain, which can bee one and only one datatype. For example, arrays can be integer-, real-, string-, or character-valued, but elements of more than one such type cannot be contained in a PASCAL array. The first value, denoted by a 1 , equals 2. The two-dimensional array shown below has four columns and three rows. Each element of the array is referenced by its row,column coordinate. For example, the element whose value equals 9. An example of a two-dimensional array. The use of row-column indices or coordinates makes referencing elements of arrays convenient. It is especially useful to note that arrays can be indexed in loops. For example, a loop that would set all the elements of the array a in Figure 5. In the above PASCAL code fragment, each dimension of the array a has a lower and upper limit to the subscripts that are allowed. One finds the size of each dimension by subtracting the lower limit from the upper limit, then adding one. If an array is dimensioned as: Multiply the dimension sizes: In the early days of computing, it was very important to know how large arrays were, because computer memory was extremely limited. Today, with large memory models, one still must be careful not to specify array dimensions too large, but it is less of a problem than in the past. As we mentioned in class, one always initializes program variables to which file data is not assigned prior to computing a given expression. One can use the preceding loop structure to assign initial values to array elements in an efficient manner. A key problem with arrays is that they have fixed size. Hence, they are called static data structures. In a more advanced class, we would examine techniques for programming data structures called lists, which can expand and contract with the data that one puts into or takes out of the list. Another problem of arrays which we mentioned previously is that they are statically typed, i. In Section 2, we mentioned file operations such as open, read, write, and close, which we now discuss in some detail. In order to understand file structures, it helps to think of a disk drive in terms of a drawer in a filing cabinet. In each drawer, there are many files, which are usually contained in manila folders. In order to view, create, or modify the contents of a folder, one must first retrieve then open the folder. This is similar to initializing and opening a computer disk file. If one wants to view the file contents, then one must read the file, which holds for either physical or computer files. Similarly, creating new file contents or modifying existing file information is accomplished by writing to the file. After one has completed operations on a given file, it is returned to the file cabinet, to keep the work area neat this helps one find the file when it is next needed. However, these are within the purview of more advanced topics, and are not part of the basic file operations reviewed in these class notes. File addresses or references are expressed in terms of symbolic file handles, which are represented in PASCAL as names assigned to a given file. The following commands pertain: There are two types of file opening statements, one of which opens the file for reading input , the other for writing output.

DOWNLOAD PDF DATA STRUCTURES WITH ABSTRACT DATA TYPES AND PASCAL

Chapter 8 : Data Structures With Abstract Data Types And Pascal by Daniel F. Stubbs

EMBED (for calendrierdelascience.com hosted blogs and calendrierdelascience.com item tags).

Chapter 9 : abstraction - Abstract Data Type and Data Structure - Software Engineering Stack Exchange

Yingxu Wang, Xinming Tan, The Formal Design Models of Tree Architectures and Behaviors, International Journal of Software Science and Computational Intelligence, v.3 n.4, p, October