

ACHIEVER'S EDUCATION CONSULTANT
08141710097
SUMMARY OF CIT341 WITH PAST QUESTIONS

Facilitators/Tutors and Tutorials.....

The overall aims of this course are to introduce you to programming concepts as and algorithm techniques. Topics related to data structures and storage management are equally discussed. The bottom-up approach is adopted in structuring this course. We start with the basic building blocks of object-oriented programming concepts and move on to the fundamental principles of data structures and algorithms.

•Build up your capacity to evaluate different algorithm

Develop your competence in analysing data structures.

•Describe the basic operations on stacks, lists and queue data

Explain the notions of trees, hashing and binary search trees. Identify the basic concepts of object-oriented programming.

Working through this Course

We designed this course in a systematic way, so you need to work through it from Module one, Unit 1 through to Module 6, Unit 6. This will enable you appreciate the course better.

Course Materials

Basically, we made use of textbooks and online materials. You are expected to search for more literature and web references for further understanding. Each unit has references and web references that were used to develop them.

Online Materials

The website is designed to integrate with the print-based course materials. The structure follows the structure of the units and all the reading and activity numbers are the same in both media.

[...]

An investment in newer, more efficient technology will more than pay for itself in improved performance results. If your system does not meet the recommended specifications, you may experience considerably slower processing when working in the application.

Study Units

Each module comprises 5 units which you are expected to complete in 3 hours. The 6 modules and their units are listed below.

[...]

It uses java as a programming language to implement a variety of data structures, while modules five and six discuss the analysis of algorithms and algorithm.

Assessment

The course, Data Structures entails attending a three-hour final examination which contributes 50% to your final grading. The examination aims at testing your ability to apply the knowledge you have gain throughout the course, rather than your ability to memorise the materials. In preparing for the examination, it is essential that you receive the activities and Tutor-marked assignments you have completed in each unit. The other 50% will account for all the TMAs at the end of each unit.

This section proposes the number of weeks that you are expected to spend on the three modules comprising 30 units and the assignments that follow each of the units. We recommend that each unit with its associated TMA is completed in one week, bringing your study period to a maximum of 30 weeks.

[...]

Specific web address will be given for your reference. You may wish to read these to extend your knowledge beyond the required materials. They will not be assessed.

Facilitators/Tutors and Tutorials

About 20 hours of tutorials will be provided in support of this course. You will be notified of the dates, time and location for these tutorials, together with the name and phone number of your tutor as soon as you are allotted a tutorial group. Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties you might encounter and provide assistance to you during the course. You must mail your TMAs to your tutor well before the due date .

They will be marked by your tutor and returned to you as soon as possible. Do not hesitate to contact your tutor by phone or e-mail, if you need help. The following might be circumstances in which you would find help necessary. You do not understand any part of the study units or the assigned readings.

You have a question or problem with your tutor's comments on an assignment or with the grading of an assignment. You can raise any problem encountered in the course of your

study. You will learn a lot from participating in discussions actively. In the longer term, we hope you will get acquainted with the National Open University of Nigeria and we wish you every success in your future.

[...]

This unit introduces some basic concepts that the student needs to be familiar with before attempting to develop any software. It describes data type and data structures, explaining the operations that may be performed on them. The unit introduces you to the fundamental notions of data structures, thus guiding you through and facilitating your understanding of the subsequent units. Explain the relevance of data structures in programming.

Some data items may be used singly whilst others may be combined together and arranged to form other data items. The former are classified as ‘simple data types’ whereas the latter are classified as ‘data structures’. However, the following classification is appropriate for study at this level. Almost all programming languages explicitly include the notion of data type, though different languages may use different terminology.

Common data types in programming languages include those that represent integers, floating point numbers, and characters, and a language may support many more. A data type is abstract in the sense that it is independent of various concrete implementations. Object-oriented languages such as C++ and Java provide explicit support for expressing abstract data types by means of classes. A data structure is the implementation of an abstract data type in a particular programming language.

Data structures can also be referred to as “data aggregate”. A carefully chosen data structure will allow the most efficient algorithm to be used. Thus, a well-designed data structure allows a variety of critical operations to be performed using a few resources, both execution time and memory spaces as possible.

Linear Data Structures

Linear data structures are data structures in which individual data elements are stored and accessed linearly in the computer memory. Non-Linear Data Structures A non-linear data structure, as the name implies, is a data structure in which the data items are not stored linearly in the computer memory, but data items can be processed using some techniques or rules. Typical non-linear data structures to be studied in this course are Trees. In the design of many types of programmes, the choice of data structures is a primary design consideration, as experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final result depends heavily on choosing the best data structure.

In this unit, you have learned about the classification of abstract data type, commonly referred to as ADT. Finally, you have been able to appreciate the significance of data structures in developing high-quality programmes. What you have learned borders on the basic notions of data structures. You have just been nominated as a Programmer of a Software firm responsible for developing software for tertiary institutions.

Multi-Dimensional Arrays Application of Arrays

In this unit, you will learn about arrays, their declaration, dimensionality and applications. You will also learn how to distinguish between static and dynamic arrays. Each data item of an array is known as an element, and the elements are referenced by a common name known as the array name. A Java array is also called an object.

An array can contain data of the primitive data types. As it is an object, an array must be declared and instantiated. An array element can be accessed using an index value. The size of an array can be found using the length attribute.

DIM LAGOS will create 45 memory locations for storage of the elements of the array called LAGOS. Multi-dimensional arrays can be stored as linear arrays in order to reduce the computation time and memory.

Multi-dimensional Arrays

An array with dimensionality k , is often called k -dimensional.

Suppose we want to represent this simple two-dimensional array

Multi-dimensional arrays are typically represented by onedimensional arrays of references to other onedimensional arrays. The subarrays can be either the rows or columns. Arrays can be classified as static arrays, or dynamic arrays, which can be resized. Applications of Arrays
Arrays are employed in many computer applications in which data items need to be saved in the computer memory for subsequent reprocessing.

Due to their performance characteristics, arrays are used to implement other data structures, such as heaps, hash tables, dequeues, queues, stacks and strings. In this unit, you have learned about the arrays and their dimensionality. Finally, you have been able to distinguish between the static and dynamic arrays as well as understand the applications of arrays. What you have learned in this unit is focused on arrays, their declaration, classification and application.

In the next unit, we will discuss another data structure known as Lists. Describe a suitable data structure for details of stock items numbered in the range 1 to 100. Each stock item may be held at each of 20 locations. The number of items held at each location needs to be recorded.

Operation

What you will learn in this unit borders on Lists, their operations and implementations. Typical examples are given to facilitate the student's understanding of these features.

What is a List Data Structure?

A list data structure is a sequential data structure, i.e. a collection of items accessible one after

the other, beginning at the head and ending at the tail. It is a widely used data structure for applications which do not need random access. Lists differ from the stacks and queues data structures in that additions and removals can be made at any position in the list.

The main primitive operations of a list are known as: updates the contents of a node returns the value at a specified index returns the index in the list of a specified element reports whether the list is empty reports whether the list is full creates/initialises the list deletes the contents of the list

Creates the structure – i.e. ensures that the structure exists but contains no elements e.g. Initialise creates a new empty queue named Q e.g. Add adds the value X to list L at position 1, shifting subsequent elements up L. .

Array Lists

The position of each element is given by an index from 0 to $n-1$, where n is the number of elements. Additions near the start of the list take longer than additions near the middle or end. When an element is removed, subsequent elements must be shifted down, so removals near the start of the list take longer than removals near the middle or end. The Linked List is stored as a sequence of linked nodes.

The list can grow and shrink as needed. This is because each element of the list must be traversed until the required index is found. The time taken to add an element at any point in the list does not depend on the size of the list, as no shifts are required.

Additions near the end of the list take longer than additions near the middle or start. The same applies to the time taken to remove an element. Singly Linked Lists A singly linked list is a data structure in which the data items are chained in one direction. Figure 1 shows an example of a singly linked list.

Circularly Linked Lists

In a circularly linked list, the tail of the list always points to the head of the list.

Doubly Linked Lists

Add method will search for the correct place in the list to insert a new data item. Represent this statement as a list, identifying the different elements. In this unit you have learned about Lists. You have also been able to identify the elements of a List.

You should also have learned about operations and implementations of lists. What you have learned in this unit concerns the Lists, their operations and implementations. In the next unit, you shall learn about another linear data structure, known as Queues.

THE STACK DATA STRUCTURE

In this unit, we will look at an abstract data structure – the Stack Data Structure. A stack is a linear data structure in which all insertions and deletions of data are made only at one end of the stack, often called the top of the stack. For this reason, a stack is referred to as a LIFO structure. In such a stack, only the top plate is visible and accessible to the user, all other plates remain hidden.

As new plates are added, each new plate becomes the top of the stack, hiding each plate below, pushing the stack of plates down. As the top plate is removed from the stack, the plates pop back up, and the second plate becomes the top of the stack. Stacks are used extensively at every level of a modern computer system. Among other uses, stacks are used to run a Java, Virtual Machine, and the Java language itself has a class called «Stack», which can be used by the programmer.

Once the function is finished, the address that was saved is removed from the stack, and execution of the programme resumes. If a series of function calls occur, the successive return values are pushed onto the stack in LIFO order so that each function can return back to calling programme. Stacks support recursive function calls, subroutine calls, especially when “reverse polish notation” is involved. Solving a search problem, regardless of whether the approach is exhaustive or optimal, needs stack space.

Examples of exhaustive search methods are brute force and backtracking. Examples of optimal search exploring methods are branch and bound and heuristic solutions. All of these algorithms use stacks to remember the search nodes that have been noticed but not explored yet. Another common use of stacks at the architecture level is as a means of allocating and accessing memory.

TOP node returns its

These define collections of data which are fixed in size when the programme is compiled. They are created as the programme executes, and grow and shrink to accommodate the data being stored. In this unit, you have learned about the stack data structure. You have also been able to understand the basic operations on a stack.

You should also have learned about applications of stacks in computing. What you have learned in this unit concerns the stack data structure, their operations and applications. In this unit, the student will gain knowledge of the queue data structure as well as its applications and operations.

Typical examples are given to facilitate your understanding of these concepts. The queue data structure is characterised by the fact that additions are made at the end, or tail, of the queue while removals are made from the front, or head of the queue. For this reason, a queue is referred to as a FIFO structure .

[...]

In operating systems, queue structures are used to represent different programmes in the computer memory in the order in which they are executed. For example, if a programme, J is submitted before programme K, then programme J is queued before programme K in the

computer memory and programme J is executed before programme K...

The main primitive operations on a queue are known as

E.g. Initialise creates a new empty queue named Q. .

[...]

The array indices at which the head and tail of the queue are currently stored must be maintained. The head of the queue is not necessarily at index 0. The array can be a “circular array” in which the queue “wraps round” if the last index of the array is reached. Figure 1.4 below is an example of storing a queue in an array of length A queue requires a reference to the head node AND a reference to the tail node.

The following diagram describes the storage of a queue called Queue. The first node is accessed using the name Queue. Its data is accessed using Queue. The second node is accessed using Queue.

The last node is accessed using Queue. The new node is to be added at the tail of the queue. Tail should point to the new node, and the NextNode reference of the node previously at the tail of the queue should point to the DataItem of the new node. DataItem is returned.

A temporary reference Temp, is declared and set to point to head node in the queue . The only reference to the original head node is now Temp and the memory used by this node can then be freed.

Why is a queue referred to as a FIFO structure?

In this unit, you have learned about the queue data structure. Queue applications and operations were equally considered. What you have learned in this unit concerns queues, their operations and applications. The units that follow shall build upon issues discussed in this unit.

Taking up again the example given in figure 1.

[...]

In this unit, we will examine the basic idea of hashing. Hash keys and functions are equally described, giving the basic implementation of hash functions.

The Basic Idea

Ideally we would build a data structure for which both the insertion and find operations are O in the worst case. However, this kind of performance can only be achieved with complete a priori knowledge. We need to know beforehand specifically which items are to be inserted into the container. Unfortunately, we do not have this information in the general case.

So, if we cannot guarantee O performance in the worst case, then we make it our design objective to achieve O performance in the average case. This is because we can access the element of an array in constant time, whereas the same operation in a linked list takes O time. In the previous chapter, we considered two searchable containers--the ordered list and the sorted list. In the case of an ordered list, the cost of an insertion is O and the cost of the find operation is O . For a sorted list, the cost of insertion is O and the cost of the find operation for the array implementation.

Clearly, neither the ordered list nor the sorted list meets our performance objectives.

Defining a Hash Function

Similarly, to locate item x , we simply check to see if it is found at position $h-1$. This example illustrates how we can achieve O performance in the worst case when we have complete, a priori knowledge. We are designing a container which will be used to hold some number of items of a given set, K . In this context, we call the elements of the set K keys. The general approach is to store the keys in an array.

The position of a key in the array is given by a function, called a hash function, which determines the position of a given key directly from that key. In the general case, we expect the size of the set of keys, $|K|$, to be relatively large or even unbounded.

Consequently, what we need is a function

This function maps the set of values to be stored in the container to subscripts in an array of length M . This function is called a hash function. In general, since, the mapping defined by hash function will be a many-to-one mapping. That is, there will exist many pairs of distinct keys, x and y , such that, for which $h_x = h_y$. A good hash function tends to spread keys evenly in the array.

A good hash function is easy to compute. In reality, we cannot expect that the keys will always be integers. That is, the hash value of a key, x , is given by $g(x)$. The second involves mapping non-negative integers into the interval $[0, M-1]$.

Ideally, the two problems would be unrelated. That is, the choice of the function, f , would not depend on the choice of g and vice versa. Unfortunately, this is not always the case. However, if we are careful, we can design the functions in such a way that function.

Consequently, we have already examined several different alternatives for the function, g . As such, its interface provides methods for putting an object into the container, finding an object in the container, and removing an object from the container. The `HashTable` interface extends the `SearchableContainerInterface` defined in programme 1.0 below. One additional method, called `getLoadFactor`, is declared.

As shown in Figure 1.0, we define an `AbstractHashTable` class from which several concrete realizations are derived. `AbstractSearchableContainer` introduced in Programme 1.1 and it implements the `HashTable` interface. The methods f , g , and h correspond to the composition

discussed. The `f` method takes as an object and calls the `hashCode` method on that object to compute an integer.

The `g` method uses the division method of hashing defined to map an integer into the interval, where M is the length of the hash table. We will consider various ways of implementing hash tables. The position of an object in the array is determined by hashing the object. The main problem to be resolved is how to deal with collisions--two different objects cannot occupy the same array position at the same time.

In the following section, we consider an approach which solves the problem of collisions by keeping objects that collide in a linked list. Figure 1.1, shows a hash table that uses separate chaining to resolve collisions. The hash table is implemented as an array of linked lists. The linked list to which it is appended is determined by hashing that item.

Figure 1.1 illustrates an example in which there are $M=16$ linked lists. The twelve character strings «ett»-«tolv» have been inserted into the table using the hashed values and in the order given in Table 1.1. Notice that in this example, since $M=16$, the linked list is selected by the least significant four bits of the hashed value given in Table 1.0. In effect, it is only the last letter of a string which determines the linked list in which that string appears.

The principal characteristic of such applications is that keyed information needs to be frequently accessed and the access pattern is either unknown or known to be random. For example, hash tables are often used to implement the symbol table of a programming language compiler. A symbol table is used to keep track of information associated with the symbols used by a programmer. In this case, the keys are character strings and each key has, associated with it, some information about the symbol.

Suppose we are required to count the number of occurrences of each distinct word contained in a text file. We can do this easily using a hash or scatter table.

The static inner class Counter extends the class Int defined in

In addition to the functionality inherited from the base class, the Counter class adds the method `increment` which increases the value by one. The `wordCounter` method does the actual work of counting the words in the input file. The objects which are put into the hash table are all instances of the class Association. If none is found, a new association is created and inserted into the hash table.

The given word is used as the key of the new association and the value is a counter which is initialised to one. On the other hand, if there is already an association for the given word in the hash table, the corresponding counter is incremented. When the `wordCounter` method reaches the end of the input stream, it simply prints the hash table on the given output stream. The running time of the `wordCounter` method depends on a number of factors, including the number of different keys, the frequency of occurrence of each key, and the distribution of the keys in the overall space of keys.

Of course, the hash/scatter table implementation chosen has an effect as does the size of the table used. For a reasonable set of keys we expect the hash function to do a good job of spreading the keys, uniformly in the table. Under these ideal conditions, the running time should be $O(n)$, where n is the number of words in the input file.

What is a Hash table?

In this unit, you have learned about hashing, hash keys and functions. You have also been able to understand what hash tables are and how to implement hash functions. Finally, you have been able to appreciate the applications of hash tables. What you have learned borders on the basic notions of hashing, hash functions and hash tables and their applications.

Tree- Basics

In this unit, we will consider different kinds of trees as well as different tree traversal algorithms. In addition, we show how trees can be used to represent arithmetic expressions and how we can evaluate an arithmetic expression by doing a tree traversal. The president of the company is shown at the top of the tree and the vice-presidents are indicated below her. The president is called the root of the tree and the clerks are the leaves.

The following is a mathematical definition of a tree

It follows from the definition that the minimal tree is a tree comprising a single root node. The similarity becomes apparent when we consider the graphical representation of these trees shown in Figure 1.1. Finally, lines are drawn from r to the roots of each of the subtrees. Of course, trees drawn in this fashion are upside down.

Nevertheless, this is the conventional way in which tree data structures are drawn. In fact, it is understood that when we speak of "up" and "down," we do so with respect to this pictorial representation. For example, when we move from a root to a subtree, we will say that we are moving down the tree. The inverted pictorial representation of trees is probably due to the way that genealogical lineal charts are drawn.

Terminology

Each root of subtree of tree T is called a child of r . The term grandchild is defined in a similar manner. Definition Given a tree, T , containing the set of nodes R , a path in T is defined as a non-empty sequence of nodes, for the parent of the is $k-1$. In fact, if you count carefully, you should find that there are exactly 29 distinct paths in tree .

Since a binary tree is an N -ary tree, all of the results derived in the preceding section apply to binary trees. However, binary trees have some interesting characteristics that arise from the restriction that N is two. For example, there is an interesting relationship between binary trees and the binary number system. Binary trees are also very useful for the representation of mathematical expressions involving the binary operations such as addition and multiplication.

Binary trees are defined as follows

Binary trees are almost always considered to be ordered trees. Therefore, the two subtrees are called the left and right subtrees, respectively. Consider the two binary trees shown in Figure 1.2. This result is true regardless of the shape of the tree.

This process of walking through the tree is called a tree traversal. There are essentially two different methods in which to visit systematically all the nodes of a tree--depth-first traversal and breadth-first traversal.

An inorder traversal of the tree visits the nodes in the following order

In this section, we will consider the implementation of trees including general trees, N-ary trees, and binary trees. The implementations presented have been developed in the context of the abstract data type framework. That is, the various types of trees are viewed as classes of containers as shown in Figure 1.4. The Tree interface extends the Container interface defined in Programme 1.0.

getHeight

When a node is visited, the appropriate methods of the visitor are applied to that node.

Inserting Items into AVL Trees

This unit introduces Search Trees, describing successful and unsuccessful searching. What makes a tree into a search tree is that the keys do not appear in arbitrary nodes of the tree. Instead, there is a data ordering criterion which determines where a given key may appear in the tree in relation to the other keys in that tree. The subsequent sections present two related types of search trees, M-way search trees and binary search trees.

Since search trees are defined recursively, it is easy to define a recursive search method. If the tree is empty, the search fails. Otherwise, the keys contained in the root node are examined to determine if the object of the search is present. Notice that when x is not found in a given node, only one of the n subtrees of that node is searched.

Therefore, a complete tree traversal is not required. A successful search begins at the root and traces a downward path in the tree, which terminates at the node containing the object of the search. Clearly, the running time of a successful search is determined by the depth in the tree of object of the search. When the object of the search is not in the search tree, the search method described above traces a downward path from the root which terminates when an empty subtree is encountered.

The search method described above applies directly to binary search trees. The tree is an example of a particularly bad search tree because it is not really very tree-like at all. In fact, it is topologically isomorphic with a linear, linked list. In the worst case, a tree which contains n items has height $O(n)$. Therefore, in the worst case an unsuccessful search must visit $O(n)$ internal

nodes.

If $h=0$, then

Similarly, if the object of the search is at depth one, two nodes are visited, and so on. We shall assume that it is equally likely for the object of the search to appear in any node of the search tree. In that case, the average number of nodes visited during a successful search is $d + 1$, where d is the average of the depths of the nodes in a given tree. That is, given a binary search tree with $n > 0$ nodes, d is the depth of the node of the tree.

The quantity is called the internal path length. Clearly, the average depth of an internal node is equal to the internal path length divided by n , the number of nodes in the tree. Unfortunately, for any given number of nodes n , there are many different possible search trees. Furthermore, the internal path lengths of the various possibilities are not equal.

Therefore, to compute the average depth of a node in a tree with n nodes, we must consider all possible trees with n nodes. In the absence of any contrary information, we shall assume that all trees having n nodes are equiprobable and then compute the average depth of a node in the average tree containing n nodes. Let I be the average internal path length of a tree containing n nodes. Now consider an arbitrary tree, having internal nodes altogether, l of which are found in its left subtree, where.

Such a tree consists of a root, the left subtree with l internal nodes and a right subtree with $n-l-1$ internal nodes. If we select the key, k , to be the root of a binary search tree, then there are l keys, $k_1, k_2, \dots, k_l$, in its left subtree and $n-l-1$ keys, $k_{l+1}, k_{l+2}, \dots, k_{n-l-1}$, in its right subtree. If we assume that it is equally likely for any of the n keys to be selected as the root, then all the subtree sizes in the range are equally likely.

To solve this recurrence, we consider the case $n > 1$ and then multiply Equation by n to get

Since this equation is valid for any $n > 1$, by substituting $n-1$ for n , we can also write which is valid for $n > 2$.

Thus, we have shown the solution to the recurrence in the equation is the same as the solution of the recurrence

All successful searches terminate when the object of the search is found. In terms of the binary tree shown in Figure 1.0, a successful search terminates in one of the nodes which are drawn as circles and an unsuccessful search terminates in one of the boxes. The preceding analysis shows that the average number of nodes visited during a successful search depends on the internal path length, which is simply the sum of the depths of all the internal nodes. Similarly, the average number of nodes visited during an unsuccessful search depends on the external path length, which is the sum of the depths of all the external nodes.

The difference between the internal path length and the external path

Since the difference between the internal and external path lengths of any tree with n internal nodes is $2n$, then we can say the same thing about the average internal and external path lengths average over all search trees. Therefore, E , the average external path length of a binary search tree is given by A binary search tree with internal n nodes has $n+1$ external nodes. What they say is that the average number of nodes visited during either a successful or an unsuccessful search in the average binary search tree having n nodes is. We must remember, however, that these results are premised on the assumption that all possible search trees of n nodes are equiprobable.

It is important to be aware that in practice, this may not always be the case. The problem with binary search trees is that while the average running times for search, insertion, and withdrawal operations are all any one operation is still O in the worst case. This is so because we cannot say anything in general about the shape of the tree. For example, consider the two binary search trees shown in Figure .

A good balance condition ensures that the height of a tree with n

A good balance condition can be maintained efficiently. Adelson-Velskii and Landis were the first to propose the following balance condition and show that it has the desired characteristics. Clearly, all perfect binary trees are AVL balanced. What is not so clear is that heights of all trees that satisfy the AVL balance condition are logarithmic in the number of internal nodes.

Clearly, must have at least one subtree of height $h-1$ and that subtree must be. Since we want the smallest number of internal nodes, it must be. In fact, it can easily be shown by induction that for all is the Fibonacci number.

[...]

So, we have shown that the AVL balance condition satisfies the first criterion of a good balance condition--the height of an AVL balanced tree with n internal nodes is. What remains to be shown is that the balance condition can be efficiently maintained. To see that it can, we need to look at an implementation.

[...]

Programme introduces the AVLTree class which extends the BinarySearchTree class introduced in Programme . However, the inherited balance, attachKey and detachKey methods are overridden and a number of new methods are declared. Programme indicates that an additional field is added in the AVLTree class. To make it possible to verify the AVL balance condition in constant time, the field, height, has been added.

In this way, it is possible for the getHeight method to run in constant time--all it needs to do is to return the value of the height field.

Inserting Items into an AVL Tree

First, the item is inserted into the tree using the usual method for insertion in binary search trees. Just as in a regular binary search tree, items are inserted into AVL trees by attaching them to the leaves. Assuming that the item is not already in the tree, the search is unsuccessful and terminates at an external, empty node. The item to be inserted is placed in that external node.

Of course, when an item is inserted in a tree, the height of the tree may increase by one. Sometimes increasing the height of a subtree does not violate the AVL balance condition. Now, suppose that insert an item into, its height may increase by one to. The resulting tree is still AVL balanced since.

On the other hand, if left subtree and the insertion of an item into the increases the height of that tree to, the AVL balance condition is no longer satisfied because. Therefore, it is necessary to change the structure of the tree to bring it back into balance. The method for removing items from an AVL tree is inherited from the BinarySearchTree class in the same way as AVL insertion. The balance method is discussed above.

GARBAGE COLLECTION AND OTHER HEAP

While Java provides the means to create an object, the language does not provide the means to destroy an object explicitly. As long as a programme contains a reference to some object instance, the Java virtual machine is required to ensure that the object exists. The situation which arises when a programme contains a reference to a destroyed object is called a dangling reference . If a programme somehow loses track of an object it has created then that object cannot be destroyed.

And if the object is never destroyed, the memory occupied by that object cannot be used again by the programme. If we run a programme that has a memory leak for a very long time, it is quite possible that it will exhaust all the available memory and eventually fail because no new objects can be created. It would seem that by disallowing the explicit destruction of objects, a Java programme is doomed to eventual failure due to memory exhaustion. Garbage collection is usually invoked when the total amount of memory allocated to a Java programme exceeds some threshold.

Typically, the programme is suspended while the garbage collection is done. The costs of doing so include the direct costs of waste disposal as well as the damage to the environment caused by the manufacturing, distribution, and ultimate disposal of products. The slogan ``reduce, reuse, recycle," prescribes three strategies for reducing the environmental costs associated with waste materials. A Java programme that creates excessive garbage may require more frequent garbage collection than a programme that creates less garbage.

Since garbage collection can take a significant amount of time to do, it makes sense to use strategies that decrease the cost of garbage collection. The preceding section presents strategies for avoiding garbage collection. Suppose the amount of memory required is very close to the amount of memory available for use by the Java virtual machine. The performance of such a programme is going to depend on the ability of the garbage collector to find and reclaim as much unused storage as possible.

Otherwise, the garbage collector will run too often. Since the garbage collector collects only unreferenced objects it is necessary to eliminate all references to objects which are no longer needed. This is done by assigning the value null to every variable that refers to an object that is no longer needed. Consequently, helping the garbage collector requires a programme to do a bit more work.

The difficulty in garbage collection is not the actual process of collecting the garbage--it is the problem of finding the garbage in the first place. An object is considered to be garbage when no references to that object exist. A simple expedient is to keep track in each object, the total number of references to that object. That is, we add a special field to each object called a reference count.

The idea is that the reference count field is not accessible to the Java programme. Instead, the reference count field is updated by the Java virtual machine itself.

Now consider the following sequence of statements

In general, every time one reference variable is assigned to another, it may be necessary to update several reference counts. As shown in Figure , two Integer objects are created, each with a reference count of one. Now, suppose we assign q to p using the code sequence given above. Typically, this means an extra word of storage must be allocated in each object.

Second, every time one reference is assigned to another, the reference counts must be adjusted as above. This increases significantly the time taken by assignment statements. The advantage of using reference counts is that garbage is easily identified. When it becomes necessary to reclaim the storage from unused objects, the garbage collector needs only to examine the reference count fields of all the objects that have been created by the programme.

If the reference count is zero, the object is garbage. It is not necessary to wait until there is insufficient memory before initiating the garbage collection process. We can reclaim memory used by an object immediately when its reference goes to zero. In this way, garbage may be collected incrementally as it is created.

Instead, garbage is allowed to accumulate until all available memory has been exhausted. When that happens, the execution of the programme is suspended temporarily while the mark-and-sweep algorithm collects all the garbage. Once all unreferenced objects have been reclaimed, the normal execution of the programme can resume. The mark-and-sweep algorithm is called a tracing garbage collector because it traces out the entire collection of objects that are directly or indirectly accessible by the programme.

The objects that a programme can access directly are those objects which are referenced by local variables on the processor stack as well as by any static variables that refer to objects. In the context of garbage collection, these variables are called the roots. An accessible object is said to be live. Conversely, an object which is not live is garbage.

The first phase is called the mark phase. The second phase is called the sweep phase. By default, all objects are unmarked when they are created. Thus, the marked field is initially false.

Notice that this recursive mark algorithm does nothing when it encounters an object that has already been marked. Consequently, the algorithm is guaranteed to terminate. And it terminates only when all accessible objects have been marked. In its second phase, the mark-and-sweep algorithm scans through all the objects in the heap, in order to locate all the unmarked objects.

The storage allocated to the unmarked objects is reclaimed during the scan. At this point, all live objects have been marked. Finally, Figure shows the objects left after the sweep phase has been completed. Only live objects remain in memory and the marked fields have all been set to false again.

Figure: Mark-and-sweep garbage collection

This is the main advantage of mark-and-sweep over the reference counting technique presented in the preceding section. The main disadvantage of the mark-and-sweep approach is the fact that normal programme execution is suspended while the garbage collection algorithm runs. In particular, this can be a problem in a programme that interacts with a human user or that must satisfy real-time execution constraints. Fragmentation is a phenomenon that occurs in a long-running programme that has undergone garbage collection several times.

The problem is that objects tend to become spread out in the heap. Live objects end up being separated by many, small unused memory regions. The problem in this situation is that it may become impossible to allocate memory for an object. While there may indeed be sufficient unused memory, the unused memory is not contiguous.

Since objects typically occupy consecutive memory locations, it is impossible to allocate storage. The mark-and-sweep algorithm does not address fragmentation.

What do you understand by garbage collection?

In this unit, you have learned about garbage and garbage collection. You have also been able to learn about reference counting garbage collection and mark-and-sweep garbage collection. Finally, you have been able to understand the fragmentation problem. What you have learned borders on the garbage, garbage collection and the fragmentation problem.

The mark-and-sweep algorithm is referred to as a tracing garbage collection.

MEMORY ALLOCATION

There are many common ways to perform this, with different strengths and weaknesses.

[...]

These techniques can often be used in combination. In the first fit algorithm, the allocator keeps

a list of free blocks and, on receiving a request for memory, scans along the list for the first block that is large enough to satisfy the request. Fragmentation is the inability to use memory because of the arrangement of memory already in use. It is usually divided into external fragmentation and internal fragmentation.

In a buddy system, the allocator will only allocate blocks of certain sizes, and has many free lists, one for each permitted size. The permitted sizes are usually either powers of two, or form a Fibonacci sequence, such that any block except the smallest, can be divided into two smaller blocks of permitted sizes. When the allocator receives a request for memory, it rounds the requested size off to a permitted size, and returns the first block from that size's free list. When blocks are recycled, there may be some attempt to merge adjacent blocks into ones of a larger permitted size .

To make this easier, the free lists may be stored in order of address. The main advantage of the buddy system is that coalescence is cheap because the «buddy» of any free block can be calculated from its address.

For example, an allocator in a binary buddy system might have sizes of

It might start off with a single block of 64 kB. If the application requests a block of 8 kB, the allocator would check its 8 kB free list and find no free blocks of that size. A Fibonacci buddy system might use block sizes 16, 32, 48, 80, 128, 208,... bytes, such that each size is the sum of the two preceding sizes. The rounding typically leads to a significant amount of wasted memory, which is called internal fragmentation.

This can be reduced by making the permitted block sizes closer together.

Suballocators are usually written for one of the following reasons

One of the most common uses of a suballocator is to supply the application with objects of one size. There are dangers involved in making use of special knowledge of the application's memory requirements. If those requirements change, then the performance of the suballocator is likely to be much worse than that of a general allocator. It is often better to have a memory manager that can respond dynamically to changing requirements.

What do you understand by the phrase ‘first fit’?

In this unit, you have learned about memory allocation. You have also been able to learn about the first fit, best fit and buddy systems of memory allocation. What you have learned borders on memory allocation and the techniques of memory allocation.

[...]

This unit will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Java programming language. An object is a

software bundle of related state and behaviour. Software objects are often used to model the real-world objects that you find in everyday life.

This unit explains how state and behaviour are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner. Objects are key to understanding object-oriented technology. Dogs have state and behaviour. Bicycles also have state and behaviour .

Take a minute right now to observe the real-world objects that are in your immediate area. «.

A bicycle modeled as a software object

Once created, an object can be easily passed around inside the system. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code. This is analogous to fixing mechanical problems in the real world. A class is a blueprint or prototype from which objects are created.

It intentionally focuses on the basics, showing how even a simple class can clearly model state and behaviour. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles. A class is the blueprint from which individual objects are created.

The following Bicycle class is one possible implementation of a bicycle: class Bicycle {

You may have noticed that the Bicycle class does not contain a main method.

Here's a BicycleDemo class that creates two separate Bicycle objects and invokes their methods: class BicycleDemo { // Create two different Bicycle objects // Invoke methods on those objects

Inheritance provides a powerful and natural mechanism for organising and structuring your software.

A hierarchy of bicycle classes

This gives MountainBike all the same fields and methods as Bicycle, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read. However, you must take care to properly document the state and behaviour that each superclass defines, since that code will not appear in the source file of each subclass. When a class implements an interface, it promises to provide the behaviour published by that interface.

In its most common form, an interface is a group of related methods with empty bodies. Implementing an interface allows a class to become more formal about the behaviour it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. A package is a namespace for organising classes and interfaces in a logical manner.

Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface provided by the Java platform. Conceptually, you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another.

« This library is known as the »Application Programming Interface” or “API for short. Its packages represent the tasks most commonly associated with general-purpose programming. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work. Object-oriented programming is a programming language model organised around «objects» rather than «actions» and data rather than logic.

Historically, a programme has been viewed as a logical procedure that takes input data, processes it, and produces output data. The programming challenge was seen as how to write the logic, not how to define the data. Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them. A real instance of a class is called an «object» or, in some environments, an «instance of a class.» The object or class instance is what you run in the computer.

Its methods provide computer instructions and the class object characteristics provide relevant data. You communicate with objects - and they communicate with each other - with well-defined interfaces called messages.

The concepts and rules used in object-oriented programming provide these important benefits

The concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics. Since a class defines only the data it needs to be concerned with, when an instance of that class is run, the code will not be able to accidentally access other programme data. The definition of a class is reusable not only by the programme for which it is initially created but also by other object-oriented programmes. The concept of data classes allows a programmer to create any new data type that is not already defined in the language itself.

One of the first object-oriented computer languages was called Smalltalk. The Java programming language is designed especially for use in distributed applications on corporate networks and the Internet. In this unit you have learned about the object-oriented programming concepts- objects, class, inheritance, interface and package. You have also been able to understand object-oriented programming in general.

What you have learned borders on the basic concepts of object-oriented programming.

VARIABLES

In this unit, you will learn about variables and their naming conventions. Explain the naming conventions of variables. In computer science, a variable is a symbolic representation used to denote a quantity or expression. However, in computer programming, a variable is a special value that has the property of being able to be associated with another value .

Variables are usually named by an identifier, but they can be anonymous, and variables can be associated with other variables. This convention of matching identifiers to values is but one of several alternative programmatic conventions for accessing values in computer memory.

The Java programming language defines the following kinds of variables

Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where auto-generated names will contain the dollar sign, but your variable names should always avoid using it. Subsequent characters may be letters, digits, dollar signs, or underscore characters. When choosing a name for your variables, use full words instead of cryptic abbreviations.

Doing so will make your code easier to read and understand. In this unit, you have learned about variables, types of Java programming language variable as well as their naming conventions. What you have learned in this unit is based on variables and the conventions for naming them.

The Conditional Operators

What you will learn in this unit borders on operators. The common types of operators will equally be discussed.

[...]

With that in mind, the following discussion focuses first on the operators that you're most likely to use on a regular basis, and ends focusing on those that are less common. Each discussion is accompanied by sample code that you can compile and run. Studying its output will help reinforce what you've just learned. The only symbol that might look new to you is «%», which divides one operand by another and returns the remainder as its result.

String secondString = "a concatenated string.

[...]

Will both end in result being incremented by one. If you are just performing a simple increment/decrement, it doesn't really matter which version you choose.

The Switch Statement

The programme displays the name of the month, based on the value of month, using the switch statement. The body of a switch statement is known as a switch block. Any statement

immediately contained by the switch block may be labeled with one or more cases or default labels. The switch statement evaluates its expression and executes the appropriate case.

You can decide which one to use based on readability and other factors. Each break statement terminates the enclosing switch statement.

Number of Days = 29

Technically, the final break is not required because flow would fall out of the switch statement anyway. However, we recommend using a break so that modifying the code is easier and less error-prone. The default section handles all values that aren't explicitly handled by one of the case sections. The while statement continually executes a block of statements while a particular condition is true.

The While and Do-While Statements

The while statement evaluates expression, which must return a boolean value. If the expression evaluates to true, the while statement executes the statement in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false.



National Open University of Nigeria
Plot 91, Cadastral Zone, Nnamdi Azikiwe Expressway, Jabi - Abuja
Faculty of Science
NOVEMBER, 2018 EXAMINATIONS

COURSE CODE: CIT341

COURSE TITLE: INTRODUCTION TO DATA STRUCTURE

CREDIT: 3 Units

TIME ALLOWED: 2½ Hours

INSTRUCTION: Answer Question One (1) and any other Four (4)

Question 1

With appropriate examples, explain Abstract Data Type?

(3 marks)

What are the four properties of array lists?

(4 marks)

Briefly explain any four properties of the linked List.

(4 marks)

State and explain two basic operations in which a stack can be implemented.

(4 marks)

What do you understand by Dynamic Programming and what are the four major steps involve in dynamic programming design?

(5 marks)

Discuss the reason why dynamic programming relies on a principle of optimality.

(2 Marks)

Question 2

Explain the procedure of storing a Queue in a Dynamic Data Structure (5 marks)

Explain the concept of Hash Function (4 marks)

What do you understand by Hash Table? (3 marks)

Question 3

Hash tables are often used to implement the *symbol table* of a programming language compiler.

Explain the function of symbol table in a compiler (3 marks)

With the aid of a diagram explain how a hash table uses Separate Chaining to resolve collisions (6 marks)

Write short on the Binary Tree (3 marks)

Question 4

Critically differentiate between a Preorder Traversal and Postorder Traversal (7 marks)

With the aid of diagram, explain the procedure of searching a Binary Tree (5 marks)

Question 5

Discuss the procedure of removing Items from an AVL Tree (3 marks)

Elucidate on what is meant by Reduce, Reuse, Recycle (6 marks)

What do you understand by the Fragmentation Problem (3 marks)

Question 6

Critically explain three different ways of performing memory allocation (12 marks)