EDO UNIVERSITY IYAMHO

Department of Computer Science

CSC 214: ALGORITHM AND COMPLEXITY ANALYSIS

Instructor: *David Acheme*, email:acheme.david@edouniversity.edu.ng
Lectures: Tuesday, 8am – 12.10 pm, LT1, phone: (+234) 8062889197
Office hours: Wednesday, 2.30 to 3.30 PM (just before class), Office: Faculty of Science Rm 4

INTENDED LEARNING OUTCOMES

General overview of lecture: In this course we are going to look at designing algorithms and how they depend on the design of suitable data structures, and how some structures and algorithms are more efficient than others for the same task. We'll concentrate on a few basic tasks, such as storing, sorting and searching data, that underlie much of computer science. Firstly, we will study some key data structures, such as arrays, lists, queues, stacks and trees, and then move on to explore their use in a range of different searching and sorting algorithms.

Learning outcomes: At the completion of this course, students are expected to:

1. Fundamental questions about algorithms
    a. Specification
    b. Verification
    c. Performance Analysis
2. Arrays, Loops, List, Stacks and Queues with their implementation in Java
3. Searching
    a. Requirements for searching
    b. Specification of the search problem
    c. A simple algorithm: Linear Search
    d. The Binary Search

Assignments & Grading

Academic Honesty: All classwork should be done independently, unless explicitly stated otherwise on the assignment handout.
You may discuss general solution strategies, but must write up the solutions yourself.
If you discuss any problem with anyone else, you must write their name at the top of your assignment, labeling them "collaborators".
NO LATE HOMEWORKS ACCEPTED
Turn in what you have at the time it's due.
All home works are due at the start of class.
If you will be away, turn in the homework early.
Exams:
Final, comprehensive (according to university schedule): ~ 70% of final grade

Books:

INTRODUCTION

In this course we are going to look at designing algorithms. We will see how they depend on the design of suitable data structures, and how some structures and algorithms are more efficient than others for the same task. We'll concentrate on a few basic tasks, such as **storing**, **sorting** and **searching** data, that underlie much of computer science, but the techniques discussed will be applicable much more generally. We will start by studying some key data structures, such as arrays, lists, queues, stacks and trees, and then move on to explore their use in a range of different searching and sorting algorithms. This will lead us on to consider approaches for the efficient storage of data in hash tables. Throughout, we'll investigate the computational efficiency of the algorithms we develop, and gain intuitions about the pros and cons of the various potential approaches for each task. While we will restrict ourselves to implementing the various data structures and algorithms in java programming language we will also specify them in simple pseudocode that can easily be implemented in any other appropriate language.

ALGORITHMS AND PROGRAMS: WHAT IS THE DIFFERENCE

An algorithm for a given task is finite sequence of steps, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time". As such, an algorithm must be precise enough to be understood by human beings. However, in order to be executed by a computer, we need a program that is written in a rigorous formal language; and since computers are quite inflexible compared to the human mind, programs usually need to contain more details than algorithms. In this course we shall concentrate on the design of algorithms but we shall also look at their implementations in Java.

Algorithms can obviously be described in plain English, and we will sometimes do that. However, for computer scientists it is usually easier and clearer to use something that comes somewhere in between formatted English and computer program code, but is not runnable because certain details are omitted. This is called **pseudocode**. Often we will use segments of psudocode that are very similar to the languages we are interested in, e.g. the overlap of C and Java, with the advantage that they can easily be inserted into runnable programs.

FUNDAMENTAL QUESTIONS ABOUT ALGORITHMS

Given an algorithm to solve a particular problem, we are naturally led to ask:

1. What is it supposed to do?

2. Does it really do what it is supposed to do?

3. How efficiently does it do it?

Technically, we refer to these as:

1. Specification
2. Verification and
3. Performance analysis

SPECIFICATION ANALYSIS

The specification formalizes the crucial details of the problem that the algorithm is trying to solve. Sometimes that will be based on a particular representation of the associated data, sometimes it will be presented more abstractly. Typically, it will have to specify how the inputs and outputs of the algorithm are related, though there is no general requirement that the specification is complete or non-ambiguous. For simple problems, it is often easy to see that a particular algorithm will always work, i.e. that it satisfies its specification.

VERIFICATION ANALYSIS

This refers to verifying whether the algorithm is indeed correct. In general, testing on a few particular inputs can be enough to show that the algorithm is incorrect. However, since the number of different potential inputs for most algorithms is infinite in theory, and huge in practice, more than just testing on particular cases is needed to be sure that the algorithm satisfies its specification, correctness proofs are used to show that an algorithm satisfies its specification. Formal verification techniques are complex and will be taught in later courses.

PERFORMANCE ANALYSIS

Finally, the efficiency or performance of an algorithm relates to the resources required by it, such as how quickly it will run, or how much computer memory it will use. This will usually depend on the problem instance size, the choice of data representation, and the details of the algorithm. Indeed, this is what normally drives the development of new data structures and algorithms.

EFFICIENCY AND COMPLEXITY ANALYSIS

As we have already noted that, when developing algorithms, it is important to consider how efficient they are, so we can make informed choices about which are best to use in particular circumstances. So, before moving on to study increasingly complex data structures and algorithms, we first look in more detail at how to measure and describe their efficiency.

TIME VERSUS SPACE COMPLEXITY

When creating software for serious applications, there is usually a need to judge how quickly an algorithm or program can complete the given tasks. For example, if you are programming a flight booking system, it will not be considered acceptable if the travel agent and customer have to wait for half an hour for a transaction to complete. It certainly has to be ensured that the waiting time is reasonable for the size of the problem, and normally faster execution is better. We talk about the time complexity of the algorithm as an indicator of how the execution time depends on the size of the data structure. Another important efficiency consideration is how much memory a given program will require for a particular task, though with modern computers this tends to be less of an issue than it used to be. Here we talk about the *space* *complexity* as how the memory requirement depends on the size of the data structure. For a given task, there are often algorithms which trade time for space, and vice versa. For example, we will see that, as a data storage device, hash tables have a very good time complexity at the expense of using more memory than is needed by other algorithms. It is usually up to the algorithm/program designer to decide how best to balance the trade-off for the application they are designing.

WORST VERSUS AVERAGE COMPLEXITY

Another thing that has to be decided when making efficiency considerations is whether it is the average case performance of an algorithm/program that is important, or whether it is more important to guarantee that even in the worst case the performance obeys certain rules. For many applications, the average case is more important, because saving time overall is usually more important than guaranteeing good behaviour in the worst case. However, for time-critical problems, such as keeping track of aeroplanes in certain sectors of air space, it may be totally unacceptable for the software to take too long if the worst case arises. Again, algorithms/programs often trade-off efficiency of the average case against efficiency of the worst case. For example, the most efficient algorithm on average might have a particularly bad worst case efficiency. We will see particular examples of this when we consider efficient algorithms for sorting and searching.

PERFORMANCE MEASURES OF ALGORITHMS

These days, we are mostly interested in time complexity. For this, we first have to decide how to measure it. Something one might try to do is to just implement the algorithm and run it, and see how long it takes to run, but that approach has a number of problems. For one, if it is a big application and there are several potential algorithms, they would all have to be programmed first before they can be compared. So a considerable amount of time would be wasted on writing programs which will not get used in the final product. Also, the machine on which the program is run, or even the compiler used, might influence the running time. You would also have to make sure that the data with which you tested your program is typical for the application it is created for. Again, particularly with big applications, this is not really feasible. This empirical method has another disadvantage: it will not tell you anything useful about the next time you are considering a similar problem.

Therefore complexity is usually best measured in a different way. First, in order to not be bound to a particular programming language or machine architecture, it is better to measure the efficiency of the algorithm rather than that of its implementation. For this to be possible, however, the algorithm has to be described in a way which very much looks like the program to be implemented, which is why algorithms are usually best expressed in a form of pseudocode that comes close to the implementation language.

What we need to do to determine the time complexity of an algorithm is count the number of times each operation will occur, which will usually depend on the size of the problem. The size of a problem is typically expressed as an integer, and that is typically the number of items that are manipulated. For example, when describing a search algorithm, it is the number of items amongst which we are searching, and when describing a sorting algorithm, it is the number of items to be sorted. So the complexity of an algorithm will be given by a function which maps the number of items to the (usually approximate) number of time steps the algorithm will take when performed on that many items.

In the early days of computers, the various operations were each counted in proportion to their particular `time cost', and added up, with multiplication of integers typically considered much more expensive than their addition. In today's world, where computers have become much faster, and often have dedicated floating-point hardware, the differences in time costs have become less important. However, we still we need to be careful when deciding to consider all operations as being equally costly - applying some function, for example, can take much longer than simply adding two numbers, and swaps generally take many times longer than comparisons. Just counting the most costly operations is often a good strategy.

ARRAYS, ITERATION, INVARIANTS

Data is ultimately stored in computers as patterns of bits, though these days most programming languages deal with higher level objects, such as characters, integers, and floating point numbers. Generally, we need to build algorithms that manipulate collections of such objects, so we need procedures for storing and sequentially processing them.

ARRAYS

In computer science, the obvious way to store an ordered collection of items is as an array. Array items are typically stored in a sequence of computer memory locations, but to discuss them, we need a convenient way to write them down on paper. We can just write the items in order, separated by commas and enclosed by square brackets. Thus,

[1, 4, 17,3, 90, 79, 4, 6, 81] is an example of an array of integers. If we call this array a, we can write it as:

a = [1, 4, 17,3, 90, 79, 4, 6, 81]

This array has 9 items, and hence we say that its size is 9. In everyday life, we usually start counting from 1. When we work with arrays in computer science, however, we more often (though not always) start from 0. Thus, for our array a, its positions are 0, 1; 2, : : : ; 7, 8. The element in the 8th position is 81, and we use the notation a[8] to denote this element. More generally, for any integer i denoting a position, we write a[i] to denote the element in the ith position. This position i is called an index (and the plural is indices). Then, in the above example, a[0] = 1, a[1] = 4, a[2] = 17, and so on. It is worth noting at this point that the symbol = is quite overloaded. In mathematics, it stands for equality. In most modern programming languages, = denotes assignment, while equality is expressed by ==. We will typically use = in its mathematical meaning, unless it is written as part of code or pseudocode.

We say that the individual items a[i] in the array a are accessed using their index i, and one can move sequentially through the array by incrementing or decrementing that index, or jump straight to a particular item given its index value. Algorithms that process data stored as arrays will typically need to visit systematically all the items in the array, and apply appropriate operations on them.

Loops and Iteration

The standard approach in most programming languages for repeating a process a certain number of times, such as moving sequentially through an array to perform the same operations on each item, involves a loop. In pseudocode, this would typically take the general form

```
For i = 1,...,N,
do something
```

and in programming languages like C and Java this would be written as the for-loop

```
for( i = 0 ; i < N ; i++ ) {
// do something
}
```

in which a counter i keep tracks of doing \the something'' N times. For example, we could compute the sum of all 20 items in an array a using

```
for( i = 0, sum = 0 ; i < 20 ; i++ ) {
sum += a[i];
}
```

We say that there is iteration over the index i. The general for-loop structure is

```
for( initialization ; condition ; update )
 {
        repeated process
}
```

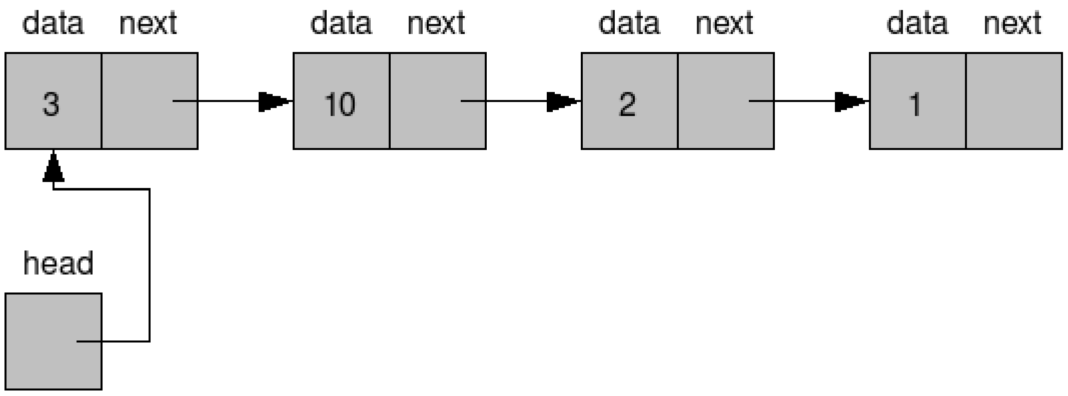
in which any of the four parts are optional. One way to write this out explicitly is initialization

```
if ( not condition ) go to loop finished
loop start
repeated process
update
if ( condition ) go to loop start
loop finished
```

In this module, we will regularly make use of this basic loop structure when operating on data stored in arrays, but it is important to remember that different programming languages use different syntax, and there are numerous variations that check the condition to terminate the repetition at different points.

Linked List Data Structure

A linked list is a data structure consisting of a group of nodes which together represent a sequence. Each node is composed of a data and a link or reference to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence. The last node is linked to a terminator used to signify the end of the list.



Linked lists are the simplest and most common data structures. They can be used to implement several other abstract data types, including lists, stacks, queues, associative arrays, and S-expressions, etc.

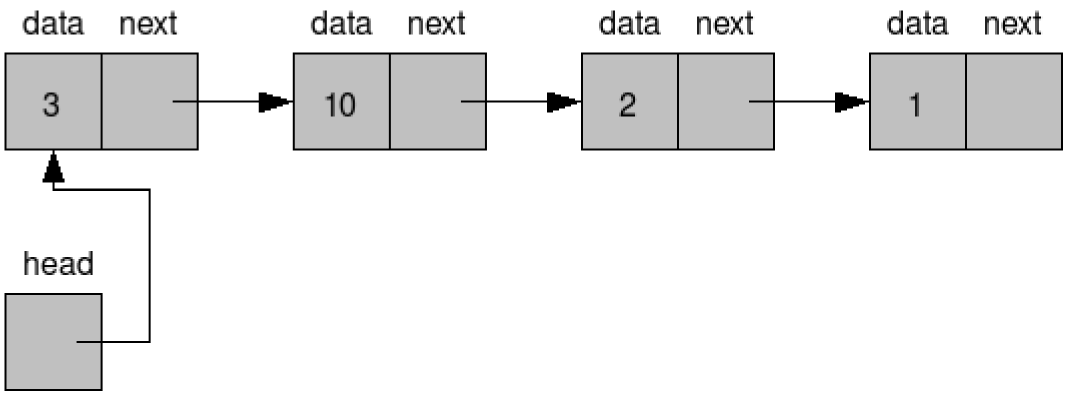
The benefits of a linked list over a conventional array is that the linked list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list.

On the other hand, simple linked lists do not allow random access to the data, or by using indexing. Thus, many basic operations like obtaining the last node of the list, or finding a node with required data, or locating the place where a new node should be inserted, may require scanning most of the list elements.

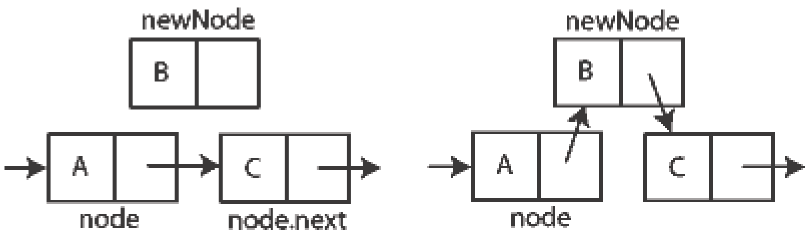
Singly linked list implementation

Singly Linked Lists are a type of data structure. It is a type of list. In a singly linked list each node in the list stores the contents of the node and a pointer or reference to the next node in the list. It does not store any pointer or reference to the previous node. It is called a singly linked list because each node only has a single link to another node. To store a single linked list, you only need to store a reference or pointer to the first node in that list. The last node has a pointer to nothingness to indicate that it is the last node.

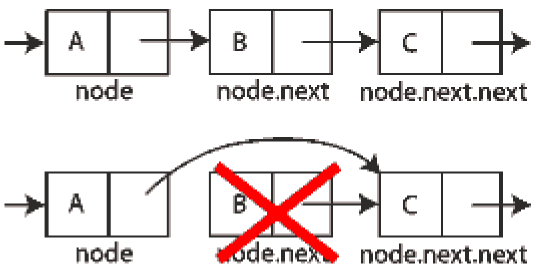
Here is the pictorial view of singly linked list:



Here is the pictorial view of inserting an element in the middle of a singly linked list:



Here is the pictorial view of deleting an element in the middle of a singly linked list:



Below shows the java implementation of singly linked list:

Queue Data Structure

A queue is a kind of abstract data type or collection in which the entities in the collection are kept in order and the only operations on the collection are the addition of entities to the rear terminal position, called as enqueue, and removal of entities from the front terminal position, called as dequeue. The queue is called as First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a peek or front operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a linear data structure, or more abstractly a sequential collection.

Queue introduction & array based implementation

A queue is a kind of abstract data type or collection in which the entities in the collection are kept in order and the only operations on the collection are the addition of entities to the rear terminal position, called as enqueue, and removal of entities from the front terminal position, called as dequeue. The queue is called as First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a peek or front operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a linear data structure, or more abstractly a sequential collection.

Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.

Queue Operations:

enqueue: Adds an item onto the end of the queue.
front: Returns the item at the front of the queue.
dequeue: Removes the item from the front of the queue.

Overflow State: A queue may be implemented to have a bounded capacity. If the queue is full and does not contain enough space to accept an entity to be pushed, the queue is then considered to be in an overflow state.

Underflow State: The dequeue operation removes an item from the top of the queue. A dequeue operation either reveals previously concealed items or results in an empty queue, but, if the queue is empty, it goes into underflow state, which means no items are present in queue to be removed.

Efficiency of Queue

The time needed to add or delete an item is constant and independent of the number of items in the queue. So both addition and deletion can be O(1) operation.

Stacks

A Stack is an abstract data type or collection where in Push,the addition of data elements to the collection, and Pop, the removal of data elements from the collection, are the major operations performed on the collection. The Push and Pop operations are performed only at one end of the Stack which is referred to as the 'top of the stack'.

In other words,a Stack can be simply defined as Last In First Out (LIFO) data structure,i.e.,the last element added at the top of the stack(In) should be the first element to be removed(Out) from the stack.

Stack introduction & implementation

A Stack is an abstract data type or collection where in Push,the addition of data elements to the collection, and Pop, the removal of data elements from the collection, are the major operations performed on the collection. The Push and Pop operations are performed only at one end of the Stack which is referred to as the 'top of the stack'.

In other words,a Stack can be simply defined as Last In First Out (LIFO) data structure,i.e.,the last element added at the top of the stack(In) should be the first element to be removed(Out) from the stack.

Stack Operations:

Push: A new entity can be added to the top of the collection.
Pop: An entity will be removed from the top of the collection.
Peek or Top: Returns the top of the entity with out removing it.

Overflow State: A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state.

Underflow State: The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into underflow state, which means no items are present in stack to be removed.

A stack is a restricted data structure, because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order.

Elements are removed from the stack in the reverse order to the order of their addition. Therefore, the lower elements are those that have been on the stack the longest.

Efficiency of Stacks

In the stack, the elements can be push or pop one at a time in constant O(1) time. That is, the time is not dependent on how many items are in the stack and is therefore very quick. No comparisons or moves are necessary.