CSCI2100 Data Structures

Ryan Chan

March 10, 2025

Abstract

This is a note for **CSCI2100 Data Structures** for self-revision purpose ONLY. Some contents are taken from lecture notes and reference book.
Mistakes might be found. So please feel free to point out any mistakes.
Contents are adapted from the lecture notes of CENG3420, prepared by Irwin King, as well as some online resources.

Contents

Chapter 1

Introduction

1.1 Overview

A **data structure** is a way to organize and store data in a computer program, allowing for efficient access and manipulation.

An **algorithm** is different from a **program**. An algorithm is a process or set of rules used for calculation or problem-solving. It is a step-by-step outline or flowchart showing how to solve a problem. A program, on the other hand, is a series of coded instructions that control the operation of a computer or other machines. It is the implemented code of an algorithm.

For example, to solve the greatest common divisor (GCD) problem, we can use the following algorithm.

Algorithm 1.1: Euclid's Algorithm

Data: $m, n \in \mathbb{Z}^+$
Result: GCD(m, n)

```
1 while m > 0 do
2 |  if n > m then
3 |  |  swap m and n
4 |  subtract n from m
5 return n
```

By using a mathematical method to prove this algorithm, we can show that it is correct, provided that it terminates.

Having proved the correctness, we also need to use different test cases to check if there is anything wrong with the coding or the proof. We should consider special cases, including large values, swapped values, etc.

We are also interested in the time and space (computer memory) it uses, which we call **time complexity** and **space complexity**. Typically, complexity is a function of the values of the inputs, and we would like to know which function. We can also consider the best case, average case, and worst-case scenarios.

For example, in the above algorithm, the best case would be $m = n$, with just one iteration. If $n = 1$, there are m iterations, which is the worst case. However, for the average case, it is difficult to analyze.

Also, for space complexity, it is constant since we only use space for the three integers: m, n, and t.

To improve the above algorithm, we can use `mod`, so we don't need to keep doing subtraction.

1.2 Algorithm

An algorithm is a finite set of instructions which, if followed, accomplishes a particular task. Every algorithm must satisfy the following criteria:

- Input: There are zero or more quantities that are externally supplied.

- Output: At least one quantity is produced.

- Definiteness: Each instruction must be clear and unambiguous.

- Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm will terminate after a finite number of steps.

- Effectiveness: Every instruction must be sufficiently basic that it can, in principle, be carried out by a person using only pencil and paper.

We also define an algorithm as any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. It is thus a sequence of computational steps that transform the input into output.

It can also be viewed as a tool for solving a well-specified computational problem. The problem statement specifies, in general terms, the desired input or output relationship, and the algorithm describes a specific computational procedure for achieving that input or output relationship.

An algorithm is said to be correct if, for every input instance, it halts with the correct output. It can solve the given computational problem. In contrast, an incorrect algorithm might not halt at all on some input instances, and sometimes it can even produce useful results.

1.3 Study of Data

A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

A data structure is a set of domains D, a designated domain $d \in D$, a set of functions F, and a set of axioms A.

An implementation of a data structure d is a mapping from d to a set of other data structures e.

Chapter 2

Analysis

2.1 Complexity

Before, we talked about the definition of an algorithm. In this part, we would like to know how we can estimate the time required for a program, how to reduce the running time of a program, what the storage complexity is, and how to deal with trade-offs.

We can analyze the runtime by comparing functions. For example, given two functions $f(N)$ and $g(N)$, we can compare their relative rates of growth. There are three types of comparisons that we can make: $f(n) = \Theta(g(n))$ represents the exact bound, $f(n) = O(g(n))$ represents the upper bound, and $f(n) = \Omega(g(n))$ represents the lower bound.

By using bounds, we can establish a relative order among functions. Here, we often use $O(n)$ to analyze time complexity.

For the definition of the upper bound, it says that there is some point n_0 past which $cf(N)$ is always at least as large as $T(N)$. Then we say that $T(N) = O(f(N))$, where $f(N)$ is the upper bound on $T(N)$.

Definition 2.1.1. We say that $f(n) = O(g(n))$ iff there exists a constant $c > 0$ and an $n_0 \geq 0$ such that

$$f(n) \leq cg(n) \quad \text{for all } n \geq n_0$$

Or we can use the following notation

$$\exists c > 0, n_0 \geq 0 \text{ such that } f(n) \leq cg(n) \forall n \geq n_0$$

There are some rules to follow:

- Transitivity
$$\text{If } f(n) = O(g(n)) \text{ and } g(n) = O(h(n)), \text{ then } f(n) = O(h(n))$$
- Rule of sums
$$f(n) + g(n) = O(max\{f(n), g(n)\})$$
- Rule of products
$$\text{If } f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)), \text{ then } f_1(n)f_2(n) = O(g_1(n)g_2(n))$$

We do not include constants or lower-order terms inside Big-O notation.

If $f(n)$ is a polynomial in n with degree r, then $f(n) = O(n^r)$, but for $s < r$, $f(n) \neq O(n^s)$.

Also, any logarithm of n grows more slowly than any positive power of n as it increases. Hence, $\log n$ is $O(n^k)$ for any $k > 0$, but n^k is never $O(\log n)$ for any $k > 0$.

Order	Time
$O(1)$	constant time
$O(n)$	linear time
$O(n^2)$	quadratic time
$O(n^3)$	cubic time
$O(2^n)$	exponential time
$O(\log n)$	logarithmic time
$O(\log^2 n)$	log-squared time

On a list of length n, sequential search has a running time of $O(n)$.

On an ordered list of length n, binary search has a running time of $O(\log n)$.

The sum of the sums of integer indices of a loop from 1 to n is $O(n^2)$.

In summary, Big-O notation provides an upper bound of the complexity in the **worst-case**, helping to quantify performance as the input size becomes arbitrarily large. However, it doesn't measure the actual time, but represents the number of operations an algorithm will execute.

2.2 Recurrence Relations

Recurrence relations are useful in certain counting problems, for example, recursive algorithms. They relate the n-th element of a sequence to its predecessors.

By definition, a recurrence relation for the sequence $a_0, a_1, \cdots$ is an equation that relates a_n to certain of its predecessors $a_0, a_1, \cdots, a_{n-1}$. Initial conditions for the sequence are explicitly given values for a finite number of the terms of the sequence.

To solve a recurrence relation, we can use iteration. We use the recurrence relation to write the n-th term a_n in terms of certain of its predecessors. We then successively use the recurrence relation to replace each of $a_{n-1}, \cdots$ by certain of their predecessors. We continue until an explicit formula is obtained.

For example, the Fibonacci sequence is also defined by the recurrence relation.

Example (Tower of Hanoi). Find an explicit formula for a_n, the minimum number of moves in which the n-disk Tower of Hanoi puzzle can be solved.

Given $a_n = 2a_{n-1} + 1$, $a_1 = 1$, by applying the iterative method, we obtain:

$$\begin{aligned} a_n &= 2a_{n-1} + 1 \\ &= 2(2a_{n-2} + 1) + 1 \\ &= 2^2 a_{n-2} + 2 + 1 \\ &= 2^2(2a_{n-3} + 1) + 2 + 1 \\ &= 2^3 a_{n-3} + 2^2 + 2 + 1 \\ &= \cdots \\ &= 2^{n-1} a_1 + 2^{n-2} + 2^{n-3} + \cdots + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2 + 1 \\ &= 2^n - 1 \end{aligned}$$

Chapter 3

ADT, List, Stack and Queue

3.1 Abstract Data Type (ADT)

We use data abstraction to simplify software development since it facilitates the decomposition of the complex task of developing a software system. To put it simply, data abstraction shows only the essential details of data, while the implementation details are hidden.

For example, as will be discussed later, List, Stack, and Queue (LSQ) are forms of data abstraction, or what we call abstract data types. We can use them to retrieve or store data, but we don't know how they are actually stored or indexed.

Data encapsulation, or information hiding, is the concealing of the implementation of a data object from the outside world. Through data abstraction, we can separate the specification of a data object from its implementation.

A data type is a collection of objects and a set of operations that act on those objects. An abstract data type (ADT) is a data type organized in such a way that we can separate the specification of the object and the specification of the operations on the object. Abstract data types are simply a set of operations, and they are mathematical abstractions.

Note that abstraction is like a functional description without knowing how to use it, while implementation, on the contrary, is something that can be used and executed.

In summary, an ADT is a high-level description of how data is organized and the operations that can be performed on it. It abstracts the details of its implementation and only exposes the operations that are allowed on data structures.

3.2 List

The first abstract data type in this chapter is List.

3.2.1 Definition

When dealing with a general list of the form $a_1, a_2, \cdots, a_n$, we say that the size of this list is n. If the list is of size 0, we call it the **null list**. Except null list, we say that a_{i+1} follows/succeeds $a_i (i < n)$ and that a_{i-1} precedes $a_i (i > 1)$.

The first element of the list is a_1, and the last element is a_n. The predecessor of a_1 and the successor of a_n is not defined.

3.2.2 Operations

A list of elements of type T is a finite sequence of elements of T together with the following operations:

- Create the list and make it empty.
- Determine whether the list is empty or not.
- Determine whether the list is full or not.
- Find the size of the list.
- Retrieve any entry from the list, provided that the list is not empty.
- Store a new entry, replacing the entry at any position in the list, provided that the list is not empty.
- Insert a new entry into the list at any position, provided that the list is not full.
- Delete any entry from the list, provided that the list is not empty.
- Clear the list to make it empty.

With these operations, we can perform various tasks on the list ADT.

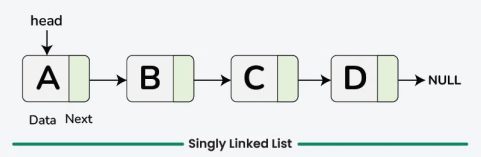
3.2.3 Implementation

We can use an array to implement a list. Most of the operations follow linear time, for example, `print_list`, `make_null`, `find`, and `find_kth`. For insertion, there could be cases where the list is full, and that's why we need dynamically allocated space. For deletion, we need to find the element, perform the deletion, and reallocate space, which might require more time. Thus, we introduce the linked list.

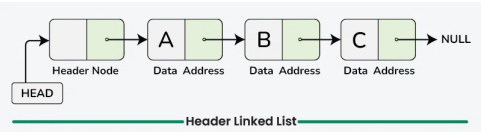
3.2.4 Linked List

There are several types of linked lists:

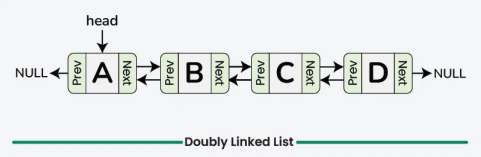
- Singly Linked List



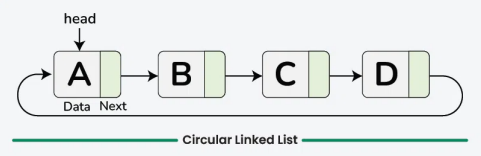
- Singly Linked List with Header



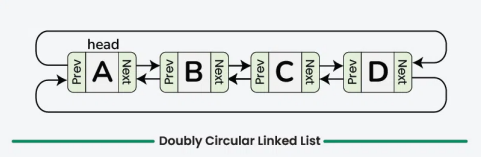
- Doubly Linked List



- Circularly Linked List



- Circularly Doubly Linked List



A polynomial can be represented as

$$F(X) = \sum_{i=0}^{N} A_i X^i$$

For example, $F(X) = 4X^3+2X^2+5X+1$. We may want to perform operations like addition, subtraction, multiplication, and differentiation. Using an array data structure, the time complexity may be larger due to the need to store all terms, including zero coefficients. However, with a linked list, we can efficiently perform these operations by traversing the linked list and processing only the non-zero terms.

Also, note that a circular list saves space but not time. It is useful for smaller datasets. However, for a larger number of students and courses, the use of such a circular list might be a waste of space.

In summary, a list abstract data type represents an ordered collection of elements. They can be added or removed at any position in the list. It provides methods to access elements by their position.

By using different types of linked lists, we can achieve various goals. For example, we can print all the elements in reverse using a doubly linked list.

3.3 Stack

3.3.1 Definition

A stack is an ordered list in which all insertions and deletions are made at one end, called the top. It follows the Last In, First Out (LIFO) rule.

3.3.2 Operations

A stack of elements of type T is a finite sequence of elements of T along with the following operations:

- Create the stack.
- Determine if the stack is empty or not.
- Determine if the stack is full or not.
- Determine the number of entries in the stack.
- Insert (Push) a new entry at one end of the stack, called its top, if the stack is not full.
- Retrieve the entry at the top of the stack, if the stack is not empty.
- Delete (Pop) the entry at the top of the stack, if the stack is not empty.
- Clear the stack to make it empty.

3.3.3 Implementation

We can use a doubly linked list to implement the stack, where both Push and Pop operations happen at the front of the list. We can use the Top operation to examine the element at the front of the list. In this case, the space complexity would be $O(3n)$, and the time complexity would be $O(c)$, where c is a constant.

Alternatively, we can use an array to implement the stack, since we only perform insertion and deletion at the top. However, we need to declare the size ahead of time and use TopOfStack as the counter to point to the top of the stack. If an array is used, the space complexity would be $O(n)$, and the time complexity would be $O(1)$.

3.3.4 Application

Balance Symbols

We can use a stack to balance symbols, which is commonly used in compilers to check for syntax errors. While compiling, an empty stack is created, and an opening symbol is pushed onto the stack. Then, when a closing symbol is encountered, it is popped from the stack. There could be four types of errors:

1. Stack Overflow: Too many brackets.
2. Mismatched Symbols: The opening and closing symbols don't match.
3. Empty Stack: Attempting to pop from an empty stack.
4. Non-Empty Stack: The stack isn't empty at the end of the process.

Reverse Polish Calculator

We can also use a stack to make a Reverse Polish Calculator. There are three forms of notation: prefix, postfix, and infix. For example, if the expression is $a \times b$, then:

1. Prefix: $\times ab$
2. Postfix: $ab\times$
3. Infix: $a \times b$

In Reverse Polish notation (postfix), parentheses are not needed. Using a stack, we can calculate the answer by evaluating the postfix expression. For example, when the character is a number, it is pushed onto the stack. If the character is an operator, two elements are popped from the stack, and the operation is performed on those two elements.

In summary, the stack abstract data type is a collection of elements with two main operations: push and pop. All operations happen at the top, and it follows the Last In, First Out (LIFO) approach.

3.4 Queue

3.4.1 Definition

A Queue is an ordered list in which all insertions take place at one end, the rear, while all deletions take place at the other end, the front. It follows the First In, First Out (FIFO) rule.

3.4.2 Operations

Several operations can be performed on a queue:

- Create the queue
- Determine if the queue is empty or not.
- Insert (Enqueue) a new entry at one end of the queue, called its rear, if the queue is not full.
- Delete (Dequeue) an entry at the other end of the queue, called its front, if the queue is not empty.
- Retrieve the entry at the front of the queue, if the queue is not empty.
- Clear the queue and make it empty.

3.4.3 Implementation

We can use both linear and circular arrays to implement a queue. For a linear array, we can have two indices that always increase. However, this might lead to overflow. Additionally, the array may need to be shifted forward or backward after each enqueue or dequeue operation. For a circular array, we have the following possibilities:

- Front and rear indices, with one position left vacant.

- Front and rear indices, with a Boolean variable indicating fullness or emptiness.

- Front and rear indices, with an integer variable counting entries.

- Front and rear indices taking special values to indicate emptiness.

3.4.4 Application

Queues are commonly used in various applications, such as in printer queues, airline control systems, and bank queues.

In summary, the physical model of a queue is a linear array, with the front always in the first position. All entries are moved up the array whenever the front is deleted. It follows the First In, First Out (FIFO) rule.