CST2550 Software Engineering
Management and Development
Student ID: M00696513

REPORT OF LIBRARY SOFTWARE PROJECT

INTRODUCTION

Brief description

This project consists of designing and developing a software simulation intent on organising, searching, and modifying books data in a library. In the following report, the type of design conceived and implemented for the realisation of this project will be deepened by illustrating various details such as the choice criterion and analysis of the data structure and algorithms employed, going through the type of approach undertaken for testing furthermore summarised as a table, and finally, a summary conclusion followed by references in Harvard format.

Report layout/index

The report layout, as required, includes 5 sections which are: Introduction, Design, Testing, Conclusion and References, and they are displayed as per followed index.

Selection of data structure and algorithms

Data structure Selection and Example:

The data structure was designed to improve searching performances but to the detriment of its overall time and space complexity. Indeed, memory and time cost are weaknesses of such a data structure due to more complex "insert", "remove", and "access" methods, shown in the time-complexity table here followed.

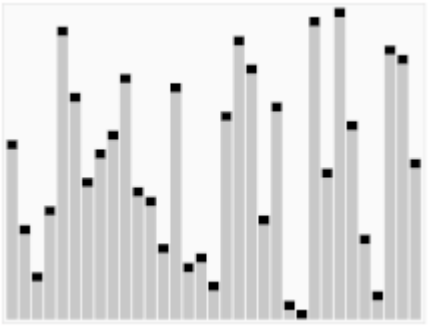
As mentioned above, its advantage lies in the search. It provides different layers of sorted data to allow faster research of multiple books just by entering any title fraction of the target. Its time-complexity worst case is quadratic for ***insertion***, polynomial for ***removing*** and polynomial/linearithmic for the ***random access***. Its space complexity is O(n*t) in the worst-case scenario and Ω(2n) in the best one.

Default Data Container *(deque)*
DDC = { e2, c2, a3, d1, b2, a2, c1, e1, a1, b1, a4, e3 }

(SDS size depends on the words amount of the longest book title in DDC)
Sorted Data Structure *(structure made of deques)*
SDS = { { a1, a2, a3, a4, b1, b2, c1, c2, d1, e1, e2, e3 }
{ a2, a3, a4, b2, c2, e2, e3 }
{ a3, a4, e3 }
{ a4 }
}

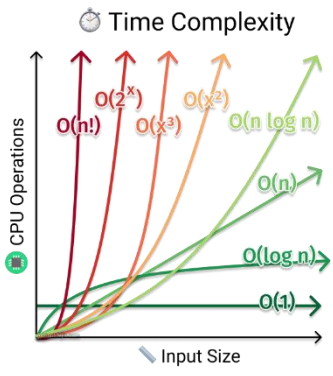
		DDC	SDS
Insert	Front	O(1)	$O(n^2)$
	Middle	O(n)	$O(n^2)$
	End	O(1)	$O(n^2)$
Remove	Front	O(1)	O(n*t)
	Middle	O(n)	O(n*t)
	End	O(1)	O(n*t)
Lookup	Front	O(1)	O(t log n)
	Middle	O(1)	O(t log n)
	End	O(1)	O(t log n)

Sorting Algorithm Selection:

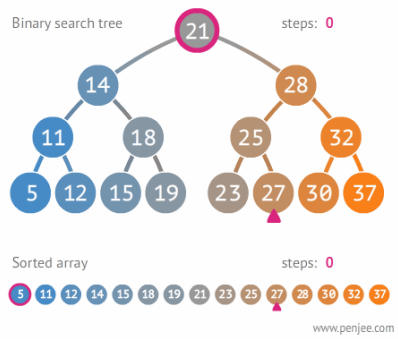


Quick-Sort is the sorting algorithm chosen and implemented in this project since, in order to experience faster research, one of the most powerful algorithms happens to be the Binary-Search, which is a searching algorithm that requires an ordered list of elements in order to work. Hence, since the data in question is sortable by its alphabetical relation, binary search appears to be the most suitable solution, and therefore a subsequential needs for a sorting algorithm, perhaps based on *divide and conquer* as well. Quick-sort *"middle pivot"* selection is due to its efficiency in terms of time-complexity since its best & average performance is linearithmic, Θ(n log n).

Searching Algorithm Selection:



As a searching algorithm, Binary-Search (divide and conquer) was my choice according to its performances in terms of time-complexity, which in the worst-case behaves logarithmically, O(log n). Its choice was suitable due to the planned implementation of a data structure containing sorted data alphabetically related to each other as a binary tree.



Eventually, such an algorithm was adapted to obtain multiple results. Hence the final time-complexity of the worst case is now O(n).

Algorithm: QUICKSORT Pseudocode

Code	Cost	Times
`Recursion(left, right)`		
`i ← left`	C1	1
`j ← right`	C2	1
`newTitle ← (left + right) / 2`	C3	1
`pivot ← A[newTitle]`	C4	1
`while i <= j do`	C5	n+1
`while A[i] < pivot do`	C10	[0, n/2+1]n
`i++`	C11	[0, n/2]n
`end while`		
`while A[j] > pivot do`	C12	[0, n/2+1]n
`j--`	C13	[0, n/2]n
`end while`		
`if i <= j then`	C14	[1, n]
`tmp ← A[i]`	C15	[0, n]
`A[i] ← A[j]`	C16	[0, n]
`A[j] ← tmp`	C17	[0, n]
`i++`	C18	[0, n]
`j--`	C19	[0, n]
`end if`		
`end while`		
`if left < j then`	C19	1
`Recursion(left, j)`	C20	T(n/2)
`end if`		
`if i < right then`	C21	1
`Recursion(i, right)`	C22	T(n/2)
`end if`		
`end of recursion`		
`Recursion(0, A.length-1)`		
end of Quicksort method		.

(Master method applied due to the recursion solution instead of an iterative one)

Worst case: O(n^2)

T(n) = *aT(n/b) + f(n/d)* = 2T(n/2) + n^2/2 ***(d = 2)***

= $log_b\, a$ = $\log_2 2$ = 1 ∴ d > 1 ∴ O(n^2)

Best case: Ω(n log n)

T(n) = *aT(n/b) + f(n/d)* = 2T(n/2) + O(n) ***(d = 1)***

= $log_b\, a$ = $\log_2 2$ = 1 ∴ d == 1 ∴ Ω(n^d log n) = Ω(n log n)

Algorithm: BINARYSEARCH Pseudocode

Code	Cost	Times
`l ← 0`		
`r ← n-1`		
`Recursion()`	T(n)	
`    mid = l + (r - l) / 2`	C1	1
`        if r >= l then`	C2	1
`            iterTitle ← A[mid]`	C3	[0, 1]
`            if iterTitle == search then`	C4	[0, 1]
`                increaseMid ← mid`	C5	[0, 1]
`                decreaseMid ← mid`	C6	[0, 1]
`                while right >= ++increaseMid do`	C7	[0, n/2+1]
`                    if A[increaseMid] <> search then`	C8	[0, n/2]
`                        break`	C9	[0, 1]
`                    end if`		
`                end while`		
`                while left <= --decreaseMid do`	C10	[0, n/2+1]
`                    if A[decreaseMid] <> search then`	C11	[0, n/2]
`                        break`	C12	[0, 1]
`                    end if`		
`                end while`		
`                decreaseMid++`	C13	[0, 1]
`                for i in A[decreaseMid] is <> A[increaseMid] do`	C14	[0, n]
`                    found[i.getId()] ← i`	C15	[0, n]
`                end for`		
`                return true`	C16	[0, 1]
`            end if`		
`            if iterTitle > search then`	C17	[0, 1]
`                r ← mid -1`	C18	[0, 1]
`                return Recursion()`	C19	T(n/2)
`            end if`		
`            l ← mid +1`	C20	[0, 1]
`            return Recursion()`	C21	T(n/2)
`        end if`		
`    return false`	C22	[0, 1]
`end of recursion`		
`Recursion()`		
`end of BinarySearch method`		.

(Master method applied due to the recursion solution instead of an iterative one)

Worst case: O(n)

T(n) = $aT(n/b) + f(n/d)$ = T(n/2) + n ***(d = 1)***

= $log_b\, a$ = $\log_2 1 = 0$ ∴ d > 0 ∴ O(n)

Best case: Ω(1)

T(n) = Ω(1)

Data structure: ADDBOOK Pseudocode *(insert method)*

Code	Cost	Times
addBook(book)		
DDC.push_back(book)	C1	1
shuffle(DDC)	C2	n
quicksort(DDC)	C3	n log n
titleSize ← book.length	C4	1
for i in 0 to titleSize-1 do	C5	t+1
if i >= SDS.length then	C6	[1, t]
if i == 0 then	C7	[0, t]
SDS.push_back(DDC)	C8	[0, t]
else do		
SDS.push_back({})	C9	[0, t]
for j in 0 to DDC.length-1 do	C10	[0, n+1]t
iterTsize ← DDC[j].length	C11	[0, n]t
if iterTsize > i then	C12	[0, n]t
SDS[i].push_back(DDC[j])	C13	[0, n]t
end if		
end of for		
quicksort(SDS[i], 0, SDS[i].length-1, i)	C14	[0, n^2]t
end of if and else		
else do		
SDS[i].insert(bookSearch(SDS[i], book, i)[1], book)	C15	[0,log(n)+n]t
end of if and else		
end of for		
end of addBook (insertion method)		.

(Master method does not apply to the current solution since it is iterative instead of recursive)

let C' = (C1 + C4) 1
let C'' = (($\sum_{i=7}^{9} Ci$)) [0, t]
let C''' = (($\sum_{i=11}^{13} Ci$) + C16) [0, n]t

Worst case: O(n^2)

T(n) = C' + C"t + C'''(n*t) + C2n + C3(n log n) + C5(t+1) + C10(n+1) + C14(n^2*t) + C15(n log n)
= C' + C"t + C'''(n*t) + C2n + C3n log n + C5t + C5 + C10n + C10 + C14n^2*t + C15n log n
=(C` + C5 + C10) + (C`` + C5)t + C```nt +(C2 + C10)n + (C3 + C15) n log n + C14n^2
= O(n^2)

Best case: Ω(n log n)

T(n) = C' + C2(n) + C3(n log n) + C5(t+1) + C6t
= C' + C2n + C3n log n + C5t + C5 + C6t
= (C' + C5)1 + (C6 + C5)t + C2(n) + C3(n log n)
= Ω(n log n)

Data structure: REMOVEBOOK Pseudocode *(remove method)*

Code	Cost	Times
removeBook(indexes)		
removed ← true	C1	1
for i in 0 to indexes.length-1 do	C2	t+1
if indexes[i] <> -1 then	C3	[1, t]
if i == 0 then	C4	[0, t]
if indexes[i] < DDC.length && indexes[i] >= 0 then	C5	[0, t]
b ← DDC[indexes[i]]	C6	[0, t]
DDC.erase(indexes[i])	C7	[0, t]
else do removed ← false	C8	[0, t]
end of if and else		
else		
if i-1 <= SDS.length then	C9	[0, t]
if indexes[i]<SDS[i-1].length&&indexes[i]>=0 then	C10	[0, t]
SDS[i-1].erase(SDS[indexes[i])	C11	[0, n]t
else do removed ← false	C12	[0, t]
end of if and else		
else do removed ← false	C13	[0, t]
end of if and else		
end of if and else		
else do removed ← false	C14	[0, t]
end of if and else		
end of for		
return removed	C15	1
end of bookRemove (remove method)		.

(Master method does not apply to the current solution since it is iterative instead of recursive)

let C' = ($(\sum_{i=4}^{10} Ci)$ + $(\sum_{i=12}^{14} Ci)$) [0, t]
let C'' = (C1 + C15) 1

Worst case: O(n*t)

T(n) = C't + C3t + C''1 + C2(t+1) + C11(n*t)
= C't + C''1 + C3t + C2t + C11n*t
= C''1 + (C' + C3 + C2)t + C11n*t
= O(n*t)

Best case: Ω(1)

T(n) = C'0 + C''1 = Ω(1)

TESTING

Testing and approach (Unit testing and RAD)

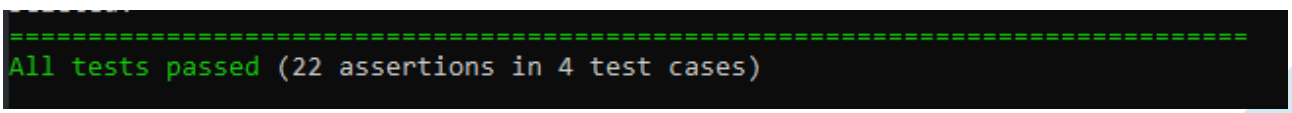
The approach undertaken is the *"functional test"* which consists of testing the application against the expected requirements. *Unit testing* is the first level of testing and is the method of my choice for developing test cases. It is the process of ensuring that the individual components of a piece of software programmatically are functional and perform as they were designed. Unit testing also simplified debugging, as finding problems earlier means smaller debugging time than if they were discovered later in the testing process.

The approach adopted for software development is *"RAD"* (Rapid Application Development). The reason why this is the approach of my choice lies in its characteristics indicated in the current context, that is, where the agreed deadline is immovable and therefore superfluous functionality are expendable.

In fact, one of the characteristics most similar to the type of context in question is the prioritization of the requirements only, therefore making use of the *"MoSCow"* concept, which consists in prioritizing the requirements by neglecting non-essential functionalities even if important, and then possibly implementing them if time permits.

Table of test cases

CLASS	STATUS	MACRO	DESCRIPTION		
			INPUT	EXPECTED OUT	OBTAINED OUT
Global *This section is where Global class methods get tested*	PASSED	CHECK(toLower());	“TeSt”	“test”	“test”
	PASSED	CHECK_FALSE(split());	“hello world”	vec.size() == 2	vec.size() == 2
	PASSED	CHECK_FALSE(vec[0]);	“hello” == “”	false	false
	PASSED	CHECK_FALSE(vec[1]);	“world” == “”	false	false
	PASSED	CHECK_FALSE(getCstate());	setColor(false);	false	false
	PASSED	CHECK(colorReset());	setColor(false);	“”	“”
	PASSED	CHECK(color());	“green”	“”	“”
	PASSED	CHECK(getCstate());	setColor(true);	true	true
	PASSED	CHECK(colorReset());	setColor(true);	!= “”	!= “”
	PASSED	CHECK(color());	“green”	!= “”	!= “”
	PASSED	CHECK(sToll());	“abc”	0	0
	PASSED	CHECK(sToll());	“0”	0	0
	PASSED	CHECK(sToll());	“123”	123	123
Operations *class methods testing*	PASSED	REQUIRE(is_class<Operations>);	::value	true	true
	PASSED	CHECK(reader());	“0”	0	0
	PASSED	CHECK_FALSE(reader());	“1”	!= 0	!= 0
Collection *class methods testing*	PASSED	CHECK(getBook(0).getId());	book1	"12345"	"12345"
	PASSED	CHECK(getBook(1).getId());	Book2	“54321”	“54321”
	PASSED	CHECK(removeBook());	BI0	true	true
	PASSED	CHECK(removeBook());	BI1	true	true
Books *testing*	PASSED	REQUIRE(is_class<Books>);	::value	true	true
	PASSED	CHECK_FALSE(emptyCheck());	b (a book obj)	false	false
4 test cases		**22 assertions**			



CONCLUSION AND REFERENCES

Summary reflection

- In conclusion, the completed project essentially consists of the development of mainly four sections. Each included in the same final report, which happens to be also one section itself.
- Another section is software programming, written in C++ language with the support of *Git Version Control* and relative repository on *Bitbucket*.
- The following section is *"unit testing"* (performed employing the *"Catch.hpp"* library for C++) included in the same software directory and both compiled by the same *"makefile"*.
- The last section consists of a report assertion of the choice justification about algorithms and data structures implemented in the software and their respective time complexity analysis.

The work performed was carried out by undertaking a *RAD (Rapid Application Development)* approach, with the main reason being the predetermined delivery deadline, which consequently limited the development flexibility in terms of time and, therefore, made the choice of this approach indicated to the case in question. Furthermore, were not for the context, a preferable approach could have also been the *"Agile"* one, as it is one of the most requested and used approaches in the professional environment.

Indeed, in a possible next project, the software development approach's choice is likely to fall into the *"Agile"* mode, to learn all the advantages and obtain the ability to approach programming in one of the most requested ways in the professional framework, and as a sorting algorithm, most probably the next chosen ones would be *Radix LSD* when values to be sorted are integers; otherwise, a variant of *Quicksort (dual pivot)*.

References

Data structure: (Page 2)
"Data Structures - GeeksforGeeks." 20 gen. 2020, https://www.geeksforgeeks.org/data-structures/.
Access: 7 apr. 2021.

Quicksort: (Page 2) "Quicksort - Wikipedia." https://en.wikipedia.org/wiki/Quicksort.
Access: 8 apr. 2021.

Binary search: (Page 2) "Binary Search and its analysis - CodesDope."
https://www.codesdope.com/course/algorithms-binary-search/.
Access: 9 apr. 2021.

Algorithms: (Page 3-4) "What is the Master Theorem? - YouTube."
https://www.youtube.com/watch?v=2H0GKdrIowU.
Access: 10 apr. 2021.

Algorithms: (Page 4) "Analysis of quicksort (article) | Quick sort ...."
https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort.
Access: 10 apr. 2021.

Testing approaches: (Page 7) "Software Testing Methodologies - SmartBear."
https://smartbear.com/learn/automated-testing/software-testing-methodologies/.
Access: 11 apr. 2021.