LO27 Report
Parsing a bibtex file in a library and managing it

Maxime Bourgeois and Nathan Olff

January 2013

Contents

Part I

Introduction

0.1 Description of the subject

A bibtex file is text file representing a bibliography. It is linked with LaTeX, a powerfull document markup language (like HTML or XML) for scientific articles.
The report you are reading now has been made in LaTeX.

Each article/book/journal... present in a Bibtex file correspond to an entry.
The subject of our project was to parse a bibtex file into a doubly linked list of entries. We had to manage every types of entry available in the language with their different types of entry fields.

After parsing the file into a library, we had to sort this library by authors and date using special data structures and the quicksort.

The next step was to export the library (sorted or not) in text files. We prefered to export it in HTML files in order to make it more readable.

The final step was to create a user-friendly interface, enabling the user to test each of the principle functions we had made before.

0.2 Describe Objectives & Problem statements.

The objectives of this project were :

- using doubly linked list
- seeing how to open / read / write / close a file
- learning the use of regular expressions
- discovering the quicksort method of sorting an array and adapting it to doubly linked list

This project is also the first project we have made in computer science. It was also an opportunity for us to apply our knowledge on a bigger project than what we were used to.

We had to create an efficient and user-friendly way of parsing a bibtex file, sorting the library obtained and exporting it in order to be read by anyone.

Part II

Project

Chapter 1

Generic linked list

1.1 How and why ?

We created generic doubly linked list by using void pointers (written *void**) which can point on any type of data (e.g : a string, a list, an integer ...).
We use the void* for two structures : *AbstractElement* (which is an element of an *AbstractList*) and *EntryField*.
But why do we use generic linked list and not the usual linked list we had already seen ?

We wanted to have a single type of list/entryField.
The problem is that an *EntryField* may contains an integer, a string or even an *Author* (a structure which contains two strings : a *firstName* and a *lastName*).
Alike an *AbstractElement* has a value which can be an integer, an *Entry*, an *EntryField*, an *Author*...

By using void pointers, we were able to simplify our structures and we have a single type of list. Therefore we made all managing functions (like *isEmpty*, *insertTail*, *removeTail*...) only once

1.2 Check if a list is empty

Lexicon

list : a generic list

Algorithm

Data: list
Result: *TRUE* or *FALSE*

function isEmpty(<AbstractList> list):Boolean
Begin
  If *head(list) = NULL* **then**
    isEmpty ← *TRUE*
  Else
    isEmpty ← *FALSE*
  EndIf
End

1.3 Initialize a list

Lexicon

sizeOfAnElement : size of one element of the list
list : a generic list

Algorithm

Data: sizeOfAnElement
Result: list

function initList(<Integer> sizeOfAnElement) : <AbstractList>
Begin
    head(list) ← *NULL*
    tail(list) ← *NULL*
    count(list) ← 0
    size(list) ← sizeOfAnElement
    initList ← list
End

1.4 Insert an element at the tail of a list

Lexicon

list : a generic list
element : the given element we want to put in the end of the list
allocMemory : a boolean which indicate if we need to allocate the memory for the element (*TRUE*) or just copy the pointer (*FALSE*)
Only needed for an optimisation in the C language.
newEl : an abstractElement

Algorithm

Data: list, element, allocMemory
Result: list (with the new element)

function insertTail(<List> list, <ListElem> element, <Boolean> allocMemory):<List>
Begin
    `/* In the C language, we use a memcpy here if we need to allocate memory (allocMemory = TRUE ) instead of value(newEl) ← element */`
    value(newEl) ← element
    next(newEl) ← *NULL*
    count(list) ← count(list) + 1
    If *isEmpty(list) = TRUE* **then**
        prev(newEl) ← *NULL*
        head(list) ← newEl
        tail(list) ← newEl
    Else
        succ(tail(list)) ← newEl
        prev(newEl) ← tail(list)
        tail(list) ← newEl
    EndIf
    insertTail ← list
End

1.5 Remove the last element of the list

Lexicon

list : a generic list

Algorithm

Data: list with N elements
Result: list with N-1 elements

```
function removeTail(<List> list):<List>
Begin
    If isEmpty(list) = FALSE then
        If prev(tail(list)) = NULL then
            head(list) ← NULL
            tail(list) ← NULL
        Else
            tail(list) ← prev(tail(list))
            succ(tail(list)) ← NULL
        EndIf
    EndIf
    removeHead ← list
End
```

Chapter 2

Initialize the library

2.1 Parse the Bibtex file

Lexicon

fileName : name of the Bibtex file
requiredFieldArray<Entry type> : array of the field required for a specified entry type
requiredFieldArray : array of the rield required for the entry that is currently processed
line : string, temporary variable
nMatch : integer, correspond to the number of matches in the regex function
tmpEntry : Entry, temporary variable
sizeOfRequiredArray : integer, number of fields required for the current entry
boolRequiredArray : array of *sizeOfRequiredArray* booleans. Each element corresponds to a required field. boolRequiredArray[i] is *TRUE* if requiredFieldArray[i] is already in the Entry, *FALSE* otherwise.
sizeOfLine : number of characters in *line*, temporary variable

Algorithm

```
Data: fileName
Result: list

function parseBibtexFile(<string> fileName) : list
Begin
    requiredFieldArrayArticle[] = {"author", "title", "journal", "year", ""}
    requiredFieldArrayBook[] = {"author/editor", "title", "publisher", "year", ""}
    ...                                  /* Declaration of every arrays for each types of entries */
    library ← initList(sizeof(Entry)) file = openFile(fileName)
    If file == NULL then
        print("The file " + fileName + " doesn't exist !")
        parseBibtexFile ← library
    EndIf
    sizeMaxOfALine ← longestLine(fileName)
    While getString(line, sizeMaxOfALine, file) ≠ NULL do
        nMatch ← 0
        While line ≠ NULL AND line[0] ≠ '@' do   /* Read the next line untill it reaches the end
        of the file or the beginning of an entry */

            line ← getString(sizeMaxOfALine, file)
                /* Read the next line from file, in the limit of sizeMaxOfALine characters */
        Done
        nMatch = testRegex(line, "@([a-zA-Z]+)\\{(.+),", arrayRegex, sizeMaxOfALine)
        key(tmpEntry) ← arrayRegex[0]
        type(tmpEntry) ← arrayRegex[1]
        If type(tmpEntry) = "Article" then requiredFieldArray = requiredFieldArrayArticle
        Else if type(tmpEntry) = "Book" then requiredFieldArray = requiredFieldArrayBook
        ...                                                  /* Test for every types of entry */
        Else
            requiredFieldArray = requiredFieldArrayMisc
        EndIf
        sizeOfRequiredArray ← 0
        While requiredFieldArray[sizeofRequiredArray] ≠ "" do sizeOfRequiredArray++
        For i from 0 to sizeOfRequiredArray do boolRequiredArray[i] = FALSE
        Do
            line ← getString(sizeMaxOfALine, file)
            If line[0] = '}' OR line[0] = '\0' OR line[0] = '\n' then nMatch ← 0 Else
                sizeOfLine ← length(line)                    /* calculate the length of the string */
                While line[sizeOfLine-2] ≠ ',' do
                    line[sizeOfLine] ← getString(sizeMaxOfALine+1, file)
                    sizeOfLine ← length(line)
                Done
                nMatch = testRegex(line, "([a-z]+) = [\\{\"](.+)[\\}\"],[\n]$", arrayRegex, sizeOfLine)
                If nMatch = 0 then testRegex(line, "([a-z]+) = (.+),[\n]$", arrayRegex, sizeOfLine)
                If nMatch > 0 then tmpEntry ← addEntryField(arrayRegex[0], arrayRegex[1], tmpEntry,
                requiredFieldArray, boolRequiredArray, sizeOfRequiredArray)
            EndIf
        While nMatch > 0
        If tmpEntry = NULL then
            i ← 0
            While i < sizeOfRequiredArray-1 AND boolRequiredArray[i] = TRUE do i++
            If boolRequiredArray[i] = FALSE then
                Print("Error, the field " + requiredFieldArray[i] + " is missing !")
            Else
                library ← insertTail(library, tmpEntry, TRUE )
            EndIf
        EndIf
    Done
    parseBibtexFile ← library
End
```

2.1.1 Compute the size of the longest line

Lexicon

sizeMaxOfALine : size of the longest line of the file
fileName : path of the Bibtex file
c : character, used to browse the file

Algorithm

```
Data: fileName
Result: sizeMaxOfALine

function longestLine(<String> fileName):<Integer>
Begin
    sizeMaxOfALine 0
    file = openFile(fileName, "r")
    Do
        c ← getChar(file)
        i ← 0
        While c ≠ '\n' AND c ≠ EndOfLine do
            i++
            c ← getChar(file)
        Done
        If i > sizeMaxOfALine then  sizeMaxOfALine ← i
    While kc ≠ EOF
    closeFile(file)
    longestLine ← sizeMaxOfALine
End
```

2.1.2 Regex

testRegex(<String> line, <String> pattern, <ArrayOfStrings> regexArray, <Integer> sizeMax) : Integer

This function apply a regular expression, the *pattern*, to *line*. Each element, which has a maximum size of *sizeMax*+1 bytes, is stored in *regexArray*.
The function returns the number of elements contained in *regexArray*.

2.1.3 Add an Entry Field to an Entry

Lexicon

name : String, name of the entry field
value : String, value of the entry field
entry : Entry in which we add the entry field
requiredFieldArray : list of strings corresponding to the required fields for the type of the entry
sizeOfRequiredArray : integer, number of fields required for the current entry
boolRequiredArray : array of *sizeOfRequiredArray* booleans. Each element corresponds to a required field. boolRequiredArray[i] is *TRUE* if requiredFieldArray[i] is already in the Entry, *FALSE* otherwise.
contained : boolean used to know if the entry field is required or not
i : integer used to browse the list of required fields
tmpEntryField : EntryField to be added to the entry
prevValue : string, temporary variable used to stored the previous string of value
string : string, temporary variable

Algorithm

Data: name, value, entry, requiredFieldArray, boolRequiredArray, sizeOfRequiredArray
Result: entry

function addEntryField(<String> name, <String> value, <Entry> entry, <String> requiredFieldArray[], <Boolean> boolRequiredArray[], <Integer> sizeOfRequiredArray) : Entry
Begin
contained ← *FALSE*
i ← -1
name(tmpEntryField) ← name
While *i< sizeOfRequiredArray-1 AND contained = FALSE* **do**
i++
contained = isContained(name, requiredFieldArray[i]); `/* check if name is present in requiredFieldArray[i] */`
Done
If *name = "Author"* **then**
value(tmpEntryField) ← initList(sizeof(Author))
While *value ≠ NULL* **do**
value ← strstr(value, " and ")
If *value = NULL* **then** string ← prevValue **Else**
strncpy(string, prevValue, size(prevValue)-size(value))
string[size(prevValue)-size(value)] ← '\0'
EndIf
sscanf(string, "%[^,], %[\t \n]", lastName, firstName) `/* divide the string into two substrings */`
lastName ← replaceSpecialChars(lastName) `/* replace special characters like é, ö, î... */`
firstName ← replaceSpecialChars(firstName)
value(tmpEntryField) ← addAuthor(value(tmpEntryField), firstName, lastName)
If *value ≠ NULL* **then**
value ← value+5
EndIf
Done
Else
value ← replaceSpecialChars(value)
value(tmpEntryField) ← value
EndIf
If *contained = TRUE AND boolRequiredArray[i] = FALSE* **then**
boolRequiredArray[i] ← *TRUE*
requiredFieldList(entry) ← insertTail(requiredFieldList(entry), tmpEntryField, *FALSE*)
Else
optionalFieldList(entry) ← insertTail(optionalFieldList(entry), tmpEntryField, *FALSE*)
EndIf
addEntryField ← entry
End

2.1.3.1 Add an Author to an Author List

Lexicon

listAuthors : a list of authors
firstName : string, first name of the new author
lastName : string, last name of the new author
tmpAuthor : author to be added to the list

Algorithm

Data: listAuthors
Result: listAuthors, firstname, lastname

function addAuthor(<List> listAuthors, <String> firstName, <String> lastName):<List>
Begin
  firstName(tmpAuthor) ← firstName
  lastName(tmpAuthor) ← lastName
  listAuthors ← insertTail(listAuthors, tmpAuthor, *FALSE*)
  addAuthor ← listAuthors
End

Chapter 3

Sorting the library

3.1 Quicksort

For sorting the different lists we have, we used the quicksort method whose algorithm were on wikipedia.
This article explains how to use the quicksort on an array of integers. We had to adapt it for sorting doubly linked lists by any type of variable.

For managing the problem of the list, we kept the notion of index and adapted it to the list.
The list's head will be the index 0, the head's successor will be the index 1 until the index (list.count-1) which is the list's tail.
We created for-loops which traverse the list until we reach the element with a given index.
It is the only way we found to adapt the quicksort to the list but it can be improved.

We also used pointers of functions in order to be able to quicksort strings, dates, authors within a single quicksort function.

How does it work ?
When we call the quicksort function, we put in parameter a pointer of a comparison function. All our comparison fonctions work the same way :

- syntax : compareSomething(elem 1, elem2)
- return an integer < 0 if elem1 $<$ elem2
- return 0 if elem1 = elem2
- return an integer > 0 if elem1 $>$ elem2

Elem1 is consider lesser than elem2 if we need to put it before elem2.
This is why we consider that "2012" is lesser than "2010" (because when we sort we put "2012" before "2010").

3.1.1 Partition

Lexicon

i, j : integers, counters for the loops
list : a generic list
left : the "left" border of the partionning
right : the "right" border of the partionning
pivotIndex : the initial index of the pivot
storeIndex : the final index of the pivot
tmpElem, tmpRight : AbstractElement used to browse the list
ptrFCompare : a pointer of function, not used in the algorithm

Algorithm

```
Data: list
Result: storeIndex (the new position of the pivot)

function partition(<List> list, int left, int right, int
pivotIndex, <PointerOfFct> ptrFCompare):Integer
Begin
    tmpElem ← head(list)
    For i from 0 to pivotIndex-1 do
    |   tmpElem ← succ(tmpElem)
    Done
    pivotValue ← value(tmpElem)

    tmpElem ← tail(list)
    For i from count(list)-1 to right+1 by-1 do
    |   tmpElem ← prev(tmpElem)
    Done

    swapValues(tmpElem, tmpRight)
    storeIndex ← left
    tmpElem ← head(list)
    For i from 0 to right-1 do
    |   If i >= left then
    |   |   If ptrFCompare(tmpElem[i], pivotValue) < 0 then
    |   |   |   storeIndexElem ← head(list)
    |   |   |   For j from 0 to storeIndex-1 do
    |   |   |   |   storeIndexElem = succ(storeIndexElem);
    |   |   |   Done
    |   |   |   swapValues(tmpElem, storeIndexElem)
    |   |   |   storeIndex ← storeIndex + 1
    |   |   EndIf
    |   EndIf
    |   tmpElem ← succ(tmpElem)
    Done

    storeIndexElem ← head(list)
    For i from 0 to storeIndex-1 do
    |   storeIndexElem ← succ(storeElemIndex)
    Done

    tmpRight ← tail(list)
    For i from count(list)-1 to right+1 by-1 do
    |   tmpRight ← succ(tmpRight)
    Done

    swapValues(storeIndexElem, tmpRight)
    partition ← storeIndex
End
```

3.1.2 Comparison

3.1.2.1 Compare two years (stored as strings)

Lexicon

year1, year2 : two string representing years

Algorithm

Data: year1, year2
Result: result

function compareYear(int year1, int year2):int
Begin
  result ← (-1) * strcmp(year1, year2)
  compareYear ← result
End

3.1.2.2 Compare entries by their year of publication

Lexicon

entry1, entry2 : two given entries
year1, year2 : two string representing years

Algorithm

Data: entry1, entry2
Result: result

function compareEntryByYear(<Entry> entry1, <Entry> entry2):Integer
Begin
  year1 ← findEntryYear(entry1)
  year2 ← findEntryYear(entry2)

  result ← (-1) * strcmp(year1, year2)
  compareEntryByYear ← result
End

3.1.2.3 Compare authors

Lexicon

author1, author2 : two given authors

Algorithm

Data: author1, author2
Result: result

function compareAuthor(<Author> author1, <Author> author2):Integer
Begin
  If *author1 = NULL AND author2 = NULL* **then**
    result ← 0
  EndIf
  Else if *author1 = NULL* **then**
    result ← (-1)
  EndIf
  Else if *author2 = NULL* **then**
    result ← 1
  EndIf
  Else
    result ← strcmp(stringToUpper(lastName(author1)), stringToUpper(lastName(author2))) **If** *result = 0* **then**
      result ← strcmp(stringToUpper(firstName(author1)), stringToUpper(firstName(author2)))
    EndIf
  EndIf
  compareAuthor ← result
End

3.1.2.4 Compare entries by their authors

It's the same principle that the previous function.

3.2 Get Lists

3.2.1 Get the list of every authors

Lexicon

library : a library (list of entry)
listYear : a list of every year which can be found in the library

tmpAbstrElem : a temporary variable for traversing a copy of the library (listElement)
tmpAbstrEntryField : a temporary entryField (listElement)
tmpEntry : a temporary tmpEntry (Entry)
tmpListYear : a temporary copy of listYear (list)

Algorithm

Data: library
Result: listYear

```
function getListDate(<List> library):<List>
Begin
    tmpAbstrElem ← head(library);

    While tmpAbstrElem ≠ NULL do
        tmpEntry ← value(tmpAbstrElem)
        tmpAbstrEntryField ← head(requiredFieldList(tmpEntry))
        While tmpAbstrEntryField ≠ NULL AND name(value(tmpAbstrEntryField)) ≠ "year" do
            tmpAbstrEntryField ← succ(tmpAbstrEntryField)
        Done

        If tmpAbstrEntryField ≠ NULL then
            tmpListYear ← head(listYear)
            While tmpAbstrEntryField ≠ NULL AND value(value(tmpAbstrEntryField)) ≠
            value(tmpListYear) do
                tmpListYear ← succ(tmpListYear)
            Done
            If tmpListYear = NULL then
                /* allocMemory = 1 because we want to copy the year in listYear (so we allocate
                the memory) */
                listYear ← insertTail(listYear, value(value(tmpAbstrEntryField)), TRUE )
            EndIf
        Else
            tmpAbstrEntryField ← head(optionalFieldList(tmpEntry))
            While tmpAbstrEntryField ≠ NULL AND name(value(tmpAbstrEntryField)) ≠ "year" do
                tmpAbstrEntryField ← succ(tmpAbstrEntryField)
            Done
            If tmpAbstrEntryField ≠ NULL then
                tmpListYear ← head(listYear)
                While tmpAbstrEntryField ≠ NULL AND value(value(tmpAbstrEntryField)) ≠
                value(tmpListYear) do
                    tmpListYear ← succ(tmpListYear)
                Done
                If tmpListYear = NULL then
                    listYear ← insertTail(listYear, value(value(tmpAbstrEntryField)), 1)
                EndIf
            EndIf
        EndIf
        tmpAbstrElem ← succ(tmpAbstrElem)
    Done
    getListDate ← listYear
End
```

3.2.2 Get the list of every years (from the entries)

Refer to the algorithm above. It is almost the same principle, but a little easier because there is only one year per Entry (and not a list as for the authors).

Chapter 4

Export library

4.0.3 exportDatePublications

Lexicon

libraryDateAuthor : the list of all date's publication
fileName : the name of the file that will be created by the export

datePub : a temporary datePublications
html : a pointer on the file opened
tmpElem : a temporary listELem
tmpEntry : a temporary entry

Algorithm

```
Data:
Result:

procedure exportDatePublications(<ListDatePublications>
libraryDateAuthor, <string> fileName)
Begin
    tmpElem ← head(libraryDateAuthor)
    html ← open(fileName)

    fputs("<html>\n<head>\n\t<title>Bibtex
    File</title>\n<link rel=\"stylesheet\" href=\"style.css\"
    />\n</head>\n<body>\n\t<h1>Bibtex
    file</h1><h2>DatePublications</h2>\n", html)

    While tmpElem ≠ NULL do
        datePub ← value(tmpElem)

        fprintf(html, "<h3>%s</h3>\n", year(datePub->year))

        tmpEntry = head(publicationsList(datePub))
        While tmpEntry ≠ NULL do
            exportEntryToHtml(value(tmpEntry), html)
            tmpEntry ← succ(tmpEntry)
        Done
        tmpElem ← succ(tmpElem)
    Done
End
```

4.0.4 exportAuthorsPublications

Refer to the algorithm above. It is almost the same principle, but a little longer.

Part III

Conclusion

We had a few difficulties at the beginning with the *void** because it is something we hadn't really seen before. Because of the absence of templates in the C language, we think it is a good way to avoid writing tone of lines identical (like the declaration of the doubly linked list).

Regular expressions (Regex) have been hard to manage because there is no premade functions in the C language or not efficient enought. Therefore, we created all the functions we needed for our use of the regex.

The quicksort was also a bit difficult because we were not used to recursive functions and we had to adapt it to the doubly linked list structure. We had segmentation faults on our function "partition" who did simply nothing.
We decided to redo the quicksort by using (and testing) portion of code as little functions. The problam was a for-loop with a wrong limit.

With this project we also learnt how to use GIT (with our deposit on bitbucket.org), LaTeX(with the Algorithm2e package), Valgrind (for debugging and searching memory leaks).

Here is some of the possible optimization we have thought of :

- propose a list of all bibtex files in the current folder (so the user would have to write the entire name and juste choose)
- check the user's inputs in the main function
- create a function to add an entry manually
- correct the last memory leaks