Design Specification
JobSeeker
Version 2

Zihao Du
Senni Tan
Gengyun Wang
Wenzhi Wang
Lab 3 Group 3
Computing and Software Department, Mcmaster University
SFWR ENG 2XB3, Software Engineering Practice and Experience:
Binding Theory to Practice

April 10, 2020

Revision Page

By virtue of submitting this document we electronically sign and date that the work being submitted by all the individuals in the group is their exclusive work as a group and we consent to make available the application developed through [CS] or [SE]-2XB3 project, the reports, presentations, and assignments (not including my name and student number) for future teaching purposes.

First revision:

Senni Tan — Edited the title page and created the contribution table.

Zihao Du — Added the attestation and consent in Revision.

Second revision:

Senni Tan — Edited the contribution table.

Zihao Du— Edited the contribution table.

Wang Wenzhi — Edit the contribution table.

Gengyun Wang — Edited the contribution table.

Third revision:

Senni Tan — Modules MIS; Description of implementation; View of uses relationship; Internal review.

Zihao Du— Modules MIS; Description of implementation; Description of Modules; Internal review.

Wang Wenzhi — Modules MIS; Description of implementation; implementation and two UML for two most interesting classes; Internal review.

Gengyun Wang — Modules MIS; Description of implementation and trace back to requirements; Internal review.

Contribution Page

Name	Role(s)	Contribution	Comments
Zihao Du	Designer	Proposal Abstract and motivation	
	Researcher	Database of jobs	
	Designer	SRS Functional requirement	
		Graphing algorithm implementation	
		Client module	
	Tester	Unit test for graphing algorithm	
		Design Specifications(refer to document revisions)	
Senni Tan	Designer	Proposal I/O	
		SRS Non-functional requirement	
		Sorting Algorithm Implementation	
	Tester	Unit test for sorting algorithm implementation	
		Design Specifications(refer to document revisions)	
Gengyun Wang	Designer	Proposal Prior Work	
		SRS Assumptions, Domain	
		Searching Algorithm Implementation	
	Tester	Unit test for searching algorithm implementation	
		Design Specifications(refer to document revisions)	
Wenzhi Wang	Designer	Proposal Reference page	
		SRS Maintenance and Development	
		Data processing implementation	
	Tester	Test and modify the client code implementation	
		Design Specifications(refer to document revisions)	

Executive Summary

JobSeeker is designed for potential immigrants, people new to Canada that are looking for a job and Canadian job seeker. It will provide the user positions of jobs they are interested in and show the relative information. The project is composed by five main modules; a Job module which constructs the Job ADT; a Data Process module which processes data from datasets, converts them to Job objects, stores the objects and return; a Sorting module which sorts the Job objects arrays, with the help from the Comparable module; a Searching module which does the searching on Job objects with some given conditions; and a DFS module which finds the relative jobs for a given job source in the graph constructed by the Graph ADT. The prototype of this design will be presented in the Demo module. This document gives the detail of the project design in the sections below.

Contents

1 Description of Modules

The design is made up with nine modules including the client module. These modules can be divided into four categories: Data Processing, Sorting, Searching and Graphing.

Job class and Dataprocess class belongs to Data Processing, which make use of data from the database and store that into some data structures in Java. Job class defines state variables for an object Job, which is an important and fundamental object for the design. The methods in Job class are all getters. Dataprocess class takes no input and use Job class to store information from dataset into its state variable "joblist".

Sorting catagory contains two classes: Comparable and Sorting. Since we need to sort by different criteria, the Comparable class provides different compareTo methods. The Sorting class inherits these methods and use quicksort algorithm to sort the input ArrayList.

Searching catagory contains only a single module Searching.Just like Sorting class, it provides static functions instead of creating objects. It uses binary search assuming that the ArrayList is already sorted to get the kind of Job the user what and return them in an ArrayList.

The Graphing part is a trival one. It contains two classes: Graph and DFS. Graph creates an undirected graph class while DFS is an object exploring reachable nodes with depth-frist search algorithm based on a Graph class.

The client module part uses outputs of Searching, Sorting and Graphing parts. It also makes use of a class called Noc which provides a list of job catagories for user selection and demostration.

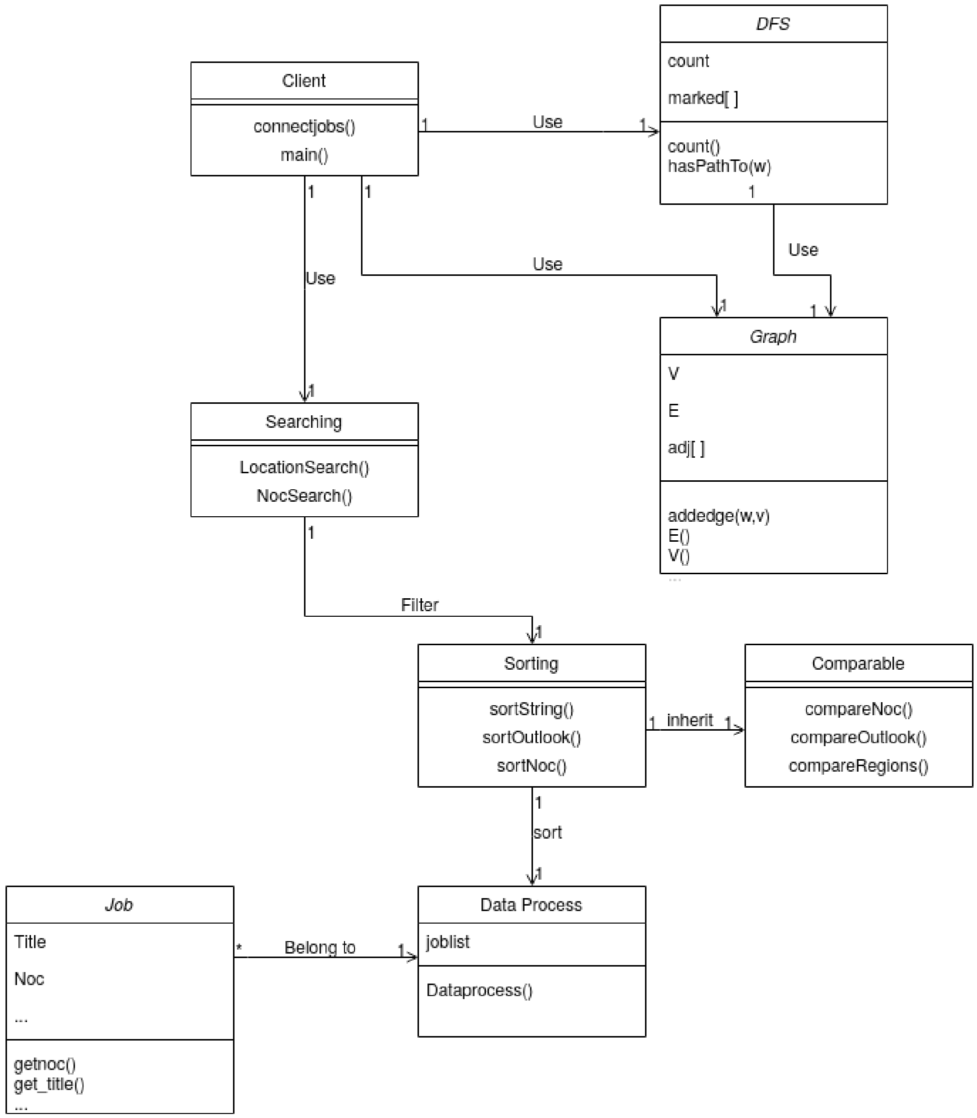

DFS
count
marked[ ]
count()
hasPathTo(w)
Client
connectjobs()
main()
Use
Graph
V
E
adj[ ]
addedge(w,v)
E()
V()
...
Searching
LocationSearch()
NocSearch()
Filter
Sorting
sortString()
sortOutlook()
sortNoc()
inherit
Comparable
compareNoc()
compareOutlook()
compareRegions()
sort
Job
Title
Noc
...
getnoc()
get_title()
...
Belong to
Data Process
joblist
Dataprocess()

Diagram.png

2 Detailed description of interfaces

Job ADT Module

Template Module

Job

Uses

N/A

Syntax

Exported Types

Job = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
Job	seq of $\mathbb{Z}$, String,$\mathbb{Z}$,$\mathbb{Z}$,String,$\mathbb{Z}$,String	Job	
get_noc	$\mathbb{Z}$	$\mathbb{Z}$	
getnoc		seq of $\mathbb{Z}$	
get_title		String	
get_location		String	
get_region		$\mathbb{Z}$	
get_outlook		$\mathbb{Z}$	
get_year		$\mathbb{Z}$	
get_regions		String	
getInfo		String	
getbriefInfo		String	
printInfo			
printbriefInfo			

Semantics

State Variables

noc: seq of $\mathbb{Z}$
title: String

$outlook$: $\mathbb{Z}$
$year$: $\mathbb{Z}$
$region$: $\mathbb{Z}$
$regions$: String
$location$: String

State Invariant

None

Assumptions

The constructor Job is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

Job($noc, title, outlook, year, location, region, regions$):

- transition: $noc, title, outlook, year, location, region, regions := noc, title, outlook, year, location, regi$
- output: $out := self$
- exception: None

get_noc($index$):

- output: $out := noc[index]$
- exception: None

getnoc( ):

- output: $out := noc[0] * 1000 + noc[1] * 100 + noc[2] * 10 + noc[3]$
- exception: None

get_title( ):

- output: $out := title$
- exception: None

get_location( ):

- output: $out := location$
- exception: None

get_region( ):

- output: $out := region$
- exception: None

get_outlook( ):

- output: $out := outlook$
- exception: None

get_year( ):

- output: $out := year$
- exception: None

get_regions( ):

- output: $out := regions$
- exception: None

printbriefInfo( ):

- exception: None

getbriefInfo( ):

- output: $out := "Job\ title : " + this.get\_title() + "Noc\ \_" + this.get\_noc(0) + "" + this.get\_noc(1) + "" + this.get\_noc(2) + "" + this.get\_noc(3)$
- exception: None

printInfo( ):

- exception: None

getInfo( ):

- output: $out := "Job\ title\ : " + this.get\_title() + "\backslash nNoc\_" + this.get\_noc(0) + "" + this.get\_noc(1) + "" + this.get\_noc(2) + "" + this.get\_noc(3) + "\backslash nOutlook : " + this.get\_outlook() + "\backslash nProvince : " + this.get\_location() + "\backslash nEconregioncode : " + this.get\_region() + "\backslash nEcon\ region\ name : " + this.get\_regions() + "\backslash nYear" + this.get\_year() + "\backslash n"$
- exception: None

DataProcess Module

Module

DataProcess

Uses

Job

Syntax

Exported Types

DataProcess = ?

Exported Access Programs

Routine name	In	Out	Exceptions
DataProcess			FileNotFoundException
get_data		seq of Job	

Semantics

State Variables

dataset: seq of Job

State Invariant

None

Assumptions

The constructor DataProcess is called only once for only one object for reading the dataset files.

Access Routine Semantics

DataProcess( ):

- transition: $dataset + Job$
- output: $out := self$
- exception: FileNotFoundException

get_data( ):

- output: $out := dataset$
- exception: None

Comparator Module

Module

Comparable

Uses

Job

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
CompareString	Job, Job	$\mathbb{Z}$	
CompareOutlook	Job, Job	$\mathbb{Z}$	
CompareNOC	Job, Job	$\mathbb{Z}$	
CompareRegionS	Job, Job	$\mathbb{Z}$	

Semantics

Access Routine Semantics

CompareString(a, b):

- output: out := a.get_title.compareTo(b.get_title)
- exception: None

// compareTo is a build in method to compare String in lexgraphical order.

CompareOutlook(a, b):

- output: out := (a.get_outlook > b.get_outlook) $\Rightarrow$ 1 | (a.get_outlook < b.get_outlook) $\Rightarrow$ -1 | 0
- exception: None

CompareNOC(a, b):

- output: out := (a.get_noc(0) > b.get_noc(0)) $\Rightarrow$ 1 | (a.get_noc(0) < b.get_noc(0)) $\Rightarrow$ -1 | 0

- exception: None

CompareRegionS(a, b):

- output: *out* := a.get_regions.compareTo(b.get_regions)
- exception: None

// compareTo is a build in method to compare String in lexgraphical order.

Sorting Module

Module

Sorting

Uses

Comparable

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
sortString	Seq of Job		
sortOutlook	Seq of Job		
sortNOC	Seq of Job		
sortRegionS	Seq of Job		

Semantics

Access Routine Semantics

sortString(a):

- transition: sortString(a, 0, $|a|$-1)
- exception: None

sortOutlook(a):

- transition: sortOutlook(a, 0, $|a|$-1)
- exception: None

sortNOC(a):

- transition: sortNOC(a, 0, $|a|$-1)
- exception: None

sortRegionS(a):

- transition: sortRegionS(a, 0, $|a|$-1)
- exception: None

Searching Module

Module

Searching

Uses

Job

Syntax

Exported Constants

None

Exported Types

Searching = seq of Job

Exported Access Programs

Routine name	In	Out	Exceptions
LocationSearch	seq of Job, String	seq of Job	
NocSearch	seq of Job, $\mathbb{Z}$	seq of Job	

Semantics

State Variables

None

State Invariant

None

Assumptions

None

Access Routine Semantics

LocationSearch($jobs$, $location$):

- output: $out := \langle e : Job | e \in jobs \wedge e.get\_regions() = location : e \rangle$ *Return a sequence of all the elements from input location in the input sequence jobs*

- exception: None

// get_regions() is a method from Job ADT class.

NocSearch($jobs$, noc):

- output: $out := \langle e : Job | e \in jobs \wedge e.get\_noc(0) = noc : e \rangle$ *Return a sequence of all the elements with same input noc number in the input sequence jobs*

- exception: None

// get_noc(int index) is a method from Job ADT class.

Graph Module

Module

Graph

Uses

N/A

Syntax

Exported Constants

None

Exported Types

Graph = ?
//An undirected graph with unweighed edges

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
Graph	$\mathbb{Z}$	Graph	NegativeArraySizeException
addedge	$\mathbb{N}$, $\mathbb{N}$		IllegalArgumentException
V		$\mathbb{N}$	
E		$\mathbb{N}$	
adj	$\mathbb{N}$	Seq of $\mathbb{N}$	IllegalArgumentException

Semantics

State Variables

V: $\mathbb{N}$
E: $\mathbb{N}$
adj: Seq of Seq of $\mathbb{N}$

State Invariant

None

Assumptions

The constructor Graph is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

//Constructor of Graph class
Graph(v):

- transition: $V, E, adj := v, 0, \text{Seq of Seq of } \mathbb{N} \text{ with length v}$
- output: $out := self$
- exception: $exc := v < 0 \Rightarrow$ NegativeArraySizeException

//Connect vertex w and vertex v
addedge(w, v):

- transition: $E, adj[w], adj[v] := E + 1, adj[w] \,||\, v, adj[v] \,||\, w$
- exception: $exc := w < 0 \vee w > V \vee v < 0 \vee v \vee V \Rightarrow$ IllegalArgumentException

//Getter, get the number of edges
E():

- output: $out := E$
- exception: None

//Getter, get the number of vertices
V():

- output: $out := V$
- exception: None

//Getter, get a list of nodes that are conneted with vertex v
adj(v):

- output: $out := adj[v]$
- exception: $exc := v < 0 \vee v > V \Rightarrow$ IllegalArgumentException

DFS Module

Module

DFS

Uses

Graph

Syntax

Exported Constants

None

Exported Types

DFS = ?
//Detect the reachable vertices from a source vertex

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
DFS	Graph, $\mathbb{N}$	DFS	IllegalArgumentException
hasPathTo	$\mathbb{N}$	$\mathbb{B}$	IllegalArgumentException
count		$\mathbb{N}$	

Semantics

State Variables

$count$: $mathbbN$
$marked$: Seq of $\mathbb{B}$

State Invariant

None

Assumptions

The constructor DFS is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

//Constructor of DFS class
Graph(g, s):

- transition: $count, \text{marked} := \text{number of reachable nodes}, \text{Seq of } \mathbb{B} \text{ recording if a vertex is reachable}$
- output: $out := self$
- exception: $exc := s < 0 \lor s >= g.V() \Rightarrow \text{IllegalArgumentException}$

//Determine if vertex w is reachable from the source vertex
hasPathTo(w):

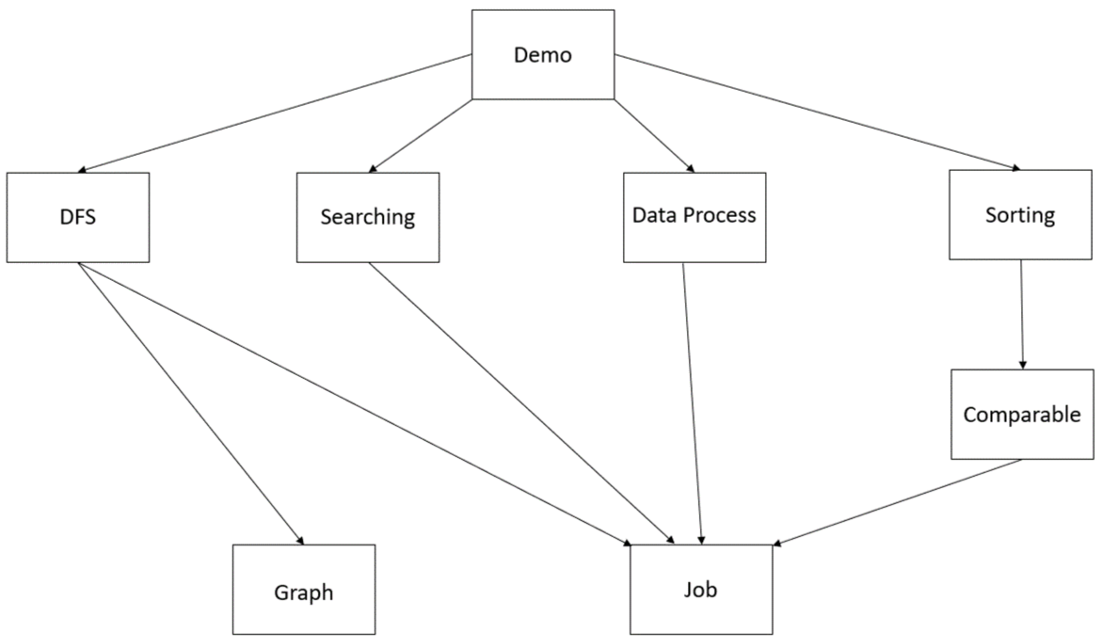
- output: $out := marked[w]$
- exception: $exc := w < 0 \lor w >= V \Rightarrow \text{IllegalArgumentException}$

//Getter, get the number of reachable vertices
count():

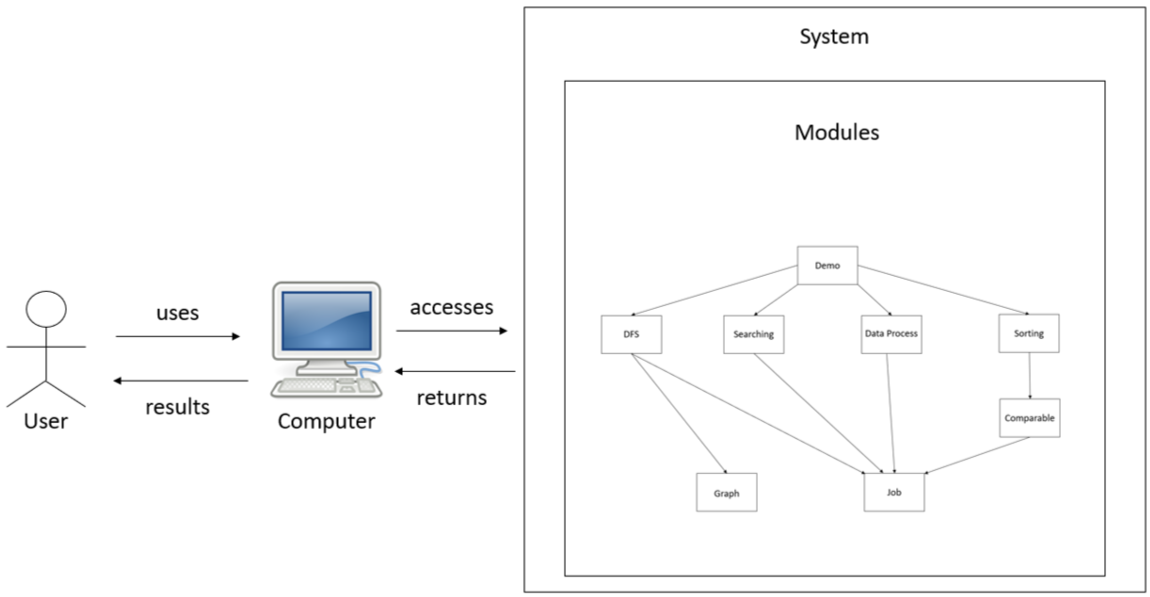
- output: $out := count$
- exception: None

3 View of uses relationship

Uses Hierarchy



Use case



4 Trace back to requirements

Demo Module (User Interface) : This module is used to do data processing operation which uses methods from DataProcess Module to read data from database files, and store data as expected type (Job ADT). Then it will give users instructions to use methods from other modules to sort, search, and create graph for the expected data. Demo Module satisfies the I/O requirements and User Interface requirements addressed in functional requirements. If users give unavailable inputs during using, the wrong inputs will not be taken and error messages will be displayed. So it also supports the robustness, usability, and understandability addressed in non-functional requirements.

DataProcess Module: As mentioned above, this module is used to read data from database files, and store data as expected type (Job ADT). It satisfies the I/O requirements addressed in functional requirements.

Job ADT Module: This module is used to build the ADT from the data in database files. The Job ADT is important to correctly use methods from other Modules like Sort, Searching, and Graph. Because the programmer provided detailed comments, thus the code is easy to understand and modify. So Job Module supports the understandability and maintainability addressed in non-functional requirements.

Comparator Module: This module is used to compare the comparable attributes (like job name, noc-code) of Job ADT. Sort module will use methods from this module to sort the dataset based on the selected attribute. The code includes clear doxygen comments. Therefore it is easy to understand and modify. Comparator Module supports the understandability and maintainability addressed in non-functional requirements.

Sort Module: This module is used to sort the dataset based on String, Outlook, NOC, and Regions by quick sort. Therefore it satisfies the sorting requirements addressed in functional requirements. Because it uses quick sort algorithm, so the performance requirements addressed in nonfunctional requirements is supported. The programmers also did Junit tests for this module, the correctness of results can be promised. Also, this programmer provided detailed comments in code for others to read and understand the code. Therefore the understandability and accuracy addressed in non-functional requirements are also satisfied.

Searching Module: This module is used to search the expected Job set based on the given location or given noc number by using binary search. Binary search algorithm can support the performance requirements. And like sort module, the code of this module has clear comments and Junit tests to ensure the understandability and accuracy.

Graph Module: This module is an ADT class used to create an undirected graph. Because the programmer provided detailed comments, thus the code is easy to understand and modify. So Job Module supports the understandability and maintainability addressed in non-functional requirements.

DFS Module: This module is used detect the reachable vertices from a source graph and source vertex by depth-first search. The algorithm satisfies the the performance requirements. Like above modules, the code of this module has clear comments and Junit tests to ensure the understandability and accuracy.

Overall: The product satisfies all the functional requirements. By the description above, the product also supports the reliability, robustness, performance, usability, maintainability, understandability, and accuracy. Besides, all the codes are implemented by Java, therefore the product can run on OS such as Windows, Mac OS, and Linux. Thus the portability is also supported.

5 Description of implementation

DFS Module

Module

DFS

Uses

Graph

Local Functions

dfs: $Graph \times \mathbb{N}$
//The private dfs method recursively call itself to detect deeper layer of the graph until it hits a sink vertex, then it will turn back to the previous layer and detect again. It updates the state variable "count" and "marked[]" to avoid repeatation of exploration

Sorting Module

Module

Sorting

Uses

Comparable

Local Functions

exch: Seq of Job $\times \mathbb{Z} \times \mathbb{Z} \rightarrow$ None
exch$(a, i, j) \equiv$ exchange a[i] and a[j] in the array

sortString: Seq of Job $\times \mathbb{Z} \times \mathbb{Z} \rightarrow$ None
sortString$(a, lo, hi) \equiv$ (hi <= lo) $\Rightarrow$ return | sortString(a, lo, j-1) && sortString(a, j+1, hi) where j = partitionString(a, lo, hi)

partitionString: Seq of Job $\times \mathbb{Z} \times \mathbb{Z} \rightarrow$ None
partitionString$(a, lo, hi) \equiv$ partition on array a using ComapreString, see detail in code

sortOutlook: Seq of Job $\times \mathbb{Z} \times \mathbb{Z} \rightarrow$ None
sortOutlook$(a, lo, hi) \equiv$ (hi <= lo) $\Rightarrow$ return | sortOutlook(a, lo, j-1) && sortOutlook(a, j+1, hi) where j = partitionOutlook(a, lo, hi)

partitionOutlook: Seq of Job $\times \mathbb{Z} \times \mathbb{Z} \rightarrow$ None
partitionOutlook$(a, lo, hi) \equiv$ partition on array a using ComapreOutlook, see detail in code

sortNOC: Seq of Job $\times \mathbb{Z} \times \mathbb{Z} \rightarrow$ None
sortNOC$(a, lo, hi) \equiv$ (hi <= lo) $\Rightarrow$ return | sortNOC(a, lo, j-1) && sortNOC(a, j+1, hi) where j = partitionNOC(a, lo, hi)

partitionNOC: Seq of Job $\times \mathbb{Z} \times \mathbb{Z} \rightarrow$ None
partitionNOC$(a, lo, hi) \equiv$ partition on array a using ComapreNOC, see detail in code

sortRegionS: Seq of Job $\times \mathbb{Z} \times \mathbb{Z} \rightarrow$ None
sortRegionS$(a, lo, hi) \equiv$ (hi <= lo) $\Rightarrow$ return | sortRegionS(a, lo, j-1) && sortRegionS(a,

j+1, hi) where j = partitionRegionS(a, lo, hi)

partitionRegionS: Seq of Job $\times \mathbb{Z} \times \mathbb{Z} \rightarrow$ None
partitionRegionS$(a, lo, hi) \equiv$ partition on array a using ComapreRegionS, see detail in code

Searching Module

Module

Searching

Uses

Job

Local Functions

Location_Search: Seq of Job $\times$ String $\rightarrow \mathbb{N}$
Location_Search$(jobs, location) \equiv (m : \mathbb{N} | m \in [0..|jobs|-1 \wedge jobs.get(m).get_regions() = location] : m)$

//The private Location_Search method is based on binary searching algorithm. It will first create two int variables, $l = 0$ and $r = jobs.size() - 1$. Then it will process a while loop : create a int variable $m = l + (r - l) \div 2$, then it will create another int variable $res = noc.compareTo(jobs.get(m).get\_noc(0))$. If $res = 0$, return m; if $res < 0$, then $r = m - 1$, continue the loop; if $res > 0$, then $l = m + 1$, continue the loop.

noc_Search: Seq of Job $\times \mathbb{Z} \rightarrow \mathbb{N}$
noc_Search$(jobs, noc) \equiv (m : \mathbb{N} | m \in [0..|jobs| - 1] \wedge jobs.get(m).get_noc(0) = noc : m)$

//The private Location_Search method is based on binary searching algorithm. It will first create two int variables, $l = 0$ and $r = jobs.size() - 1$. Then it will process a while loop : create a int variable $m = l + (r - l) \div 2$, then it will create another int variable $res = location.compareTo(jobs.get(m).get\_regions())$. If $res = 0$, return m; if $res < 0$, then $r = m - 1$, continue the loop; if $res > 0$, then $l = m + 1$, continue the loop.

6 Internal review

7 References

- GNOME Project. An Icon from the GNOME-Icon-Theme. WEKIMEDIA COMMONS, GNOME, 12 Jan. 2008, https://commons.wikimedia.org/wiki/File:Gnome-computer.svg.

- Sedgewick Robert, and Kevin Wayne. Algotithms, 4th Edition.

- Open Government Portal (2018, May 15). 3-year Employment Outlooks . Retrieved from https://open.canada.ca/data/en/dataset/b0e112e9-cf53-4e79-8838-23cd98debe5b