**Comparative Study of three Memory Allocation Algorithms used in Dynamic Partitioning: First Fit, Worst Fit, Best Fit**

COMP4721 Assignment #2

Michael Anderson

201449410

October 9, 2017

COMP4721

Ashoke Deb

**Table of Contents:**

**User’s Guide**

* 1. **Overview……………………………………………………………………………………………………………………………………3**
  2. **System Requirements………………………………………………………………………………………………………3**
  3. **Operating Guide…………………………………………………………………………………………………………………4**
  4. **Instruction (What to type)………………………………………………………………………………4-5**

**Logical Analysis**

* 1. **Logical Structure of the System………………………………………………………………………6**
  2. **Black Box Definitions…………………………………………………………………………………………7-17**

**Dictionary**

**2.1 Dictionary of Functions…………………………………………………………………………………18-22**

**2.1.1 Built-in Functions……………………………………………………………………………………18**

**2.1.2 User-defined Functions…………………………………………………………………19-21**

**2.2 Dictionary of Variables…………………………………………………………………………………22-23**

**Java Code & Testing**

**3.1 The Code…………………………………………………………………………………………………………………………24-36**

**3.2 Run-time output………………………………………………………………………………………………………37-38**

**3.3 instructionHandler() Function Test……………………………………………………………39**

**3.4 removeJob() & reloadJob Function Tests……………………………………………40-41**

**3.5 FirstFit, BestFit, WorstFit() Function Tests…………………………42-44**

**3.6 checkValidity() Function Test…………………………………………………………………45-46**

**3.7 recallOrExit() Function Test……………………………………………………………………………47**

**Issues and Limitations**

**4.1 Error Handling…………………………………………………………………………………………………………………48**

**4.2 Generality……………………………………………………………………………………………………………………………48**

**User’s Guide**

* 1. **Overview:**

The goal of the system is to create a comparative study to analyze three memory allocation algorithms used in Dynamic Partitioning: First Fit, Best Fit, and Worst Fit. Dynamic Partitioning is similar compared to other partitioning methods discussed in CS3725 due to the fact that a job’s logical address is bound to its corresponding physical address at the time of loading, however the number and sizes of the partitions evolve during the process lifetime of the system.

This was achieved by using Java’s LinkedList feature primarily.

Given the user’s input of the Memory size and a sequence of instructions, the system will maintain three instances of a Memory, and execute the sequence of instructions on each based off of their respective algorithms.

First Fit will load jobs into the first available partition of adequate size.

Best Fit will load jobs into the smallest available partition of adequate size.

Worst Fit will load jobs into the largest available partition of adequate size.

Snapshots of the Memory as well as Allocation/Free Lists are printed either after each instruction, or each sequence of instructions, based on user desire.

**0.2 System Requirements**

The program may be access and modified using Java 7 and newer, through any function IDE. The program can be ran through the system console (CMD.exe) or through an IDE with runtime support.Java 7 may be ran on Microsoft Windows XP and newer operating systems, as well as an Intel-based Mac running Mac OS X 10.7.3 or later operating system. For full Java 7 system requirements visit <http://www.oracle.com/technetwork/java/javase/config-417990.html>

**0.3 Operating Guide**

The file name of the program is MemoryAlgorithms.java and can be saved under any supporting public directory. For the sake of testing, it was saved in the directory of:

C:\Users\Michael\Documents\4721 A2\src\com\company\MemoryAlgorithms.java

To locate and access the program, the user can open the Start menu and navigate to the Search feature. Typing “cmd” and pressing enter will open the system’s Command Line. The user may run the program by navigating to the directory in which they have the program saved, and then then executing it. For the example above, the user would enter

“cd Documents … cd 4721 A2 … cd src\com\company”

To compile and execute the program the user types:

Compile:

“javac MemoryAlgorithms.java”

Execute:

“java –classpath com.company.MemoryAlgorithms”

**0.4 Instructions (What to type)**

When first running the program, it will prompt you with the following statement:

*Please enter the size of your Memory.*

What you are required to enter here is a natural number that you wish to be the size of the memory in which the program is working with.

You will then be prompted with:

*Please choose one the follow options by entering the corresponding value:*

*1 - Print Allocation/Availability Tables & Memory Snapshots after the execution of EACH instruction*

1. *- Print Allocation/Availability Tables & Memory Snapshots after the execution of the sequence of instructions*

You can enter either “1” or “2”. Nothing more, nothing less.

After selecting an option, you will be prompted with:

*Instruction formats:*

*To load a new job, type: (load, job #, [jobSize]k)*

*To remove a job, type: (remove, job #)*

*To reload a job, type: (reload, job #)*

*At this time you may input your sequence in the following format (please note brackets):*

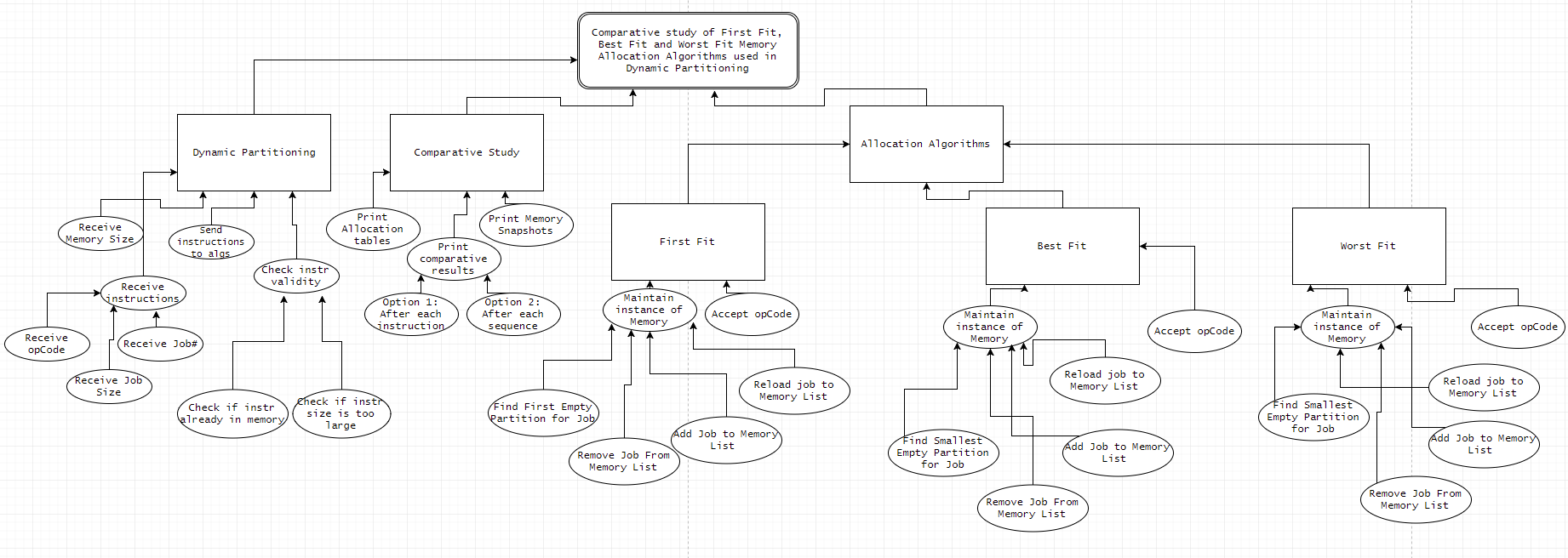
*(instruction 1), (instruction 2), (instruction 3), ...*

This is your first opportunity to input instruction. All instructions must be wrapped in parentheses “()” and follow the spacing as illustrated in the example, for the sake of substrings.

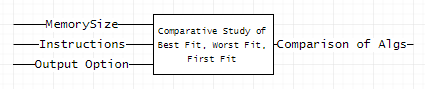
**Logical Analysis**

**1.1 Logical Structure of the System**

Below is the top-down logic follow when developing the logical structure of the code.



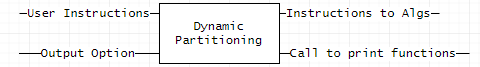
**1.2 Black Box Definitions**



**Black Box Definition for Top Node**

**Input:** Inputted Size of the Memory, Sequence of Instructions, and Output Option.

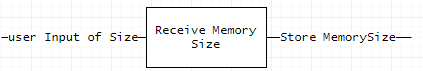
**Output:** Organized comparison of all 3 memory allocation algorithms, outputted to user-described standard.



**Black Box Definition for Dynamic Partitioning Node**

**Input:** Given user instructions and output option.

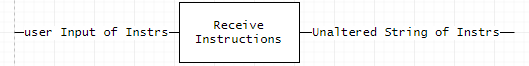
**Output:** Sends instructions to Algorithms, calls print functions.



**Black Box Definition for Receive Memory Size Node**

**Input:** User inputted size.

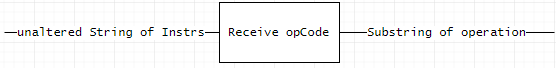
**Output:** Stores the size of the memory to be executed upon.



**Black Box Definition for Receive Instructions Node**

**Input:** Sequence of user inputted instructions.

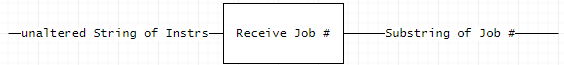
**Output:** Stores unaltered string of instructions in preparation for dissection



**Black Box Definition for Receive opCode Node**

**Input:** Unaltered String of user-inputted instructions.

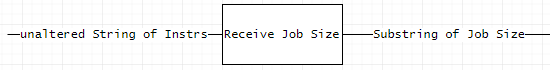
**Output:** Substrings containing individual relevant segments of information within instruction string containing the action to be executed.



**Black Box Definition for Receive Job # Node**

**Input:** Unaltered String of user-inputted instructions.

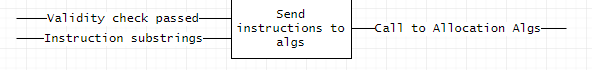
**Output:** Substrings containing individual relevant segments of information within instruction string containing the Job #.



**Black Box Definition for Receive Job Size Node**

**Input:** Unaltered String of user-inputted instructions.

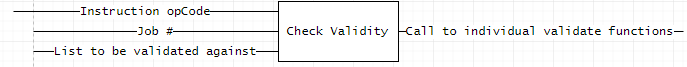
**Output:** Substrings containing individual relevant segments of information within instruction string containing the Job Size.



**Black Box Definition for Send Instructions to Algorithms Node**

**Input:** Substrings of instruction information & a Boolean variable stating the pass of a validity check.

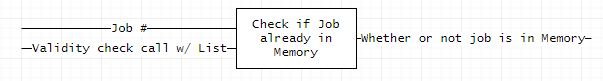
**Output:** Individual calls to each allocation algorithm.



**Black Box Definition for Check Validity Node**

**Input:** Instruction opcode variable, so the function knows which instruction is trying to be performed. Job # of job to be checked for validity. List variable initiating validity check against a specific list.

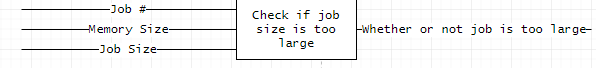
**Output:** Individual calls to each validation algorithm.



**Black Box Definition for Check if Job is Already in Memory**

**Input:** Validity check call from handler function, including Job #, List, and Instruction.

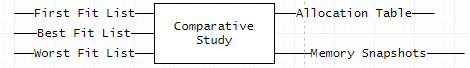
**Output:** Boolean whether or not job is already in Memory.



**Black Box Definition for Check if Job Size is too Large**

**Input:** Job #, Memory Size, Job Size.

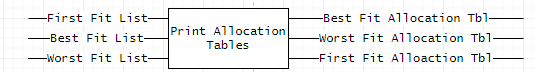
**Output:** Whether or not job is too large to fit in Memory in current state.



**Black Box Definition for Comparative Study Node**

**Input:** First Fit, Best Fit and Worst Fit Lists, altered entirely after instruction sequence execution.

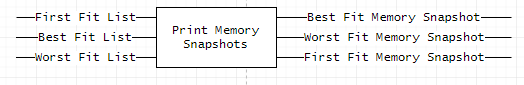
**Output:** Allocation Tables & Memory Snapshots for each algorithm.



**Black Box Definition for Print Allocation Tables Node**

**Input:** First Fit, Best Fit and Worst Fit Lists, altered entirely after instruction sequence execution.

**Output:** Best Fit Allocation Table, Worst Fit Allocation Table, First Fit Allocation Table.



**Black Box Definition for Print Memory Snapshots Node**

**Input:** First Fit, Best Fit and Worst Fit Lists, altered entirely after instruction sequence execution.

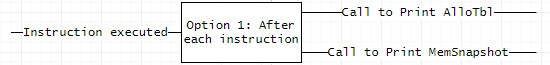
**Output:** Best Fit Memory Snapshot, Worst Fit Memory Snapshot, First Fit Memory Snapshot.

https://puu.sh/xSHAf/0ea37c422f.png

**Black Box Definition for Print Comparative Results Node**

**Input:** User Option of output frequency.

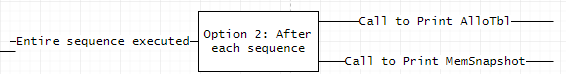
**Output:** Call to print Allocation Tables & Memory Snapshots, as often as user prescribed.



**Black Box Definition for Option 1: After each instruction Node**

**Input:** Instruction executed

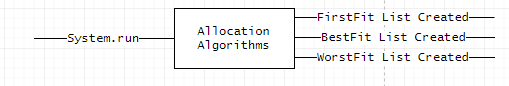
**Output:** Call to print Allocation Tables & Memory Snapshots.



**Black Box Definition for Option 2: After each sequence of instructions Node**

**Input:** Entire sequence of instructions executed.

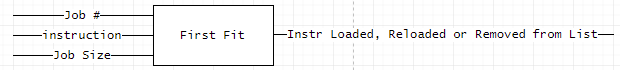
**Output:** Call to print Allocation Tables & Memory Snapshots.



**Black Box Definition for Allocation Algorithms Node**

**Input:** System run.

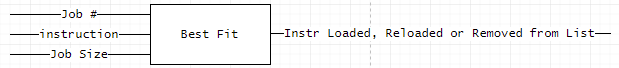
**Output:** Creation of FirstFit, BestFit and WorstFit Lists.



**Black Box Definition for First Fit Node**

**Input:** Job #, Instruction to be executed and Job Size.

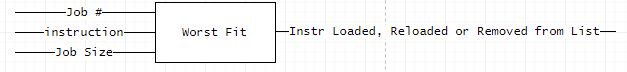
**Output:** Instruction either loaded, reloaded or removed from List.



**Black Box Definition for Best Fit Node**

**Input:** Job #, Instruction to be executed and Job Size.

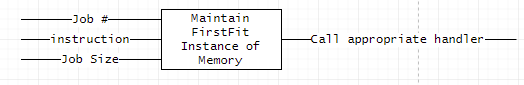
**Output:** Instruction either loaded, reloaded or removed from List.



**Black Box Definition for Worst Fit Node**

**Input:** Job #, Instruction to be executed and Job Size.

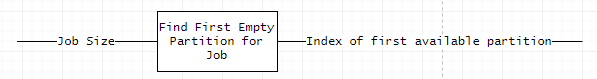
**Output:** Instruction either loaded, reloaded or removed from List.



**Black Box Definition for Maintain FirstFit Instance of Memory Node**

**Input:** Job #, Instruction to be executed and Job Size.

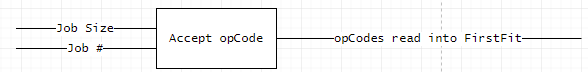
**Output:** Call to appropriate handler, based off of instruction.



**Black Box Definition for Find First Empty Partition for Job**

**Input:** Job Size to be allocated.

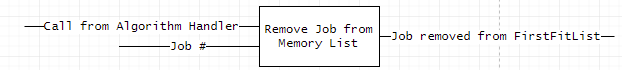
**Output:** Index of first available partition to be allocated.



**Black Box Definition for Accept opCode Node (First Fit)**

**Input:** Job # and Job Size.

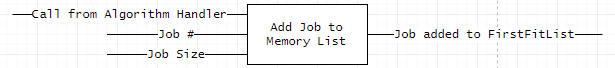
**Output:** opCodes read into FirstFit algorithm.



**Black Box Definition for Remove Job from Memory List Node (First Fit)**

**Input:** Call from Algorithm handler (to prompt removal), Job # to be removed.

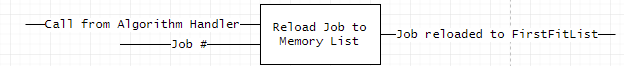
**Output:** Job removed from FirstFitList.



**Black Box Definition for Add Job from Memory List Node (First Fit)**

**Input:** Call from Algorithm handler (to prompt addition), Job # to be added, Job Size.

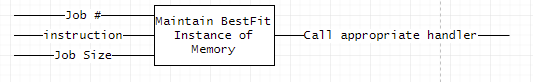
**Output:** Job added to FirstFitList.



**Black Box Definition for Add Job from Memory List Node (First Fit)**

**Input:** Call from Algorithm handler (to prompt reload), Job # to be reloaded.

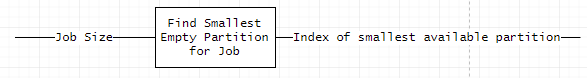
**Output:** Job reloaded to FirstFitList.



**Black Box Definition for Maintain BestFit Instance of Memory Node**

**Input:** Job #, Instruction to be executed and Job Size.

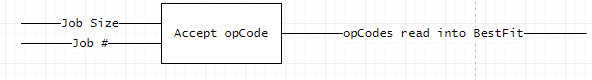
**Output:** Call to appropriate handler, based off of instruction.



**Black Box Definition for Find Smallest Empty Partition for Job**

**Input:** Job Size to be allocated.

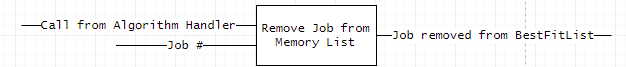
**Output:** Index of smallest available partition to be allocated.



**Black Box Definition for Accept opCode Node (Best Fit)**

**Input:** Job # and Job Size.

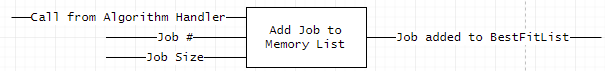
**Output:** opCodes read into BestFit algorithm.



**Black Box Definition for Remove Job from Memory List Node (BestFit)**

**Input:** Call from Algorithm handler (to prompt removal), Job # to be removed.

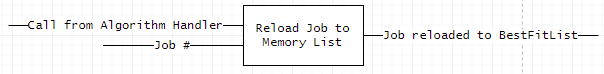
**Output:** Job removed from BestFitList.



**Black Box Definition for Add Job from Memory List Node (BestFit)**

**Input:** Call from Algorithm handler (to prompt addition), Job # to be added, Job Size.

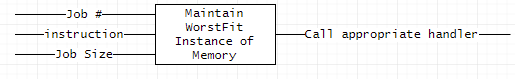
**Output:** Job added to BestFitList.



**Black Box Definition for Reload Job from Memory List Node (Best Fit)**

**Input:** Call from Algorithm handler (to prompt reload), Job # to be reloaded.

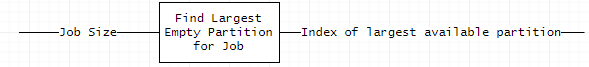
**Output:** Job reloaded to BestFitList.



**Black Box Definition for Maintain WorstFit Instance of Memory Node**

**Input:** Job #, Instruction to be executed and Job Size.

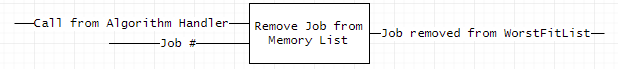
**Output:** Call to appropriate handler, based off of instruction.



**Black Box Definition for Find Largest Empty Partition for Job**

**Input:** Job Size to be allocated.

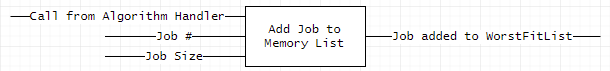
**Output:** Index of largest available partition to be allocated.



**Black Box Definition for Remove Job from Memory List Node (WorstFit)**

**Input:** Call from Algorithm handler (to prompt removal), Job # to be removed.

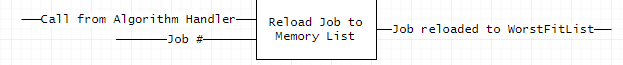
**Output:** Job removed from WorstFitList.



**Black Box Definition for Add Job from Memory List Node (WorstFit)**

**Input:** Call from Algorithm handler (to prompt addition), Job # to be added, Job Size.

**Output:** Job added to WorstFitList.



**Black Box Definition for Reload Job from Memory List Node (Worst Fit)**

**Input:** Call from Algorithm handler (to prompt reload), Job # to be reloaded.

**Output:** Job reloaded to WorstFitList.

**Dictionary**

**2.1 Dictionary of Functions**

Within the program, many functions are used. Some are defined within the program while others are built-in Java functions:

**2.1.1 Built-in Functions:**

**println()** – Prints the strings contained in the argument to the user.

**hasNext()** – returns whether or not the prefix has a pending input.

**nextInt()** – returns the next integer in the prefix.

**add()** – adds the argument to the list described in the prefix.

**size()** – returns the size of the list in the prefix.

**remove()** – removes the first-entered element in the queue in the prefix.

**LinkedList<Integer>()** – creates a new LinkedList of integers.

**Scanner()** – creates a new Scanner objects that reads from the argument.

**split** – splits the array of Strings in the prefix at the given delimeter.

**length** – returns the length of the array in the prefix.

**substring** – breaks the given string up into smaller substrings.

**parseInt** – converts a given string into its integer equivalent.

**compareTo** – compares the value contents of 2 strings, and returns an integer value describing the comparison.

**get** – returns the value of the list at the given index.

**remove** – removes the value of the list at the given index.

**valueOf** – returns the value of the string in the parameter.

**2.1.2 User-defined Functions:**

**instructionHandler() –** This instructionHandler function takes the untouched input from the user and uses substrings to dissect it into relevant parts of the string. This does require, however, that the user inputs their instruction  
in a specific format, as commas and brackets are used as delimiters.  
Once the instructions are in a usable format, they are sent to respective functions to alter the Memory accordingly.  
Each instruction is checked for validity in EACH list, as there are rare circumstances in which 1 list may contain a job that another  
did not (as one may not have had enough space to hold it). If the instruction passes the validity check, it moves forward unannounced,  
otherwise it lets the user know that the instruction will not be performed.  
The first part of each instruction will always be the operation to take place (opcode), the second part  
will always be the job #, and the third, in the case of the load instruction, is the size of the job.  
On subsequent loads of a job (reload function), the size is already known as it is maintained in a previouslyLoadedJobs list.

**reloadJob() –** reloadJob function takes the jobNumber from the instructionHandler and searches through the list of maintained  
and corresponding job sizes associated with each previously loaded job. If it finds the job number in question,  
the job is loaded back into the memory.

**removeJob() –** The removeJob function iterates through each of the memory lists in search for the job Number in question.  
Once it is found, it removes the entire 3-tuple from the list.

**firstFit() –** First Fit Memory Allocation follows the logic of allocating any given job with size n in the first partition  
to be found with size>=n. Thus, this function works by taking a job size and its number, iterating through the memory list,  
and checking to see if there is a space available between any 2 subsequent instructions. If there is, then if that space is greater than or  
equal to the size of the job, then the job number is inputted at the location of the free space.

**bestFit() –** Best Fit Memory Allocation follows the logic of allocating any given job with size n in the smallest partition of size>=n.Thus, this function works by taking a job size and its number, iterating through the memory list, and checking to  
see if there is a space available between any 2 subsequent instructions. The partition with the smallest found size to date  
has its index maintained in a temporary variable until the end of the iteration. Once it has completed, the job is assigned at  
the index of the smallest partition.

**worstFit() –** Worst Fit Memory Allocation follows the logic of allocating any given job with size n in the largest partition of size>=n. Thus, this function works by taking a job size and its number, iterating through the memory list, and checking to  
see if there is a space available between any 2 subsequent instructions. The partition with the largest found size to date  
has its index maintained in a temporary variable until the end of the iteration. Once it has completed, the job is assigned at  
the index of the largest partition.

**printAllocationTables() –** printAllocationTables function prints the allocation table for each instruction (or each sequence of instructions, depending on user inputted decision) in the format depicted in CS3725 lectures. E.g. the function iterates through the memory and prints all of its containing data in a table format, with columns Size, Location, State and Job. Each row contains 1 partition.  
This is done for all 3 of the memory allocation algorithms.

**printSnapshots() –** printSnapshots function prints a snapshot of the Memory for each instruction (or each sequence of instructions, depending  
on user inputted decision) in the format depicted in CS3725 lectures. E.g. the function iterates through the memory and prints all of its containing data in the format of Starting Location (of each partition) printed on the left side, and the  
Job that is contained in each partition is written inside of the block. Each block represents 1 partition.  
This is done for all 3 of the memory allocation algorithms.

**checkValidity() –** The checkValidity function takes 3 arguments, Job #, sender and list. The sender is the instruction opcode from the instructionHandler function that is being executed and requires the validity check. The list is which of the 3  
Memory lists to reference.  
For the load instruction, each list is independently checked through a separate call of checkValidity, as  
the job can be loaded into 1 list even if it is already contained in the other 2, etc.  
For remove, the job is searched for & removed in applicable in each list. 3 boolean variables are maintained,  
representing whether or not the job was found in each list. If it is not found in all 3, this is made known  
to the user.  
For reload, the load sender is also called, with the list set as previous (to check the list of previously loaded jobs),  
but the boolean is simply inverted in the instructionHandler function.

**recallOrExit() –** The recallOrExit function simply prompts the user to either continue running the program, or exit. This just serves as a simple way to loop the program until the user decides to exit on their own, allowing them to input as many  
instructions as they desire.

**2.2 Dictionary of Variables**

The following is a list of all variables used within the program code:

**MemorySize –** integer containing user inputted size of Memory.

**firstFitList –** List depicting first fit instance of Memory.

**bestFitList –** List depicting best fit instance of Memory.

**worstFitList –** List depicting worst fit instance of Memory.

**previouslyLoadedJobs –** List containing information on previously loaded jobs for reloadable ease.

**in –** variable to be used with Scanner object to read in user data.

**operatingSystemSize –** hard-coded size of Operating System.

**userOption –** integer deciding way in which user prefers data outputted.

**instructionList –** list of unaltered user instructions.

**Instructions –** array of strings containing user instructions.

**x –** iterative variable.

**pass –** Boolean variable for deciding whether or not instruction passes validity check.

**decider –** final array of strings that user input is held in.

**jobNumber** – Job # associated with given instruction.

**jobSize –** Size associated with loading of given instruction.

**size –** size of List at given time, used for iteration.

**i –** iterative variable.

**jobLocation –** variable used to store location of a given job.

**firstRemoved** – Boolean used to depict if job was removed from first fit list.

**bestRemoved** – Boolean used to depict if job was removed from best fit list.

**worstRemoved** – Boolean used to depict if job was removed from worst fit list.

**added -** – Boolean used to depict if job was added to given list.

**lastLocation** – integer variable storing location of the last job in the given list.

**lastJobSize -** integer variable storing size of the last job in the given list.

**smallestFit -** integer variable storing size of smallest partition in the Best Fit List.

**indexOfSmallest -** integer variable storing index of smallest partition in the Best Fit List.

**currentFit - -** integer variable storing size of current partition in the Best Fit List, while iterating through.

**largestFit -** integer variable storing size of largest partition in the Worst Fit List.

**indexOfLargest -** integer variable storing index of largest partition in the Worst Fit List.

**remainingSpace** - remaining space in Memory after all partitions.

**emptyPartitionLoc –** location of indexed empty partition

**sender –** variable used to determine which opcode is being referenced for the checkValidity function.

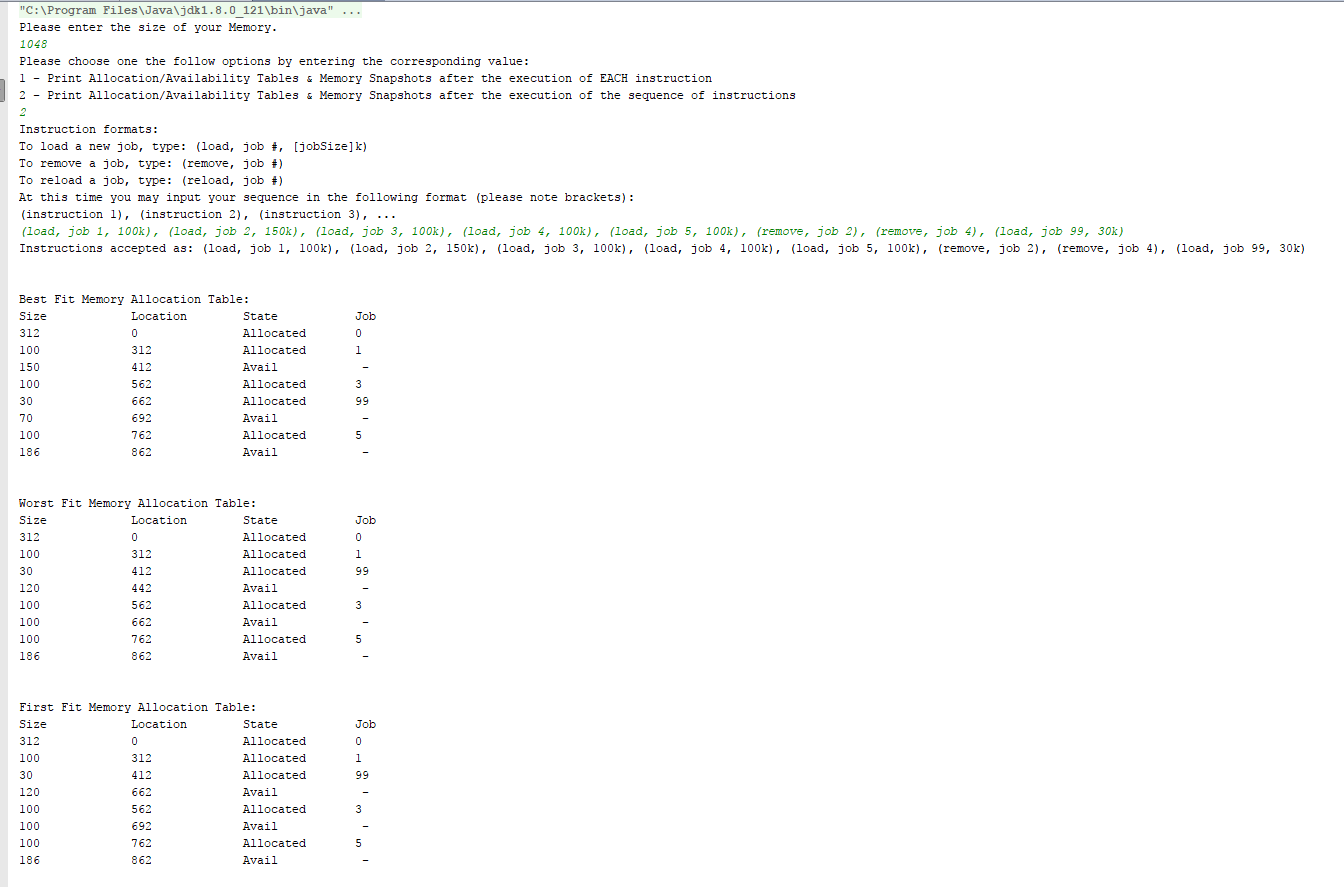
**List –** variable referencing which list to be checked for the checkValidity function.

**Java Code & Testing**

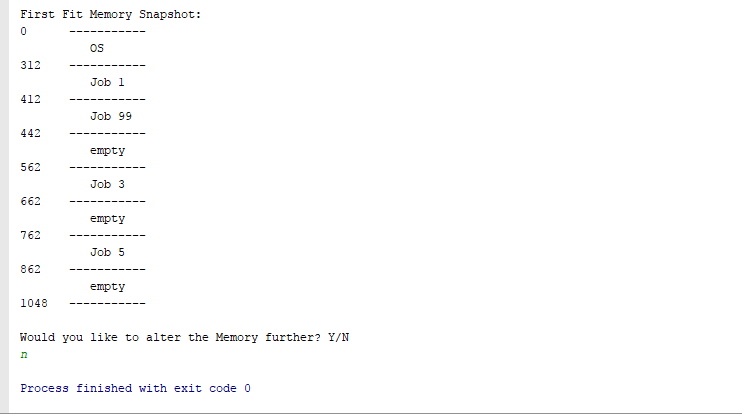
**3.1 The Code**

Below is the full Java code for the program:

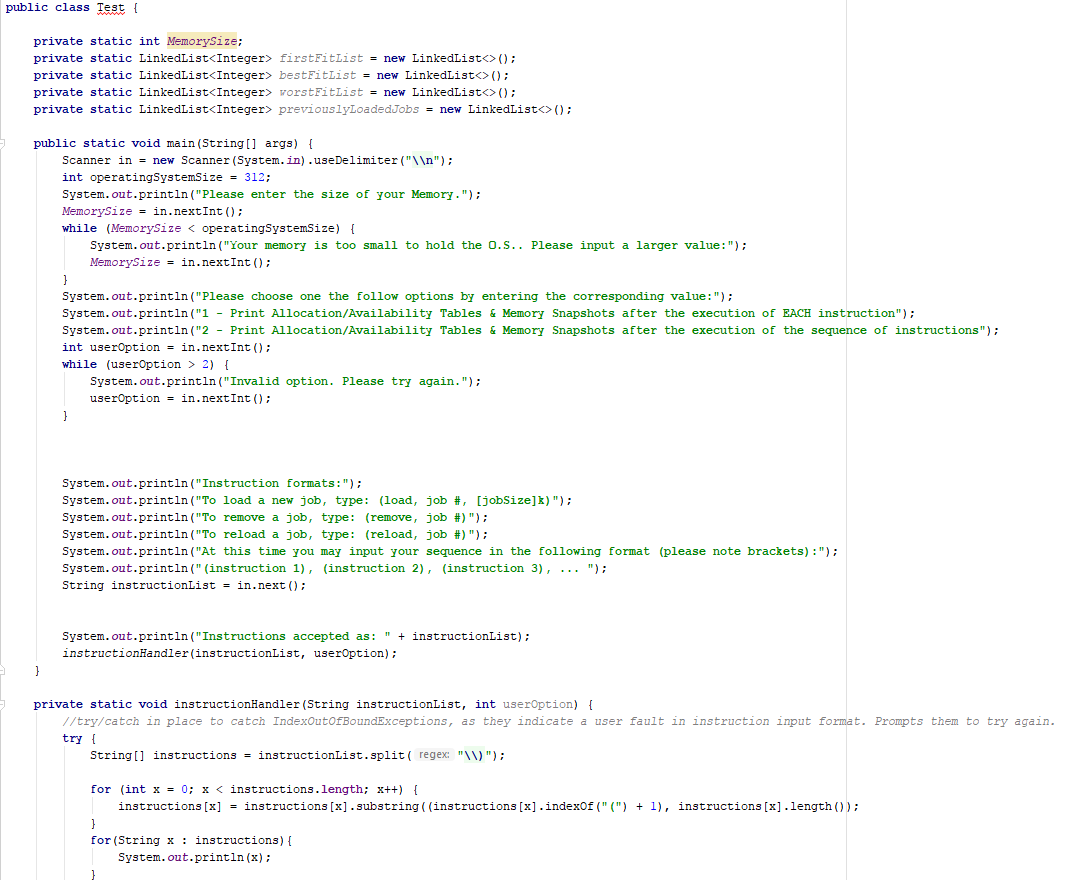
**package** com.company;  
**import** java.util.LinkedList;  
**import** java.util.Scanner;  
*/\*  
 Program designed for a comparative study of Best Fit, Worst Fit and First Fit Memory Allocation  
 techniques, used in Dynamic Partitioning. Given the user's input of Memory Size, the program then prompts  
 the user to select an output format. Once decided, the user is free to input 1 or more instructions to be  
 executed on the memory (3 instances of the same memory, handled different by each algorithm). Once the inputted  
 instruction(s) is/are executed, the system will prompt the user for either more input, or to exit.  
 Michael Anderson - 20144910 - Oct/2017  
 \*/***public class** MemoryAlgorithms {  
 */\*  
 Lists to handle the memory management of each algorithm. All lists are ordered 3-tuples,  
 following the format of Job #, Job Size, Job Location, ...  
 \*/* **private static int** *MemorySize*;  
 **private static** LinkedList<Integer> *firstFitList* = **new** LinkedList<>();  
 **private static** LinkedList<Integer> *bestFitList* = **new** LinkedList<>();  
 **private static** LinkedList<Integer> *worstFitList* = **new** LinkedList<>();  
 **private static** LinkedList<Integer> *previouslyLoadedJobs* = **new** LinkedList<>();  
  
 **public static void** main(String[] args) {  
 *//Scanner set to use delimiter of line break rather than white space, to ease the readability of instruction sequences.* Scanner in = **new** Scanner(System.***in***).useDelimiter(**"\\n"**);  
 *//Operating System Size hard coded to represent a generic operating system, as typically this is not up to the user to decide the size.* **int** operatingSystemSize = 312;  
 System.***out***.println(**"Please enter the size of your Memory."**);  
 *MemorySize* = in.nextInt();  
 **while**(*MemorySize* < operatingSystemSize) {  
 System.***out***.println(**"Your memory is too small to hold the O.S.. Please input a larger value:"**);  
 *MemorySize* = in.nextInt();  
 }  
 System.***out***.println(**"Please choose one the follow options by entering the corresponding value:"**);  
 System.***out***.println(**"1 - Print Allocation/Availability Tables & Memory Snapshots after the execution of EACH instruction"**);  
 System.***out***.println(**"2 - Print Allocation/Availability Tables & Memory Snapshots after the execution of the sequence of instructions"**);  
 **int** userOption = in.nextInt();  
 **while**(userOption > 2) {  
 System.***out***.println(**"Invalid option. Please try again."**);  
 userOption = in.nextInt();  
 }  
 */\*  
 For all 3 memory instances, a base case and operating system are automatically added to the list.  
 This allows for more general calls in the future when iterating through the list. The base case  
 is never shown or referenced to the user.  
 \*/  
 firstFitList*.add(0); *//base case  
 firstFitList*.add(0);  
 *firstFitList*.add(0);  
 *firstFitList*.add(0); *//os  
 firstFitList*.add(operatingSystemSize);  
 *firstFitList*.add(0);  
 *bestFitList*.add(0); *//base case  
 bestFitList*.add(0);  
 *bestFitList*.add(0);  
 *bestFitList*.add(0); *//os  
 bestFitList*.add(operatingSystemSize);  
 *bestFitList*.add(0);  
 *worstFitList*.add(0); *//base case  
 worstFitList*.add(0);  
 *worstFitList*.add(0);  
 *worstFitList*.add(0); *//os  
 worstFitList*.add(operatingSystemSize);  
 *worstFitList*.add(0);  
  
  
 System.***out***.println(**"Instruction formats:"**);  
 System.***out***.println(**"To load a new job, type: (load, job #, [jobSize]k)"**);  
 System.***out***.println(**"To remove a job, type: (remove, job #)"**);  
 System.***out***.println(**"To reload a job, type: (reload, job #)"**);  
 System.***out***.println(**"At this time you may input your sequence in the following format (please note brackets):"**);  
 System.***out***.println(**"(instruction 1), (instruction 2), (instruction 3), ... "**);  
 String instructionList = in.next();  
  
 */\*This output is largely for testing, but decided to leave it to ensure the user knows their input  
 was accepted as intended. Not exactly necessary, but nice quality of life.  
 \*/* System.***out***.println(**"Instructions accepted as: "**+instructionList);  
 *instructionHandler*(instructionList, userOption);  
 }  
  
 */\*  
 This instructionHandler function takes the untouched input from the user and uses substrings to dissect it  
 into relevant parts of the string. This does require, however, that the user inputs their instruction  
 in a specific format, as commas and brackets are used as delimiters.  
 Once the instructions are in a usable format, they are sent to respective functions to alter the Memory accordingly.  
 Each instruction is checked for validity in EACH list, as there are rare circumstances in which 1 list may contain a job that another  
 did not (as one may not have had enough space to hold it). If the instruction passes the validity check, it moves forward unannounced,  
 otherwise it lets the user know that the instruction will not be performed.  
 The first part of each instruction will always be the operation to take place (opcode), the second part  
 will always be the job #, and the third, in the case of the load instruction, is the size of the job.  
 On subsequent loads of a job (reload function), the size is already known as it is maintained in a previouslyLoadedJobs list.  
 \*/* **private static void** instructionHandler(String instructionList, **int** userOption){  
 *//try/catch in place to catch IndexOutOfBoundExceptions, as they indicate a user fault in instruction input format. Prompts them to try again.* **try** {  
 String[] instructions = instructionList.split(**"\\)"**);  
  
 **for** (**int** x = 0; x < instructions.**length**; x++) {  
 instructions[x] = instructions[x].substring((instructions[x].indexOf(**"("**) + 1), instructions[x].length());  
 }  
 **for** (String instruction : instructions) {  
 **boolean** pass;  
 String[] decider = instruction.split(**","**);  
 String jobNumber = decider[1].substring(5, decider[1].length());  
 **if** (decider[0].compareTo(**"load"**) == 0) {  
 pass = *checkValidity*(Integer.*parseInt*(jobNumber), **"load"**, **"first"**);  
 String jobSize = decider[2].substring(1, decider[2].length() - 1);  
 **if** (pass) {  
 *firstFit*(Integer.*parseInt*(jobNumber), Integer.*parseInt*(jobSize));  
 } **else** System.***out***.println(**"Job#"** + Integer.*parseInt*(jobNumber) + **" is already in the First Fit Memory."**);  
 pass = *checkValidity*(Integer.*parseInt*(jobNumber), **"load"**, **"best"**);  
 **if** (pass) {  
 *bestFit*(Integer.*parseInt*(jobNumber), Integer.*parseInt*(jobSize));  
 } **else** System.***out***.println(**"Job#"** + Integer.*parseInt*(jobNumber) + **" is already in the Best Fit Memory."**);  
 pass = *checkValidity*(Integer.*parseInt*(jobNumber), **"load"**, **"worst"**);  
 **if** (pass) {  
 *worstFit*(Integer.*parseInt*(jobNumber), Integer.*parseInt*(jobSize));  
 } **else** System.***out***.println(**"Job#"** + Integer.*parseInt*(jobNumber) + **" is already in the Worst Fit Memory."**);  
 pass = *checkValidity*(Integer.*parseInt*(jobNumber), **"load"**, **"previous"**);  
 **if** (pass) {  
 *previouslyLoadedJobs*.add(Integer.*parseInt*(jobNumber));  
 *previouslyLoadedJobs*.add(Integer.*parseInt*(jobSize));  
 }  
  
 }  
 **if** (decider[0].compareTo(**"remove"**) == 0) {  
 pass = *checkValidity*(Integer.*parseInt*(jobNumber), **"remove"**, **null**);  
 **if** (!pass) {  
 System.***out***.println(**"Job#"** + Integer.*parseInt*(jobNumber) + **" is not contained in all 3 lists,"**);  
 System.***out***.println(**"and thus will only be removed from containing list(s)."**);  
 }  
 *removeJob*(Integer.*parseInt*(jobNumber));  
 }  
 **if** (decider[0].compareTo(**"reload"**) == 0) {  
 pass = *checkValidity*(Integer.*parseInt*(jobNumber), **"load"**, **"previous"**);  
 **if** (!pass) {  
 *reloadJob*(Integer.*parseInt*(jobNumber));  
 } **else** {  
 System.***out***.println(**"Job#"** + Integer.*parseInt*(jobNumber) + **" has not been loaded,"**);  
 System.***out***.println(**"and thus cannot be reloaded into the Memory."**);  
 }  
 }  
 **if** (userOption == 1) {  
 System.***out***.println(**"\n\nOutput after performing action "**+decider[0]+**" on Job #"**+jobNumber+**".\n"**);  
 *printAllocationTables*();  
 *printSnapshots*();  
 }  
 }  
 **if**(userOption == 2) {  
 *printAllocationTables*();  
 *printSnapshots*();  
 }  
 *recallOrExit*(userOption);  
  
 } **catch**(IndexOutOfBoundsException e){  
 System.***out***.println(**"Your instructions were not entered in the correct format."**);  
 *recallOrExit*(userOption);  
 }  
 }  
 */\*  
 reloadJob function takes the jobNumber from the instructionHandler and searches through the list of maintained  
 and corresponding job sizes associated with each previously loaded job. If it finds the job number in question,  
 the job is loaded back into the memory.  
 \*/* **private static void** reloadJob(**int** jobNumber){  
 **int** size = *previouslyLoadedJobs*.size();  
 **for** (**int** i = 0; i < size; i++) {  
 **if** (*previouslyLoadedJobs*.get(i).compareTo(jobNumber) == 0) {  
 **int** jobSize = *previouslyLoadedJobs*.get(i + 1);  
 **int** jobLocation = *previouslyLoadedJobs*.get(i + 2);  
 *firstFit*(jobNumber, jobSize);  
 *bestFit*(jobNumber, jobSize);  
 *worstFit*(jobNumber, jobSize);  
 }  
 i++;  
 i++;  
 }  
 }  
  
  
 */\*  
 The removeJob function iterates through each of the memory lists in search for the job Number in question.  
 Once it is found, it removes the entire 3-tuple from the list.  
 \*/* **private static void** removeJob(**int** jobNumber){  
 **int** size = *firstFitList*.size();  
 **boolean** firstRemoved = **false**;  
 **boolean** bestRemoved = **false**;  
 **boolean** worstRemoved = **false**;  
 **for**(**int** i=0; i<size; i++){  
 **if**(!firstRemoved) {  
 **if** (*firstFitList*.get(i).compareTo(jobNumber) == 0) {  
 *firstFitList*.remove(i);  
 *firstFitList*.remove(i);  
 *firstFitList*.remove(i);  
 firstRemoved = **true**;  
 }  
 }  
 i++;  
 i++;  
 }  
  
 size = *bestFitList*.size();  
 **for**(**int** i=0; i<=size; i++){  
 **if**(!bestRemoved) {  
 **if** (*bestFitList*.get(i).compareTo(jobNumber) == 0) {  
 *bestFitList*.remove(i);  
 *bestFitList*.remove(i);  
 *bestFitList*.remove(i);  
 bestRemoved = **true**;  
 }  
 }  
 i++;  
 i++;  
 }  
 size = *worstFitList*.size();  
 **for**(**int** i=0; i<=size; i++){  
 **if**(!worstRemoved) {  
 **if** (*worstFitList*.get(i).compareTo(jobNumber) == 0) {  
 *worstFitList*.remove(i);  
 *worstFitList*.remove(i);  
 *worstFitList*.remove(i);  
 worstRemoved = **true**;  
 }  
 }  
 i++;  
 i++;  
 }  
  
 }  
 */\*  
 First Fit Memory Allocation follows the logic of allocating any given job with size n in the first partition  
 to be found with size>=n. Thus, this function works by taking a job size and its number, iterating through the memory list,  
 and checking to see if there is a space available between any 2 subsequent instructions. If there is, then if that space is greater than or  
 equal to the size of the job, then the job number is inputted at the location of the free space.  
 \*/* **private static void** firstFit(**int** jobNumber, **int** jobSize){  
 *//linkedlist follows pattern Job #, Job Size, Location* **int** size = *firstFitList*.size();  
 **boolean** added = **false**;  
 **for**(**int** i=2; i<size-2; i++) {  
 **if** ((*firstFitList*.get(i + 3) - (*firstFitList*.get(i) + *firstFitList*.get(i - 1))) >= jobSize && !added){  
 *firstFitList*.add(i + 1, (*firstFitList*.get(i) + *firstFitList*.get(i - 1)));  
 *firstFitList*.add(i + 1, jobSize);  
 *firstFitList*.add(i + 1, jobNumber);  
 added = **true**;  
 }  
 i++;  
 i++;  
 }  
 **if**(!added) {  
 **int** lastLocation = *firstFitList*.peekLast();  
 size = *firstFitList*.size();  
 **int** lastJobSize = *firstFitList*.get(size - 2);  
 **if**((lastLocation + lastJobSize + jobSize) < *MemorySize*) {  
 *firstFitList*.add(jobNumber);  
 *firstFitList*.add(jobSize);  
 *firstFitList*.add(lastLocation + lastJobSize);  
 }  
 **else**{  
 System.***out***.println(**"There is not enough space in the Memory for Job #"**+jobNumber+**" using First Fit, consider compaction."**);  
 }  
 }  
 }  
 */\*  
 Best Fit Memory Allocation follows the logic of allocating any given job with size n in the smallest partition of size>=n.  
 Thus, this function works by taking a job size and its number, iterating through the memory list, and checking to  
 see if there is a space available between any 2 subsequent instructions. The partition with the smallest found size to date  
 has its index maintained in a temporary variable until the end of the iteration. Once it has completed, the job is assigned at  
 the index of the smallest partition.  
 \*/* **private static void** bestFit(**int** jobNumber, **int** jobSize){  
 *//linkedlist follows pattern Job #, Job Size, Location* **int** size = *bestFitList*.size();  
 **int** smallestFit = *MemorySize*;  
 **int** indexOfSmallest = 0;  
 **boolean** added = **false**;  
 **for**(**int** i=2; i<size-2; i++) {  
 **if** ((*bestFitList*.get(i + 3) - (*bestFitList*.get(i) + *bestFitList*.get(i - 1))) >= jobSize){  
 **int** currentFit = (*bestFitList*.get(i + 3) - (*bestFitList*.get(i) + *bestFitList*.get(i - 1)));  
 **if**(currentFit<smallestFit){  
 smallestFit = currentFit;  
 indexOfSmallest = i;  
 }  
 }  
 i++;  
 i++;  
 }  
 **if**(indexOfSmallest != 0){  
 *bestFitList*.add(indexOfSmallest + 1, (*bestFitList*.get(indexOfSmallest) + *bestFitList*.get(indexOfSmallest - 1)));  
 *bestFitList*.add(indexOfSmallest + 1, jobSize);  
 *bestFitList*.add(indexOfSmallest + 1, jobNumber);  
 added = **true**;  
 }  
 **if**(!added) {  
 **int** lastLocation = *bestFitList*.peekLast();  
 size = *bestFitList*.size();  
 **int** lastJobSize = *bestFitList*.get(size - 2);  
 **if**((lastLocation + lastJobSize + jobSize) < *MemorySize*) {  
 *bestFitList*.add(jobNumber);  
 *bestFitList*.add(jobSize);  
 *bestFitList*.add(lastLocation + lastJobSize);  
 }  
 **else**{  
 System.***out***.println(**"There is not enough space in the Memory for Job #"**+jobNumber+**", using Best Fit, consider compaction."**);  
 }  
 }  
 }  
 */\*  
 Worst Fit Memory Allocation follows the logic of allocating any given job with size n in the largest partition of size>=n.  
 Thus, this function works by taking a job size and its number, iterating through the memory list, and checking to  
 see if there is a space available between any 2 subsequent instructions. The partition with the largest found size to date  
 has its index maintained in a temporary variable until the end of the iteration. Once it has completed, the job is assigned at  
 the index of the largest partition.  
 \*/* **private static void** worstFit(**int** jobNumber, **int** jobSize){  
 *//linkedlist follows pattern Job #, Job Size, Location* **int** size = *worstFitList*.size();  
 **int** largestFit = 0;  
 **int** indexOfLargest = 0;  
 **boolean** added = **false**;  
 **for**(**int** i=2; i<size-2; i++) {  
 **if** ((*worstFitList*.get(i + 3) - (*worstFitList*.get(i) + *worstFitList*.get(i - 1))) >= jobSize){  
 **int** currentFit = (*bestFitList*.get(i + 3) - (*bestFitList*.get(i) + *bestFitList*.get(i - 1)));  
 **if**(currentFit>largestFit){  
 largestFit = currentFit;  
 indexOfLargest = i;  
 }  
 }  
 i++;  
 i++;  
 }  
 **if**(indexOfLargest != 0){  
 *worstFitList*.add(indexOfLargest + 1, (*worstFitList*.get(indexOfLargest) + *worstFitList*.get(indexOfLargest - 1)));  
 *worstFitList*.add(indexOfLargest + 1, jobSize);  
 *worstFitList*.add(indexOfLargest + 1, jobNumber);  
 added = **true**;  
 }  
 **if**(!added) {  
 **int** lastLocation = *worstFitList*.peekLast();  
 size = *worstFitList*.size();  
 **int** lastJobSize = *worstFitList*.get(size - 2);  
 **if**((lastLocation + lastJobSize + jobSize) < *MemorySize*) {  
 *worstFitList*.add(jobNumber);  
 *worstFitList*.add(jobSize);  
 *worstFitList*.add(lastLocation + lastJobSize);  
 }  
 **else**{  
 System.***out***.println(**"There is not enough space in the Memory for Job #"**+jobNumber+**", using Worst Fit, consider compaction."**);  
 }  
 }  
 }  
 */\*  
 printAllocationTables function prints the allocation table for each instruction (or each sequence of instructions, depending  
 on user inputted decision) in the format depicted in CS3725 lectures. E.g. the function iterates through the memory and prints  
 all of its containing data in a table format, with columns Size, Location, State and Job. Each row contains 1 partition.  
 This is done for all 3 of the memory allocation algorithms.  
 \*/* **private static void** printAllocationTables() {  
  
 **int** size = *bestFitList*.size();  
 System.***out***.println(**"\n\nBest Fit Memory Allocation Table:"**);  
 System.***out***.printf (**"%-15s %-15s %-15s %-15s"**, **"Size"**, **"Location"**, **"State"**, **"Job"**);  
 System.***out***.println();  
 **for** (**int** i = 3; i < size; i++) {  
 *//linkedlist follows pattern Job #, Job Size, Location* System.***out***.printf(**"%-15s %-15s %-15s %-15s"**, *bestFitList*.get(i+1).toString(), *bestFitList*.get(i+2).toString(),**"Allocated"**, *bestFitList*.get(i).toString());  
 System.***out***.println();  
 **try** {  
 **if** (!(*bestFitList*.get(i + 5) - (*bestFitList*.get(i + 2) + *bestFitList*.get(i + 1)) == 0)) {  
 **int** emptyPartitionSize = (*bestFitList*.get(i + 5) - (*bestFitList*.get(i + 2) + *bestFitList*.get(i + 1)));  
 **int** emptyPartitionLoc = (*bestFitList*.get(i + 2) + *bestFitList*.get(i + 1));  
 System.***out***.printf(**"%-15s %-15s %-15s %-15s"**, String.*valueOf*(emptyPartitionSize), String.*valueOf*(emptyPartitionLoc), **"Avail"**, **" - "**);  
 System.***out***.println();  
 }  
 }**catch**(IndexOutOfBoundsException ignored){}  
 i++; i++;  
 }  
 **int** remainingSpace = *MemorySize*-(*bestFitList*.get(size-1)+*bestFitList*.get(size-2));  
 **int** lastLocation = (*bestFitList*.get(size-1)+*bestFitList*.get(size-2));  
 System.***out***.printf(**"%-15s %-15s %-15s %-15s"**, String.*valueOf*(remainingSpace), String.*valueOf*(lastLocation),**"Avail"**, **" - "**);  
 System.***out***.println();  
  
 size = *worstFitList*.size();  
 System.***out***.println(**"\n\nWorst Fit Memory Allocation Table:"**);  
 System.***out***.printf (**"%-15s %-15s %-15s %-15s"**, **"Size"**, **"Location"**, **"State"**, **"Job"**);  
 System.***out***.println();  
 **for** (**int** i = 3; i < size; i++) {  
 *//linkedlist follows pattern Job #, Job Size, Location* System.***out***.printf(**"%-15s %-15s %-15s %-15s"**, *worstFitList*.get(i+1).toString(), *worstFitList*.get(i+2).toString(),**"Allocated"**, *worstFitList*.get(i).toString());  
 System.***out***.println();  
 **try** {  
 **if** (!(*worstFitList*.get(i + 5) - (*worstFitList*.get(i + 2) + *worstFitList*.get(i + 1)) == 0)) {  
 **int** emptyPartitionSize = (*worstFitList*.get(i + 5) - (*worstFitList*.get(i + 2) + *worstFitList*.get(i + 1)));  
 **int** emptyPartitionLoc = (*worstFitList*.get(i + 2) + *worstFitList*.get(i + 1));  
 System.***out***.printf(**"%-15s %-15s %-15s %-15s"**, String.*valueOf*(emptyPartitionSize), String.*valueOf*(emptyPartitionLoc), **"Avail"**, **" - "**);  
 System.***out***.println();  
 }  
 }**catch**(IndexOutOfBoundsException ignored){}  
 i++; i++;  
 }  
 remainingSpace = *MemorySize*-(*worstFitList*.get(size-1)+*worstFitList*.get(size-2));  
 lastLocation = (*worstFitList*.get(size-1)+*worstFitList*.get(size-2));  
 System.***out***.printf(**"%-15s %-15s %-15s %-15s"**, String.*valueOf*(remainingSpace), String.*valueOf*(lastLocation),**"Avail"**, **" - "**);  
 System.***out***.println();  
  
 size = *firstFitList*.size();  
 System.***out***.println(**"\n\nFirst Fit Memory Allocation Table:"**);  
 System.***out***.printf (**"%-15s %-15s %-15s %-15s"**, **"Size"**, **"Location"**, **"State"**, **"Job"**);  
 System.***out***.println();  
 **for** (**int** i = 3; i < size; i++) {  
 *//linkedlist follows pattern Job #, Job Size, Location* System.***out***.printf(**"%-15s %-15s %-15s %-15s"**, *firstFitList*.get(i+1).toString(), *firstFitList*.get(i+2).toString(),**"Allocated"**, *firstFitList*.get(i).toString());  
 System.***out***.println();  
 **try** {  
 **if** (!(*firstFitList*.get(i + 5) - (*firstFitList*.get(i + 2) + *firstFitList*.get(i + 1)) == 0)) {  
 **int** emptyPartitionSize = (*firstFitList*.get(i + 5) - (*firstFitList*.get(i + 2) + *firstFitList*.get(i + 1)));  
 **int** emptyPartitionLoc = (*bestFitList*.get(i + 2) + *bestFitList*.get(i + 1));  
 System.***out***.printf(**"%-15s %-15s %-15s %-15s"**, String.*valueOf*(emptyPartitionSize), String.*valueOf*(emptyPartitionLoc), **"Avail"**, **" - "**);  
 System.***out***.println();  
 }  
 }**catch**(IndexOutOfBoundsException ignored){}  
 i++; i++;  
 }  
 remainingSpace = *MemorySize*-(*firstFitList*.get(size-1)+*firstFitList*.get(size-2));  
 lastLocation = (*firstFitList*.get(size-1)+*firstFitList*.get(size-2));  
 System.***out***.printf(**"%-15s %-15s %-15s %-15s"**, String.*valueOf*(remainingSpace), String.*valueOf*(lastLocation),**"Avail"**, **" - "**);  
 System.***out***.println();  
 }  
 */\*  
 printSnapshots function prints a snapshot of the Memory for each instruction (or each sequence of instructions, depending  
 on user inputted decision) in the format depicted in CS3725 lectures. E.g. the function iterates through the memory and prints  
 all of its containing data in the format of Starting Location (of each partition) printed on the left side, and the  
 Job that is contained in each partition is written inside of the block. Each block represents 1 partition.  
 This is done for all 3 of the memory allocation algorithms.  
 \*/* **private static void** printSnapshots() {  
  
 **int** size = *bestFitList*.size();  
 System.***out***.println(**"\n\nBest Fit Memory Snapshot:"**);  
 System.***out***.printf (**"%-6s %-15s"**,**"0"**, **"-----------"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-9s %-12s"**,**" "**, **"OS"**);  
 System.***out***.println();  
 **for** (**int** i = 6; i < size; i++) {  
 *//linkedlist follows pattern Job #, Job Size, Location* System.***out***.printf (**"%-6s %-15s"**,*bestFitList*.get(i+2).toString(), **"-----------"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-9s %-12s"**, **" "**, **"Job "**+*bestFitList*.get(i).toString());  
 System.***out***.println();  
 *// System.out.printf("%-10s %-20s", bestFitList.get(i+2).toString(), bestFitList.get(i).toString());* **try** {  
 **if** (!(*bestFitList*.get(i + 5) - (*bestFitList*.get(i + 2) + *bestFitList*.get(i + 1)) == 0)) {  
 **int** emptyPartitionLoc = (*bestFitList*.get(i + 2) + *bestFitList*.get(i + 1));  
 System.***out***.printf (**"%-6s %-15s"**,emptyPartitionLoc, **"-----------"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-9s %-12s"**, **" "**, **"empty"**);  
 System.***out***.println();  
 }  
 }**catch**(IndexOutOfBoundsException ignored){}  
 i++; i++;  
 }  
 **int** lastLocation = (*bestFitList*.get(size-1)+*bestFitList*.get(size-2));  
 System.***out***.printf (**"%-6s %-15s"**,lastLocation, **"-----------"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-9s %-12s"**, **" "**, **"empty"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-6s %-15s"**,*MemorySize*, **"-----------"**);  
  
 size = *worstFitList*.size();  
 System.***out***.println(**"\n\nWorst Fit Memory Snapshot:"**);  
 System.***out***.printf (**"%-6s %-15s"**,**"0"**, **"-----------"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-9s %-12s"**,**" "**, **"OS"**);  
 System.***out***.println();  
 **for** (**int** i = 6; i < size; i++) {  
 *//linkedlist follows pattern Job #, Job Size, Location* System.***out***.printf (**"%-6s %-15s"**,*worstFitList*.get(i+2).toString(), **"-----------"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-9s %-12s"**, **" "**, **"Job "**+*worstFitList*.get(i).toString());  
 System.***out***.println();  
 *// System.out.printf("%-10s %-20s", bestFitList.get(i+2).toString(), bestFitList.get(i).toString());* **try** {  
 **if** (!(*worstFitList*.get(i + 5) - (*worstFitList*.get(i + 2) + *worstFitList*.get(i + 1)) == 0)) {  
 **int** emptyPartitionLoc = (*worstFitList*.get(i + 2) + *worstFitList*.get(i + 1));  
 System.***out***.printf (**"%-6s %-15s"**,emptyPartitionLoc, **"-----------"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-9s %-12s"**, **" "**, **"empty"**);  
 System.***out***.println();  
 }  
 }**catch**(IndexOutOfBoundsException ignored){}  
 i++; i++;  
 }  
 lastLocation = (*worstFitList*.get(size-1)+*worstFitList*.get(size-2));  
 System.***out***.printf (**"%-6s %-15s"**,lastLocation, **"-----------"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-9s %-12s"**, **" "**, **"empty"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-6s %-15s"**,*MemorySize*, **"-----------"**);  
  
 size = *firstFitList*.size();  
 System.***out***.println(**"\n\nFirst Fit Memory Snapshot:"**);  
 System.***out***.printf (**"%-6s %-15s"**,**"0"**, **"-----------"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-9s %-12s"**,**" "**, **"OS"**);  
 System.***out***.println();  
 **for** (**int** i = 6; i < size; i++) {  
 *//linkedlist follows pattern Job #, Job Size, Location* System.***out***.printf (**"%-6s %-15s"**,*firstFitList*.get(i+2).toString(), **"-----------"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-9s %-12s"**, **" "**, **"Job "**+*firstFitList*.get(i).toString());  
 System.***out***.println();  
 *// System.out.printf("%-10s %-20s", bestFitList.get(i+2).toString(), bestFitList.get(i).toString());* **try** {  
 **if** (!(*firstFitList*.get(i + 5) - (*firstFitList*.get(i + 2) + *firstFitList*.get(i + 1)) == 0)) {  
 **int** emptyPartitionLoc = (*firstFitList*.get(i + 2) + *firstFitList*.get(i + 1));  
 System.***out***.printf (**"%-6s %-15s"**,emptyPartitionLoc, **"-----------"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-9s %-12s"**, **" "**, **"empty"**);  
 System.***out***.println();  
 }  
 }**catch**(IndexOutOfBoundsException ignored){}  
 i++; i++;  
 }  
 lastLocation = (*firstFitList*.get(size-1)+*firstFitList*.get(size-2));  
 System.***out***.printf (**"%-6s %-15s"**,lastLocation, **"-----------"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-9s %-12s"**, **" "**, **"empty"**);  
 System.***out***.println();  
 System.***out***.printf (**"%-6s %-15s"**,*MemorySize*, **"-----------"**);  
  
 }  
 */\*  
 The checkValidity function takes 3 arguments, Job #, sender and list. The sender is the instruction opcode from the  
 instructionHandler function that is being executed and requires the validity check. The list is which of the 3  
 Memory lists to reference.  
 For the load instruction, each list is independently checked through a separate call of checkValidity, as  
 the job can be loaded into 1 list even if it is already contained in the other 2, etc.  
 For remove, the job is searched for & removed in applicable in each list. 3 boolean variables are maintained,  
 representing whether or not the job was found in each list. If it is not found in all 3, this is made known  
 to the user.  
 For reload, the load sender is also called, with the list set as previous (to check the list of previously loaded jobs),  
 but the boolean is simply inverted in the instructionHandler function.  
 \*/* **private static boolean** checkValidity(**int** jobNumber, String sender, String list){  
 **if**(sender.compareTo(**"load"**) == 0 && list.compareTo(**"first"**) == 0){  
 **for**(**int** i=0; i<*firstFitList*.size(); i++){  
 **if** (*firstFitList*.get(i).compareTo(jobNumber) == 0) {  
 **return false**;  
 }  
 i++; i++;  
 }  
 }  
 **if**(sender.compareTo(**"load"**) == 0 && list.compareTo(**"best"**) == 0){  
 **for**(**int** i=0; i<*bestFitList*.size(); i++){  
 **if** (*bestFitList*.get(i).compareTo(jobNumber) == 0) {  
 **return false**;  
 }  
 i++; i++;  
 }  
 }  
 **if**(sender.compareTo(**"load"**) == 0 && list.compareTo(**"worst"**) == 0){  
 **for**(**int** i=0; i<*worstFitList*.size(); i++){  
 **if** (*worstFitList*.get(i).compareTo(jobNumber) == 0) {  
 **return false**;  
 }  
 i++; i++;  
 }  
 }  
 **if**(sender.compareTo(**"load"**) == 0 && list.compareTo(**"previous"**) == 0){  
 **for**(**int** i=0; i<*previouslyLoadedJobs*.size(); i++){  
 **if**(*previouslyLoadedJobs*.get(i).compareTo(jobNumber) == 0){  
 **return false**;  
 }  
 i++;  
 }  
 }  
 **if**(sender.compareTo(**"remove"**)==0){  
 **boolean** worstList = **false**;  
 **boolean** firstList = **false**;  
 **boolean** bestList = **false**;  
 **for**(**int** i=0; i<*worstFitList*.size(); i++){  
 **if** (*worstFitList*.get(i).compareTo(jobNumber) == 0) {  
 worstList = **true**;  
 }  
 i++; i++;  
 }  
 **for**(**int** i=0; i<*bestFitList*.size(); i++){  
 **if** (*bestFitList*.get(i).compareTo(jobNumber) == 0) {  
 bestList = **true**;  
 }  
 i++; i++;  
 }  
 **for**(**int** i=0; i<*firstFitList*.size(); i++){  
 **if** (*firstFitList*.get(i).compareTo(jobNumber) == 0) {  
 firstList = **true**;  
 }  
 i++; i++;  
 }  
 **if**(!worstList || !bestList || !firstList){  
 **return false**;  
 }  
 }  
 **return true**;  
 }  
 */\*  
 The recallOrExit function simply prompts the user to either continue running the program, or exit. This just serves as a  
 simple way to loop the program until the user decides to exit on their own, allowing them to input as many  
 instructions as they desire.  
 \*/* **private static void** recallOrExit(**int** userOption){  
 System.***out***.println(**"\n\nWould you like to alter the Memory further? Y/N"**);  
 Scanner in = **new** Scanner(System.***in***).useDelimiter(**"\\n"**);  
 String input = in.next();  
 **if**(input.compareTo(**"N"**) == 0 || input.compareTo(**"n"**) == 0){  
 System.*exit*(0);  
 }  
 **else**{  
 System.***out***.println(**"Instruction formats:"**);  
 System.***out***.println(**"To load a new job, type: (load, job #, [jobSize]k)"**);  
 System.***out***.println(**"To remove a job, type: (remove, job #)"**);  
 System.***out***.println(**"To reload a job, type: (reload, job #)"**);  
 System.***out***.println(**"At this time you may input your sequence in the following format (please note brackets):"**);  
 System.***out***.println(**"(instruction 1), (instruction 2), (instruction 3), ... "**);  
 String instructionList = in.next();  
 *//* System.***out***.println(**"List: "**+instructionList);  
 *instructionHandler*(instructionList, userOption);  
 }  
 }  
}

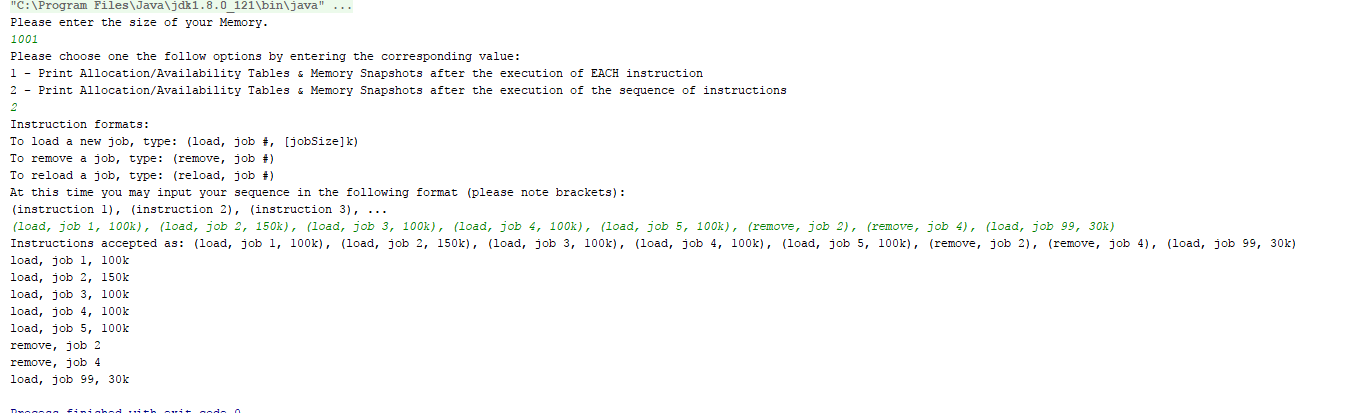
**3.2 Run-time Output**





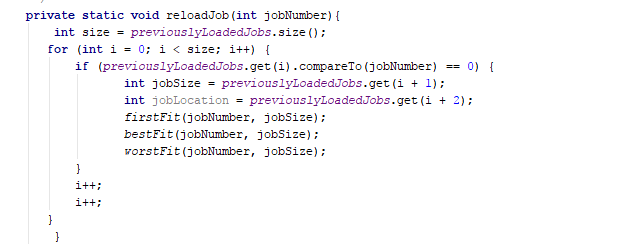
**3.3 instructionHandler() Function Test**

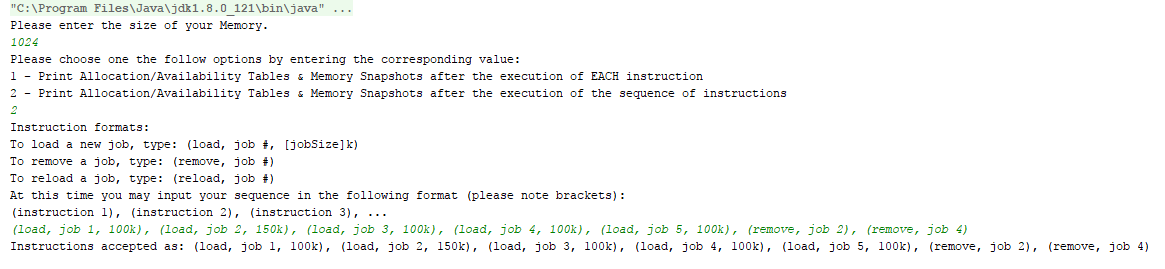
Using the following segment of code, the instructionHandler()function was tested.

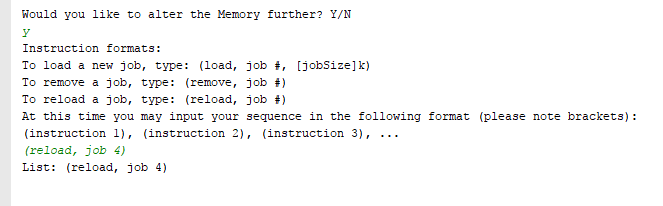
**Output:**

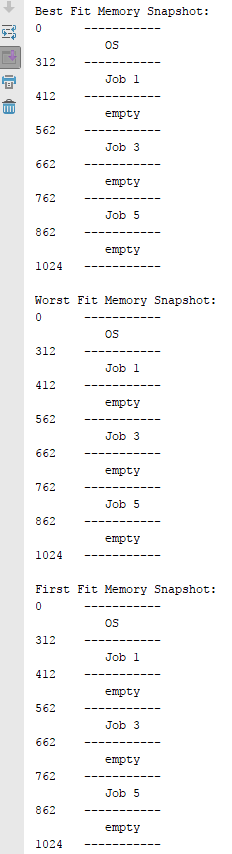
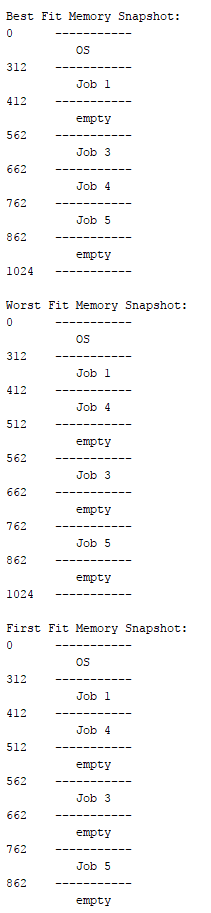
**3.4 removeJob() and reloadJob Function Tests**

Using the follow segment of code, the reloadJob() and removeJob() functions were tested.



**Output (using tables to show):**



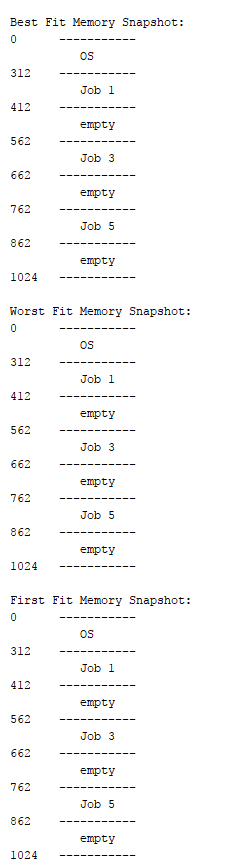


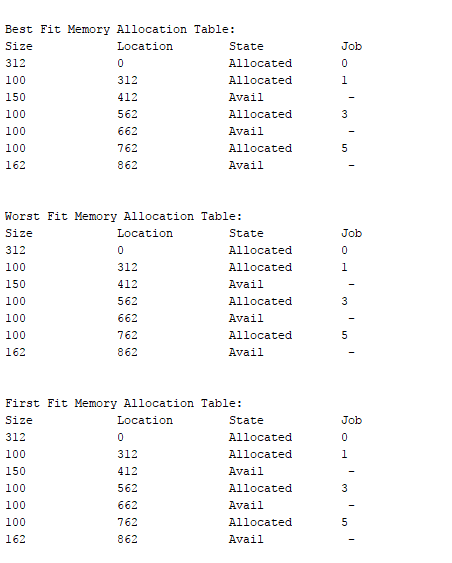
**3.5 FirstFit, WorstFit, BestFit Function Tests**

As all 3 should have very predictable results, they were tested simultaneously, using the follow code:

Note, this also tests the PrintAllocationTable and PrintSnapshot functions.

**private static void** firstFit(**int** jobNumber, **int** jobSize){  
 *//linkedlist follows pattern Job #, Job Size, Location* **int** size = *firstFitList*.size();  
 **boolean** added = **false**;  
 **for**(**int** i=2; i<size-2; i++) {  
 **if** ((*firstFitList*.get(i + 3) - (*firstFitList*.get(i) + *firstFitList*.get(i - 1))) >= jobSize && !added){  
 *firstFitList*.add(i + 1, (*firstFitList*.get(i) + *firstFitList*.get(i - 1)));  
 *firstFitList*.add(i + 1, jobSize);  
 *firstFitList*.add(i + 1, jobNumber);  
 added = **true**;  
 }  
 i++;  
 i++;  
 }  
 **if**(!added) {  
 **int** lastLocation = *firstFitList*.peekLast();  
 size = *firstFitList*.size();  
 **int** lastJobSize = *firstFitList*.get(size - 2);  
 **if**((lastLocation + lastJobSize + jobSize) < *MemorySize*) {  
 *firstFitList*.add(jobNumber);  
 *firstFitList*.add(jobSize);  
 *firstFitList*.add(lastLocation + lastJobSize);  
 }  
 **else**{  
 System.***out***.println(**"There is not enough space in the Memory for Job #"**+jobNumber+**" using First Fit, consider compaction."**);  
 }  
 }  
 }  
 */\*  
 Best Fit Memory Allocation follows the logic of allocating any given job with size n in the smallest partition of size>=n.  
 Thus, this function works by taking a job size and its number, iterating through the memory list, and checking to  
 see if there is a space available between any 2 subsequent instructions. The partition with the smallest found size to date  
 has its index maintained in a temporary variable until the end of the iteration. Once it has completed, the job is assigned at  
 the index of the smallest partition.  
 \*/* **private static void** bestFit(**int** jobNumber, **int** jobSize){  
 *//linkedlist follows pattern Job #, Job Size, Location* **int** size = *bestFitList*.size();  
 **int** smallestFit = *MemorySize*;  
 **int** indexOfSmallest = 0;  
 **boolean** added = **false**;  
 **for**(**int** i=2; i<size-2; i++) {  
 **if** ((*bestFitList*.get(i + 3) - (*bestFitList*.get(i) + *bestFitList*.get(i - 1))) >= jobSize){  
 **int** currentFit = (*bestFitList*.get(i + 3) - (*bestFitList*.get(i) + *bestFitList*.get(i - 1)));  
 **if**(currentFit<smallestFit){  
 smallestFit = currentFit;  
 indexOfSmallest = i;  
 }  
 }  
 i++;  
 i++;  
 }  
 **if**(indexOfSmallest != 0){  
 *bestFitList*.add(indexOfSmallest + 1, (*bestFitList*.get(indexOfSmallest) + *bestFitList*.get(indexOfSmallest - 1)));  
 *bestFitList*.add(indexOfSmallest + 1, jobSize);  
 *bestFitList*.add(indexOfSmallest + 1, jobNumber);  
 added = **true**;  
 }  
 **if**(!added) {  
 **int** lastLocation = *bestFitList*.peekLast();  
 size = *bestFitList*.size();  
 **int** lastJobSize = *bestFitList*.get(size - 2);  
 **if**((lastLocation + lastJobSize + jobSize) < *MemorySize*) {  
 *bestFitList*.add(jobNumber);  
 *bestFitList*.add(jobSize);  
 *bestFitList*.add(lastLocation + lastJobSize);  
 }  
 **else**{  
 System.***out***.println(**"There is not enough space in the Memory for Job #"**+jobNumber+**", using Best Fit, consider compaction."**);  
 *Worst Fit Memory Allocation follows the logic of allocating any given job with size n in the largest partition of size>=n.  
Thus, this function works by taking a job size and its number, iterating through the memory list, and checking to  
see if there is a space available between any 2 subsequent instructions. The partition with the largest found size to date  
has its index maintained in a temporary variable until the end of the iteration. Once it has completed, the job is assigned at  
the index of the largest partition.  
 \*/* **private static void** worstFit(**int** jobNumber, **int** jobSize){  
 *//linkedlist follows pattern Job #, Job Size, Location* **int** size = *worstFitList*.size();  
 **int** largestFit = 0;  
 **int** indexOfLargest = 0;  
 **boolean** added = **false**;  
 **for**(**int** i=2; i<size-2; i++) {  
 **if** ((*worstFitList*.get(i + 3) - (*worstFitList*.get(i) + *worstFitList*.get(i - 1))) >= jobSize){  
 **int** currentFit = (*bestFitList*.get(i + 3) - (*bestFitList*.get(i) + *bestFitList*.get(i - 1)));  
 **if**(currentFit>largestFit){  
 largestFit = currentFit;  
 indexOfLargest = i;  
 }  
 }  
 i++;  
 i++;  
 }  
 **if**(indexOfLargest != 0){  
 *worstFitList*.add(indexOfLargest + 1, (*worstFitList*.get(indexOfLargest) + *worstFitList*.get(indexOfLargest - 1)));  
 *worstFitList*.add(indexOfLargest + 1, jobSize);  
 *worstFitList*.add(indexOfLargest + 1, jobNumber);  
 added = **true**;  
 }  
 **if**(!added) {  
 **int** lastLocation = *worstFitList*.peekLast();  
 size = *worstFitList*.size();  
 **int** lastJobSize = *worstFitList*.get(size - 2);  
 **if**((lastLocation + lastJobSize + jobSize) < *MemorySize*) {  
 *worstFitList*.add(jobNumber);  
 *worstFitList*.add(jobSize);  
 *worstFitList*.add(lastLocation + lastJobSize);  
 }  
 **else**{  
 System.***out***.println(**"There is not enough space in the Memory for Job #"**+jobNumber+**",**

**Output:**

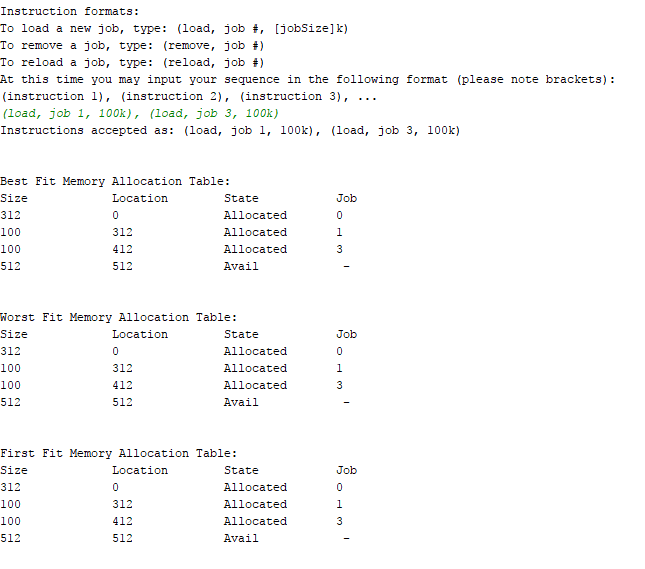


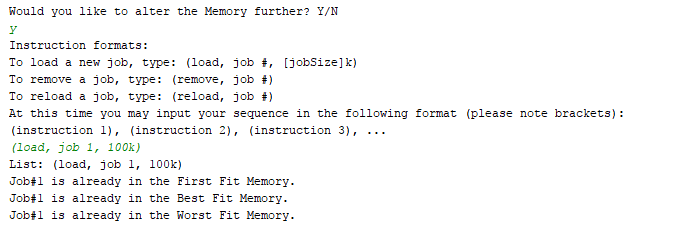
**3.6 checkValidity Function Test**

**Using the following segment of code, the checkValidity()function was tested.**

**private static boolean** checkValidity(**int** jobNumber, String sender, String list){  
 **if**(sender.compareTo(**"load"**) == 0 && list.compareTo(**"first"**) == 0){  
 **for**(**int** i=0; i<*firstFitList*.size(); i++){  
 **if** (*firstFitList*.get(i).compareTo(jobNumber) == 0) {  
 **return false**;  
 }  
 i++; i++;  
 }  
 }  
 **if**(sender.compareTo(**"load"**) == 0 && list.compareTo(**"best"**) == 0){  
 **for**(**int** i=0; i<*bestFitList*.size(); i++){  
 **if** (*bestFitList*.get(i).compareTo(jobNumber) == 0) {  
 **return false**;  
 }  
 i++; i++;  
 }  
 }  
 **if**(sender.compareTo(**"load"**) == 0 && list.compareTo(**"worst"**) == 0){  
 **for**(**int** i=0; i<*worstFitList*.size(); i++){  
 **if** (*worstFitList*.get(i).compareTo(jobNumber) == 0) {  
 **return false**;  
 }  
 i++; i++;  
 }  
 }  
 **if**(sender.compareTo(**"load"**) == 0 && list.compareTo(**"previous"**) == 0){  
 **for**(**int** i=0; i<*previouslyLoadedJobs*.size(); i++){  
 **if**(*previouslyLoadedJobs*.get(i).compareTo(jobNumber) == 0){  
 **return false**;  
 }  
 i++;  
 }  
 **if**(sender.compareTo(**"remove"**)==0){  
 **boolean** worstList = **false**;  
 **boolean** firstList = **false**;  
 **boolean** bestList = **false**;  
 **for**(**int** i=0; i<*worstFitList*.size(); i++){  
 **if** (*worstFitList*.get(i).compareTo(jobNumber) == 0) {  
 worstList = **true**;  
 }  
 i++; i++;  
 }  
 **for**(**int** i=0; i<*bestFitList*.size(); i++){  
 **if** (*bestFitList*.get(i).compareTo(jobNumber) == 0) {  
 bestList = **true**;  
 }  
 i++; i++;  
 }  
 **for**(**int** i=0; i<*firstFitList*.size(); i++){  
 **if** (*firstFitList*.get(i).compareTo(jobNumber) == 0) {  
 firstList = **true**;  
 }  
 i++; i++;  
 }  
 **if**(!worstList || !bestList || !firstList){  
 **return false**;  
 }  
 }  
 **return true**;  
}

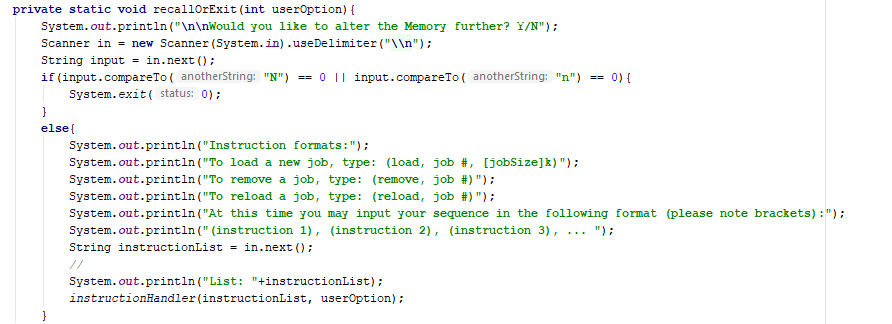
**Output:**



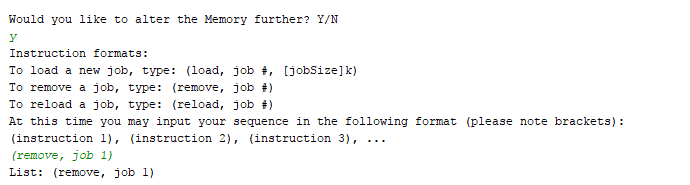


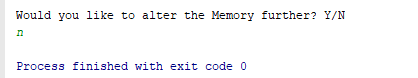
**3.6 recallOrExit Function Test**

**Using the following segment of code, the recallOrExit function was tested.**



**Output:**





**Issues and Limitations**

**4.1 Error Handling**

Given time restraints, certain areas of error handling could be more prevalent. There is the possibility that the user inputs an integer value too large which causes a system overflow, however unlikely.

**4.2 Generality**

As with any program, it can be more general to allow for easier adaptation. Adaptation was taken into account, as seen with the user option for selecting output format, there however could be far more generality in using a more complicated data structure than a List.

Global variables are also a bad practice. Keeping to strict object-oriented principles, they shouldn’t be used as they aren’t objects. A solution to this would have been a separate class for each allocation algorithm which is altered through .get() and .set() functions, which in a larger program is easier to keep track of where the value is being altered from. In this example though, and for its intended use, the global variable serves the purpose, and is of course still static.