Programming Assignment 2:

CPU Process Scheduling System

using Binary Search Trees and Hash Tables

Chelsea Jaculina

May 2, 2018

Professor Wu

CS 146

Table of Contents

System Overview 2

Binary Search Tree Concept Design 3

Hash Table Concept Design 4

Explanations of Functions 5

- **ProcessBST.java** 5
- **HashTable.java** 12
- **Process.java** 19
- **Node.java** 20
- **LinkedList.java** 22
- **MainMethod.java** 24

How to Export the Application 29

- **Zip File** 29
- **How to Unzip File** 35
- **Jar File** 38

How to Run the Application 45

Installation Needed 46

Problems Encountered 47

Lessons Learned 48

System Overview

The system that has been created is based off a CPU simulation software application. This application is entitled to replicate a dynamic keep track of each process and schedule them in accordance to the sorted priority code (indexes) in waiting queue. In all, the system is implemented so that any process with a larger priority code (index) will be able to pre-empt others even if a process has been waiting for a longer amount of time in the CPU. Two separate approaches were taken to implement the CPU process scheduling simulation software application. The two approaches included Binary Search Tree and Hash Tables. The Binary Search Tree approach was for sorting and the Hash Table was for searching.

Binary Search Tree Concept Design

According to Introduction to Algorithms by Thomas Cormen a binary search tree is an organized tree that satisfies the binary search tree property. The binary search tree property states all node values that are in the left subtrees will be smaller and all the node values to the right will be larger. In order to perform the binary search tree approach, a binary search class needed to be implemented. The system included its own individual class called ProcessBST.java. This is because Binary Search Trees has different implementations of printing, inserting, searching, and deleting in comparison to Hash Tables. Binary Search Tree consists of nodes that contain processes. The binary search tree uses both the Node.java and Process.java classes. The class contains a root instance variable of type Node and has a constructor that sets the root to null initially. Each function that has been implemented for a binary search tree will be thoroughly explained in the following section explanations.

Hash Table Concept Design

The concept behind hash tables according to Thomas Cormen is to perform dictionary three operations. The three operations include insert, search, and delete. When hash tables are implemented with a good hash function, there are not many collisions involved. The CPU system implements hash table in its own individual class called HashTable.java since it has the same operations as binary search trees. In this class, a hash table creates an array of linked lists. This class uses the linked list class LinkedList.java as well as the Node.java and Process.java class. The functions of how to insert, search by priority index, and delete will be explained further in the following section.

Explanations of Functions

The CPU process scheduling simulation software application was built with 6 classes. The classes include ProcessBST.java, HashTable.java, Process.java, Node.java, LinkedList.java, and MainMethod.java. Majority of the functions described below were implemented using the pseudo codes from Chapter 10, 11, and 12 in the Intro to Algorithms textbook by Thomas Cormen.

ProcessBST.java

The ProcessBST.java class uses the Node.java class since every node in the binary search tree contains a process.

ProcessBST()

The no-argument constructor of the class. The root instance variable is initialized to null.

getRoot()

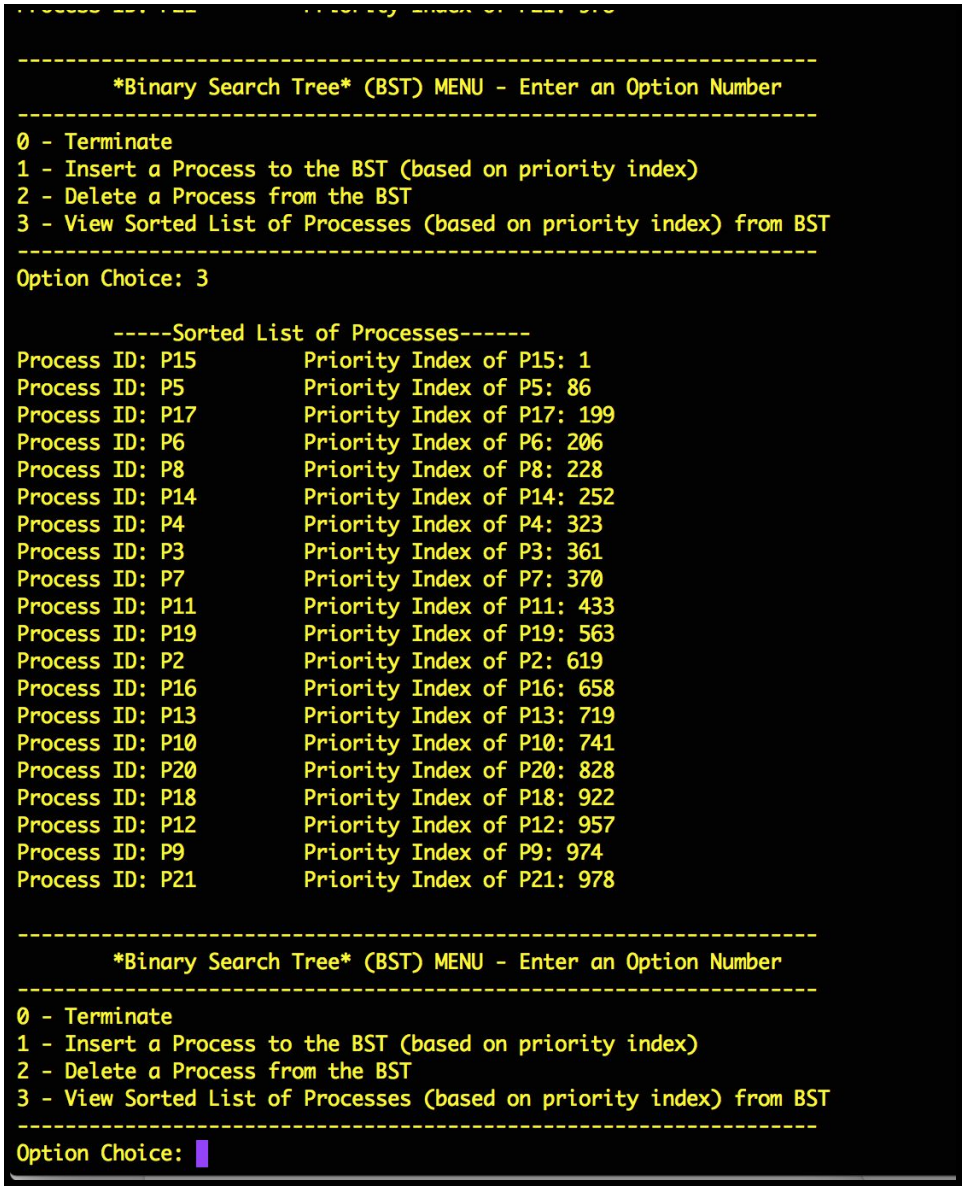
A getter method that gets the root of the binary search tree.

setRoot()

A setter method that sets the root to a another node.

inorderTreeWalk(Node x)

This method is a simple recursive algorithm that allows us to print out all the keys in the binary search tree in sorted order. It prints in the following format: left-node-right.



Output for Option 3 - View Sorted List of Processes (based on priority index) from BST

This output uses inOrderTreeWalk.

iterativeTreeSearch(Node x, Process k)

This method allows the CPU to search for a node with a given key in a binary search tree. It gives a pointer to the root of the binary search tree and a key value. In our instance it will be the priority index code. The function returns a pointer to a node with key value (priority

index code) if one exists; otherwise, it returns null. There are two approached when it comes to implementing tree search, but modern computers tend to use the iterative fashion algorithm because it uses recursion and the version is much more efficient and straightforward. Search for a node inside a binary search tree takes O(lg n) time.

treeMinimum(Node x)

This function returns the smallest node in the binary search tree. This is typically the most left node in the most left subtree.

treeMaximum(Node x)

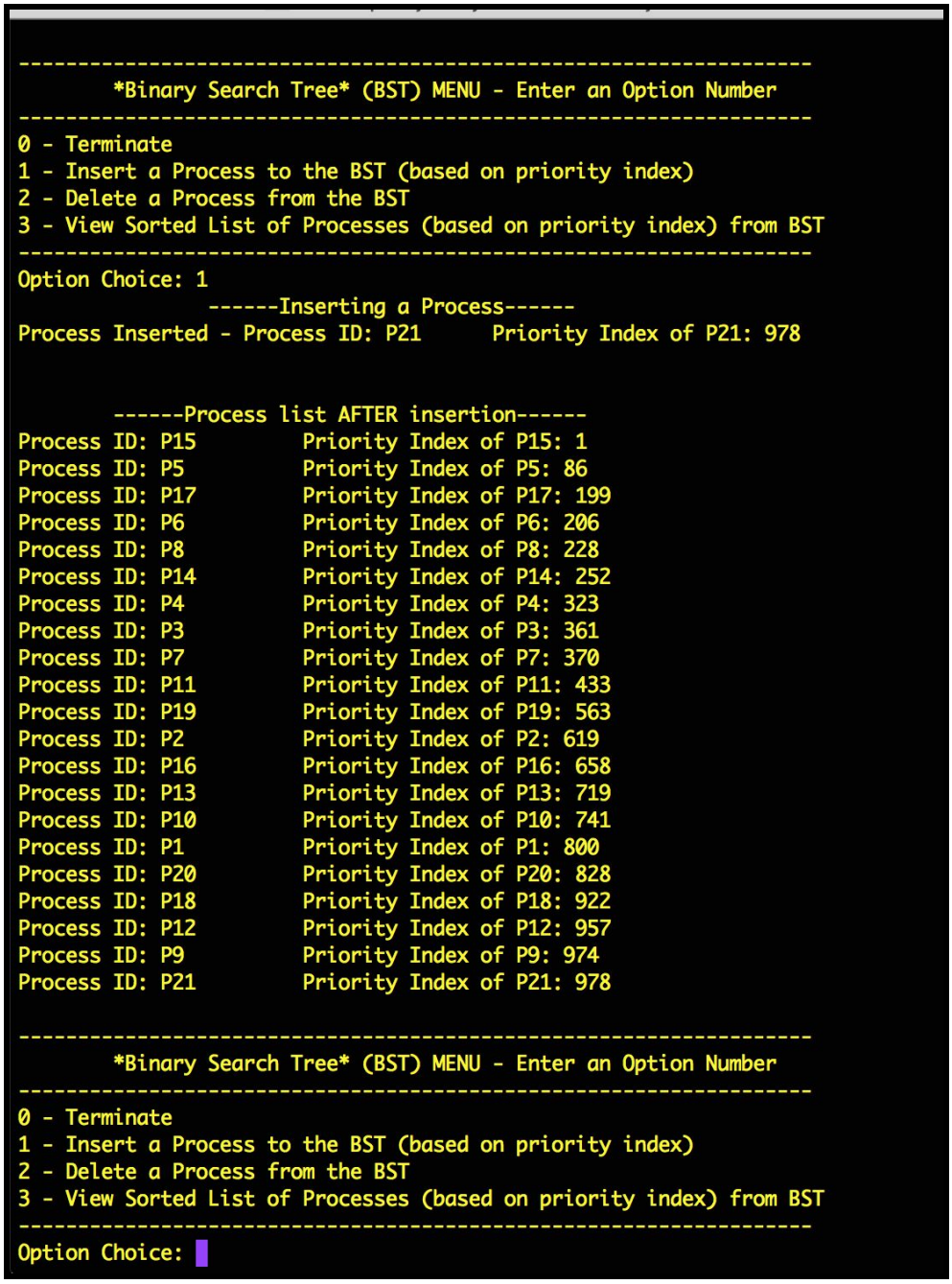
This function returns the largest node in the binary search tree. This is typically the most right node in the most right subtree.

treeSuccessor(Node x)

This function finds the successor of a specific node in the binary search tree. The function passes in a Node ‘x’. It then checks if the right subtree is not null. If the right subtree is non-empty then we call the treeMinimum function in which will give the minimum value within the entire subtree. Then we create a new Node ‘y’ and set it to Node ‘x’s parent. We then check if Node ‘x‘ shallow equals Node ‘y’ subtree. If it does then we enter and set Node ‘x’ to y and set Node ‘y’ to the parent.

treeInsert(ProcessBST T, Node z)

This function inserts a new value ‘v’ into the binary search tree T. It first takes in a binary search tree to insert to and then takes in a Node ‘z’. We create a new Node ‘y’ and set it to null. Then we set Node ‘x’ to the root in the binary search tree. If Node ‘x’ is non-empty we then set Node ‘y’ to Node ‘x’ and compare their key values by calling getInfo on both nodes. If Node ‘z’ value is less than Node ‘x’ then we are going to insert to the left. Vice versa is the key value of Node ‘z’ is greater than Node ‘x’ then we will insert to the right. We then need to adjust the parent of Node ‘z’. We first set the parent of it to be Node ‘y’ which contains null. If Node’y’ is null we then set the root to Node ‘z’, else we determine to set the left or right of Node ‘y’ based on the priority index. Inserting a node into a binary search tree takes O(lg n) time.



Output for Option 1 - Inserting a Process to the BST (based on the priority index)

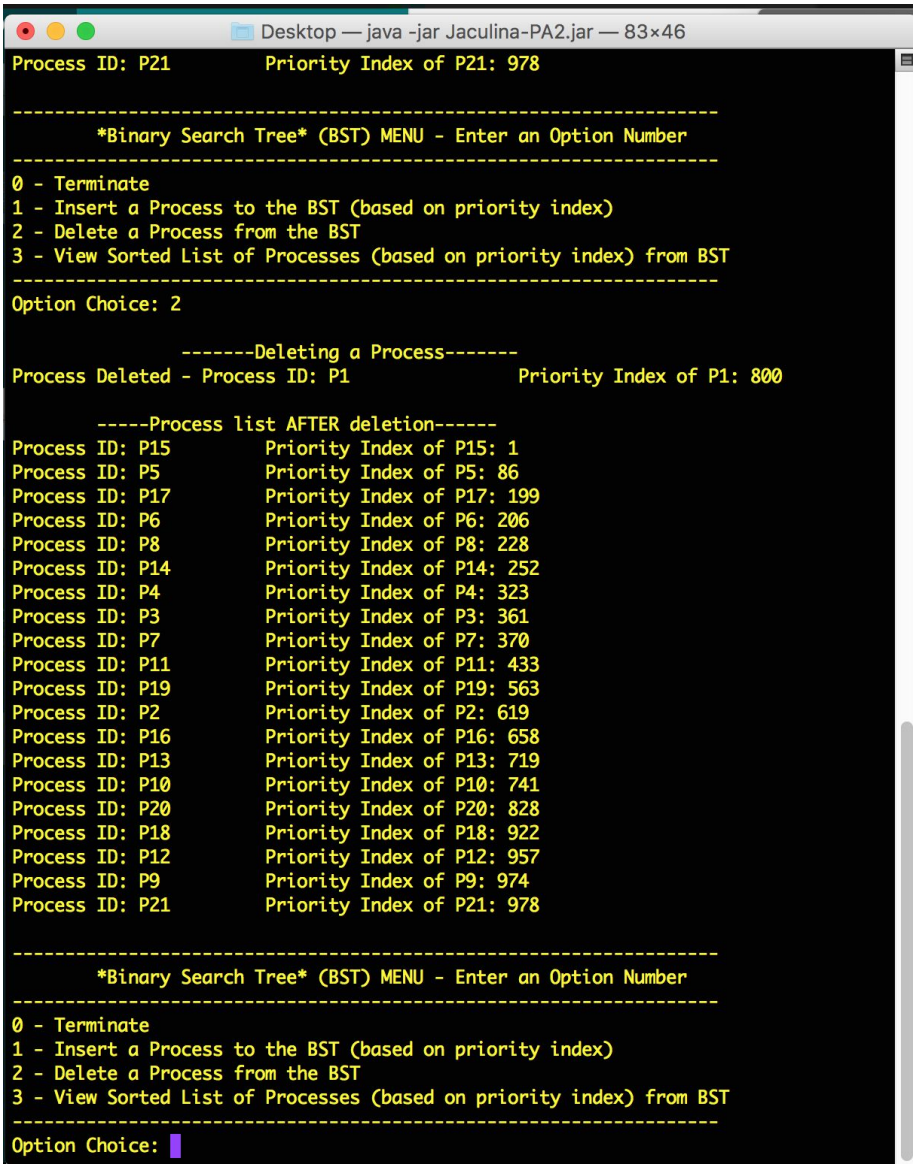
transplant(ProcessBST T, Node u, Node v)

This method is a subroutine in which it allows subtrees to moved around within the binary search tree. It replaces one subtree as a child of its parent with another subtree. After

executing transplant we will see that TRANSPLANT replaces the subtree rooted at node 'u' with the subtree rooted at node 'v', and the parent of node 'u' becomes the parent of node 'v' and the 'u's parent ends up having 'v' as its appropriate child. Transplant is primarily used whenever a node is deleted in the binary search tree.

treeDelete(ProcessBST T, Node z)

This method deletes a specific node from the binary search tree. When it comes to deleting a node within a binary search tree there are 3 scenarios. The node that we are deleting may not have a left child, right child, or have both of their children. The delete method uses the method transplant to rearrange the children and parent. The method also uses minimum in order to get the minimum value in the right subtree. Deleting a node in a binary search tree takes O(lg n) time, that is the height of the tree.



Output for Option 2 - Deleting a Processing from the BST

HashTable.java

In HashTable.java we are asked to create a table that would be of size 11 and use the technique of chaining to resolve any collisions.

HashTable()

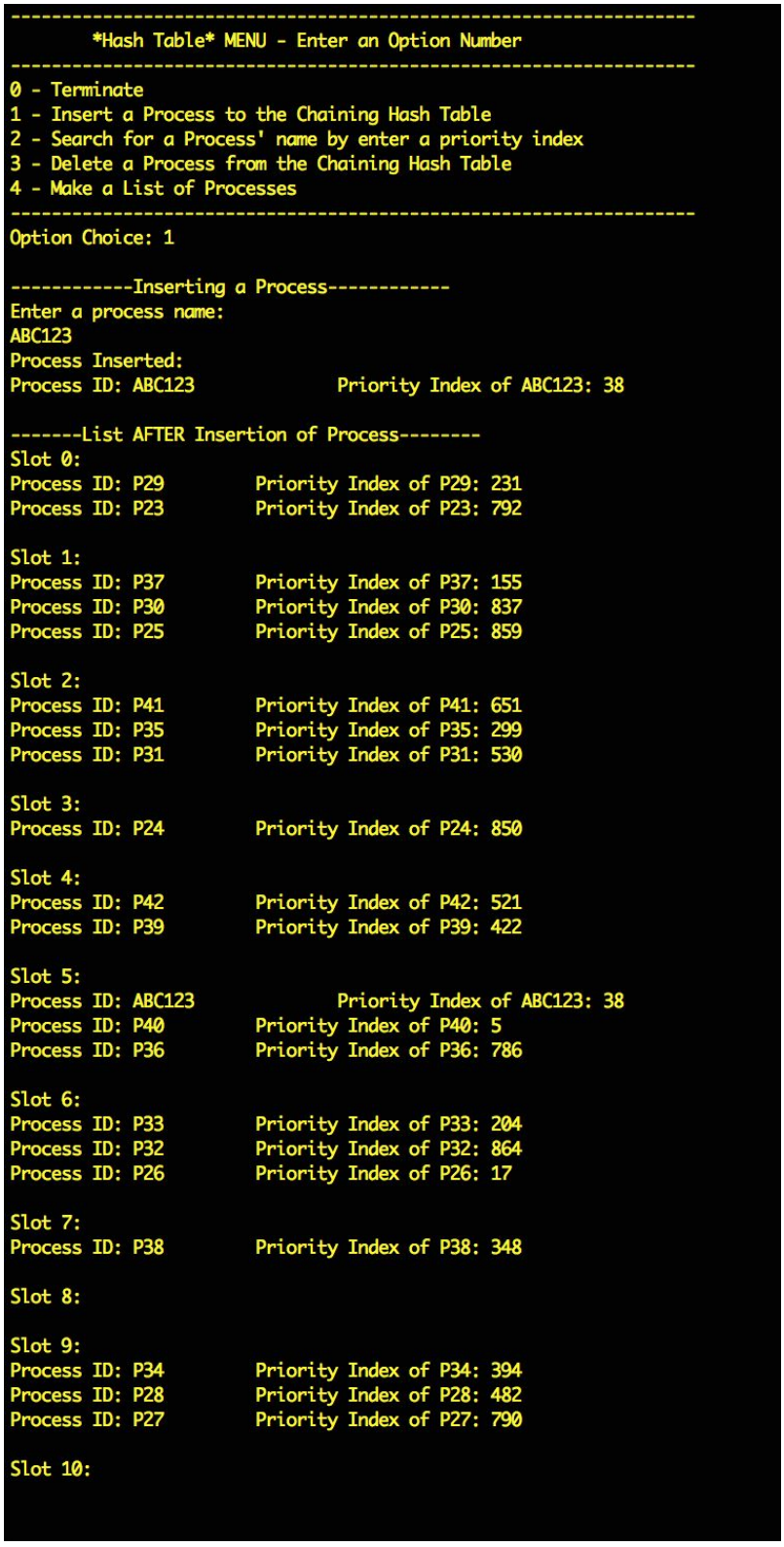
The no-argument constructor of HashTable.java constructs a linked list and sets the size to the hash table size. In this particular CPU system we are hashing 20 processes into 11 slots. Then when we create each slot in the hash table, we will create a linked list for it so that chaining can occur if the CPU scheduler experiences any collisions.

hashFunction(Node x)

A good hash function is to be said when it distribute keys uniformly into slots and it should not depend on patterns in the data. In our system, we are chaining each process into a hash table with a size of 11. The node that is passed in gets its priority index and we take the mod of it by 11. Doing so will give us the table slot in the hash table.

chainedHashInsert(HashTable T, Node x)

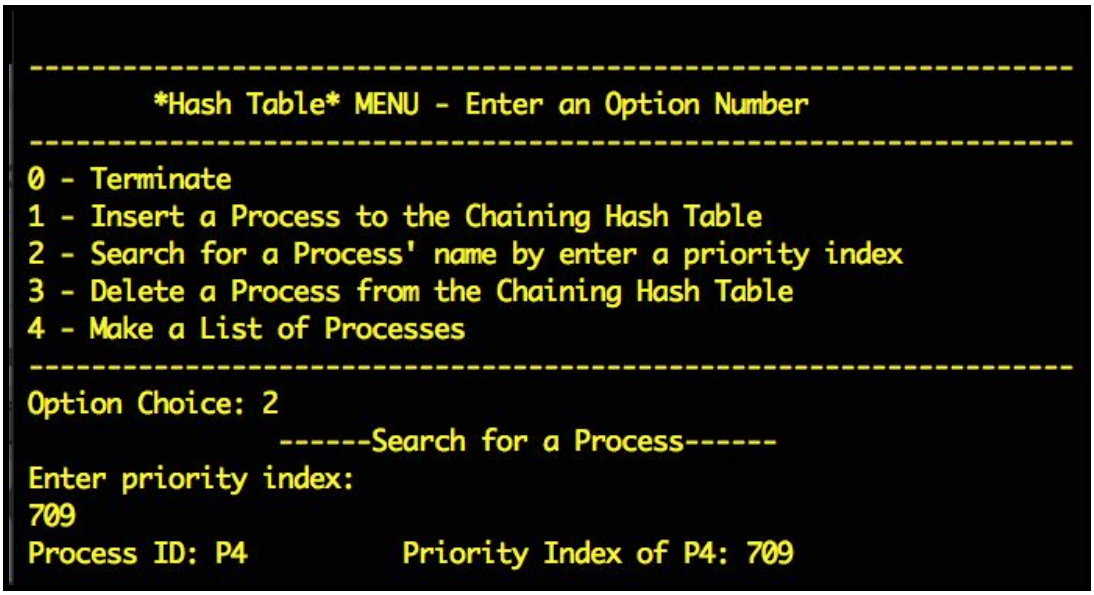
In chainHashInsert we are passing in a hashtable and also a node. The node that we are passing in is the node that we want to insert inside the hashtable. We first get the Node 'x' priority index by calling getInfo() and then pass it into the hashFunction to obtain the table slot for the hash table. We then insert to the table by calling the linkedlist function listInsert and pass in a linked list along with the node to add. The worst-case running time for insertion is O(1).



Output for Option 1 - Insert a Process to the Chaining Hash Table

chainedHashSearch(HashTable T, int k)

In chainedHashSearch function we are passing in a hash table ‘T’ and an integer. The hashtable ‘T’ is the table that we want to iterate through. The key ‘k’ is the a value in which we first calculate the table slot. Then with that table slot integer we use the listSearch method to check if the node is in the linkedlist for a specific hash table slot. We also check if the node is empty or not and have an error message. This function search and checks if a Node is in the hash table and returns a Node.

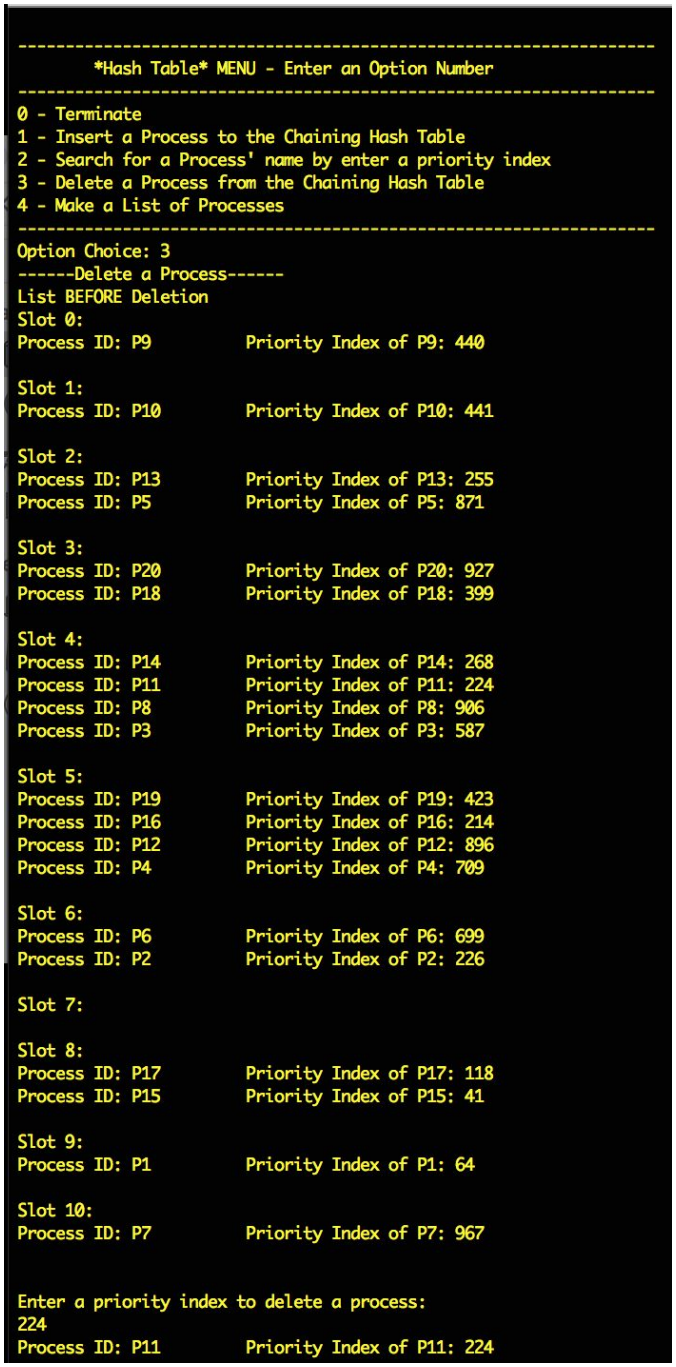


```
------------------------------------------------------------------
       *Hash Table* MENU - Enter an Option Number
------------------------------------------------------------------
0 - Terminate
1 - Insert a Process to the Chaining Hash Table
2 - Search for a Process' name by enter a priority index
3 - Delete a Process from the Chaining Hash Table
4 - Make a List of Processes
------------------------------------------------------------------
Option Choice: 4

--------Sorted List of Processes based on Priority Index--------
Slot 0:
Process ID: P9          Priority Index of P9: 440

Slot 1:
Process ID: P10         Priority Index of P10: 441

Slot 2:
Process ID: P13         Priority Index of P13: 255
Process ID: P5          Priority Index of P5: 871

Slot 3:
Process ID: P20         Priority Index of P20: 927
Process ID: P18         Priority Index of P18: 399

Slot 4:
Process ID: P14         Priority Index of P14: 268
Process ID: P11         Priority Index of P11: 224
Process ID: P8          Priority Index of P8: 906
Process ID: P3          Priority Index of P3: 587

Slot 5:
Process ID: P19         Priority Index of P19: 423
Process ID: P16         Priority Index of P16: 214
Process ID: P12         Priority Index of P12: 896
Process ID: P4          Priority Index of P4: 709

Slot 6:
Process ID: P6          Priority Index of P6: 699
Process ID: P2          Priority Index of P2: 226

Slot 7:

Slot 8:
Process ID: P17         Priority Index of P17: 118
Process ID: P15         Priority Index of P15: 41

Slot 9:
Process ID: P1          Priority Index of P1: 64

Slot 10:
Process ID: P7          Priority Index of P7: 967
```

Output for Option 2 - Search for a Process' name by entering a priority index

chainedHashDelete(HashTable T, Node x)

In chainHashDelete function we are passing in a hash table ‘T’ and also a Node ‘x’. With the node that is passed through we call hashFunction on it to obtain a table slot integer. Then with the table slot integer we call the listDelete from the linked list class. This function deletes a process from the hash table according to the integer the user inputs. The integer (priority index) must occur in the hash table. Using a doubly linked list allows deleting an element to take O(1) time.



Output for Option 3 - Delete a Process from the Chaining Hash Table

sop(Object x)

This shortcut function implements the input/output print statement System.out.println. It is used to make the main method have a cleaner feel. Whatever is passed as the Object "x" will be printed out.

toHashString()

This function prints out the entire hash table with the slot number.

Process.java

Process()

This is a no-argument constructor to construct a process in the CPU scheduler system.

Process(String processId)

This is a one-argument constructor to construct a process. It passes in a process id (name of the process).

getProcessId()

This is a getter method to get the process id number. The process id number is concatenated with the character "P."

setProcessId(String s)

This set the process name for a single process. It passes in a new string 's' to set it to.

getPriorityIndex()

This method gets the priority index for a particular process.

setPriorityIndex(int priorityIndex)

This method sets the priorityIndex for a particular process.

Node.java

This class implements a Node used for the ProcessBST and HashTable class.

Node(Process key)

One argument constructor. This constructors a node that contains a process. A process contains a process name and priority index.

getNext()

This method gets the next node in the linked list.

setNext(Node no)

This method sets the node with another node.

getPrev()

This method gets the previous node in the linked list.

setPrev(Node p)

This method sets the previous node with another node that is passed in.

setLeft(Node left)

This method sets the left node from the node that is passed in.

getLeft()

This method gets the left node.

setRight(Node right)

This method sets the right node from the nodes that is passed in.

getRight()

This method gets the right node.

setKey(Process key)

This method passes in a process object and sets it to another process.

getKey()

This method gets the key of a type Process.

setParent(Node parent)

This method sets the parent to the new node that is passed it. Used in the method transplant.

getParent()

This method gets the parent of a Node. Used in the method transplant.

getInfo()

This method returns an integer which is a Process' priority index.

toNodeString()

This function prints out the the process id and the priority index of a Node object.

LinkedList.java

getHead()

This getter method is to retrieve the head of a Node.

setHead(Node head)

This setter method sets the head for a node.

listInsert(LinkedList L, Node x)

This function was implemented from Chapter 10 in which it passes in a linked list object and a node. The linked list object "L" is the linkedlist that we want to insert to. The Node 'x' is the node that we want to append to the linked list. The procedure splices 'x' onto the front of the linked list. The running time for listInsert on a list of *n* elements is O(1).

listSearch(LinkedList L, int k)

This function uses a simple linear search in a given linked list and returns a pointer to the element. If no object with key *k* appears to be in the linked list, then the function returns null. The running time for listSearch on a list of *n* elements is O(1).

listDelete(LinkedList L, Node x)

This method removes the passed in Node 'x' from the LinkedList 'L'. It must be given a pointer to x, and then splices 'x' out of the list by updating points. The worst-case running time is O(n) and the average case runs in O(1).

toLinkedListString()

This function prints out the linked list. It uses the toNodeString to print out each individual node that is in the linked list.

sop(Object x)

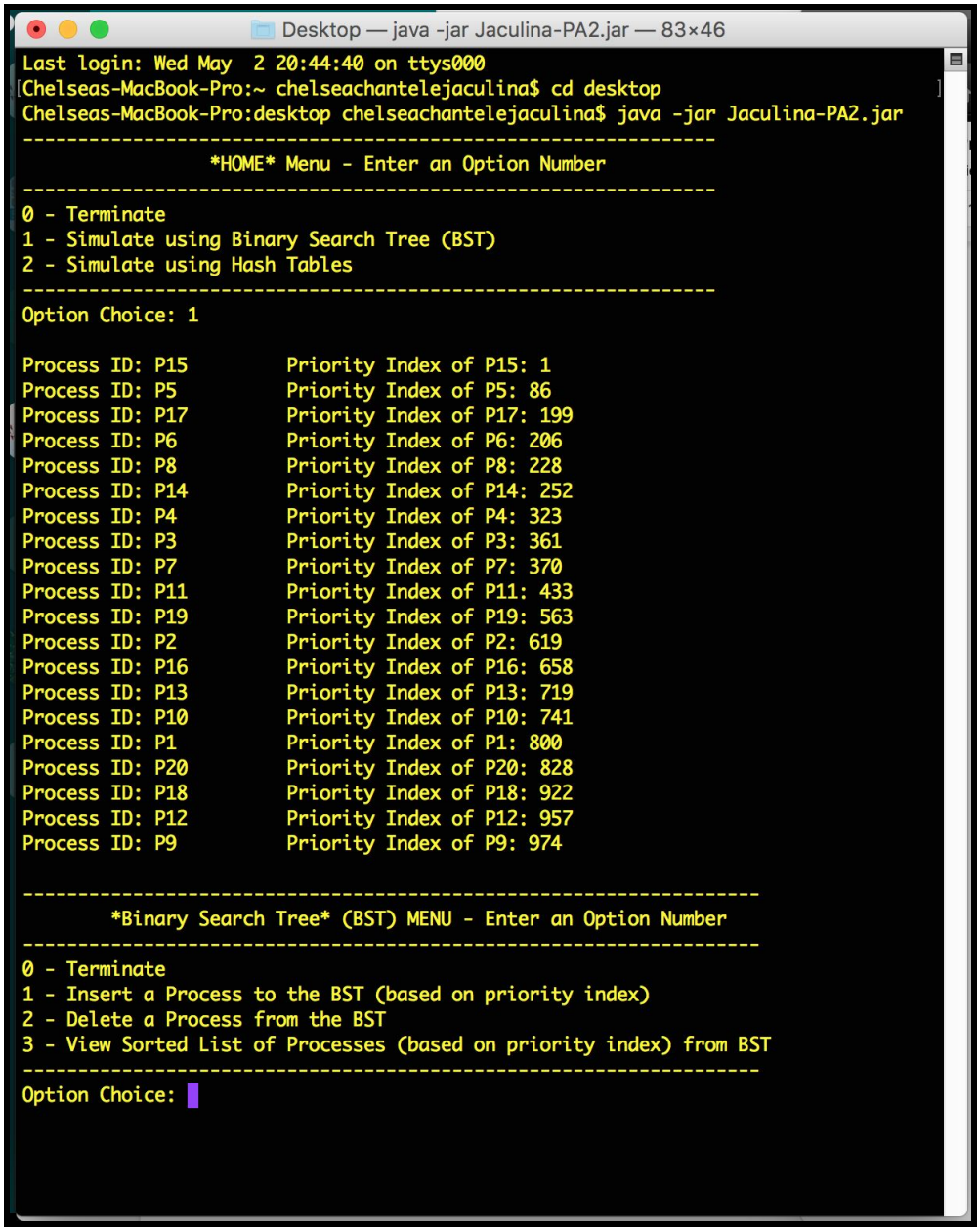
This shortcut function implements the input/output print statement System.out.println. It is used to make the main method have a cleaner feel. Whatever is passed as the Object “x” will be printed out.

MainMethod.java

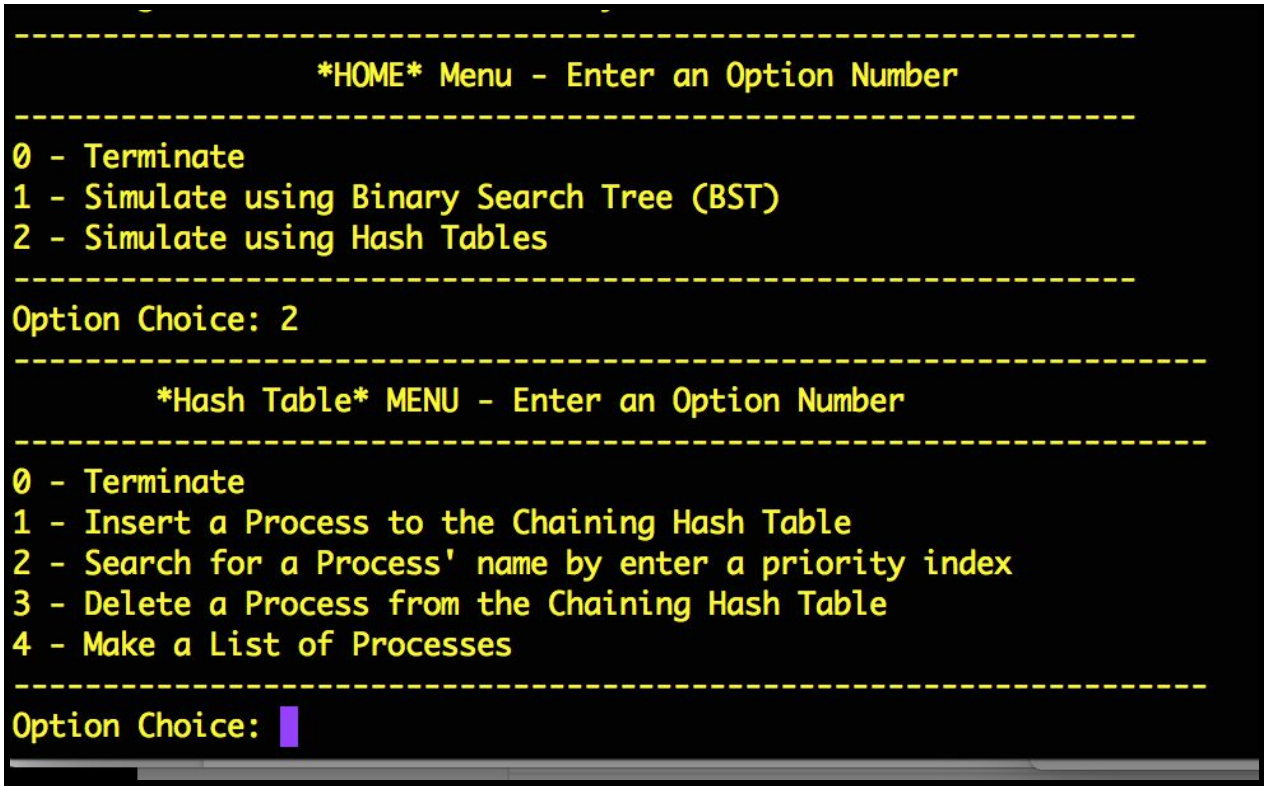
main(String[] args)

This method runs contains the user interface that runs the home main menu, binary search menu, and the hash table menu.

Output for Option 0 - Terminate Home Menu



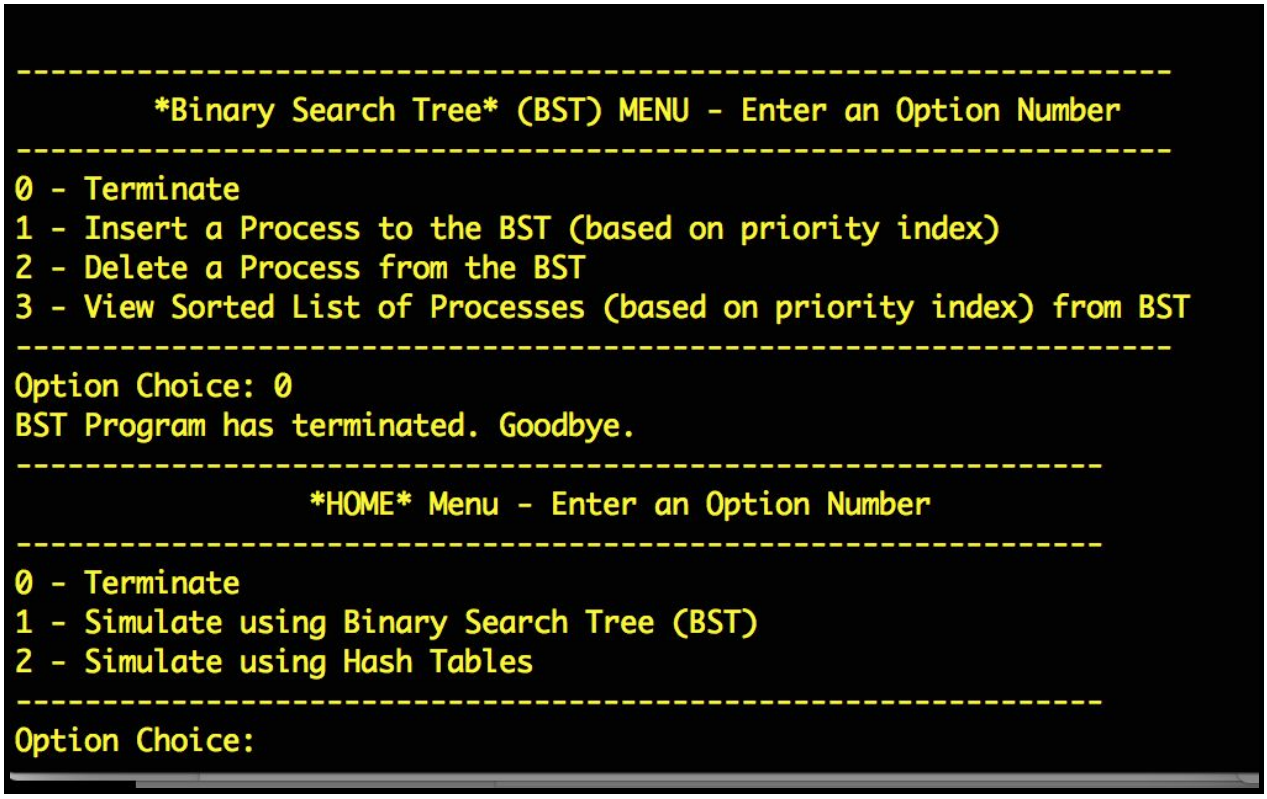
Output for Option 1 - Simulate Using Binary Search Tree (BST)



Output for Option 2 - Simulate using Binary Search Tree (BST)

BSTMenu(ProcessBST bst)

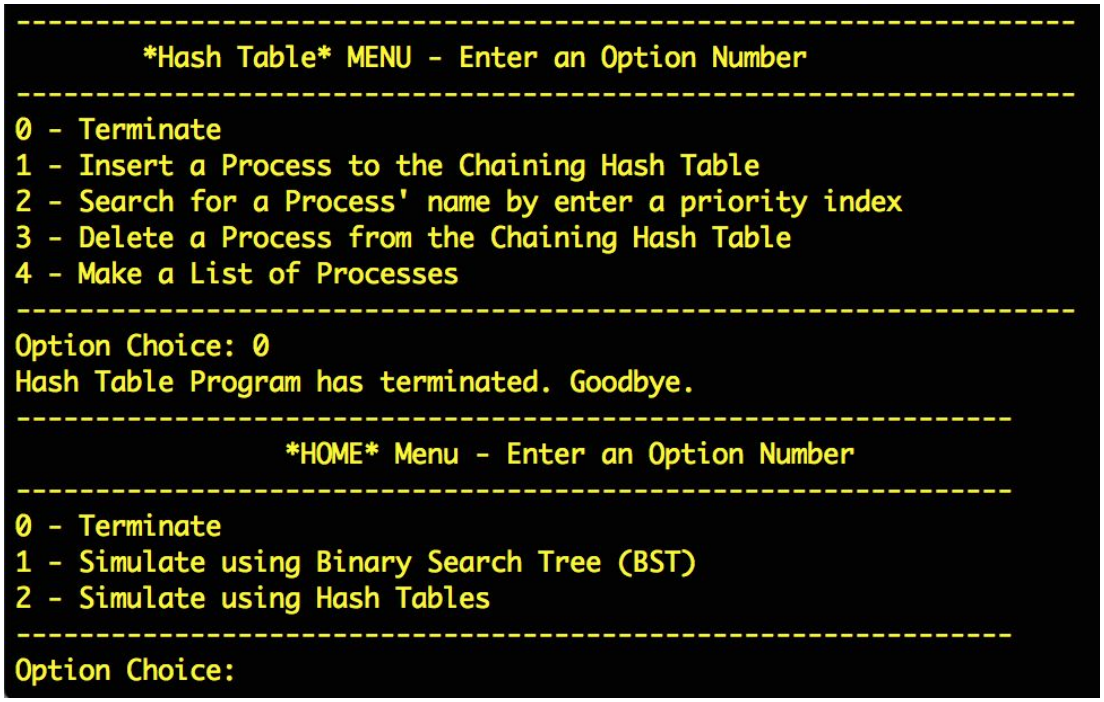
This method displays the user interface for the binary search tree menu. The binary search menu contains 4 different options. The options include terminating the program, inserting a process based on the priority index, deleting a process, and display a sorted list of processes based on the priority index.



Output for Option 0 - Terminate Program for Binary Search Tree Menu

hashTableMenu(HashTable hash)

This method displays the user interface for the hash table tree menu. The hash table menu contains 5 different options. The options include terminate the program, inserting a process based on the priority index, searching for a process by a priority index deleting a process, and display a sorted list of processes based on the priority index.



Output for Option 0 - Terminate in Hash Table Menu

sop(Object x)

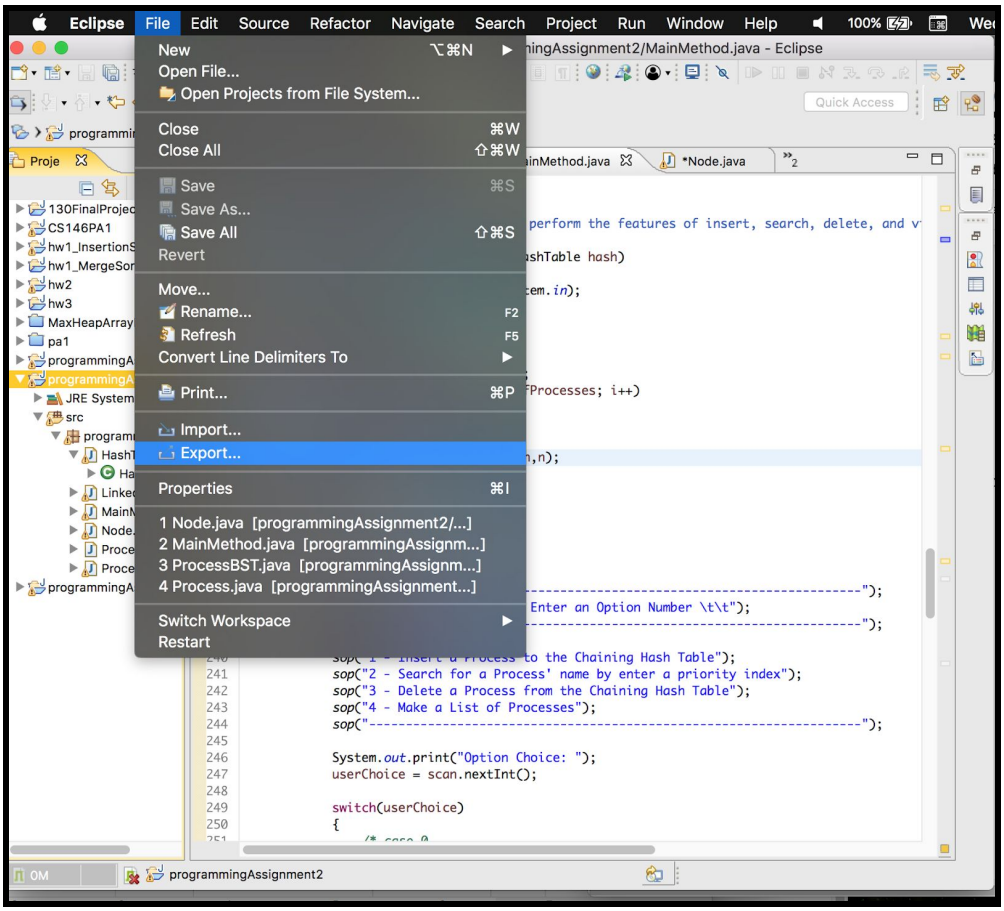
This shortcut function implements the input/output print statement System.out.println. It is used to make the main method have a cleaner feel. Whatever is passed as the Object "x" will be printed out.

How To Run the Application

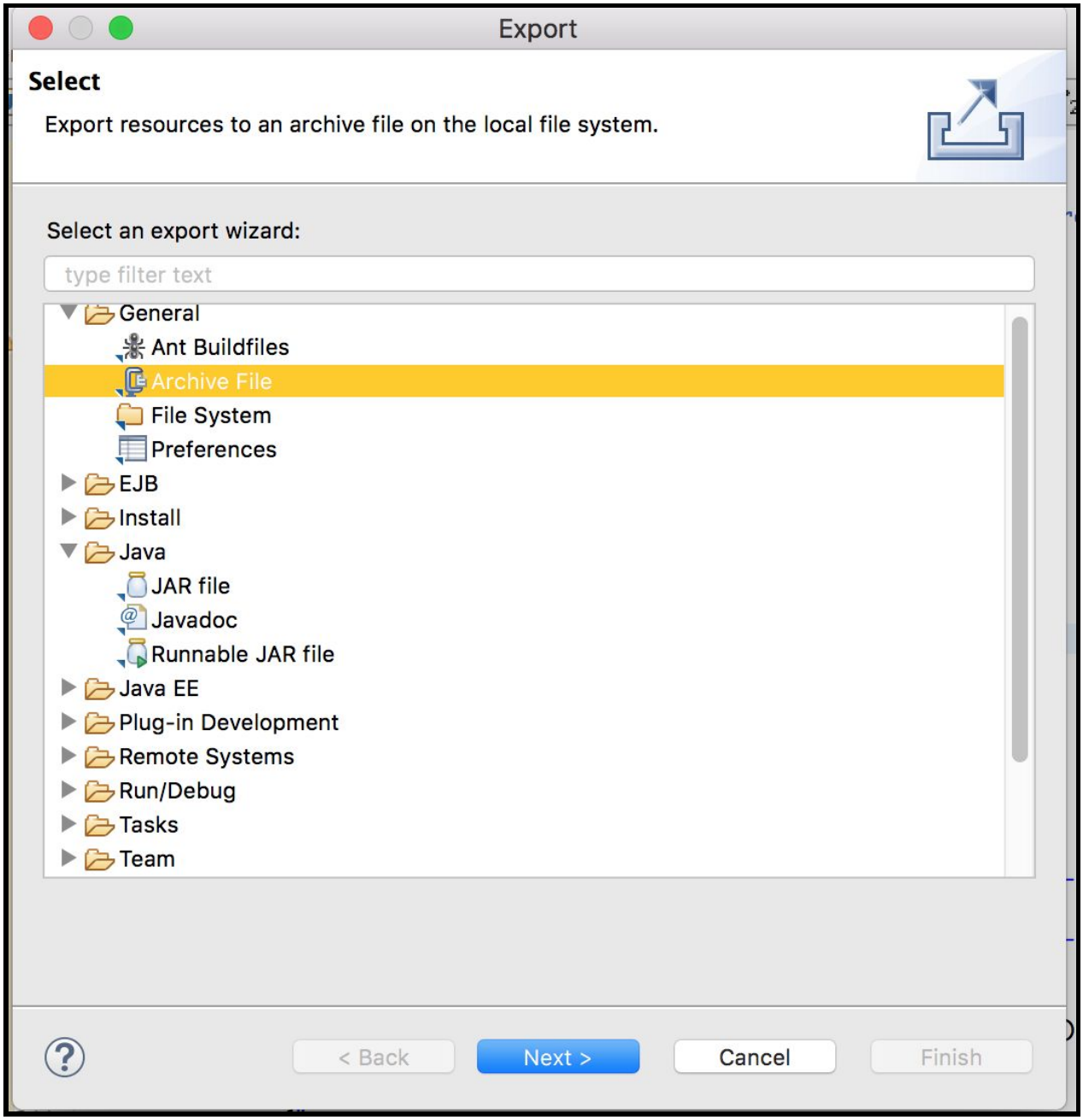
Step 1 Export Files

Zip File

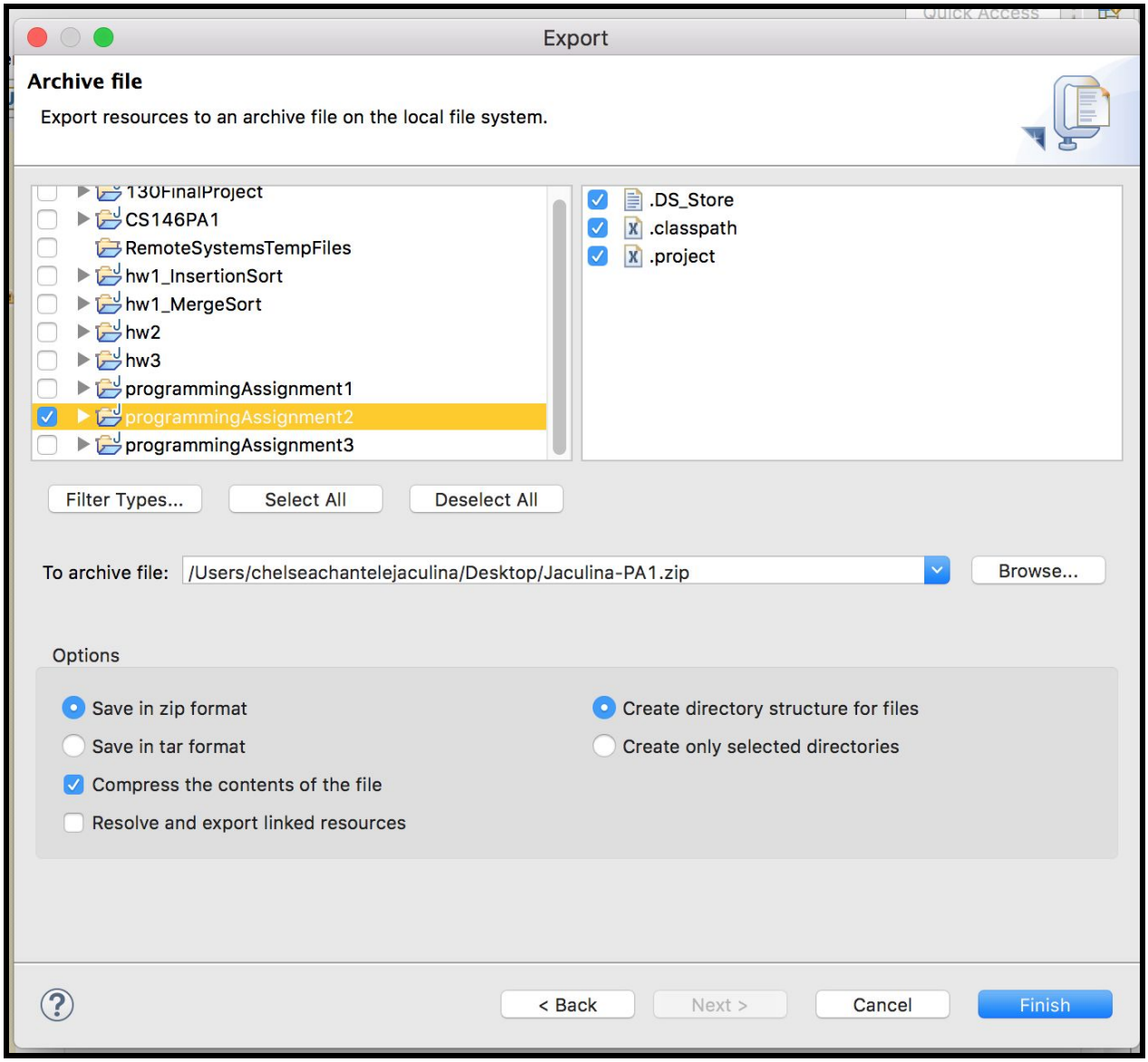
1. File → Export



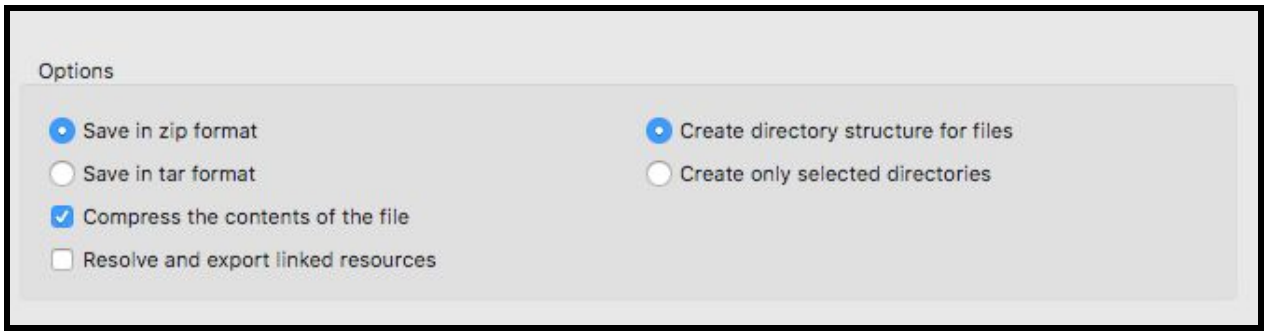
- Select the General → Archive File export Wizard



- Select the Project to be exported. In this case the CPU scheduling system is under "programmingAssignment2."

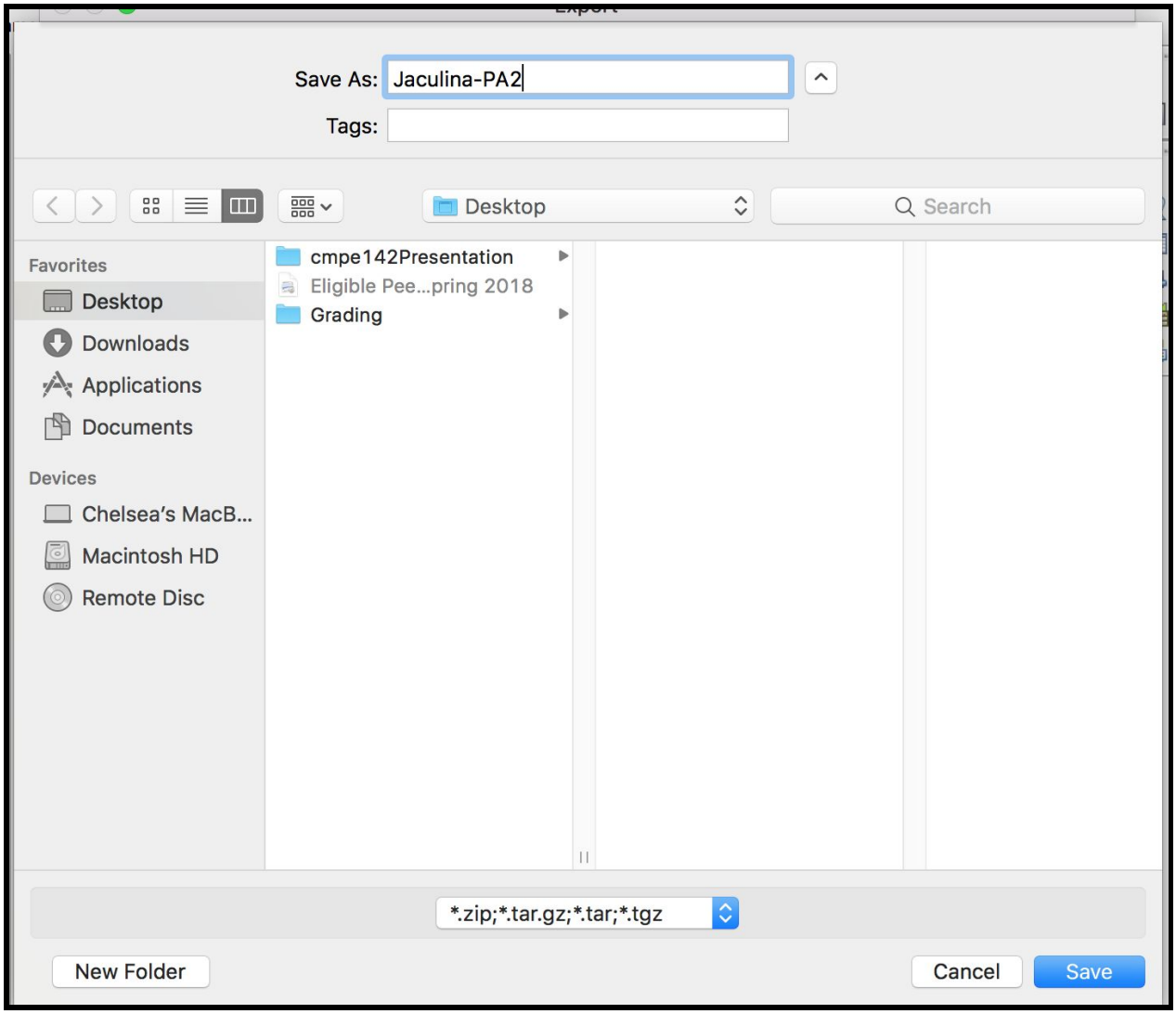


- Choose the archive file type **ZIP**. To create a zip file.

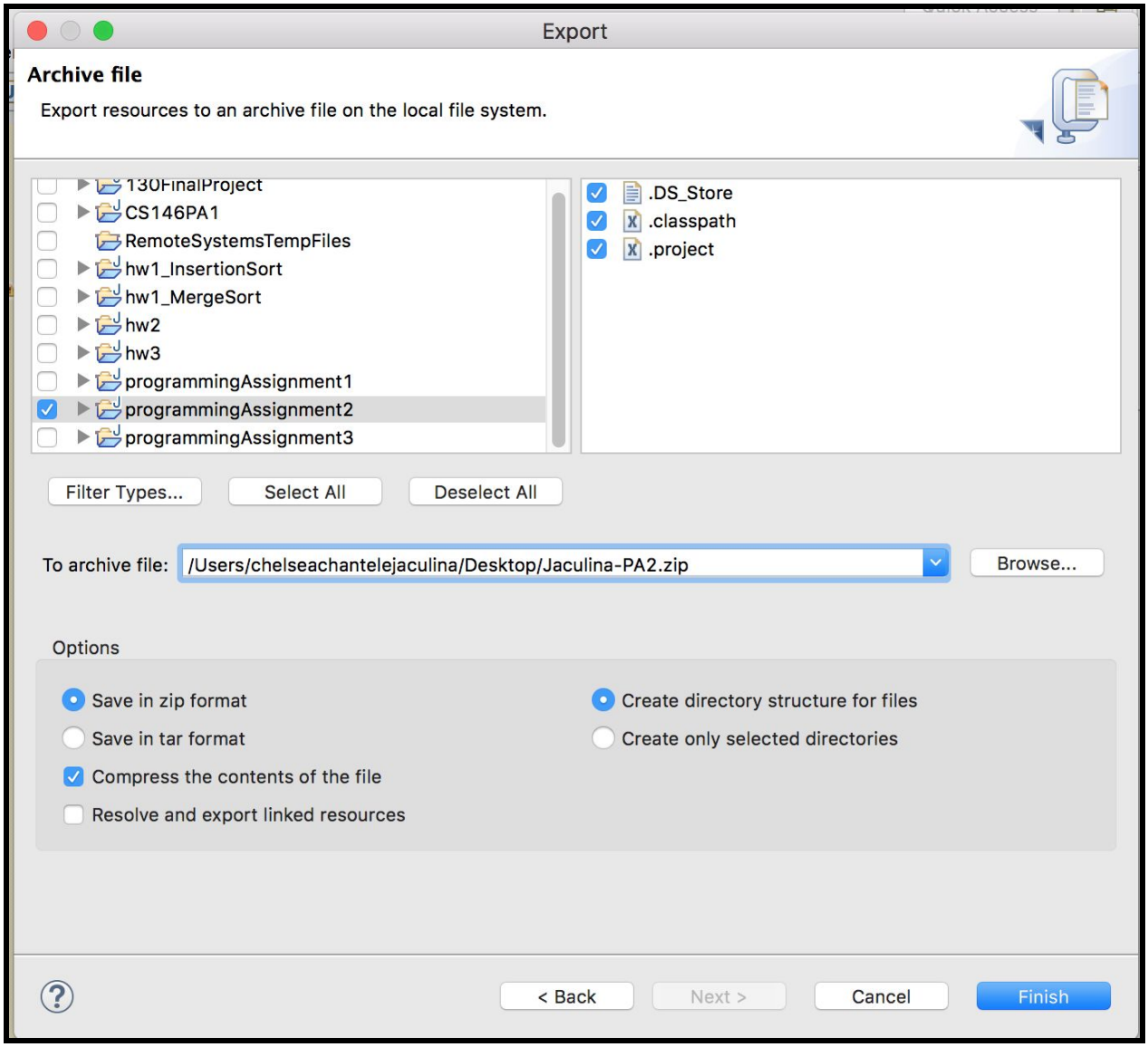


- Save archive file name as Jaculina-PA2.zip (Your-Last-Name)-PA2zip

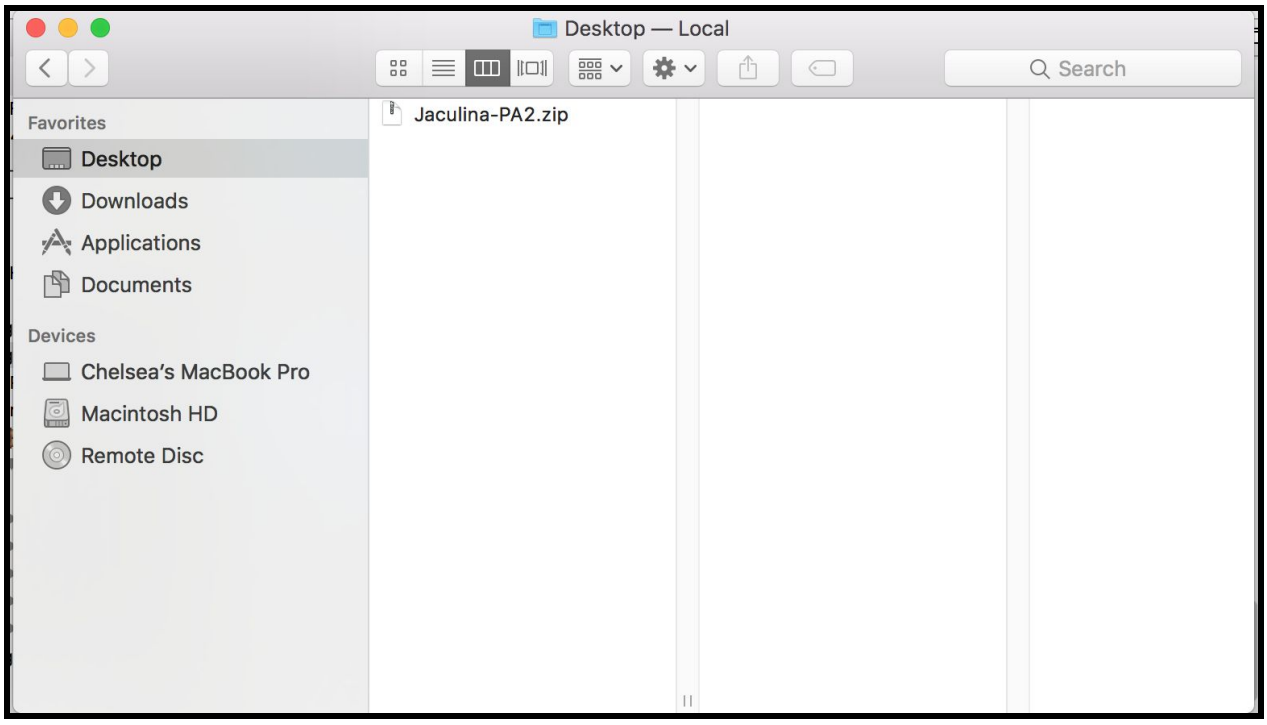
- Click on Browse to save to the Desktop. Make sure it is saved as zip file.



- Click **Save** to save zip file on Desktop. Then click on Finish.



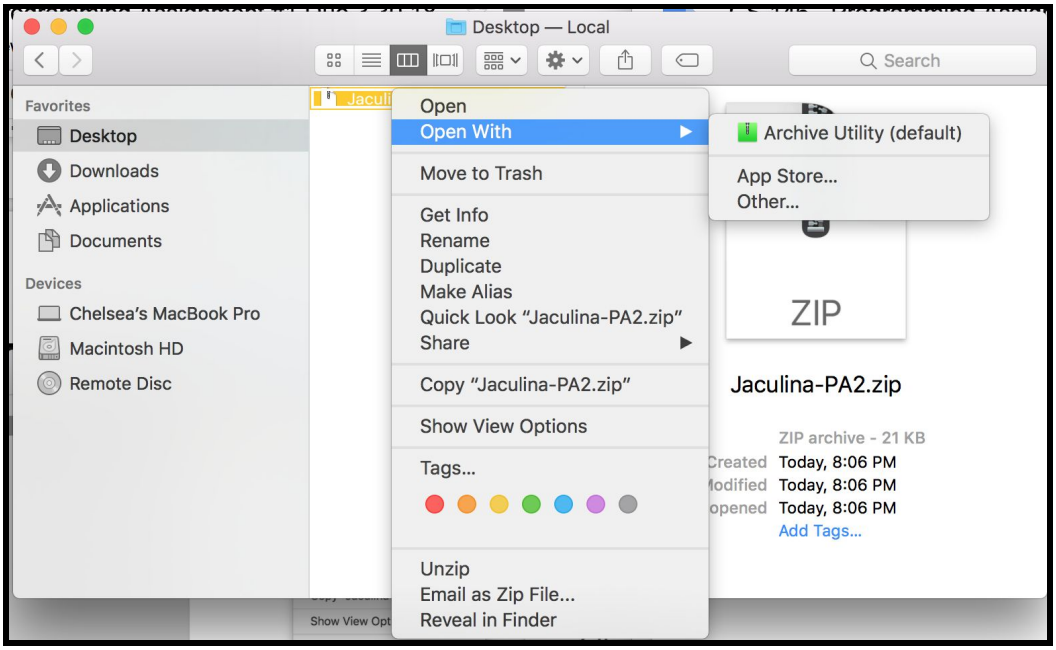
- Zip File should be saved on Desktop as "**Jaculina-PA2.zip**".



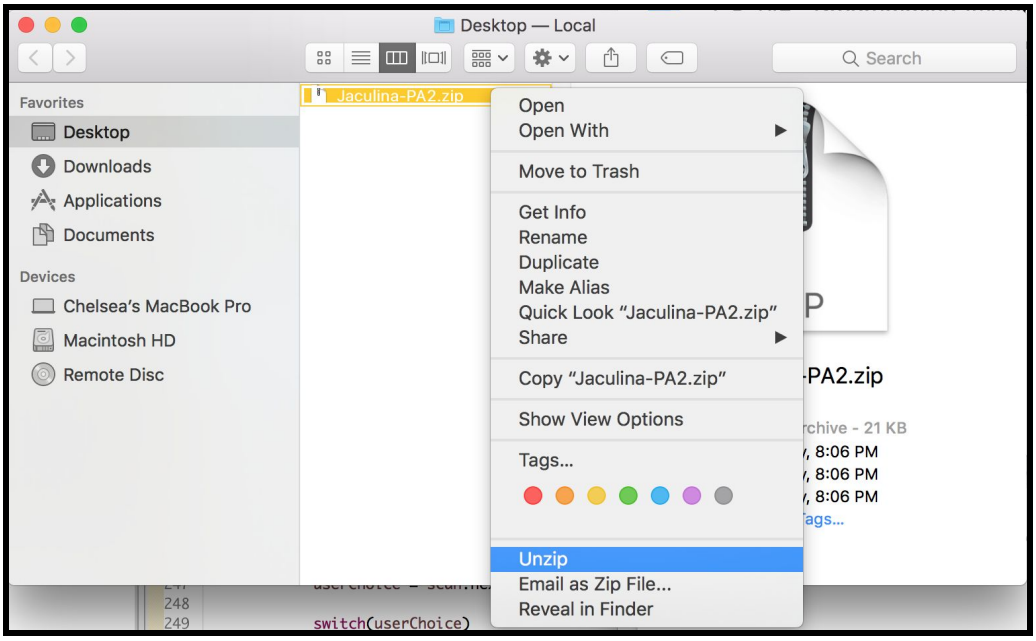
How to Unzip File

There are two ways to download the zip file and unzip it.

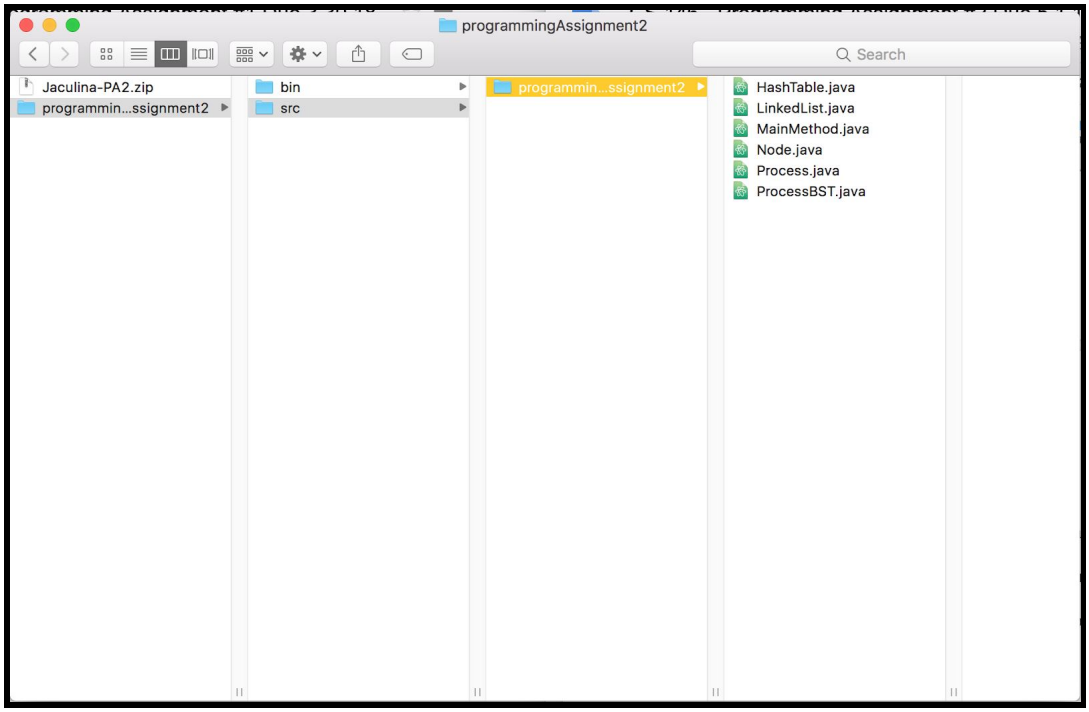
1. One way is to right click on the zip file and select the option **"Archive Utility (default)."**



2. The second way is to right click on the zip file and select the option **"Unzip."**



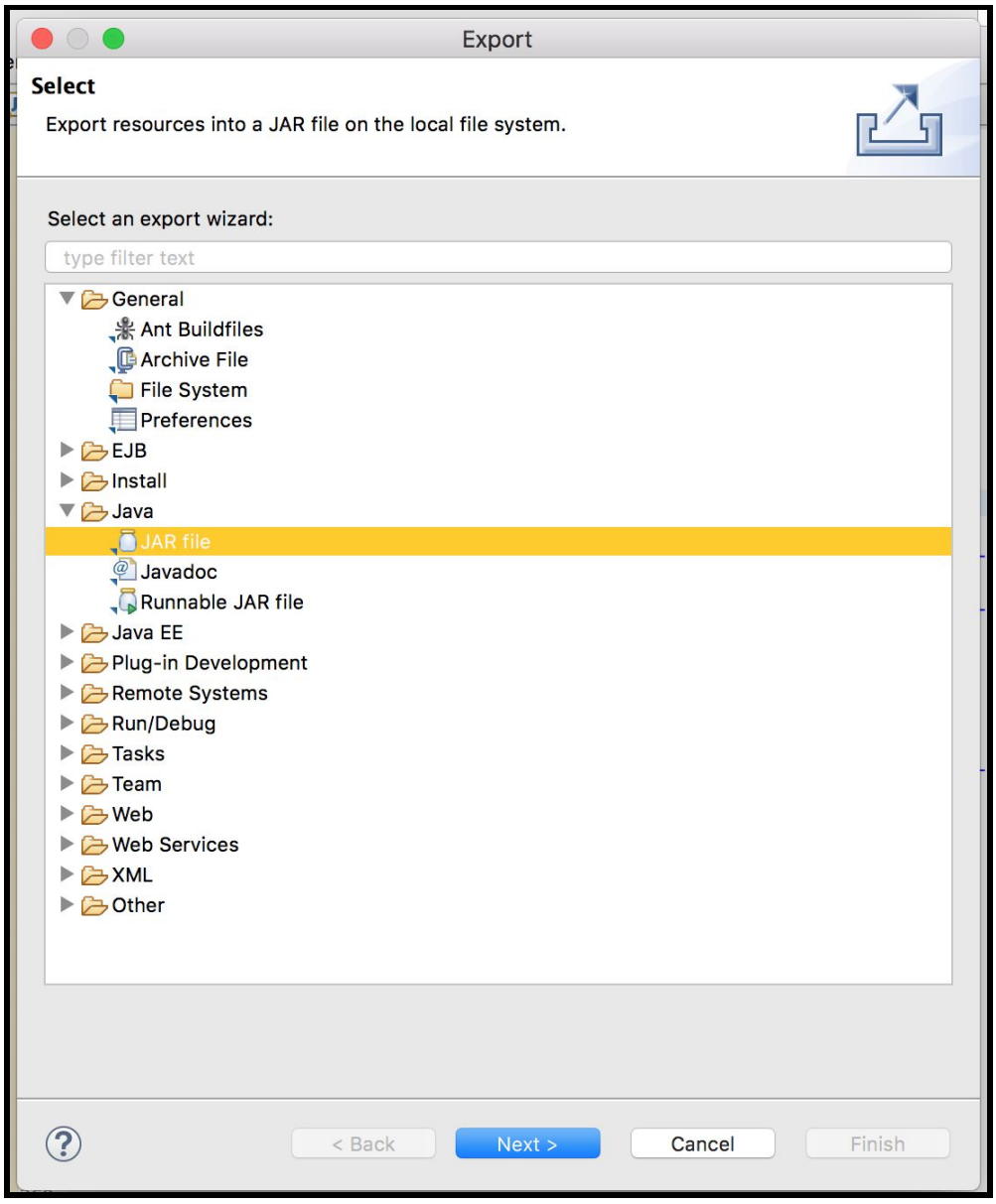
The result will be of unzipping the file will contain a folder that has a subfolder called **src** which has a subfolder called **programmingAssignment2**. In **programmingAssignment2** will contain the 6 java source code files (HashTable, LinkedList, MainMethod, Node, Process, ProcessBST).


programmingAssignment2
Search
Jaculina-PA2.zip
programmin...ssignment2
bin
src
programmin...ssignment2
HashTable.java
LinkedList.java
MainMethod.java
Node.java
Process.java
ProcessBST.java

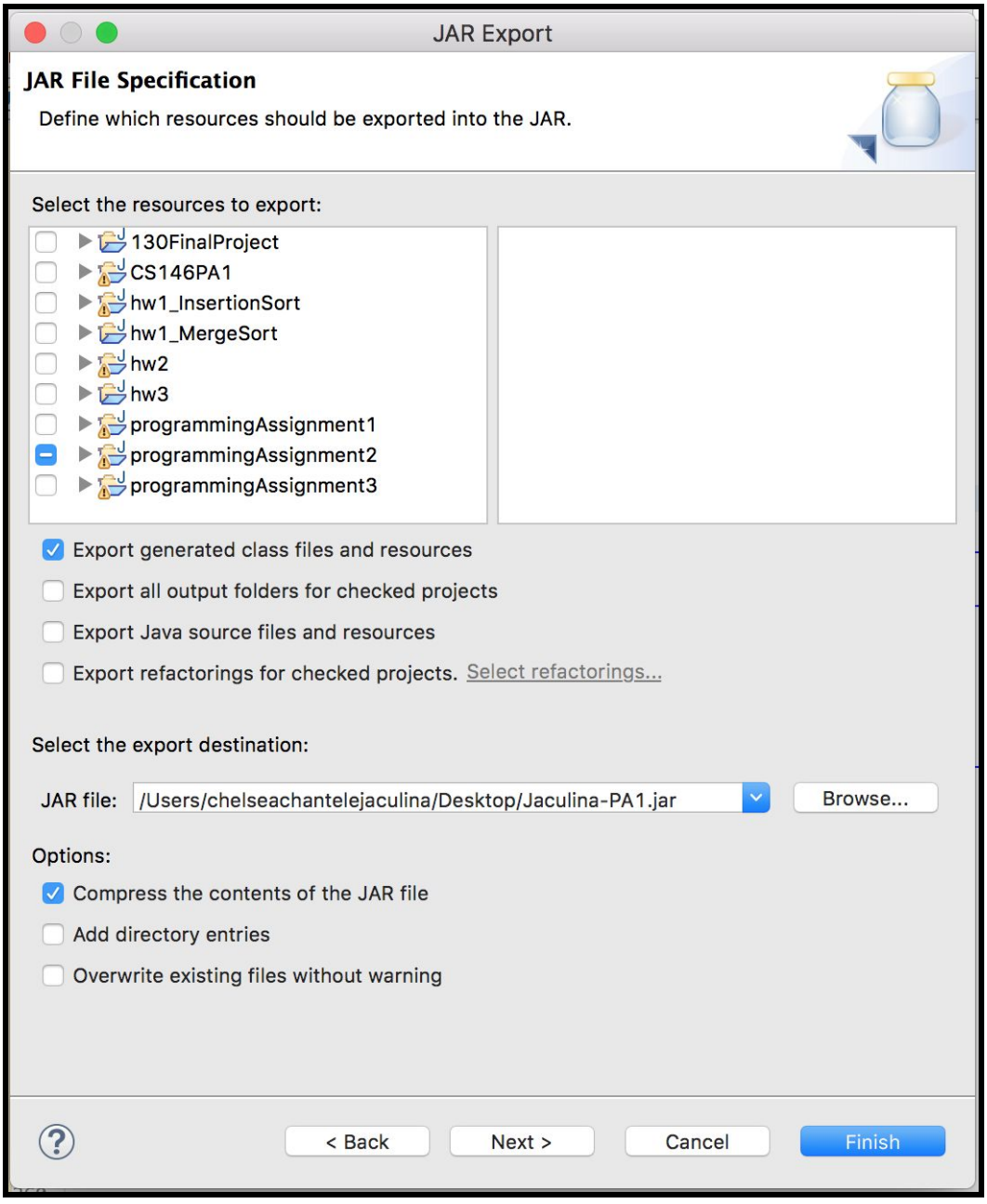
Jar File

1. File → Export

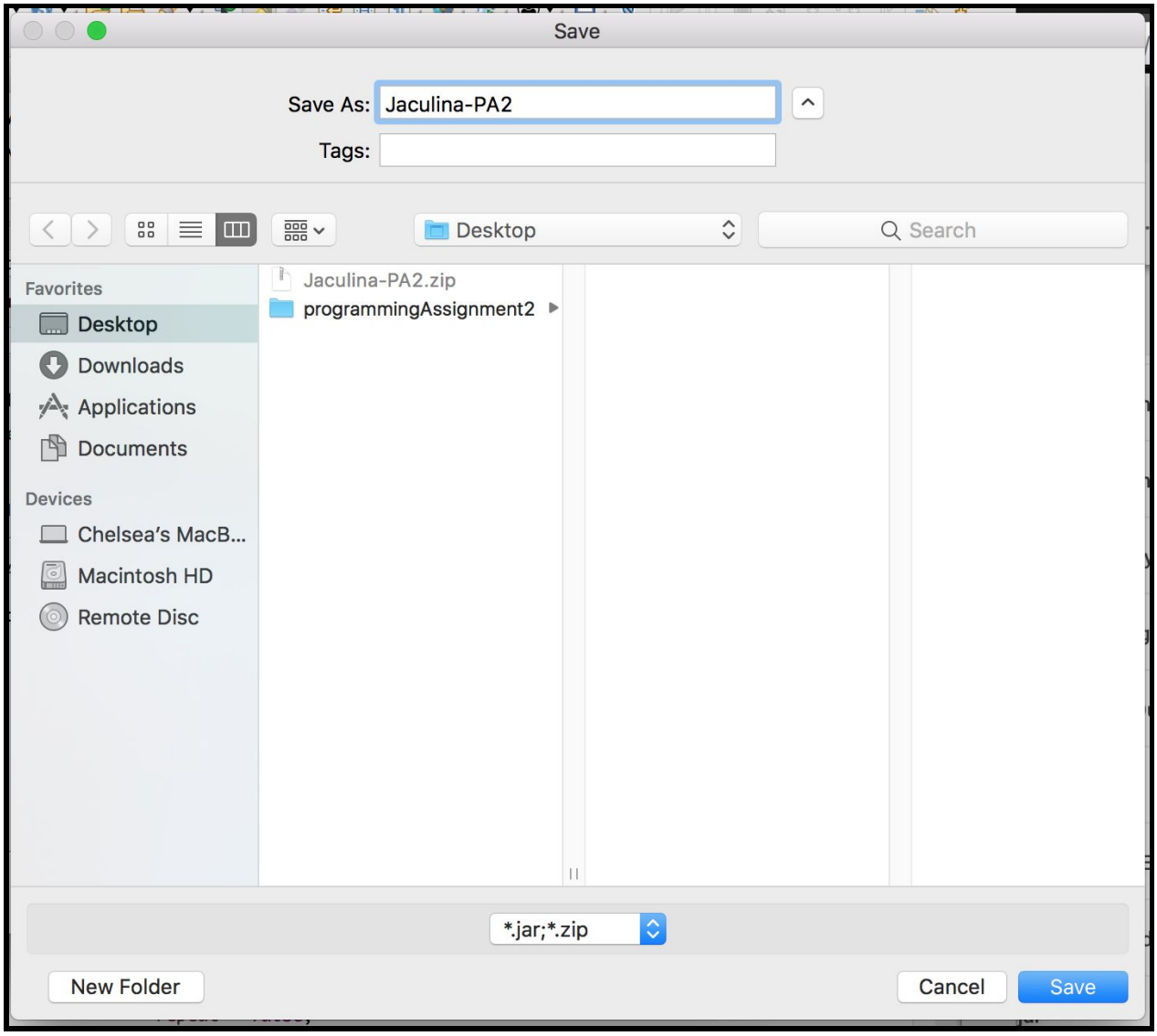
2. Selection the Java node and choose JAR file



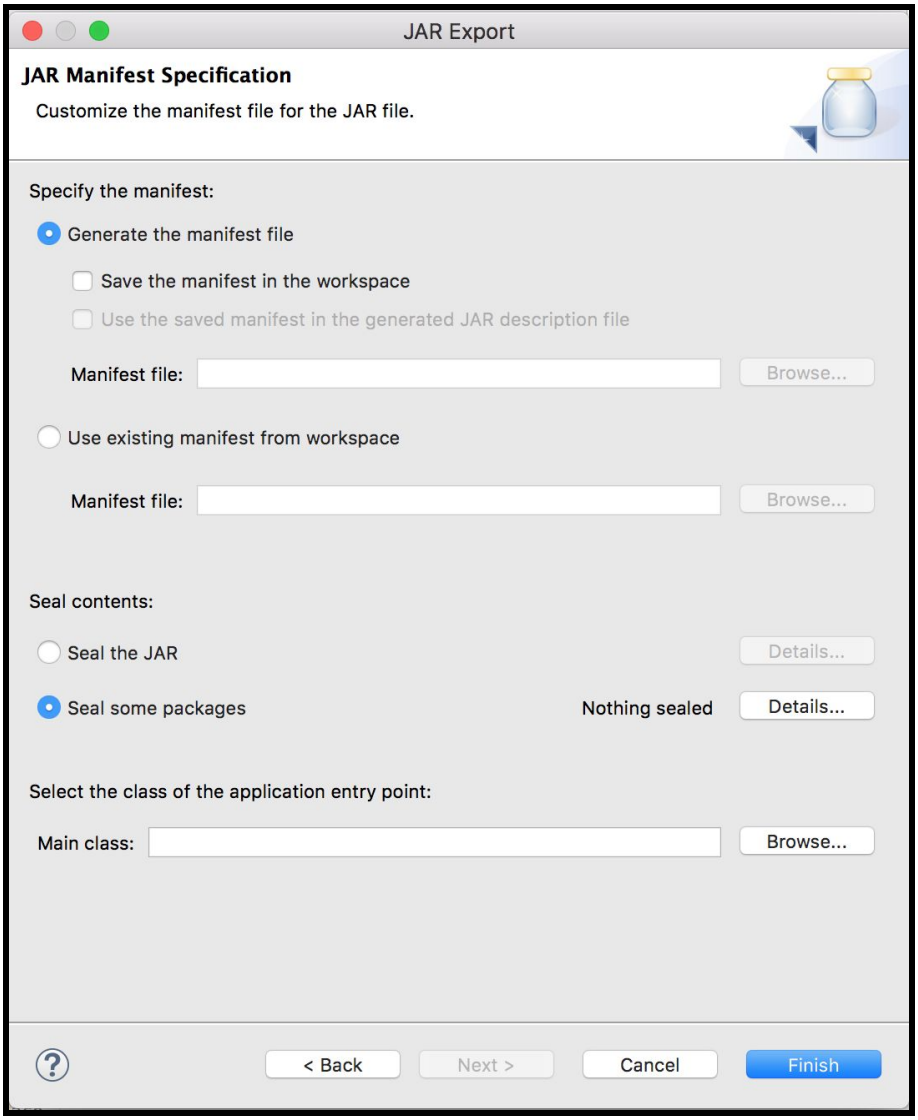
3. Select the resources to export: **“programmingAssignment2”.**



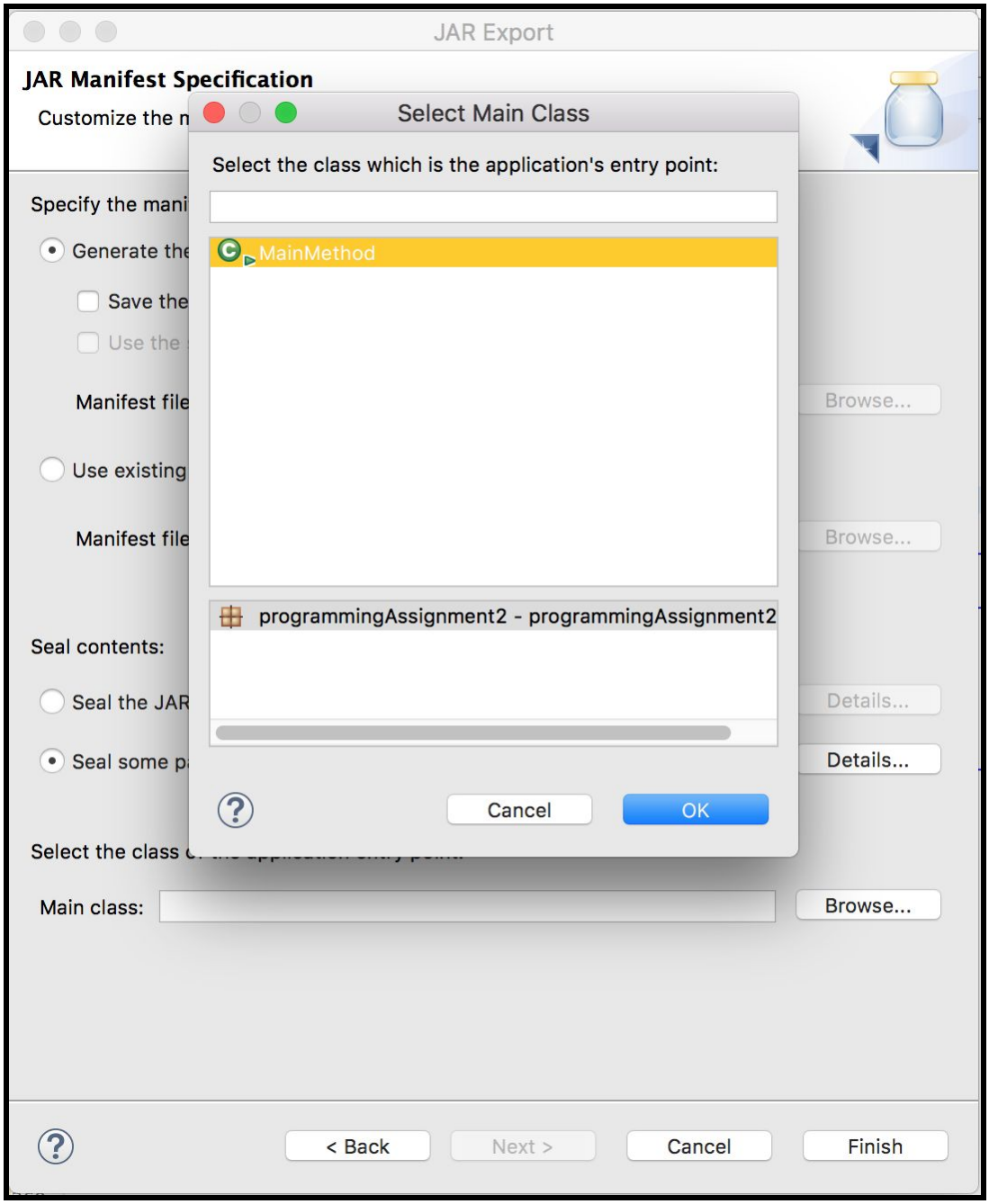
4. Select Browse and choose Desktop to save your Jar file. Name it with the extension lastname-PA2. Click save.



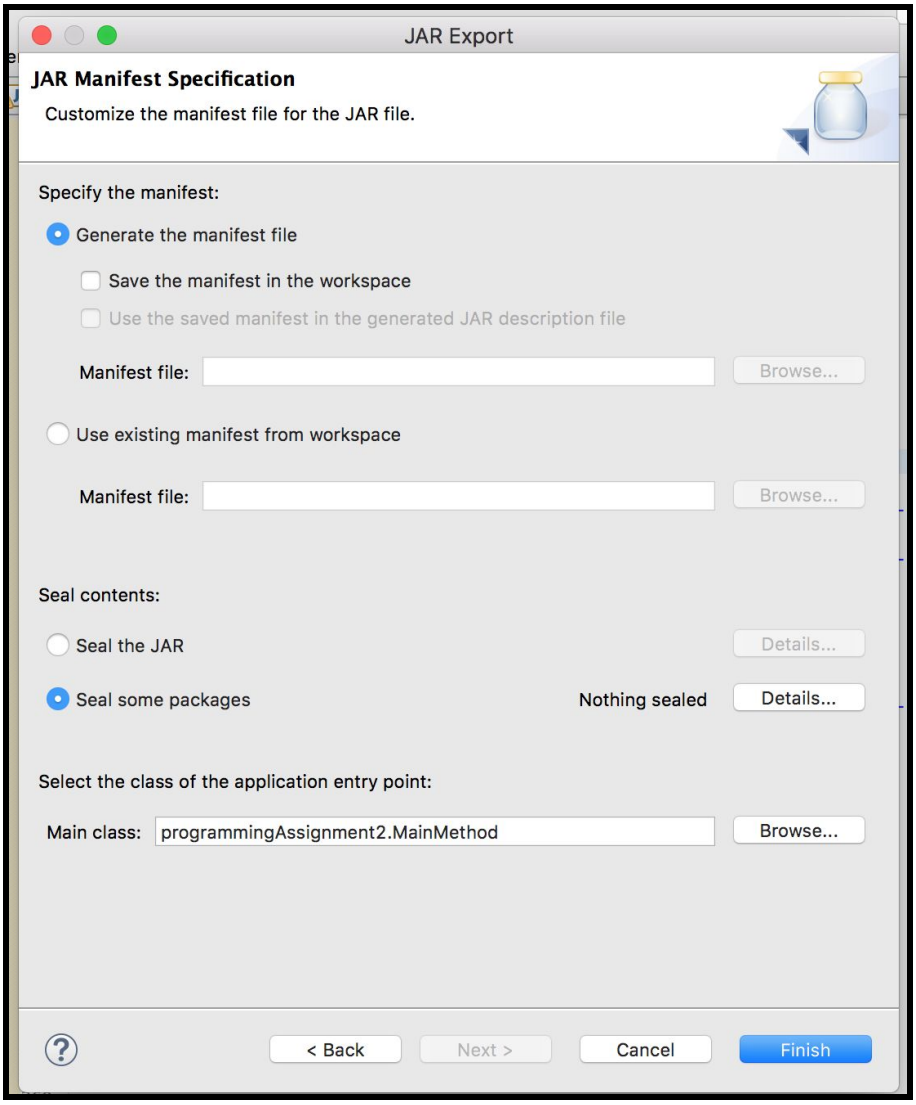
5. Click on **“Next”** Button **twice**.



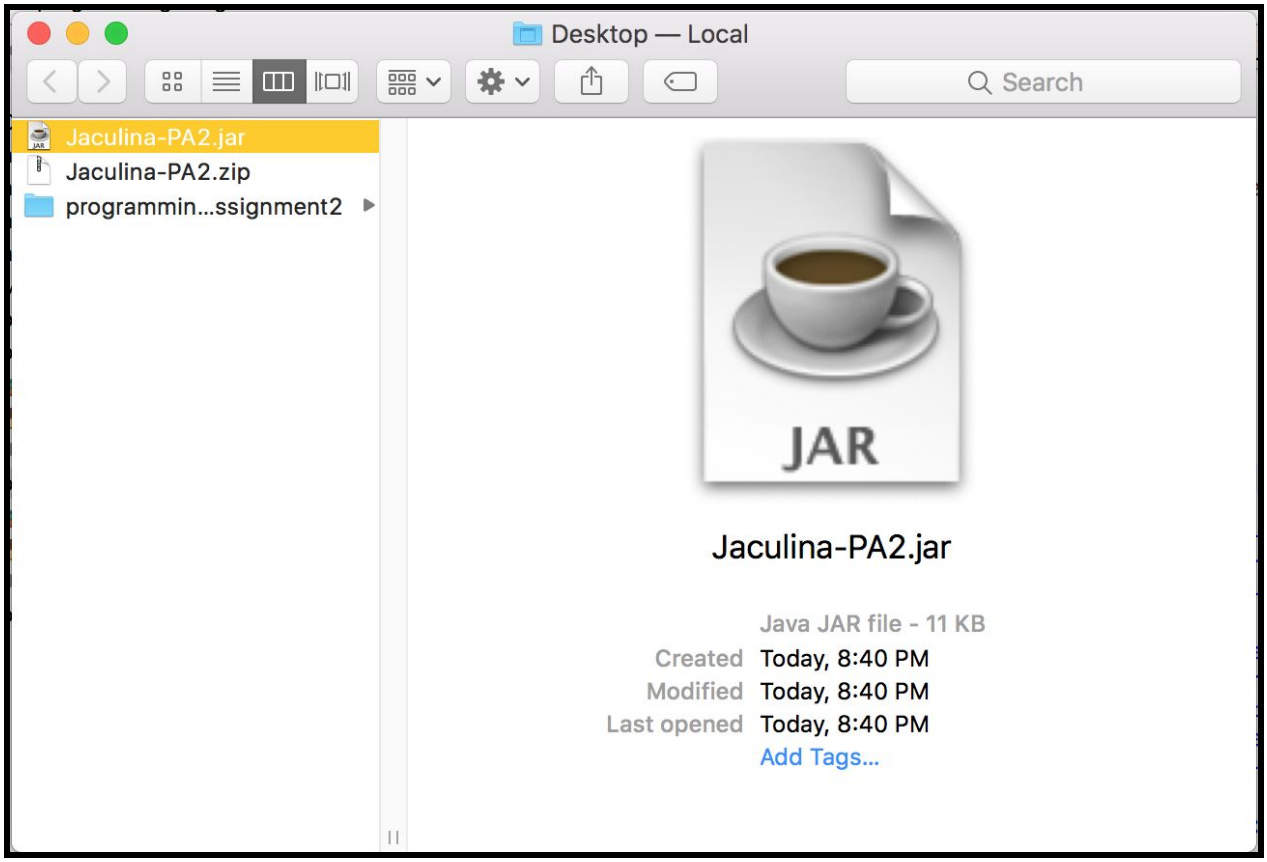
6. Select the class of the application entry point and choose Browse. Select the MainMethod in the project. Choose Ok.



7. Select Finish



8. Jar file (Jaculina-PA2.jar) should be saved on the Desktop



Step 3 How to Run the Application

To run this CPU application it will be easier to run it through the jar file.

Jar File

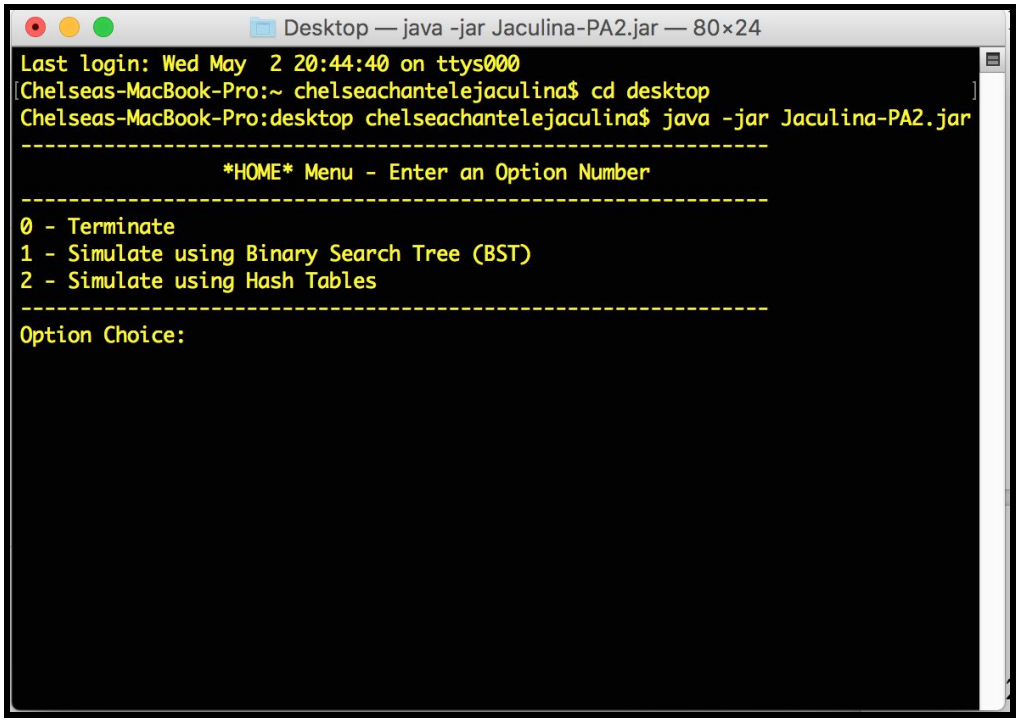
1. **Open Terminal or Command Prompt**

   Type in the command **cd desktop**

   Then type in **java -jar Jaculina-PA2.jar**

2. Output

The following home menu should display.



Step 4 Installation Needed

To install the application to your computer, you must have a Java IDE. It is recommended to use Eclipse IDE since the application has been developed on Eclipse Java IDE for Web Developers Version Oxygen.1a Release (4.7.1a). In addition, a MacBook that runs terminal or a Windows computer that runs command prompt.

Problems Encountered

There were many problems encountered when implementing the CPU scheduling system. The first problem was translating the Intro to Algorithms pseudocodes to Java source code. This was a major problem because when trying to declare the methods it was hard to wrap my head around on the type of parameters I needed to pass in. Additionally it was tough to implement chained-hash-insert, chained-hash-search, and chained-hash-delete since the pseudocode was based off of linked lists. This meant in creating a linked list class of my own. Another problem that I encountered was writing print methods for the hash table, node, process, and linked list class. Initially it was hard to conceptually see what I needed to print for the hash table. I then figured out that the hash table consisted of an array of linked lists, and within those linked lists contained nodes, and inside a node contains a process that includes a priority name and priority index.

Lessons Learned

Many lessons were learned in developing this CPU priority process simulator. The first and most important lesson included what binary search trees and hash tables exactly were. There are many applications in creating a CPU priority process simulator but it can be done efficiently with binary search trees or with hash tables. What I learned primarily about binary search trees is that binary search trees support many dynamic operations, including search, insert, delete, minimum, maximum, and successor. In binary search trees they give a O(lg n) running time when it comes to searching, inserting, and deleting a specific value of each node. Binary search trees are also good for sorting purposes. With hash tables I learned that they are great for searching and are also efficient when it comes to inserting and deleting a value. I learned how the chaining technique worked when collisions occurred and got to practice working with doubly linked lists.

I also learned the advantages of binary search trees and how the two approaches differed. Although hash table beats binary search tree for the basic dynamic operations, binary search trees have the advantage when it comes to sorting since the values can be printing in inorder traversal. Another edge that binary search trees have is that they have the operations to find the maximum, minimum, successor, and predecessor fast. The implementation of binary search trees was also easier to implement than hash tables.

What I also learned was how to create a user interface that contained multiple methods for the tester and the ability to make me work on formatting properly.

The last lesson that I learned from this project is that a CPU priority simulation can be implemented efficiently with hash tables, binary search trees, and also max heaps.