CS 251, Data Structures and Algorithms, Fall 2016

Project 4: Resource Allocations Queries

- **Due Date:** Friday, November 18, 11:59pm
- **Late Penalty**: 20% per day. A maximum of 3 late days (including slip days).
- **Skeleton Code**: Project4

Introduction

In this project you will develop a search-tree based technique to manage the assignment of jobs to machines. You are given N identical machines, each having maximum free space (available memory) of size M. The machines are numbered from 0 to N-1. Arriving jobs request a specified amount of space which must be assigned from one machine. If the request cannot be satisfied using the free space in a single machine, the job is discarded. If it can be satisfied, it is scheduled. Deletion queries remove jobs from the assigned machine and free up space for future use. You will answer a number of analytical questions and experimentally compare two job assignment strategies.

Overview

An arriving job is specified by a unique `jobid` and the necessary space to satisfy the job of size m. You need to determine whether a machine with sufficient free space is available. You are asked to consider and compare two strategies for assigning jobs to machines, the **min-space** strategy and the **min-job** strategy. We illustrate the two strategies using an example with N=4 machines and M=100. Assume 9 jobs have already been assigned as shown below.

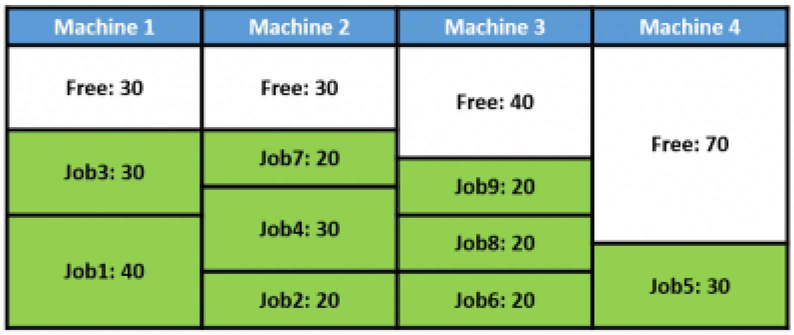

Figure 1: Example with 4 machines and 9 jobs

The min-space strategy assigns an incoming job to the machine that has the smallest available free space out of all machines with the necessary available space. Suppose a new job with jobid=10 and m=35 arrives. Machines 3 and 4 each have enough free space available for job 10. When using the min-space strategy, job 10 is assigned to machine 3 (because it has less free space than machine 4). Jobs will also be deleted when they complete. For example, deletion of job4 results in machine 2 having a remaining free space of size 60. The figure below shows the 4 machines after insertion of job10 (using the min-space strategy) and deletion of job4. Deletion behaves the same under either strategy. This assignments leaves a free space of size 5 available, as shown in the following image.

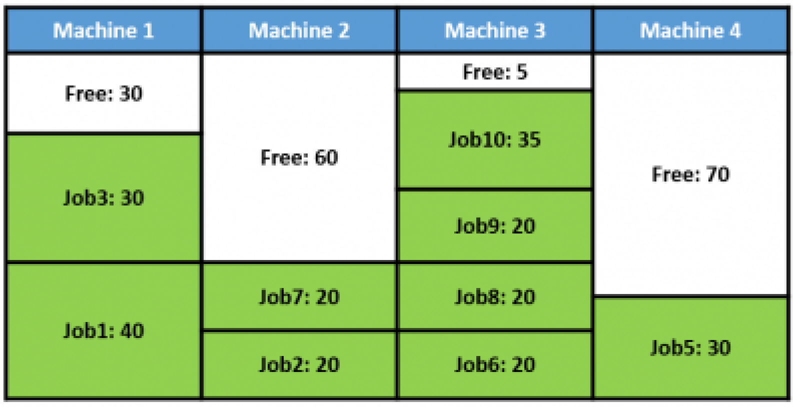

Figure 2: After removing Job4 and assigning Job10 using Min-Space Strategy

On the other hand, the min-job strategy assigns the job to the machine that has the fewest jobs already assigned out of all machines with the necessary available free space. Under the min-job strategy, job10 is assigned to machine 4 rather than machine 3 because machine 4 only has one assigned job when job10 is added. The figure below again shows the 4 machines in *figure 1* after insertion of job10 and deletion of job4, but the insertion follows the min-job strategy. As mentioned previously, deletion behaves the same under either strategy.

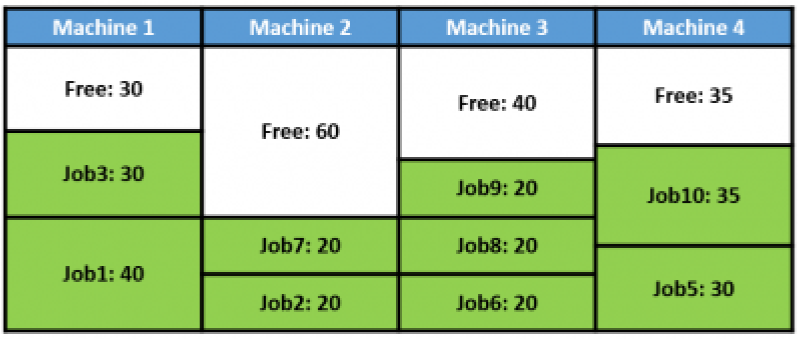

Figure 3: After removing Job4 and assigning Job10 using Min-Job strategy

The space available in the N machines will be maintained in a balanced search tree (rather than tables). Your main task is to use a red-black tree with additional entries to efficiently manage and assign free space under insertion and deletion of jobs using the min-space and min-job strategies. Information about current jobs will be stored in a hash table. The two data structures are described in the next section.

The Data Structures

You will be using two data structures:

1. A balanced search tree to store and manage the space available in the N machines
2. A hash table for keeping track of information about the jobs assigned to machines. Hashing is used to map the job IDs to some cell in the table, from index 0 to H-1 (where H is the size of the hash table).

Note that the two data structures do not mirror Figures 1-3 whose main purpose is to illustrate the problem and the two different job assignment strategies. In the balanced search tree, one node represents one machine. The node will have an entry representing how much free space the machine currently has, but the node will **not** record the assigned jobs. The job table maintains one entry per job and records the machine the job is assigned to. Both data structures are provided to you in the files `RedBlackBST.java` and `JobTable.java`. You are also provided the file `Node.java` which contains the definition of the Node class which comprises the entries in the balanced search tree. Your main task is to write new queries using the search tree data structure. The queries will use additional entries in the nodes maintained under insertion and deletion.

The class SearchTree in `SearchTree.java` contains two important class variables which you will use to interact with the search tree and hash table respectively:

- **Node root** - the root of the red black tree
- **JobTable jobs** - the reference to the hash table containing job associated information

The Search Tree

You are provided with an implementation of a left-leaning red black tree in the file `RedBlackBST.java`. The balanced search tree is used to assign jobs to machines, manage free space, and execute other queries. Each node in the search tree represents one of the N machines. At any point, the tree will consist of N nodes. Insertions and deletions are used to manage the tree as entries change (a change is a delete followed by an insert).

The key used by the search tree is the free space currently available in each machine. Initially, each one of the N nodes has a key of M, the maximum amount of free space. For any node u with key m, nodes having a key of m can be in the left as well as the right subtree of u. Assume node u represents machine k (i.e., `machineid=k`) and available space m. Assume u is the node of key m closest to the root. If there are multiple machines with key m, the search tree property applies to the machine id's. More specifically, the machines with key m which have `machineid<k` are in the left subtree of u; machines with key m which have `machineid>k` are in the right subtree of u.

A tree for N=11 and M=110 is shown below. The current available space is the black integer in each node; the `machineid` is the green integer inside each node. The tree contains two pairs of duplicate nodes (55 and 75). Machines 4 and 7 have m=75: machine 4 is closer to the root and machine 7 is in its right subtree. The nodes of the search tree will contain additional entries described later. Later figures will not explicitly show red and black nodes (they are only relevant to insertions and deletions, whose code is provided).

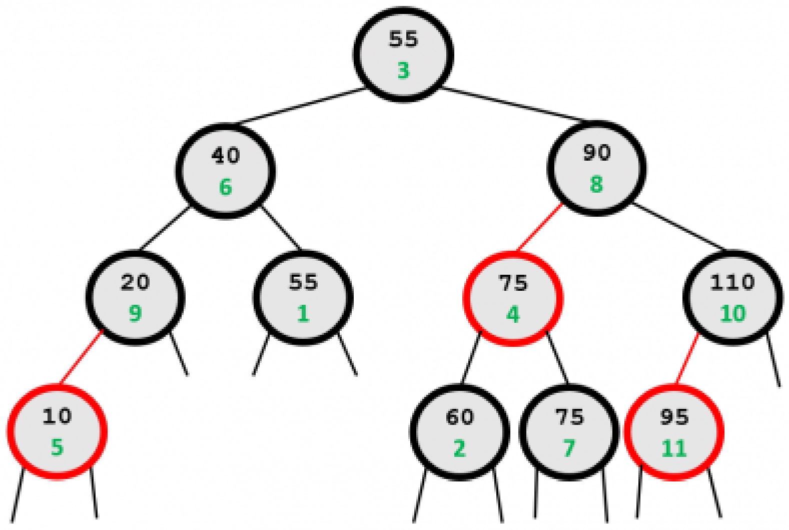

Figure 4: LL Red-Black Tree for N=11 and M=110
(free memory in black; machine number in green)

Don't alter the contents of the file RedBlackBST.java; it will be replaced during testing. You will need to interact with the red-black tree using the following two functions:

- **RedBlackBST.insert(Node root, Node u)**
  - This will insert the node u into the tree rooted by 'root'
- **RedBlackBST.delete(Node root, Node u)**
  - This will delete the node u from the tree rooted by 'root'
  - Make sure you create a deepcopy of u before you delete it; after deletion, node u will point to it's original right child using the code that is provided.

Nodes of the Search Tree

The N nodes in the search tree are defined in the file `Node.java`. You will interact with the nodes directly when adding or deleting a job and when carrying out a query on the tree. Adding and deleting jobs from a node will be done using the two void functions, addJob and removeJob in `Node.java` (both take only jobid as the parameter). In order to carry out queries, you will search for nodes in the tree starting from the root and advancing to the left or right children as necessary.

The nodes of the tree contain and maintain more fields than the search trees you have seen in class. The fields for a node u are:

- **int free** - entry free is the key of node u; free corresponds to the free space currently available in the machine represented by node u
  - This field will be used and updated when inserting and deleting jobs.
    - Addition of a new job will take up the free space m required by the job.
    - Deletion of a job will free up the free space m which was being used by the job.
  - The free field is used by both assignment strategies.
  - All nodes start with a free space of M before any jobs are added.
- **int id** - the unique integer `machineid` identifies the machine represented by node u
- **Node left** and **Node right** - the left and right children of node u
  - You will access the children of a node during the searches.
- **int numjobs** - the number of jobs currently assigned to machine id
  - This field will be used when assigning jobs under the min-job strategy.

- **Node minJobsNode** - the node in the subtree rooted at u (including u) with the fewest assigned jobs.
  - In the event of ties, the left-most node will be chosen as the minJobNode.
  - Immediate access to the specified node is used in the min-job strategy.
- **int size** - the number of nodes in the subtree rooted at node u
  - Field size is defined as seen in class. It will be used during the `count` query.

The Job Table

For the Job Table, a hash table will map a job's unique `jobid` to a job structure, which stores the size of that job as well as a reference to the node in the search tree corresponding to the machine which was assigned job `jobid`. The fully functional hash table is provided for you to use; you will interact with the JobTable using the functions provided in `JobTable.java`:

- **void addJob(int jobid, int size, Node machine)**: adds a new job to the JobTable; machine points to the Node object in the search tree corresponding to the machine which was assigned the job.
- **void deleteJob(int jobid)**: removes the job specified by jobid from JobTable.
- **int jobSize(int jobid)**: returns the amount of free space required by job `jobid`.
- **Node jobMachine(int jobid)**: returns the pointer to the Node of the machine which was assigned job `jobid`

Figure 5 illustrates the relationship between the job table and the search tree (only a subset of the jobs and nodes in the tree are shown). Depicted in the table are the two job fields: (i) the amount of space being used by the job and (ii) a reference to the node corresponding to the machine which has been assigned that job. The blue arrows indicate these references. For example, both jobs 101 and 370 have been assigned to machine 1. Since we have M=110 and machine 1 has free space of 30, there must be at least one other job assigned to machine 1 (not shown). If job 101 were to be removed, machine 1 would have its free space increase by 40 and the new free space at machine 1 would be 70.

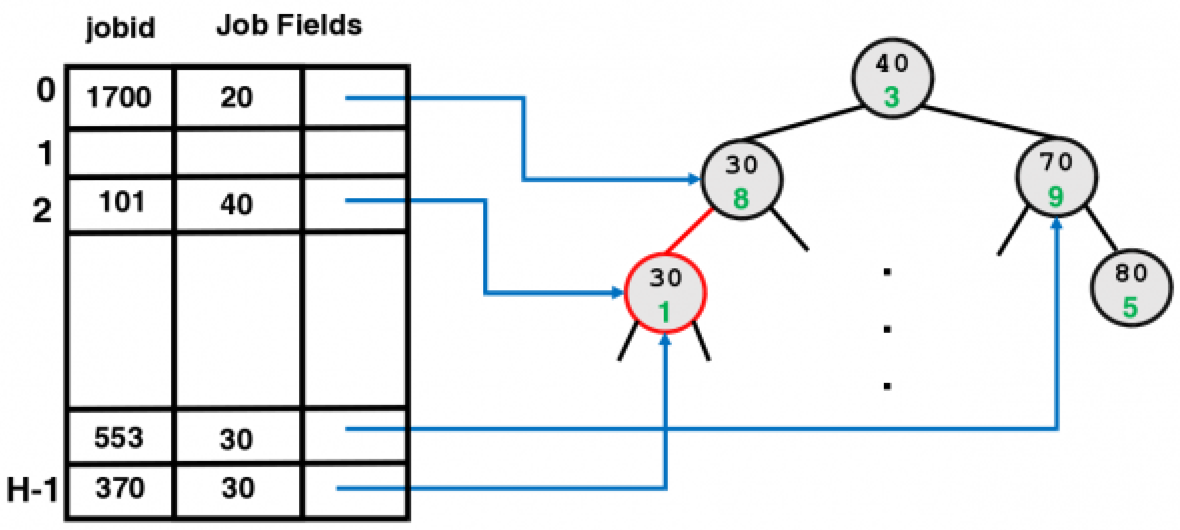

Figure 5: Job Table and Search Tree illustration

The Overall Strategy

The program receives a series of queries including job requests (for specified amounts of free space) and job deletions. When a job request arrives, first determine whether the job can be assigned to a machine using the chosen strategy (either min-space or min-job). In either case, if no machine has the necessary free space, the scheduling function will return -1. Otherwise, the job scheduling function adds the new job to the assigned machine, adds the new job to the JobTable (which makes the shown link between the job table and search tree), and returns the `machineid` of the machine which was assigned the job.

When a job deletion arrives, look up the job with the corresponding jobid in the JobTable to find its assigned machine. Then, update the search tree as follows: when updating the free space of the machine, you delete the node from the tree and insert it again after updating the values of the node appropriately. The code for tree insertion and deletion are given. The code automatically updates the fields of entries maintained in the nodes during additions and deletions.

Warmup: The Count Query

To get familiar with the tree structure used, you will implement the following count query:

- Given a required free space of size m, determine the number of machines that could be assigned to a job requesting size m.
- If count is zero, the request for a space of size m cannot be satisfied and the job is discarded.

Recall that entry free represents the available space in machines and is the key of the search tree. Implement the count query in O(log N) time using the `size` entries in the nodes. We suggest you review the rank and select queries presented in class before working on the count-query. Your implementation of the count query should use recursion and be based on the idea described below.

Starting at the root of the binary search tree, assume node u is the node being considered:

1. If node u has at least m free space, return the sum of the size-entry of the right child of u (all nodes in the subtree rooted at the right child represent machines with enough free space) plus 1 (for u itself) plus what is returned by the recursive call made on the left child of u.
2. If the node u does not have enough free space, no machine represented by nodes in the left subtree has enough free space. Make a recursive call on the right subtree of node u.

Min-Space Strategy

A job request for a space of size m arrives. We can use count query to determine whether it can be satisfied. In the min-space strategy, the job is assigned to the machine that has free space of at least m and its free space is as small as possible. As an example consider Figure 6: if m=35, the space should be allocated from machine 12 (which initially has free space of size 36).

The min-space strategy assigns space so that machines with greater free space remain available for future jobs with larger space requirements. How to handle ties is discussed in more detail later. The min-space method needs to run in O(log N) time. We next describe the idea you should be using.

The method starts at the root. As the search moves from the root towards a leaf, keep track of the node seen so far with minimum available space whose free space is at least m (it may be the one you assign to the job). We refer to this node as the **best-option node**. Assume we are currently at node u.

- If the request for space m **can be satisfied** by the machine represented by node u, proceed with the subtree rooted at the u's left child. There is no reason to explore the subtree at the right child of u. Also, as node u can satisfy the request, we may need to remember it (more explanation below).
- If the request for space m **cannot be satisfied** by the machine represented by node u, proceed with the subtree rooted at u's right child.

Consider Figure 6 and a job request of m=35. The root has free=55 and is thus the first best-option node. The search proceeds with the left child of the root. The left child has free=40 which becomes the new best-option node. The search moves to the left child, node u, with free=20. Node u represents machine 9 which does not have enough free space. Proceed with the right child of u where the search terminates at a leaf node with free=36. The links traversed by the search are shown in green.

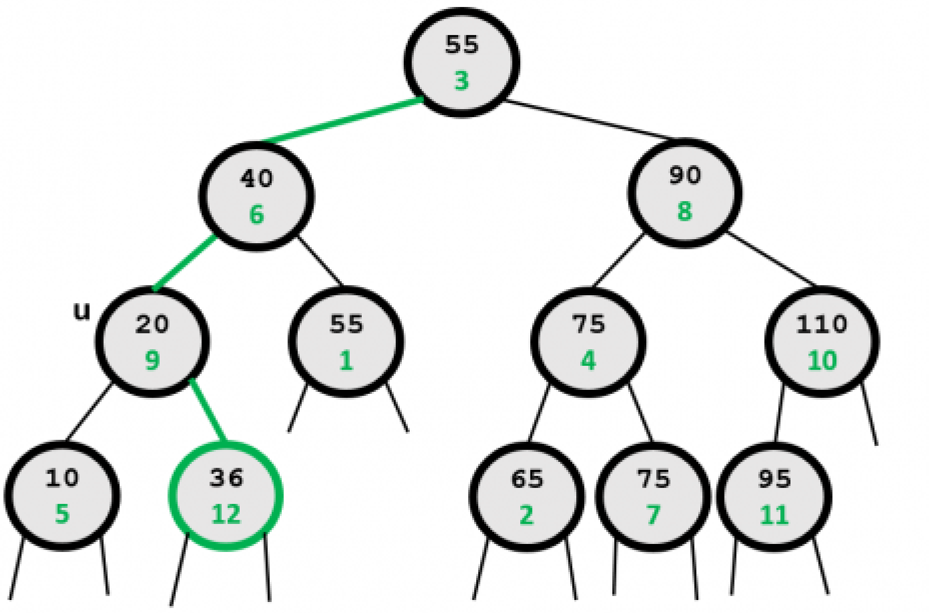

Figure 6: Searching for a machine for a job requiring space=35 using the min-space strategy (search path in green)

As described previously, deletion of a job is handled by first looking up the machine the job is assigned to using the JobTable. The corresponding machine should then be deleted from the search tree, updated to reflect the change in available space, and then immediately inserted back into the search tree. The job itself can then be removed from the JobTable. Figure 7 shows the tree from *figure 6* after the request for space of size 35 has been handled (machine 12 has now free = 1) and after deleting a job of size 15 from machine 2.

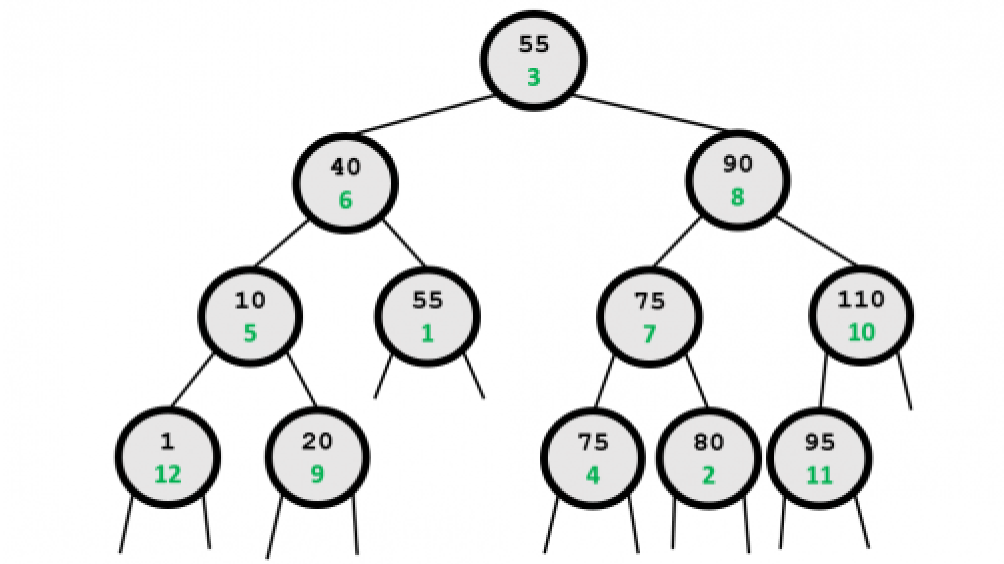

Figure 7: Search tree after assigning the job with size=35 to machine 12 and deleting a job with size=15 from machine 2

The case of identical keys in the search tree is handles as follows. Assume the best-option node is node u. As the search continues, another node, say node v, with the same amount of free space is encountered. At this point, node u **remains** the best-option node for the min-space strategy. **Always keep the node closest to the root as the best-option node.** In the event that that the search encounters a node with the exact amount of free space requested, the search terminates at this node.

Figure 8 illustrates how identical keys are handled. Assume m=63. The root node does not have enough space available and its right child, node u, with free=90 is the first best-option node. The search proceeds to v and node v becomes the new best-option node (with free=75). The search proceeds in the left subtree to node w. Node w also has free=75, but the best-option node is not updated. The request for m=63 is satisfied by node v.

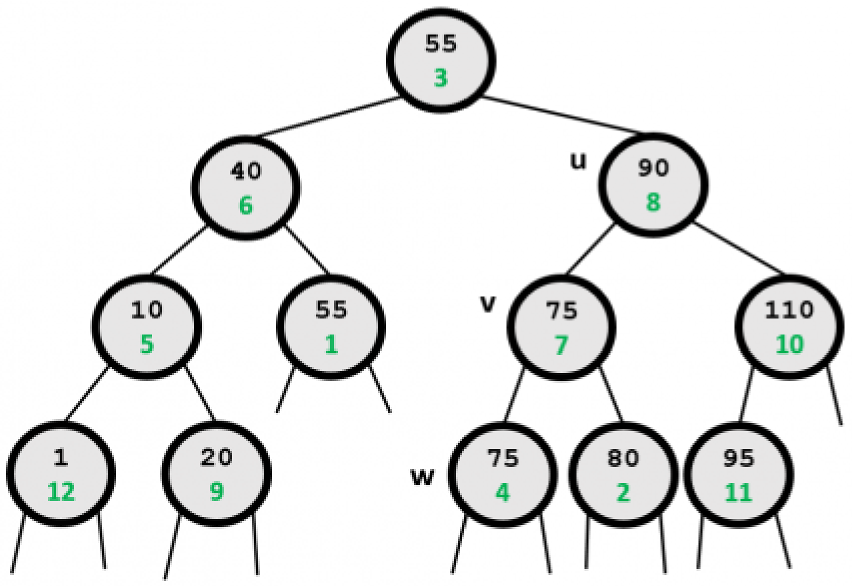

Figure 8: Handing a request for m=63 using Min-Space. In case of ties, the first node encountered is used.

Min-Job Strategy

Running many jobs on a machine can degrade performance and individual jobs may takes longer to complete. In the min-job strategy, a job requesting a space of size m is scheduled on a machine that has at least m free space **and** the number of jobs already assigned is a minimum (over all qualifying machines). Determining the machine receiving the new job should again take O(log N) time.

To execute the query in O(log N) time, use two of the entries maintained in a node: numJobs and minJobsNode. For node u:

- numJobs represents the number of jobs assigned to the machine corresponding to node u
- minJobsNode is the node in the subtree rooted at u which has the minimum number of jobs assigned. The minJobsNode can be in the left or the right subtree of node u.

Consider Figure 9: the black and green numbers in a node correspond again to the free space and `machineid` of the nodes; the red numbers represent the number of jobs assigned to the node. The minJobsNode entries are shown for three nodes, the root and both children of the root. The minJobsNode entry of the root points to a node not having any jobs assigned.

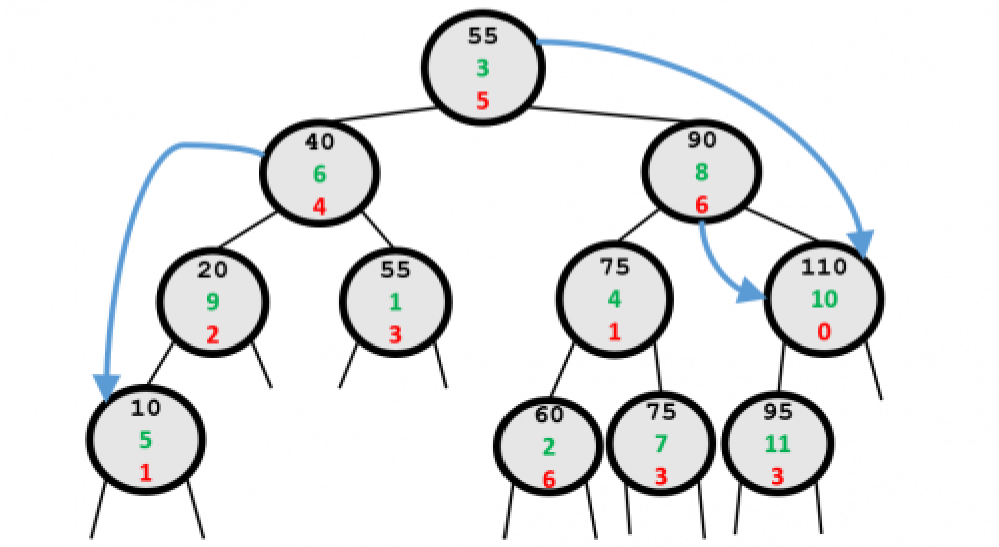

Figure 9: Search tree with entries used by Min-job strategy. Black integer=free space, green = processor id, red = # of jobs assigned. MinJobNodes links shown for three nodes in blue

How do we find a machine with available space of at least m and having a minimum number of jobs assigned? We search down a path starting at the root and we again maintain a best-option node, which is initially null. Assume the search is at a node u:

1. If node u cannot satisfy the request, the left subtree of u does not need to be checked (since all nodes in the left subtree will have no more free space than that of u). Continue with the right child of u.
2. Assume node u can satisfy the request.
    1. First, find and compare the number of jobs assigned to u itself with the number of jobs assigned to the minJobNode of the right subtree of node u (all nodes in the right subtree satisfy the request). Let w be the node with fewer assigned jobs of the two.
    2. Next, compare the number of jobs in w with the current best-option node and update the best-option node as appropriate (node w may be the new best-option node).

3. We still have to check whether a node in the left subtree of u provides a better machine. Hence, the search continues with recursion on the left child of node u.

Once we have reached a (null) leaf node, the best-option for the new job has either been identified, or if the best-option is still null, no machine had enough free space to satisfy the job request.

How do we break ties? If we have a tie on the number of jobs assigned, we give preference to the node furthest to the left in the tree.

- If two machines have the same number of jobs, the machine with smaller free space is selected as the best option.
- If two machines have the same number of jobs and the same free space, the machine with the smaller `machineid` will be selected as the best option.

Figure 10 illustrates the beginning stages of this process for some job requesting a free space of m<180. The blue links represent the MinJobNode links used during the initial stages of the search.

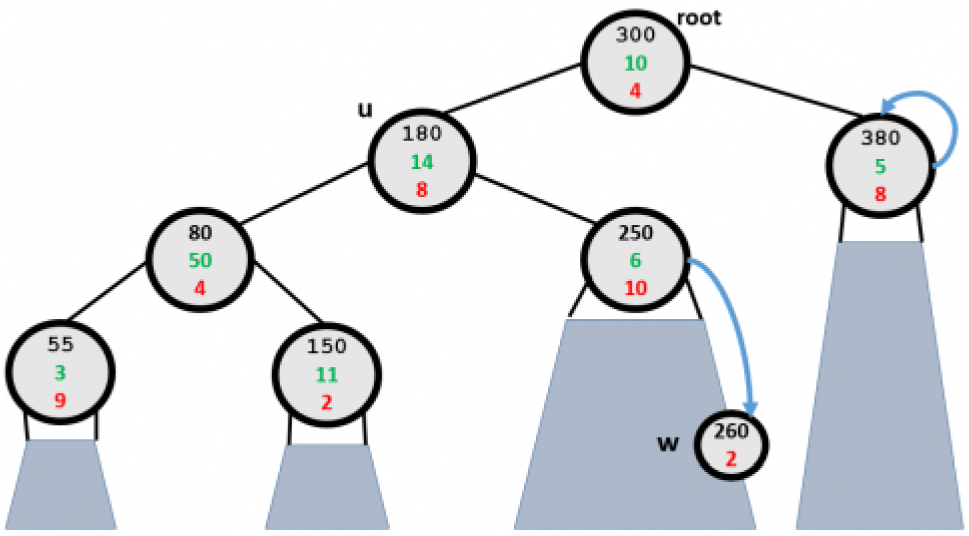

Figure 10: Min-Jobs strategy for m<180.
Black integer = free space, green = processor id, red = # of jobs assigned.

In the partial search shown below, the root is large enough to satisfy the request, and is found to have fewer assigned jobs than any node to the right, so the root is the first best option. The search then recurses on the left subtree (root at node u), where the best-option node will be updated to be node w. The search will continue through recursion down the appropriate subtrees until a leaf has been encountered.

Implementing Queries and Operations with the Search Tree

Tasks

You are provided with a red-black tree implementation (in `RedBlackBST.java`) that implements insert, search, and delete operations and updates the additional entries size and minJobsNode as the tree changes. We encourage you to take a look at the code. However, do NOT edit the code. What you submit will be run with the original version of the red-black tree code. You will need to implement the following methods in `SearchTree.java`:

- **int count(int jobsize)**:
  - Return the total number of machines which can handle a job of size `jobsize`. Method must use recursion.
- **int scheduleJobMinSpace(int jobid, int jobsize)**:
  - Find the machine that can accommodate the job requesting space of size `jobsize` using the Min-Space strategy. The method can be recursive or iterative.
  - Return the `machineid` of the machine which is assigned the job.
  - If no machine has enough space available, the request is discarded. You will return -1.
- **int scheduleJobMinJob(int jobid, int jobsize)**:
  - Find the machine that can accommodate the job requesting space of size `jobsize` using the Min-Job strategy. The method can be recursive or iterative.
  - Return the `machineid` of the machine which is assigned the job.
  - If no machine has enough space available, the request is discarded. You will return -1.
- **void releaseJob(int jobid)**:
  - Remove the job from the job table.
  - Update the available free space of the machine running this job and the search tree.

All operations and queries need to run in O(log N) time using the approaches outlined earlier.

Input Format

Input queries having the following format:

1. Scheduling a new job: `S jobID free_space`
2. Release: `R jobID`
3. Count: `C minFreeSpace`
4. Machine Utilization (used in the experiments): `M`

Examples:

- Schedule job XYZ requiring a space of 200: S XYZ 200
- Release job 37: R 37
- Count the number of machines with free space at least 30: C 30

Testing Your Implementation

Two test engines are provided in the project skeleton directory:

- TestTree.java takes input queries and outputs information based on your configuration.
- QueryGenerator.java creates input queries based on your configuration. Use the query generator to test the scheduling strategies and count query.

TestTree.java

This test engine is designed to help you test your algorithm. To compile it, use the following command:

```
javac TestTree.java
```

To execute it, use the following command:

```
java TestTree [options] [< query input]
```

Here is the list of options:

1. -h or -help: list all the options and their format
2. -m [N] [M]: Allocate N machines, each with a maximum free space of size M.
3. -s [strategy]: Scheduling strategy used, either minSpace or minJob

Notice that you must have “-m N M” as an argument in order to run the program, as it sets N and M and creates the initial search tree with N identical machines.

When the test engine receives a count query, it prints out your solution. When it receives a utility measure query, it prints out the measurement of your machine's load status.

QueryGenerator.java

This program is designed to generate random queries for you to test your programs. To compile it, execute the following command:

```
javac QueryGenerator.java
```

To execute it, use the following command:

```
java QueryGenerator [options] [> output file]
```

Here is the list of options:

1. -h or -help: list all the options and their format
2. -s [n] [maxload] Schedule n jobs, each job takes up to maxload space. Loads are created randomly from 1 to maxload. Note that maxload can be larger than M.
3. -c Enable generating counting query. When this flag is on, after every 0.1*n schedule queries, a counting query is generated. A random argument M is generated from [1, 2M].
4. -r Enable random release query. When this flag is on, a random release query is created with certain probability.
5. -m Enable machine utilization query. When this flag is on, after every 0.1*n schedule queries, a machine utilization query is executed.

Note that the -s [n] [maxload] flag must be passed to generate meaningful input.

Comparing the two Assignment Strategies

Assume you are given a sequence of operations consisting of S (schedule), R (release), and C (count). You execute the sequence using the Min-Space strategy as well as the Min-Job strategy. Which strategy does better? Neither strategy guarantees that the best decision is made. At the time that the decision is made, you don't know which assignment is better because you don't know what future operations will occur or how much free space a future job will need.

A machine utilization query has been written for you to use (it is measureUtility() in SearchTree.java). It uses two metrics, throughput and fairness. Do not change this method. The parameters throughput and fairness are described below.

Given a sequence of operations containing R job requests with r of them scheduled on machines (i.e., R-r job requests are discarded), we define **throughput** = $\frac{r}{R}$. A throughput of 1 indicates that all job requests were able to be scheduled.

Fairness measures the load on machines in terms of number of jobs assigned. Assume at the time the measurement is taken, the N machines contain a total of r jobs. Then, an ideal load distribution assigns every machine an equal number of jobs: r/N. We define **fairness** = $\frac{medianLoad}{ideal}$, where medianLoad is the median of the numJobs-entries in the search tree.

The best load distribution has a fairness=1. The fairness value can be smaller or larger than 1. For example, for N=5 and numJobs-entries 1, 1, 1, 1, 11, the ideal load would assign every machine ideal=3 jobs and fairness is 1/3. The numJobs-entries 1, 1, 11, 11, 11 give a fairness >1.

You can use the Query Generator to generate queries to test your code. Make sure you enable “-m” flag when you are generating the input file. Then run TestTree.java and observe the result from your input.

Analysis Questions

Your **typed** report must answer each question briefly and clearly. Data presented in any experimental analysis should be presented in tables or charts. All material must be readable and clear and prepared in a professional manner. For charts and tables that can easily be produced with Excel (or equivalent software), handwritten material **will not** be accepted.

Experimental Analysis

You are required to report the results based on the input data provided in the skeleton. (3k.txt and Identical.txt)

`3k.txt`: This file contains 3000 query schedule requests with additional release queries, measure queries, and count queries. This data set follows real-life 80-20 rule. 80% of the jobs requires few amount of space while the rest 20% requires a lot.

`Identical.txt`: This file contains many schedule queries with required space = 1(for the simplicity of testing). It represent the fact some machines are designed for a specific known tasks.

Fill in the following table and generate a plot based on the input files we provide for each strategy, in total you should fill 8 tables (2 strategies * 2 tables * 2 files) and generate plots for the corresponding tables. You can merge tables into plots as long as the visualization is clear and understandable.

Explain your observations for the min-space and min-job strategies with respect to utility and fairness.

Which of the two job scheduling strategies (min-space and min-job) consistently has better values in terms of throughput? What about in terms of fairness? Explain what you observed in experiments. Can you make any conclusions?

Number of Machines	**Machine Free Space**	**Throughput**	**Fairness**
40	100		
40	200		
40	300		
40	400		
40	500		
40	600		
40	700		
40	1500		
40	3000		

Number Of Machines	**Machine Free Space**	**Thoroughput**	**Fairness**
40	100		
60	100		
80	100		
100	100		
150	100		
300	100		
350	100		
400	100		
600	100		
1000	100		
2000	100		
4000	100		

Analytical Questions

1. All three queries (count, scheduleJobMinSpace, and scheduleJobMinJob) should run in O(log N) time. Explain how and why your implementation of each query achieves O(log N) time. Be precise.
2. Give an example consisting of at least 10 jobs and at least 4 machines for which the min-space strategy discards at least 2 jobs, while there exists an assignment that schedules all 10 jobs. You set the value of M. Explain your example. Give and explain the assignment made by the min-space strategy as well as the assignment scheduling all jobs.
3. Give an example consisting of at least 10 jobs and at least 4 machines for which the min-job strategy discards at least 2 jobs, while there exists an assignment that schedules all 10 jobs. You set the value of M. Explain your example. Give and explain the assignment made by the min-job strategy as well as the assignment scheduling all jobs.
4. Give an example of at least 20 jobs scheduled on 20 machines with M=50 for which the fairness is larger than 1.8. Explain your answer. You do not need to explain how the jobs were scheduled.
5. Given the data structures used, describe how a machine represented by node u in the search tree can determine the jobs (Id and size) assigned to it. What is the time complexity of your approach? Do not code this task.

Grading

Project is 100 points.

- **10** count
- **20** scheduleJobMinSpace
- **20** scheduleJobMinJob
- **10** releaseJob
- **30** Report: experimental results; discussion of experimental results and responses to analytical questions
- **10** Code Quality

Submission Instructions

You are responsible for ensuring your submission meets the requirements. If not, points will be deducted from code quality allocation. You project will not be graded before the deadline. Your score on Vocareum will remain 0 until we grade your project.

Code

- Submit only the source files of your implementation. For this lab, your entire implementation should be in SearchTree.java.
  - You can make additional .java files if needed.
  - No .jar files or .class files should be submitted.
  - If a precompiled binary file is submitted without the .java file, you will receive a 0.
- In `SearchTree.java` (and any additional `.java` files you make), you must correctly fill your name, login ID, your project completion date, and your PSO section number.
- All files other than `SearchTree.java` provided in the skeleton code will be replaced during testing.
  - Your code must compile to be graded by the autograder. Be sure to test your code before submitting.
  - Make sure to test your implementation using the files provided in the skeleton code.
- Your should submit a single folder named as “project4” containing the followings
  - SearchTree.java (with measureUtility unchanged)
  - Any extra .java files you created or used. You are allowed to use the standard Java libraries, but nothing from the Princeton library besides the files used and adapted and in the skeleton code.
  - Your report in one .pdf file. Maximum size allowed is 10MB.

Report

- File name: `<username>.pdf`
- Your report must be typed.
  - Use LaTeX, OpenOffice Write, or Microsoft Word.
  - State your name on top of the report.
  - Handwritten reports, including digitized versions, will not be accepted.
  - Figures and diagrams that **cannot** be easily generated may be hand drawn and digitized for inclusion into the .pdf file representing your report. Note that tables and graphs that can easily be generated by Excel or similar **cannot** be hand drawn!
- Your report must be submitted as a PDF.
  - Other file formats will not be accepted.

- Your report must be submitted using Vocareum.
  - Submissions via any other means, to any other venue, will not be accepted.

All work must be submitted in Vocareum following the submission instructions at Vocareum Submission Guidelines. Please be aware of the following:

- Only your final submission will be graded.
- A submission **after** the deadline is considered late. DON'T wait until the last second to submit.
- If you are submitting late and want to use your slip days, you need to select “use slip days” when submitting on Vocareum.
- For project 4, submissions and late policies are described on the top along with the project description.