```java
import java.util.LinkedList;

public class MemoryManager
{
    // For the freeSpaceTable
    class FreeSpace
    {
        int start;
        int size;

        FreeSpace (int _start, int _size)
        {
            start = _start;
            size = _size;
        }
    }

    /**
     * VARIABLES***************************************************************
     */
    final int MEMSIZE = 100;
    LinkedList<FreeSpace> freeSpaceTable;
    LinkedList<Integer> jobsInMemory;
    LinkedList<Integer> unswappedQueue;
    LinkedList<Integer> swappedQueue;
    LinkedList<Integer> blockedQueue;
    LinkedList<Integer> terminated;

    /**
     * CONSTRUCTOR*************************************************************
     */
    MemoryManager ()
    {
        // Initial freeSpace is all of memory
        FreeSpace empty = new FreeSpace (0, MEMSIZE);
        freeSpaceTable = new LinkedList<FreeSpace>();
        freeSpaceTable.add(empty);
        // Keeps track of jobs in memory and their base addresses
        jobsInMemory = new LinkedList<Integer>();
        unswappedQueue = new LinkedList<Integer>();
        swappedQueue = new LinkedList<Integer>();
        blockedQueue = new LinkedList<Integer>();
        terminated = new LinkedList<Integer>();
    }

    /**
     * PRIVATE METHODS*********************************************************
     */

    /**
     * Adds new job to the correct queue
     * @param jobID [description]
     */
    void addToQueues (int jobID)
    {
        if (JobTable.getAddress(jobID) == -1) {
            blockedQueue.remove((Integer)jobID);
            unswappedQueue.remove((Integer)jobID);
            swappedQueue.remove((Integer)jobID);
            if (JobTable.isBlocked(jobID)) {
                blockedQueue.add(jobID);
            }
            else if (JobTable.getSwapped(jobID)) {
                swappedQueue.add(jobID);
            }
            else {
                unswappedQueue.add(jobID);
            }
        }
    }

    /**
     * Finds a job from queues to add to memory
     * @return jobID of job to add
     */
    int find ()
    {
        int swapInJob = -1;
        // Send another job
        // System.out.print("-MemoryManager attempts to add another job to memory");

        // See if we can find free space
        swapInJob = findFreeSpace();
        // If freespace is found
        if (swapInJob != -1) {
            // Update address in jobTable
            // System.out.println("--" + swapInJob + " added and sent to swapper");
        }
        else {
            // System.out.println("--No Space found");
        }

        return swapInJob;
    }

    /**
     * Checks for availability in freeSpaceTable
     * If found, adds job to jobsInMemory, updates freeSpaceTable
     * Else, add jobs to memQueue
     * @param job to be added
     * @return  jobID of job which space was found
     */
    int findFreeSpace ()
    {
        // set initial address/job to invalid
        int jobID = -1;
        // Checks unswapped first
        if (!unswappedQueue.isEmpty()) {
            jobID = iterateFreeSpace(unswappedQueue);
        }
        // Then swapped, but only if unswapped is empty
        else if (!swappedQueue.isEmpty()) {
            jobID = iterateFreeSpace(swappedQueue);
        }
        // Then blocked, but only if both unswapped & swapped are empty
        else if (!blockedQueue.isEmpty()) {
            jobID = iterateFreeSpace(blockedQueue);
        }
        return jobID;
    }

    /**
     * Given a list of freespaces, checks if there is freespace for any
     * pending jobs
     * @param  queue LinkedList of freespace
     * @return       the jobID of job which fits
     */
    int iterateFreeSpace (LinkedList<Integer> queue)
    {
        int jobID = -1;
        int address = -1;
        for (int j = 0; j < queue.size(); j++) {
            // Gets next element of queue
            int jobSize = JobTable.getSize(queue.get(j));
            // System.out.println("--Checking for " + jobSize
            //  + " free space...");

            // Checks freeSpaceTable for first available memory location
            FreeSpace iterator;
            for (int i = 0; i < freeSpaceTable.size(); i++) {
                iterator = freeSpaceTable.get(i);
                // System.out.println("---FreeSpace : Address=" +
                //  iterator.start + " Size=" + iterator.size);

                // If the space will hold new job
                if (iterator.size >= jobSize) {
                    if (jobSize > 40) {
                        if (iterator.start + jobSize > 60 &&
                            iterator.start < 50) {
                            break;
                        }
                    }
                    address = iterator.start;
                    iterator.start = iterator.start + jobSize;
                    iterator.size = iterator.size - jobSize;
                    // System.out.println("----Fit success");

                    FreeSpace newSpace =
                        new FreeSpace(iterator.start, iterator.size);

                    freeSpaceTable.remove(i);
                    if (iterator.size != 0) {
                        freeSpaceTable.add(i, newSpace);
                        // System.out.println("----New : Address=" + newSpace.start + " Size=" +
newSpace.size);
                    }
                    else {
                        // System.out.println("----Used entire space");
                    }
                    jobID = queue.get(j);
                    JobTable.setAddress(jobID, address);
                queue.remove((Integer)jobID);
                break;
            }
            // If the space is too small
            else {
                // System.out.println("----Too small");
            }
        }
        // Checks whether freespace was found for job and ends iterative check
        if (address != -1) {
            break;
        }
    }
    return jobID;
}

/**
 * PUBLIC METHODS**********************************************************
 */

/**
 * adds new job to memory from id number
 * checks to see if space is available
 * Sets address in job table
 * returns true if space is found and swapper should run
 * false if otherwise
 * @param  idNum job to add into memory
 * @return       jobID of new job to swap into memory
 */
public int add (int jobID)
{
    // If the job is not in memory or in the queue already & is valid
    if (jobID != -1 && !jobsInMemory.contains((Integer)jobID)) {
        // Sets swap direction to-Memory
        JobTable.setDirection(jobID, 0);
        // System.out.println ("-MemoryManager adds job " +
        //  jobID + " to queue");
        addToQueues(jobID);

        // With new element in memQueue, attemp  ts to find space
        jobID = findFreeSpace();
        return jobID;
    }
    else {
        return -1;
    }
}

/**
 * Adds job to jobsInMemory
 * @param jobID [description]
 */
public void addToMemory (int jobID)
{
    if (jobID != -1) {
        //memQueue.remove((Integer)jobID);
        jobsInMemory.add(jobID);
```

```java
        }
    }

    /**
     * Allocates jobs memory to freespace table, will append current
     * freespace if they are contiguous
     * Removes job from jobsInMemory
     * @param jobID jobID of job to be swapped
     */
    public void free (int jobID)
    {
        if (jobID != -1 && !JobTable.doingIO(jobID)) {
            // System.out.println("-MemoryManager begins to free job " + jobID + " with " +
JobTable.getTimeLeft(jobID) + " left");

            // Variables needed in iteration
            // Iterates across freespaces in mem
            FreeSpace iterator;
            // Used when checking to append between 2 freespaces
            FreeSpace iterator2;
            // Will replace current freespace if appended, or be added if no
            // appending occurs
            FreeSpace newSpace;
            Job job = JobTable.returnJob(jobID);

            // If the just freed job still has CPU time remaining,
            // then add it back into the memqueue
            if (JobTable.getTimeLeft(jobID) > 0 &&
                !JobTable.isTerminated(jobID) &&
                    !JobTable.isBlocked(jobID)) {
                JobTable.setDirection(jobID, 0);
                // System.out.println("ADDEDTOQUEUS");
                addToQueues(jobID);
            }

            // Free the space
            // First check to see if current freespace can be appended to
            boolean append = false;
            for (int i = 0; i < freeSpaceTable.size(); i++) {
                iterator = freeSpaceTable.get(i);
                // Status print
                // System.out.println("--Job to Free start=" + job.address +
                //  " end=" + (job.size + job.address - 1));
                // System.out.println("--Iterator start=" + iterator.start +
                //  " end=" + (iterator.size+iterator.start - 1));

                // Check to see if perfect fit between freespaces
                // If freespace does not start at zero && there is
                // another freespace
                if ((i + 1) < freeSpaceTable.size()) {
                    iterator2 = freeSpaceTable.get(i+1);
                    // If the boundaries match on both sides
                    if (job.address == (iterator.start + iterator.size) &&
                       (job.address + job.size) == iterator2.start) {
                        // Details to print
                        // System.out.println("--Freespace appended btw");
                        // System.out.println("--Existing1: Address=" +
                //  iterator.start + " Size=" + iterator.size);
                // System.out.println("--Addition : Address=" +
                //  job.address + " Size=" + job.size);
                // System.out.println("--Existing2: Address=" +
                //  iterator2.start + " Size=" + iterator2.size);
                // Updates iterator to new size
                iterator.size =
                    iterator.size + job.size + iterator2.size;
                // System.out.println("--New      : Address=" +
                //  iterator.start + " Size=" + iterator.size);
                newSpace =
                    new FreeSpace(iterator.start, iterator.size);

                // Replaces 2 iterators with newSpace
                freeSpaceTable.remove(i);
                freeSpaceTable.remove(i);
                freeSpaceTable.add(i, newSpace);

                append = true;
                break;
            }
        }
        // If freedspace ends at existing freespace
        if (job.address == (iterator.start + iterator.size)) {
            // Details to print
            // System.out.println("--Freespace appended to end");
            // System.out.println("--Addition : Address=" +
            //  job.address + " Size=" + job.size);
            // System.out.println("--Existing : Address=" +
            //  iterator.start + " Size=" + iterator.size);
            // Updates iterator to new size
            iterator.size = iterator.size + job.size;
            // System.out.println("--New      : Address=" +
            //  iterator.start + " Size=" + iterator.size);
            newSpace =
                new FreeSpace(iterator.start, iterator.size);
            // Replaces iterator with newSpace
            freeSpaceTable.remove(i);
            freeSpaceTable.add(i, newSpace);

            append = true;
            break;
        }
        // If freedspace starts at existing freespace
        if ((job.address + job.size) == iterator.start) {
            // Details to print
            // System.out.println("--Freespace appended to start");
            // System.out.println("--Existing : Address=" +
            //  iterator.start + " Size=" + iterator.size);
            // System.out.println("--Addition : Address=" +
            //  job.address + " Size=" + job.size);
            // Updates iterator to new size
            iterator.start = job.address;
            iterator.size = iterator.size + job.size;
            // System.out.println("--New      : Address=" +
            //  iterator.start + " Size=" + iterator.size);
            newSpace = iterator;
```

```java
                // Replaces iterator with newSpace
                freeSpaceTable.remove(i);
                freeSpaceTable.add(i, newSpace);

                append = true;
                break;
            }
        }
        // If the space couldn't be appended, need to add into correct location
        if (!append) {
            // System.out.println("--New freespace added : " +
            //  job.address + ", " + job.size);
            newSpace = new FreeSpace(job.address, job.size);
            for (int i = 0; i < freeSpaceTable.size(); i++) {
                iterator = freeSpaceTable.get(i);
                if (i == 0 && newSpace.start < iterator.start) {
                    // System.out.println("---At "+ i);
                    freeSpaceTable.add(i, newSpace);
                    break;
                }
                if ( (i+1) < freeSpaceTable.size()) {
                    iterator2 = freeSpaceTable.get(i+1);
                    if (iterator.start < newSpace.start
                        && newSpace.start < iterator2.start) {
                        // System.out.println("---At "+ (i+1));
                        freeSpaceTable.add(i+1, newSpace);
                        break;
                    }
                }
                if ( i == freeSpaceTable.size()-1) {
                    // System.out.println("---At end");
                    freeSpaceTable.add(newSpace);
                    break;
                }
            }
            if (freeSpaceTable.isEmpty()) {
                freeSpaceTable.add(newSpace);
            }
        }
        // Removes freed job from memory & clears address in jobTable
        jobsInMemory.remove((Integer) jobID);
        JobTable.clearAddress(jobID);
    }
}

/**
 * Frees memory of terminated jobs if all I/O done
 */
public void freeTerminated()
{
    for (int i = 0; i < terminated.size(); i++) {
        if (JobTable.getIO(terminated.get(i)) == 0) {
            free(terminated.remove(i));
        }
    }
}
```

```java
/**
 * Adds to list of jobs terminated with memory needing to be freed
 * @param jobID [description]
 */
public void newTerminated(int jobID)
{
    if (jobID != -1) {
        if (JobTable.getIO(jobID) > 0 || JobTable.doingIO(jobID)) {
            terminated.add(jobID);
        }
        else {
            free(jobID);
        }
    }
}

/**
 * Prints the status of freeSpace and the jobs in memory
 */
public void print()
{
    // // System.out.println("-Memory Report");
    // FreeSpace iterator;
    // for (int i = 0; i < freeSpaceTable.size(); i++)
    // {
    //  iterator = freeSpaceTable.get(i);
    //  System.out.println("--FreeSpace " + i + " : Address=" +
    //      iterator.start + " Size=" + iterator.size);
    // }
    // System.out.println("--Jobs waiting for memory: ");
    // System.out.print("---Unswapped : ");
    // for (int i = 0; i < unswappedQueue.size(); i++)
    // {
    //  System.out.print(unswappedQueue.get(i) + ", ");
    // }
    // System.out.println("");
    // System.out.print("---Swapped : ");
    // for (int i = 0; i < swappedQueue.size(); i++)
    // {
    //  System.out.print(swappedQueue.get(i) + ", ");
    // }
    // System.out.println("");
    // System.out.print("---Blocked : ");
    // for (int i = 0; i < blockedQueue.size(); i++)
    // {
    //  System.out.print(blockedQueue.get(i) + ", ");
    // }
    // System.out.println("");
    // System.out.print("--Jobs in memory: ");
    // for (int i = 0; i < jobsInMemory.size(); i++)
    // {
    //  System.out.print(jobsInMemory.get(i) + ", ");
    // }
    // System.out.println("");
}

/**
```

```java
     * Returns positive if there are not enough running jobs in memory
     * @return boolean whether it is wise to swap a job out
     */
    public boolean smartSwap ()
    {
        int blockedCount = 0;
        for (int i = 0; i < jobsInMemory.size(); i++) {
            if (JobTable.isBlocked(jobsInMemory.get(i))) {
                blockedCount++;
            }
        }
        if (blockedCount > 0) {
            return true;
        }
        else {
            return false;
        }
    }
}
```