CS 2223 D23 Term. Homework 2

Homework Instructions

- This homework is to be completed individually. If you have any questions as to what constitutes improper behavior, review the examples as I have posted online http://web.cs.wpi.edu/~heineman/html/teaching_/cs2223/d23/#policies
- Due Date for this assignment is **on Canvas**. Homework submissions received after 6PM receive a 25% late penalty. Extension is good for 48 hours.
- Submit your assignments electronically using the canvas site for CS2223. Login to canvas.wpi.edu and locate HW2. You must submit a single ZIP file that contains all of your code as well as the written answers to the assignment.
- All of your Java classes must be defined in a package, such as USERID.hw2, where USERID is your login name

Questions

1. Mathematical Analysis [20 points] (1 bonus point)
2. Instrumentation and Tournament algorithm [10 points]
3. Data Type Exercise [70 points] (2 bonus point)

Getting Started

Copy the following files into your USERID.hw2 package:

- **`algs.hw2.Analysis`**
- **`algs.hw2.Memory`**
- **`algs.hw2.Tournament`**
- **`algs.hw2.StressProgram`** ← Use to validate **`Memory`**
- **`algs.hw2.ValidateMemory`** ← Smaller program to validate **`Memory`**

If you attempt the bonus questions, write your results in a **`writtenQuestions.txt`** file.

And remember that this is an **individual assignment**. I encourage that you discuss the problem on Discord and with peers and friends and talk about ways you might solve the problem. BUT once you start writing code, that is where the **sharing** should stop. If we detect submissions that violate the WPI academic honesty policy, we will pursue each case. If you don't know how to get started, PLEASE come to office hours (either in person or in evening sessions).

This homework assesses the following skills:

- How to assemble linked lists (such as adding nodes, removing nodes)
- How to do "telescoping" mathematical analysis to count key operations
- How to use queues and stacks to solve problems

Q1. Mathematical Analysis [20 points] (REPLACE WITH NEW ONE)

This question is a more complicated version of what you will see on the midterm exam. You can find this code in the **`algs.hw2.Analysis`** class. Copy this class into your **`USERID.hw2`** package and modify it based on the requirements below. You will also need to instrument the code to count the number of times **`power()`** is invoked.

Given the following `proc` function, let S(N) be the number of times **`power(base, exp)`** is invoked when calling **`proc(A, 0, n-1)`** on an array, **`A`**, of length **`n`** containing integer values from **`0`** to **`n-1`**.

```
static long power(long base, int exp) {
  return (long) Math.pow(base, exp);
}

static long proc(int[] A, int lo, int hi) {
  long v = power(A[lo], 2) + power(A[hi], 2);
  if (lo == hi) {
    return v + power(v, 2);
  }

  int m = (lo + hi) / 2;
  long total = proc(A, lo, m) + 3*proc(A, m+1, hi);
  while (hi > lo) {
    total += power(A[lo], 2);
    lo += 2;
  }

  return total;
}
```

For this assignment, develop the recurrence relationship for S(N) and compute its closed-form formula. Then modify the **`Analysis`** class to output an updated table that shows the computed counts and the result of your model.

Question 1.1 (8 pts.)

Identify the Base Case for S() and the Recursive Case for S(N). Refer back to lecture for the format of this question.

Question 1.2 (12 pts.)

Derive an exact solution to the recurrence for S(N) when N is a power of 2. Be sure to show your work either in the **`WrittenQuestions.txt`** file or by a picture that you upload. Validate that your formula is correct by modifying the **`model(int n)`** implementation so it prints proper result when run.

BONUS QUESTION 1.3 (1 pt.)

Derive a formula that predicts the **`Value`** printed for **`proc(A, 0, A.length-1)`** when **`A`** contain values that alternate **`0, N, 0, N, 0, N, ...`** and N is a power of 2. Note that the **`main()`** method in **`Analysis`** already prepares **`A`** in this fashion.

Q2. Max and Second Largest (10 pts)

Copy the **`algs.hw2.Tournament`** class into **`USERID.hw2`** and complete its implementation. You will instrument this class to accumulate the total number of comparisons between two integers (using **less**) and the total number of exchanges of two elements (using **exch**).

This program uses a Max Priority Heap to compute the largest and second largest values assuming that N integers are inserted one at a time into an empty heap that is initially constructed with capacity N.

There are three cases to consider:

- The values are inserted in ascending order from 0 to N-1
- The values are inserted in descending order from N-1 to 0
- N random integers – drawn uniformly using **`Random.nextInt(N)`** – are inserted in order.

For each of these three cases, you are to compute (a) the total number of **less** invocations; and (b) the total number of **exch** invocations.

You must print out three tables, first the BEST case table, then an AVERAGE case table, then the WORST case table. You must determine which of the cases above are the BEST, AVERAGE and WORST cases. Note that every time you run the AVERAGE case trial, the number of comparisons might change, because the input has changed.

Q2.1 Mathematical Modeling

For the WORST case, design a formula that predicts S(N), the number of combined times **less** and **exch** are invoked. This S(N) must be a closed formula, which means that it cannot be recursive.

Q2.2 BONUS QUESTION [1pt]

Use a memory-efficient algorithm to compute largest and second largest, with least number of comparisons (as identified by Tournament algorithm on day 2 of class). Sample code is provided.

This took me way to long to complete, so do not even attempt this until totally done. If you complete, you will be able to have an algorithm that computes 1st and 2nd largest with minimal number of comparisons (though will require extra storage).

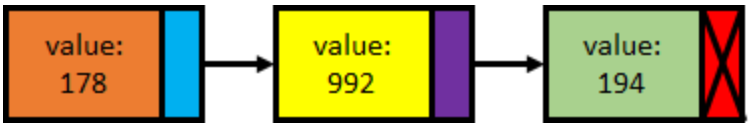
Q3. Memory Allocation Exercise (70 pts)

This assignment gives you a chance to demonstrate your ability to program with Linked Lists. This assignment is non-trivial and at the end of this assignment, I provide some thoughts on the order in which you should tackle this problem.

```
struct Node {
  int   value;
  Node  next;
};
```

Every programmer needs to know how to store a linked list in memory. Let's assume each **Node** in the linked list can store a single **int value** (which is 32-bits, as you know). The **next** reference is a 32-bit address in memory declaring where next node can be found. In total, you need 64-bits or 8 **char** to store each **Node**.

Given the following linked list, it would require a total of 24 **char** in memory. This entire linked list could be stored within a contiguous 32 **char** as shown below. Gray cells are unused. Values in memory are in top row and addresses are in the bottom.



		178				<25>							194				<0>						992				<14>			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

The first node in the linked list is found at address **<3>**. Its **value** (stored in the **char** from **<3>** to **<7>**) is **178** and the address of the **next** node in the linked list (stored in the next four **char**) is **<25>**. The last node in the linked list is found at address **<14>** since its **next** reference (found in **char** from **<18>** to **<21>**) is **0**. Addresses appear like **<address>**.

The C programming standard API offers the ability to dynamically allocate memory; you can allocate **struct Node** for a linked list using **malloc(8)** and then call **free()** when you are done with each node. Operating systems have sophisticated implementations for these operations. If you really want to see how it is done, check out these slides (https://moss.cs.iit.edu/cs351/slides/slides-malloc.pdf). For this assignment, you are going to implement this API by using linked lists.

Let's get started!

In this assignment, you will implement an API to manage dynamic memory by supporting the allocation and freeing of memory. To do so, a **`Memory(int size)`** object keeps a linked list of **`StorageNode`** that records what **`char`** has already been **`allocated`** in a computer's memory and a linked list of **`StorageNode`** that records what **`char`** remains **`available`**. Each **`StorageNode`** refers to a contiguous

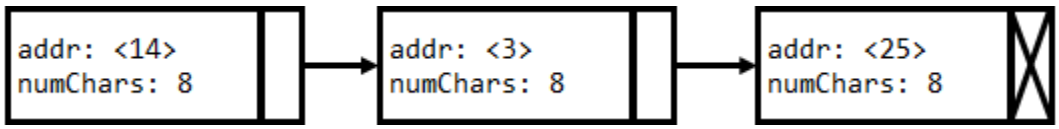
```
class StorageNode {
  int         addr;
  int         numChars;

  StorageNode next;
}
```

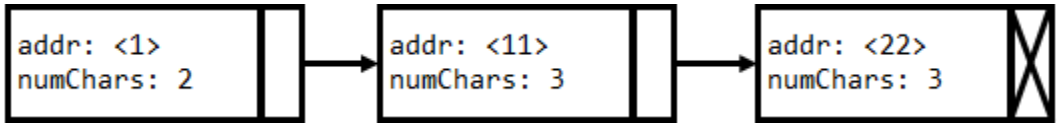
chunk of **`char`** in memory.

In the example introduced at the start of this question, there were 32 **`char`** of memory being managed.

Given the earlier color-coded chunk being managed by **`mem = new Memory(32)`**, the following could be the **`allocated`** linked list:



And the following is the **`available`** linked list (note that addresses must appear in ascending order):



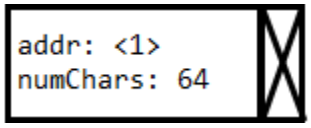
Ok, now let's get to the actual assignment!

Allocating and freeing memory

Implement the following **Memory** data type to manage the allocation (and subsequent free) of dynamic memory in a computer. For simplicity, the memory is defined by a one-dimensional **char[] storage** array that contains the **char** to be managed. **Memory** maintains two linked lists containing information about **storage.**

- **available** – records the **char** in **storage[]** that are available; these are broken into "chunks" of contiguous **char** in **storage**.
- **allocated** – records the **char** in **storage[]** that have already been allocated. These are discrete "chunks" from **storage** that have been claimed for use.

Memory manages a one-dimensional array of **char** of a given size containing **'\0' char** values. For example, using **Memory mem = new Memory(64)** the **allocated** linked list is **null** and the following represents the **available** linked list with a single **StorageNode**:

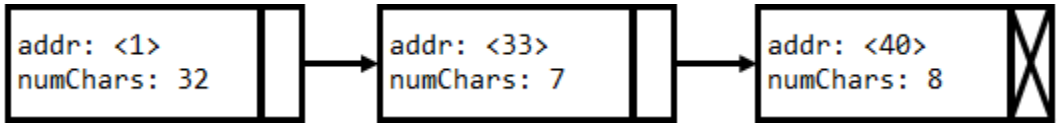


A valid address is an integer greater than zero; the first valid address is **<1>**.

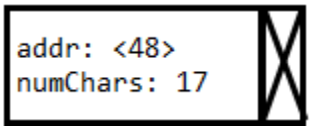
From this block that represents a one-dimensional array of 64 **char**, you can make multiple requests to allocate consecutive **char** sequences:

- **mem.alloc(32)** – allocate 32 consecutive **char** and return <1>, the address of the first **char**
- **mem.alloc(7)** – allocate 7 consecutive **char** and return <33>, the address of the first **char**
- **mem.alloc(8)** – allocate 8 consecutive **char** and return <40>, the address of the first **char**

The resulting **allocated** linked list contains three **StorageNode** entries. The programmer can **setChar(addr, ch)** or **getChar(addr)** any address that has been allocated (in this case any address from **<1>** to **<47>**).

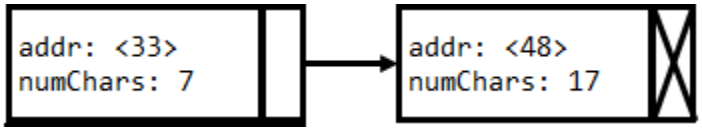


The original **available** linked list contains a single **StorageNode** entry:

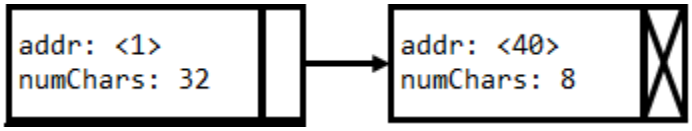


You can request to release an allocated chunk that had previously been allocated by calling **mem.free(addr)** using the address of the first **char** in the consecutive sequence of **char** that had previously been allocated.

Given the above situation, after calling **`mem.free(33)`**, the resulting **`available`** linked list looks like:



While the **`allocated`** linked list looks like:

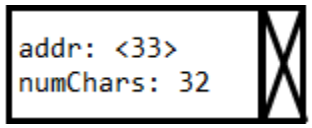


You can only call **`setChar()`** with an address that remains on **`allocated`**. In the example above, this means addresses 1 to 32 and 40 to 47.

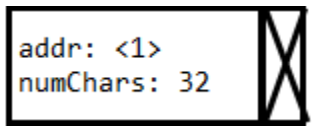
Organizing StorageNode in available

Now comes the tricky part. For **full credit** (a) the **`StorageNode`** entries in **`available`** must appear in ascending order by **`addr`**; and (b) **`available`** must contain the fewest number of **`StorageNode`** entries by merging adjacent **`StorageNode`** entries where possible when memory is freed.

Specifically, if given the above configuration, after calling **`mem.free(40)`**, the **`available`** linked list would have three **`StorageNode`** entries: **`(addr=<33>, numChars=7)`** and **`(addr=<40>, numChars=8)`** and **`(addr=<48>, numChars=17)`**. But as you can see, these are all directly adjacent to each other and so **`allocated`** can be collapsed into just a single **`StorageNode`** as follows:



While the **`allocated`** linked list looks like:



Accessing memory

You can call **`mem.getChar(addr)`** anytime to return the **`char`** stored at that address location. You can also call **`mem.getChars(addr, numChars)`** to return a **`char[]`** containing the desired number of **`char`**. I have provided the implementation of a helper method, **`getInt(addr)`**, that returns the 32-bit value stored in the 4 consecutive **`char`** found in address **`addr`** through **`addr+3.`**

However, to set a **`char`** in **`Memory`** at a specified address, the address must be wholly contained within a previously allocated chunk. Any attempt to **`setChar()`** using an address that was not previously allocated should throw a **`RuntimeException`**. I have provided the implementation of a helper method **`setInt(addr,value)`** that encodes **`value`** in the four consecutive **`char`** from **`addr`** to **`addr+3`**.

You need to complete the implementation of **validateAllocated(addr)** and **validateAllocated(addr,numChars)** to protect against invalid **setChar()** requests.

Reallocating memory

Memory provides the ability to resize an existing chunk of allocated memory of size **n** to a new size, **ns**. The original allocated memory must be released and **min(n, ns)**[1] **char** from the original chunk is copied to the newly allocated chunk. One thing that will be true is that after a valid **realloc()** request, the number of allocated blocks remains the same. Below are the methods that you must complete:

```
public class Memory {
  final char[] storage;     // actual memory being managed

  class StorageNode {
    int  addr;              // address into storage[] array
    int  numChars;          // how many chars are allocated

    StorageNode   next;     // the next StorageNode in linked
  }

  // operations that must be independent of the number of StorageNode objects
  public int charsAllocated() { ... }
  public int charsAvailable() { ... }
  public char getChar(int addr) { ... }
  public char[] getChars(int addr, int len) { ... }

  // operations that are dependent (in some way) on # of StorageNode objects
  public int blocksAllocated() { ... }
  public int blocksAvailable() { ... }

  public void setChar(int addr, char value) throws RuntimeException;
  public void setChars(int addr, char[] values) { ... }
         void validateAllocated(int addr) throws RuntimeException;
         void validateAllocated(int addr, int numChars) throws RuntimeException;

  public int alloc(int numChars) { ... }
  public int realloc(int addr, int numChars) { ... }
  public int copyAlloc(char[] chars) { ... }
  public boolean free(int addr) { ... }

  // bonus method only
  public Iterator<StorageNode> iterator(char[] pattern) { ... }
}
```

Copy **algs.hw2.Memory** into **USERID.hw2** and complete its implementation, which must conform to performance specifications that are included in the sample code. More documentation is found in the sample file. In the performance specifications, N refers to the number of **StorageNode** in **Memory**.

[1] The minimum of **n** and **ns**.

How you should approach this question

This question will prove to be a challenging programming assignment. Do not procrastinate! I recommend that you work on this question as follows:

- Get **`alloc()`** to work and validate by implementation **`charsAllocated()`**
- Get **`getChars()`** working and **`copyAlloc()`**
- Make sure **`charsAvailable()`** work
- Get **`free()`** to work
- Now handle all the cases where the **`free()`** of a chunk could be merged with a neighboring existing chunk as already found in the **`available`** linked list
- Lastly get **`realloc()`** to work
- Note that the **`blocksAvailable()`** and **`blocksAllocated()`** are for testing purposes and you actually might want to have them working first so you can debug your program.

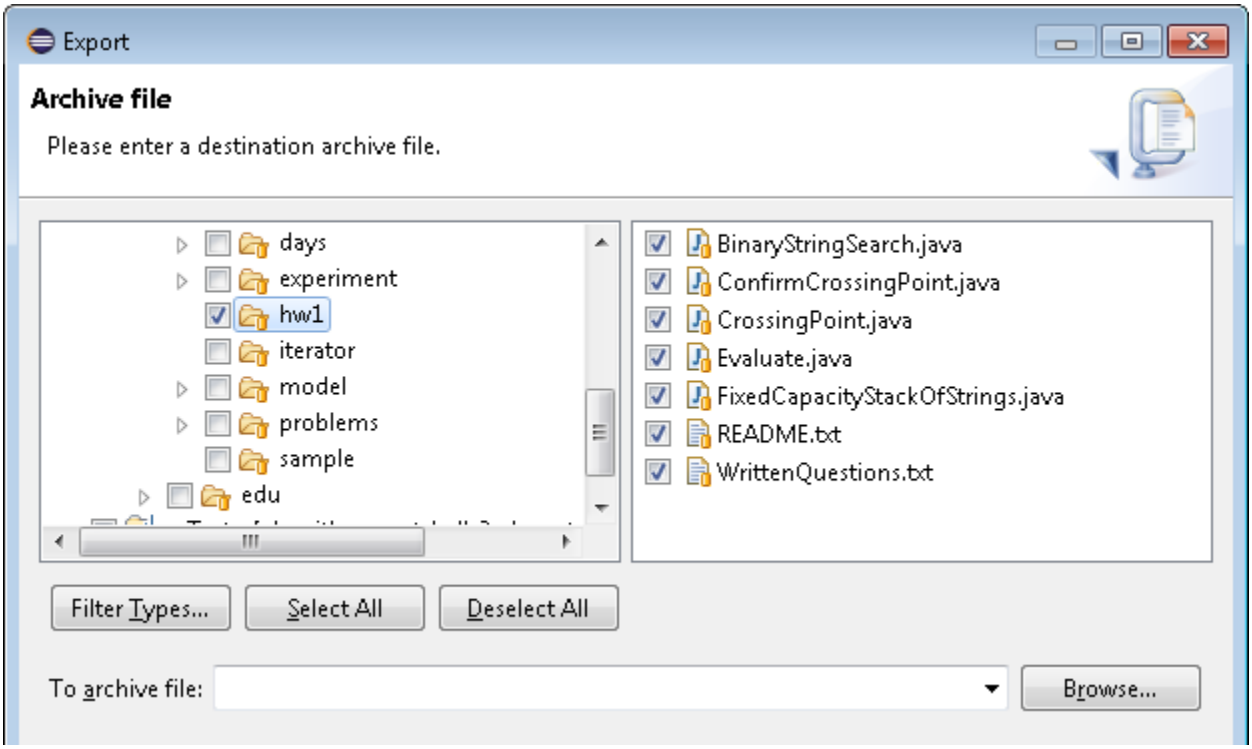
This assignment is challenging if you have not programmed extensively with linked lists. Do not wait!! Get started on this assignment as soon as you can. Come to office hours if you have questions.

Bonus Question 3.1. (1 pt) Implement **`java.util.Iterator<StorageNode> match(char[] pattern)`** that produces an **`Iterator`** object that returns the allocated chunks of **`char[]`** in memory that match the given **`pattern`** sequence exactly.

Submission Details

Each student is to submit a single ZIP file that will contain the implementations. In addition, there is a file "WrittenQuestions.txt" in which you are to complete the short answer problems on the homework.

The best way to prepare your ZIP file is to export your entire **`USERID.hw2`** package to a ZIP file using Eclipse. Select your package and then choose menu item "**Export…**" which will bring up the Export wizard. Expand the **General** folder and select **Archive File** then click **Next**.



You will see something like the above. Make sure that the entire "hw2" package is selected and all files within it will also be selected. Then click on **Browse…** to place the exported file on disk and call it USERID-HW2.zip or something like that. Then you will submit this single zip file in Canvas as your homework2 submission.

Addendum

If you discover anything materially wrong with these questions, be sure to contact the professor or TA/SAs posting to the discussion forum for HW2 on Discord;

When I make changes to the questions, I enter my changes in red colored text as shown here.

Change Log

1. Clarified that blocksAllocated() and blocksAvailable() are to be implemented for Q3
2. Clarify that in proc for Q1 the inner loop has "lo += 2;"

3. Clarify that for Q2, you are to derive a formula for the combined number of less and exch operations