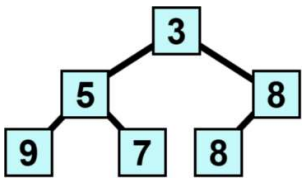
ECE220: Computer Systems & Programming **Spring 2024 ZJUI**
Machine Problem 2 **due: Saturday 2 March, 11:59:59 p.m.**

Heap Subroutines



Your task this week is to write three subroutines that together implement a data structure called a heap. A heap uses an array to make the cost of inserting new elements and extracting the smallest element fairly efficient, requiring only a number of operations logarithmic in the number of elements in the heap. Heap operations are efficient because a heap keeps its elements in a partially-sorted order. The figure to the right illustrates a heap conceptually. The elements form a binary tree and obey the **heap property**, which means that the children of any element have values no smaller than the value of the element itself. As the figure illustrates, the heap property only extends within a subtree: all descendants of a node have values at least as large as that node's value, but we cannot reason about the values of two sibling or cousin nodes. In the example, the left child of the root has a value smaller than that of the right child of the root, but looking at the children of node with value 5, the left child's value is larger than that of the right.

How the heap is sorted—deciding what "larger" means in the heap property—does not matter, but we must choose an order for any particular heap. In this assignment, the heap stores node numbers from a graph, which means that each heap element requires only one LC-3 memory location. We want to then sort the heap based on the distances from a source node in the graph. In other words, the node specified by the number in any heap element has a distance that is no greater than the distance of the nodes specified by the numbers stored in the heap element's children. Node distances are stored in a separate array, which is then indexed by node number.

The three subroutines that you must write are SWAP_ELEMENTS, which swaps two heap elements and updates associated information about the nodes in those elements; PERCOLATE_DOWN, which moves a heap element downward (towards the leaves at the bottom of the illustration) to maintain the heap property; and GET_CLOSEST, which removes the element at the root of the heap (the root of the tree in the figure, which contains the node with the least distance in our heap) then acts to preserve the heap property. Together, these three routines require a little over 200 lines of LC-3 assembly code, including comments. To help you understand the data structures and to keep the assignment from getting too long, we have provided you with a fourth subroutine, PERCOLATE_UP, which moves a heap element upward (towards the root of the tree at the top of the illustration) to maintain the heap property.

The objective for this week is to give you some experience with understanding and manipulating arrays of data in memory as well as some exposure to important data structures.

Warning About Outside Material

To make this document and your task easier to understand, we refer to things stored in a heap as **elements**. Each heap element has an **index**, which is simply the index of the element in the array used to store the heap. In contrast, a graph such as the ones that we showed you last week consists of a set of **nodes**, which are numbered, so every node has a **number**. The heap that you will build stores **node numbers**: in other words, **each heap element contains a node number** and thus requires only one LC-3 memory address. As mentioned last week, please be aware that, should you choose to look at any outside material, that material is likely to use the same terms (nodes, numbers, indices, and so on) to refer to different things. Be careful.

How to Proceed

First, read this document completely.

Next, add a copy of your working MP1 code to your **`mp2.asm`** file. We usually allow you to make use of a peer's MP1 if you were not able to get yours working in time. However, **to avoid academic integrity violations, you must give explicit credit** to that person in your code and make clear exactly what code was taken from them, **and** you **must follow the rules about when** you are allowed to receive the code—obviously not before MP1 is due. Check Piazza for details.

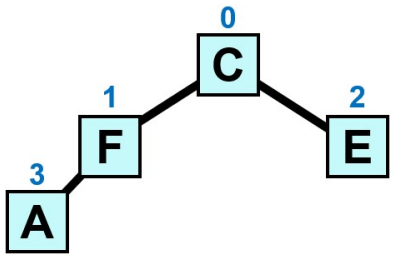
We suggest that you then read through the PERCOLATE_UP subroutine provided to you in **`mp2.asm`** and make sure that you understand both how it works and that its operation corresponds to your understanding of the data structures provided to your subroutines.

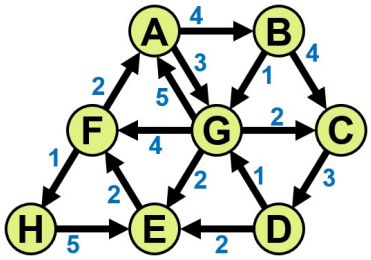
Following the order of subroutines suggested in this document is not mandatory, but we strongly encourage you to develop and test them in the order described.

The Heap

All subroutines in this assignment operate on a heap stored starting at address x8000 in LC-3 memory. The length of the heap is stored just before the heap, at address x7FFF. Remember that the heap stores node numbers from a graph. In your subroutines, you will also need to access two additional arrays, both indexed by node number. The first array hold heap indices, and its contents should be consistent with the heap. In other words, if heap element ***E*** holds node number ***N***, position ***N*** in the array of heap indices must hold the value ***E***. The second array holds node distances, and its contents define the heap property for the heap: any element's node's distance should be no greater than the distance to any node stored in a child element. Remember that all arrays are indexed starting from 0. The array of heap indices starts at address x7000, and the array of node distances starts at x6000.

For clarity, below is an example set of data structures generated while searching from graph node H. A heap element with index ***E*** has as its parent the element with index $P = \lfloor (E-1)/2 \rfloor$ (except for element 0, which has no parent) and has children with indices $2E+1$ and $2E+2$ (when they exist).





Address	Contents	Distance
x6000	x0005	to A is 5
x6001	x7FFF	B unreachable
x6002	x0002	to C is 2
x6003	x7FFF	D unreachable
x6004	x0002	to E is 2
x6005	x0004	to F is 4
x6006	x0000	to G is 0
x6007	x7FFF	H unreachable

Address	Contents	Heap Index
x7000	x0003	3
x7001	xFFFF	not in heap
x7002	x0000	0
x7003	xFFFF	not in heap
x7004	x0002	2
x7005	x0001	1
x7006	xFFFF	not in heap
x7007	xFFFF	not in heap

Address	Contents	Meaning
x7FFF	x0004	length is 4
x8000	x0002	node C
x8001	x0005	node F
x8002	x0004	node E
x8003	x0000	node A

The First Task

The first subroutine is SWAP_ELEMENTS. Two heap indices are provided to your code in R0 and R1. The subroutine also uses the heap (starting at address x8000, with length stored at address x7FFF) and the array of heap indices (starting at address x7000). Your SWAP_ELEMENTS subroutine may assume that both arrays exist and are valid, although you will need to create these arrays yourself when testing your code. The subroutine may also assume that the two indices provided in R0 and R1 are valid—in other words, both indices are non-negative and smaller than the length of the heap.

Your subroutine must swap the two node numbers contained in the heap elements with indices given by R0 and R1. Your subroutine must also update the array of heap indices to reflect the changes made to the heap. Remember that the array of heap indices is indexed by node numbers, and that the node numbers are contained in the heap elements. Your subroutine must preserve all register values except for R7 (the return address).

The PERCOLATE_UP subroutine provided to you requires only SWAP_ELEMENTS and FIND_PARENT from MP1, so once you have written and debugged SWAP_ELEMENTS, you can test PERCOLATE_UP. Be sure that you read it and understand it—see the section below with pseudo-code for what the subroutine is doing—before you try to tackle the next subroutine, which is similar but slightly more complex.

Pseudo-code for Upwards Percolation

Remember that a heap maintains the heap property by percolating elements upward and downward as necessary. For the upward direction, the code has been provided to you. Upward percolation is necessary in two situations, both of which occur when executing Dijkstra's single-source shortest path algorithm, which you will implement in MP3.

First, upwards percolation is used whenever a heap element's distance is reduced, as happens when we find a shorter path to a graph node to which we've already found a path. Second, upwards percolation is used whenever a new element is added to a heap, as happens when we first find a path to a previously unseen graph node.

Since a heap is implemented as an array, a new element is initially placed at the end of the array and the length of the array (the heap) incremented. Unfortunately, the new element's value may not obey the heap property. If the new element's value is in fact smaller than the value of its parent, we fix the problem by swapping the new element with its parent. You should convince yourself that if we swap a new element with its parent, the new element's value is ALSO at least as small as that of the parent's other child, if that child exists. But the new element's value may be smaller than that of its grandparent. So we could swap again and again. If we reach the root, though, we are done: the new element has the smallest value in the heap.

Let's see that in pseudo-code:

Given: heap ***H*** with length ***L***, parent index of a non-root element ***E*** given by *parent(**E**)*, node number ***N*** stored in an element ***E*** given by *value(**E**)*, and the distance to a node ***N*** given by *distance(**N**)*,

Problem: percolate element ***X*** upward as necessary to maintain the heap property.

Pseudo-code Solution:

```
while X ≠ 0 and distance(value(X)) < distance(value(parent(X)))
        SWAP_ELEMENTS(X, parent(X))
        X ← parent(X)
end while
```

The value of X is passed to PERCOLATE_UP in R0, and all registers except R7 are callee-saved.

The Second Task

The second subroutine is PERCOLATE_DOWN. Downward percolation is necessary when we extract the smallest element from the heap, the root of the conceptual tree. In our heap, element 0 contains the node number of the graph node with least distance, which is the node explored next by Dijkstra's algorithm. In fact, your third task is to write the algorithm that extracts this element from the heap, so you can skip ahead to the next section to see when downward percolation is used.

When PERCOLATE_DOWN is called, the heap index of the element to be percolated down is provided in R0. The subroutine also uses the heap (starting at address x8000, with length stored at address x7FFF) and the array of node distances (starting at address x6000). As with PERCOLATE_UP, your subroutine must use SWAP_ELEMENTS to percolate the specified node (contained in the element with index given in R0) downward in the heap. Thus your subroutine also uses and changes the array of heap indices at x7000 implicitly. Your subroutine may assume that all three arrays exist and are valid, but again you will need to create these arrays yourself when testing your code. The subroutine may also assume that the index provided in R0 is valid—in other words, non-negative and smaller than the length of the heap. Finally, you may assume that the distances to all nodes in the heap are non-negative and less than x4000, so overflow should not be an issue when comparing distances. Graph nodes NOT in the heap may be marked as "infinite" distance (x7FFF in MP3), but your subroutine should not need to access those distances. Your subroutine must preserve all register values except for R7 (the return address).

Pseudo-code for downward percolation appears below. Given ***X*** in R0, PERCOLATE_DOWN re-establishes the heap property by moving the node contained in element X downward in the heap using SWAP_ELEMENTS until either the node reaches the lowest layer of the heap (with no children), or any children contain nodes with distances at least as large as that of the node being moved (in other words, the element obeys the heap property). Be sure to break ties correctly: do not move the node down if the distances are equal, and if both children have equal distances which are less than that of their parent, swap with the left child. PERCOLATE_DOWN is similar but more complicated than PERCOLATE_UP. In the pseudo-code, child element indices are calculated explicitly. For upward percolation, the *parent(**E**)* function corresponded to FIND_PARENT from MP1. Here, we have not asked you to write such a subroutine, although you should feel free to do so.

Given: heap ***H*** with length ***L***, the node number ***N*** stored in an element ***E*** given by *value(**E**)*, and the distance to a node ***N*** given by *distance(**N**)*,

Problem: percolate node ***X*** downward as necessary to maintain the heap property.

Pseudo-code Solution:

```
while 2X + 1 < L
        if 2X + 2 < L and distance(value(2X + 2)) < distance(value(2X + 1))
                if distance(value(2X + 2)) < distance(value(X))
                        SWAP_ELEMENTS(X, 2X + 2)
                        X ← 2X + 2
                else
                        break (out of while loop)
                end if
        else if distance(value(2X + 1)) < distance(value(X))
                SWAP_ELEMENTS(X, 2X + 1)
                X ← 2X + 1
        else
                break (out of while loop)
        end if
end while
```

The Third Task

The final subroutine that you must write is GET_CLOSEST. This subroutine removes and returns the contents of element 0 from the heap in R0. Remember that element 0, the root of the heap, contains the node number of the closest graph node. If the heap is empty, the subroutine returns -1 in R0. Otherwise, a valid node number is returned, and the subroutine changes that node's heap index to -1 (xFFFF) in the heap index array at x7000. To re-establish the heap property, GET_CLOSEST then copies the last element of the heap (with index given by the heap length minus 1) over element 0, decrements the length of the heap (at x7FFF), and calls PERCOLATE_DOWN.

Since GET_CLOSEST uses PERCOLATE_DOWN, GET_CLOSEST also requires that all three arrays be in memory before it is called: the heap (starting at address x8000, with length stored at address x7FFF), the array of node distances (starting at address x6000), and the array of heap indices (starting at address x7000). Your subroutine may assume that all three arrays exist and are valid, but again you will need to create these arrays yourself when testing your code. You may also assume that the distances to all nodes in the heap are non-negative and less than x4000. Your subroutine must preserve all register values except for R0 (the return value) and R7 (the return address).

Test Code Elements

When you open **`mp2.asm`**, you will see that we have given you some test code this time in addition to the PERCOLATE_UP subroutine. **Be sure that you follow the rules** given in the comments when adding your code! **Failure to do so will break the test code and result in a fairly significant loss of points for you.**

The test code enables outside code to easily call your subroutines. Such code has also been provided to you, along with a number of test scripts, all within the **`tests`** subdirectory. The easiest way to make use of these tests is to copy your **`mp2.asm`** file into that subdirectory and then run the tests from within the subdirectory. Start by assembling all six of the graph-related files and all four of the test code files as well as your MP2 solution. The test scripts are called "t<function><number>," where function is "se" for SWAP_ELEMENTS, "pu" for PERCOLATE_UP, and so forth, and number is the test number. The correct output for the test is given in the file with the same name but with "out" appended as a suffix. For example, to execute the first upward percolation test,

```
lc3sim -s tpu1 > myresult
diff tpu1out myresult
```

The **`diff`** command should produce no output if your code produces the correct results. If you fail to preserve registers or fail to update the heap or heap index array correctly, these differences will appear as output based on a simple dump of memory in both cases.

Specifics

- Your code must be written in LC-3 assembly language and must be contained in a single file called **`mp2.asm`**. We **will not grade** files with any other name.
- Do not make any assumptions unless they are stated explicitly to you in this document.
- You must obey the rules given in the file with regard to placing your code, including a copy of your MP1. Be sure to follow the rules and to credit your peer properly if you choose to use someone else's MP1 code.
- All subroutines must preserve all registers except R7 (and R0 in the case of GET_CLOSEST).
- Your subroutine for swapping two elements in a heap must be called SWAP_ELEMENTS.
    - You may assume that R0 and R1 hold non-negative heap indices and that each is less than the length of the heap when your function is called.

    - You may assume that a heap starting at x8000 with length at x7FFF has been provided, and that a heap index array starting at x7000 has also been provided. The heap is indexed by heap indices, and the heap index array is indexed by node numbers. Be aware that the number of graph nodes is usually more than the number of heap elements.
    - Your subroutine must swap the contents (node numbers) of the two elements specified by R0 and R1 and must swap the corresponding heap indices in the heap index array.
- Your subroutine for percolating downward in the heap must be called PERCOLATE_DOWN.
    - You may assume that R0 holds a non-negative heap index that is less than the length of the heap when your function is called.
    - You may assume that a heap starting at x8000 with length at x7FFF, a heap index array starting at x7000, and a node distance array starting at x6000 have also been provided. The heap is indexed by heap indices, and the other two arrays are indexed by node numbers. Be aware that the number of graph nodes is usually more than the number of heap elements. You may also assume that any node in the heap has a non-negative distance less than x4000.
    - Remember that heap element ***E***'s children are 2***E*** +1 and 2***E*** +2, which may or may not exist (compare the indices with the heap length to check).
    - Your routine must percolate the node contained within the specified element downward in the heap until the distance of the node is less than that of any children (having no children implicitly satisfies this requirement).
    - Your routine must make use of SWAP_ELEMENTS when percolating downward.
- Your subroutine for getting the closest node from a heap must be called GET_CLOSEST.
    - You may assume that a heap starting at x8000 with length at x7FFF, a heap index array starting at x7000, and a node distance array starting at x6000 have been provided. The heap is indexed by heap indices, and the other two arrays are indexed by node numbers. Be aware that the number of graph nodes is usually more than the number of heap elements. You may also assume that any node in the heap has a non-negative distance less than x4000.
    - If the heap is empty, your routine must return -1 in R0.
    - If the heap is not empty, your routine must return the node number contained in element 0 of the heap at the start of the call.
        - The heap index for the node number returned should be changed to -1,
        - If any elements remain, the last element of the heap must be copied over the first, and the corresponding heap index updated.
        - The length of the heap must be decremented.
        - If the heap has more than one element remaining, PERCOLATE_DOWN must be called with R0=0 to re-establish the heap property.
- Your code must be well-commented, must include a header explaining each subroutine, and must include a table describing how registers are used within each subroutine. Follow the style of examples provided to you in class and in the textbook.
- None of your subroutines may access memory (neither to read nor to write) outside of your code and the regions provided for your subroutines' use, nor write to memory outside of your code unless specifically allowed in this document.
- None of your MP2 subroutines may produce any output to the display (neither directly nor indirectly). Also, none of your subroutines may read input from the keyboard.
- All subroutines must operate correctly even when called many times without reloading your code.

Testing

You are responsible for testing your code to make sure that it executes correctly and follows the specification exactly. See "Test Code Elements" for some help getting started, but note that the given files are meant as examples. You should create more tests to ensure that your code works correctly. You will also find it useful to step through your code in the simulator and inspect the state after each step to make sure that your code does exactly what you intended.

Grading Rubric

Functionality (70%)

- SWAP_ELEMENTS
  - 5% - swaps node numbers contained in heap elements given by R0 and R1 correctly
  - 5% - swaps corresponding heap indices correctly in array
- PERCOLATE_DOWN
  - 5% - handles elements with no children correctly (stops percolation)
  - 10% - handles elements with one child correctly (compares distances and percolates if distance is strictly larger than that of child)
  - 10% - handles elements with two children correctly (compares distances and percolates if distance is strictly larger than that of one or both children)
  - 5% - breaks ties between two children correctly (choose left)
  - 5% - uses SWAP_ELEMENTS correctly to percolate down
- GET_CLOSEST
  - 5% - produces correct return value in R0 and updates node's heap index (if any)
  - 5% - skips copy and percolation if heap holds one element at start
  - 5% - copies last element to index 0 and changes heap index (if any)
  - 5% - uses PERCOLATE_DOWN correctly
- ALL subroutines (any failure loses ALL points)
  - 5% - preserves all registers except R7 when called (and R0 for GET_CLOSEST)

Style (5%)

- 5% - Percolate down uses only two distance comparisons (left vs. right child and smaller child vs. parent)

Comments, Clarity, and Write-up (25%)

- 5% - each subroutine has a paragraph explaining the subroutine's purpose and interface (these are given to you; you just need to document your work)
- 10% - each subroutine includes a table explaining how each register is used
- 10% - code is clear and well-commented

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if your code does not assemble, you will receive no functionality points. If you name a subroutine incorrectly, you will receive no points for it, whether it works or not. Note also that MP3 will build on these subroutines, so you may have difficulty testing those MPs if your code does not work properly for this MP.

Finally, we will deduct **10 points** if the LC-3 VSCode extension developed by former ZJUI student Qi Li (look in the marketplace—you'll see his name) gives **even a single warning** on your assembly code. Fix them. (There are two exceptions. The first is the same as with MP1. The second is we will ignore save/restore sequence differences for GET_CLOSEST, since use of PERCOLATE_DOWN at the end makes that warning likely but specious.)