CSE 12 – Basic Data Structures

Prof. Christine Alvarado
[Slides borrowed/adapted from slides by Cynthia Lee, Rakesh Vama, & Roshni Chandrashekar]
Announcements

1. GREAT JOB on the midterm. Mean 82, median 86! WOW!
   1. If you didn’t do so well, please come talk to me. There is still time to improve.
2. HW3 is posted. You can start now, but be sure to also do the reading for Friday.
   1. 1-day extension: Due next WEDNESDAY
   2. We will talk about deep and shallow copy on Monday, but understanding the copy constructor is not hard
Your feedback: What’s going well (in order of # of times cited, cited by at least 3 people)

- Reading quizzes (mentioned by MANY!)
- HW assignments
- Lectures/clickers/discussion with peers
- Review quizzes
Your feedback: What I can do to help your learning

- Clearer HW instructions
  - Please ASK about what’s not clear
- Better advice from the tutors/more tutor hours on the weekends
- More review material
  - We’ll try, but every quiz IS a review sheet
Today’s topics

- More with Java Generics
- Priority Queues, Trees and Heaps
Heap insert and delete
Heap insert

(a) A minheap prior to adding an element. The circle is where the new element will be put initially.

(b) Add the element, 6, as the new rightmost leaf. This maintains a complete binary tree, but may violate the minheap ordering property.

(c) “Bubble up” the new element. Starting with the new element, if the child is less than the parent, swap them. This moves the new element up the tree.

(d) Repeat the step described in (c) until the parent of the new element is less than or equal to the new element. The minheap invariants have been restored.
Heap delete

(a) Moving the rightmost leaf to the top of the heap to fill the gap created when the top element (5) was removed. This is a complete binary tree, but the minheap ordering property has been violated.

(b) “Trickle down” the element. Swapping top with the smaller of its two children leaves top’s right subtree a valid heap. The subtree rooted at 18 still needs fixing.

(c) Last swap. The heap is fixed when 18 is less than or equal to both of its children. The minheap invariants have been restored.
Removing from a heap

When you remove an element from an array-backed heap, at what index is the element to remove located? Assume the variable size stores the number of elements currently in the heap, and arr.length is the length of the array storing the heap.

A. 0
B. 1
C. size – 1
D. arr.length – 1
E. You can’t tell with the information given
Removing from a heap

![Heap Diagram]

```
size  9
```

```
5  6  10  7  14  11  21  27  18
0  1  2  3  4  5  6  7  8
```
Removing from a heap

```java
theHeap[0] = theHeap[size-1];
```

NOTE: The assignment uses ArrayLists not arrays
Removing from a heap

Now 18 needs to trickle down... this should be a separate method. Can be iterative or recursive, but recursive is easier, really!

```java
theHeap[0] = theHeap[size-1];
size--;
trickleDown( ____?____ );
```
TrickleDown (min heap)

This is the main challenge of writing the heap, so I am not going to write it for you. But I will give you some hints and the general idea behind a recursive approach.

When we trickle 18 down, which value should become the root?
A. 6
B. 10
C. 27
D. 18 should stay there
E. Other
TrickleDown (min heap)

This is the main challenge of writing the heap, so I am not going to write it for you. But I will give you some hints and the general idea behind a recursive approach.

Assume the node to be trickled down is at \( \text{index} \) and that its left and right children are at \( l\text{Ind} \) and \( r\text{Ind} \), respectively. Also assume the node at index has two children. Which line correctly completes the code below?

```c
if ( ___________________ ) childInd = lInd;
else childInd = rInd;
swap( theHeap, childInd, index );
```

A. \( l\text{Ind} < r\text{Ind} \)
B. \( l\text{Ind} < \text{index} \)
C. \( \text{theHeap}[l\text{Ind}] < \text{theHeap}[r\text{Ind}] \)
D. \( \text{theHeap}[l\text{Ind}] < \text{theHeap}[\text{index}] \)
This is the main challenge of writing the heap, so I am not going
to write it for you. But I will give you some hints and the general
idea behind a recursive approach.

There are two base cases for this method.
Which of these is one?
A. 18 is in a leaf node
B. 18 is in a node with one child
C. 18 is a node with 2 children
D. 18 is at the root of the heap
This is the main challenge of writing the heap, so I am not going to write it for you. But I will give you some hints and the general idea behind a recursive approach.

There are two base cases for this method. Which of these is another?
A. 18 has no children less than itself
B. 18 has no more than one child less than itself.
C. 18 has exactly one child, which is greater than or equal to it
TrickleDown

Here’s a rough recursive algorithm for trickleDown. It’s up to you to translate this to code! And careful, because there are subtleties not mentioned here (e.g., what if the node has only one child?

```
TrickleDown(index) // You might need more arguments
    If value at index is a leaf, return
    If value at index has no children less than it, return
    Swap value at index with its smaller child (at childInd)
    TrickleDown(childInd)
```
Adding to a heap (offer)

When you add an element to a heap, at what index do you put it initially?
A. 0
B. size – 1
C. size
D. Somewhere else
Adding to a heap (offer)

If the ArrayList holding your heap is at capacity, you must stop and make a new one! DO NOT simply let Java resize the ArrayList.
For offer, you will need a helper method called BubbleUp. I also suggest recursion, and here’s a very rough recursive algorithm, though it’s up to you to figure out the base case. You can talk to others in the class about this.

```java
BubbleUp(index) // You might need more arguments
    If value at index shouldn't move any more, return
    Swap value at index with its parent (at parentInd)
    BubbleUp(parentInd)
```

This is the base case for you to figure out.

HINT: There will be more than one
TRUE OR FALSE

- There is only one configuration of a valid min-heap containing the elements \{34, 22, 3, 9, 18\}

A. TRUE
B. FALSE
Time cost

What is the worst-case time cost for each heap operation: Add, Remove, Peek?

A. $O(n)$, $O(1)$, $O(1)$
B. $O(\log n)$, $O(\log n)$, $O(1)$
C. $O(n)$, $O(\log n)$, $O(\log n)$
D. Other/none/more
Heapsort
Heap sort is super easy

1. Insert unsorted elements one at a time into a heap until all are added
2. Remove them from the heap one at a time (we will always be removing the next biggest item, for max-heap; or next smallest item, for min-heap)

THAT’S IT!
Implementing heapsort

Devil’s in the details
We can do the entire heapsort in place in one array

- Unlike mergesort, we don't need a separate array for our workspace.
- We can do it all in place in one array (the same array we were given as input).

For your HW, you do not need to heap sort in place.
Build heap by inserting elements one at a time:

1. 8
2. 12 8
3. 12 8 2
4. 12 10 2 8
5. 12 10 2 8 6
6. 12 10 4 8 6 2

12 2 10 6 4
2 10 6 4
10 6 4
Sort array by removing elements one at a time:

1. 10 8 4 2 6
   
   12

2. 8 6 4 2
   
   10 12

3. 6 2 4
   
   8 10 12

4. 4 2
   
   6 8 10 12

5. 2
   
   4 6 8 10 12

6. 2 4 6 8 10 12
Build heap by inserting elements one at a time IN PLACE:

1. \[ \begin{array}{cccccc}
8 & 12 & 2 & 10 & 6 & 4
\end{array} \]

2. \[ \begin{array}{cccccc}
12 & 8 & 2 & 10 & 6 & 4
\end{array} \]

3. \[ \begin{array}{cccccc}
12 & 8 & 2 & 10 & 6 & 4
\end{array} \]

4. \[ \begin{array}{cccccc}
12 & 10 & 2 & 8 & 6 & 4
\end{array} \]

5. \[ \begin{array}{cccccc}
12 & 10 & 2 & 8 & 6 & 4
\end{array} \]

6. \[ \begin{array}{cccccc}
12 & 10 & 4 & 8 & 6 & 2
\end{array} \]
Sort array by removing elements one at a time IN PLACE:

1. 10 8 4 2 6 12
2. 8 6 4 2 10 12
3. 6 2 4 8 10 12
4. 4 2 6 8 10 12
5. 2 4 6 8 10 12
6. 2 4 6 8 10 12