CSE 12 – Basic Data Structures

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

1. HW6 posted. Have you started
   A. No, not at all
   B. I’ve read it
   C. I’ve started coding
   D. I’ve made significant progress
   E. I’m done!
Today’s topics

- Copying in Java: Shallow vs Deep
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
Sort array by removing elements one at a time:

1. 10 8 4 2 6
2. 8 6 4 2
3. 6 2 4
4. 4 2
5. 2
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
Copying Objects
Copying Objects

Objects can contain other objects and primitive types. This leads to varying degree of “copying” within Java.

class List12<E> implements List<E>
{
    Node<E> head;
    int size;
    ...
}
Very shallow copy

Just copying the pointer to the data structure object. Now the two references point to the same object.

```java
List<Integer> x = new LinkedList<Integer>();
List<Integer> y = x;
```
Very shallow copy

Diagram:
- X
  - Head
    - size
- Y
- 10 Data
- 20 Data
- 30
Very shallow copy

Just copying the pointer to the data structure object. Now the two references point to the same object.

```java
List<Integer> x = new LinkedList<Integer>();
List<Integer> y = x;
System.out.println( y == x );
```

What does this line print?

A. True
B. False
Shallow copy

Create a new object, but “very-shallow-copy” the instance variables. For ex, using a copy constructor.

```java
public List12(List12<E> other) {
    this.head = other.head;
    this.size = other.size;
}

List12<Integer> x = new List12<Integer>();
...
List12<Integer> y = new List12<Integer>(x);
```
Shallow copy
Shallow copy

Create a new object, but “very-shallow-copy” the instance variables. For ex, using a copy constructor.

```java
public List12(List12<E> other) {
    this.head = other.head;
    this.size = other.size;
}
```

List12<Integer> x = new List12<Integer>();
...
List12<Integer> y = new List12<Integer>(x);
System.out.println( y == x );
```

What does this line print?
A. True
B. False
Deep copy

Create a new data structure object and new data structure elements, but share data item objects.
Deep copy

X

Head
size

10
Data

20
Data

30
Data

Y

Head
size
Deeper copy

Create a new data structure object, new data structure elements and create copies of data elements.
Even though the two resulting data structures contain the same data, they are independent of one another. Modifying one does not affect the other.
Hard to achieve, since the data objects might not be ‘cloneable’.
Deeper copy

[Diagram showing a deep copy process with nodes labeled X and Y, each with a Head and size, connected to nodes with data values 10, 20, and 30.

The diagram illustrates the process of creating a deep copy where each node is duplicated, ensuring that references to the original data are not copied, thus maintaining data integrity.]
Copying objects in Java

- If you want to design instances of a class so that they are copyable, how can you do that? There are several options

1. Define a copy constructor for your class: a constructor that takes as argument a pointer to an existing instance of the class, and initializes the new instance to be a copy of the existing one

2. Define a static factory method in the class that takes as argument a pointer to an existing instance of the class, creates a new instance, and initializes the new instance to be a copy of the existing one

3. Use the Cloneable interface and override the clone() instance method
Binary tree traversals
Binary trees

- Recall from before, a binary tree is any tree where each node has 0, 1, or 2 children
- That’s the only restriction
  - Recall: heaps are a special case of binary trees, and they have two additional restrictions
Trees vs. Lists

- Lists have an obvious ordering
  - 1\textsuperscript{st} element is first, 2\textsuperscript{nd} element is second, ...
- Trees don’t
  - More than one reasonable order
Pre-order traversal

```java
preorder(node) {
  if (node != null) {
    visit this node
    preorder(node.left)
    preorder(node.right)
  }
}
```

A. D B E A F C G
B. A B D E C F G
C. A B C D E F G
D. D E B F G C A
E. Other/none/more
**Post-order traversal**

```java
postorder(node) {
    if (node != null) {
        postorder(node.left)
        postorder(node.right)
        visit this node
    }
}
```

A. D B E A F C G  
B. A B D E C F G  
C. A B C D E F G  
D. D E B F G C A  
E. Other/none/more
In-order traversal

```java
inorder(node) {
    if (node != null) {
        inorder(node.left)
        visit this node
        inorder(node.right)
    }
}
```

A. D B E A F C G
B. A B D E C F G
C. A B C D E F G
D. D E B F G C A
E. Other/none/more
Level-order traversal
AKA Breadth-first search
Level-order traversal

- Also commonly called **Breadth-First Search** or **BFS**
- As opposed to something like pre-order or post-order, which are **Depth-First Search (DFS)** algorithms
Level-order challenges

- How do we know where to go next?
- 1, 2, 3—but there is no edge from 2 to 3!
- While we’re still at 1, we must “remember” to come back to 3 after we’re done with 2
  - At 2, remember to come back to 4 and 5 after 3
  - At 8, remember to come back to 16 and 17 after 15
- Etc …
How to “remember”

- When we visit a node, add references (pointers) to its children to a **queue**
- To know which node to visit next, remove next element from the queue
Tracing the queue in BFS

Which shows the state of the queue at the end of “visiting” node #10? (“head” of the queue, i.e. the next element that will be removed, is the leftmost)

A. 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11
B. 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21
C. 20, 21
D. 1, 2, 5
E. Other/none/more
BFS code

BFS(node)
{
    Queue<Bnode> q = new Queue<Bnode>();
    q.add(node);
    while (!q.isEmpty()) {
        current = q.remove();
        System.out.println(current.getData());
    }
}

A. q.add(current.getLeftChild());
B. q.add(current.getRightChild());
C. q.add(q.remove());
D. q.sort();
E. Other/none/more
Is it bigger than a breadbox?
- no
  - Do you eat it with eggs?
    - no: a mouse
    - yes: Spam

- yes
  - Is it worth a lot of money?
    - no
      - a bag of trash
    - yes
      - Does it know Java?
        - no: a house
        - yes: a computer scientist