Middlebury

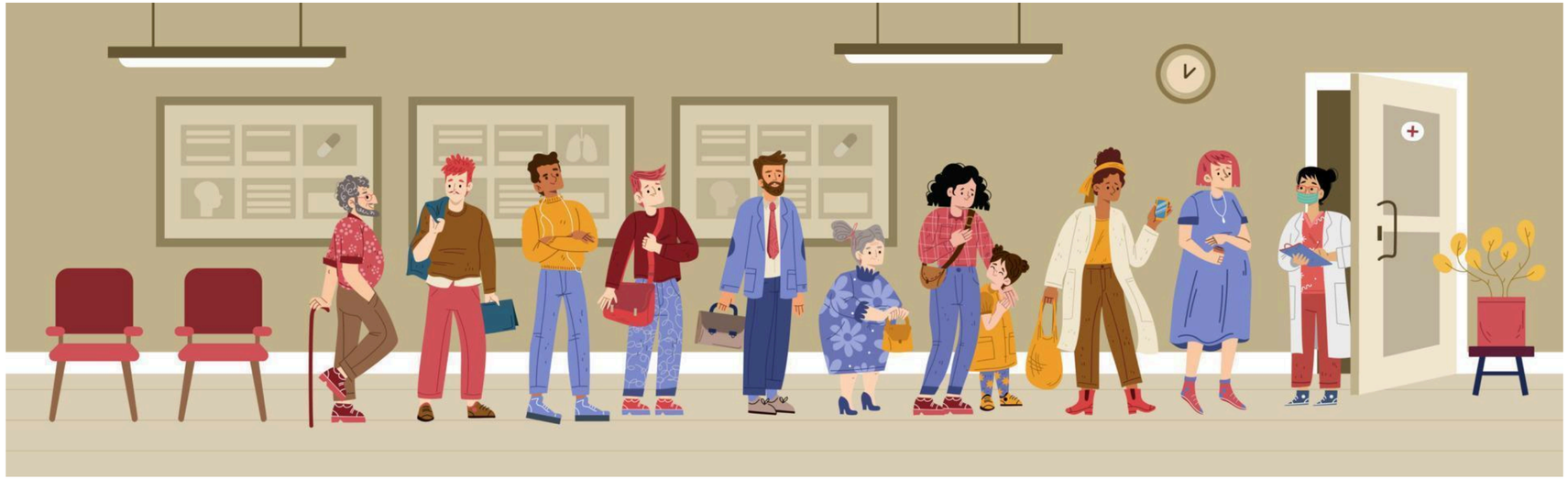
CSCI 201: Data Structures

Spring 2025

Lecture 8W: Heaps, priority queues

Goals for today:

- Design a **priority queue** -- queues where items are `remove`d according to a *priority*.
- Implement a **priority queue** using the structures we've learned so far.
- Use a **complete binary tree** to implement a **heap** (min & max).
- Represent a complete binary tree using an **array**.
- Encode a message and decode a sequence of bits using a **Huffman Coding Tree**.



A priority queue is an abstract data type that can be implemented with different data structures. Which should we use?

Main things we want:

- Ability to `add` a new item into a priority queue.
- Ability to **query** (`peek`) or **remove** (`poll`) next item with highest priority.

```java
import java.util.PriorityQueue;

public class PriorityQueueExample {
  public static void main(String[] args) {
    PriorityQueue<Integer> queue = new PriorityQueue<>();

    queue.add(10);
    queue.add(1);
    queue.add(5);
    queue.add(3);

    while (queue.size() > 0) {
      // remove the next item and print it out
      System.out.println(queue.poll());
    }
  }
}
```

- How does this work?
- For regular queues, we used either `ArrayList` or `LinkedList`.
- Let's consider how these could be used for priority queues too.
- But how do we find the *highest* priority item?

💡 **Idea 1:** look for it!

💡 **Idea 2:** keep the items sorted!

Exploring these ideas with an `ArrayList` or `LinkedList`.

Idea 1: (unsorted) add to beginning or end, *look* for highest priority item. (hpi)

- `ArrayList`:  add to end: O(1) search for hpi: O(n)
- `LinkedList`:  with tail add to end: O(1) search for hpi: O(n)

Idea 2: (sorted) add in appropriate place, remove highest priority item from beginning or end.

- `ArrayList`:  add: O(n) remove from end: O(1)
- `LinkedList`:  with tail add: O(n) remove from end or beginning: O(1)

Is there a way to have something in between $O(1)$ and $O(n)$ for both `add` and `poll`?

Yes. We can use a *heap* (like what Java uses).

Class PriorityQueue<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractQueue<E>
      java.util.PriorityQueue<E>

Type Parameters:

`E - the type of elements held in this collection`

All Implemented Interfaces:

`Serializable, Iterable<E>, Collection<E>, Queue<E>`

```
public class PriorityQueue<E>
extends AbstractQueue<E>
implements Serializable
```

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a `Comparator` provided at queue construction time, depending on which constructor is used. A priority queue does not permit `null` elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in `ClassCastException`).

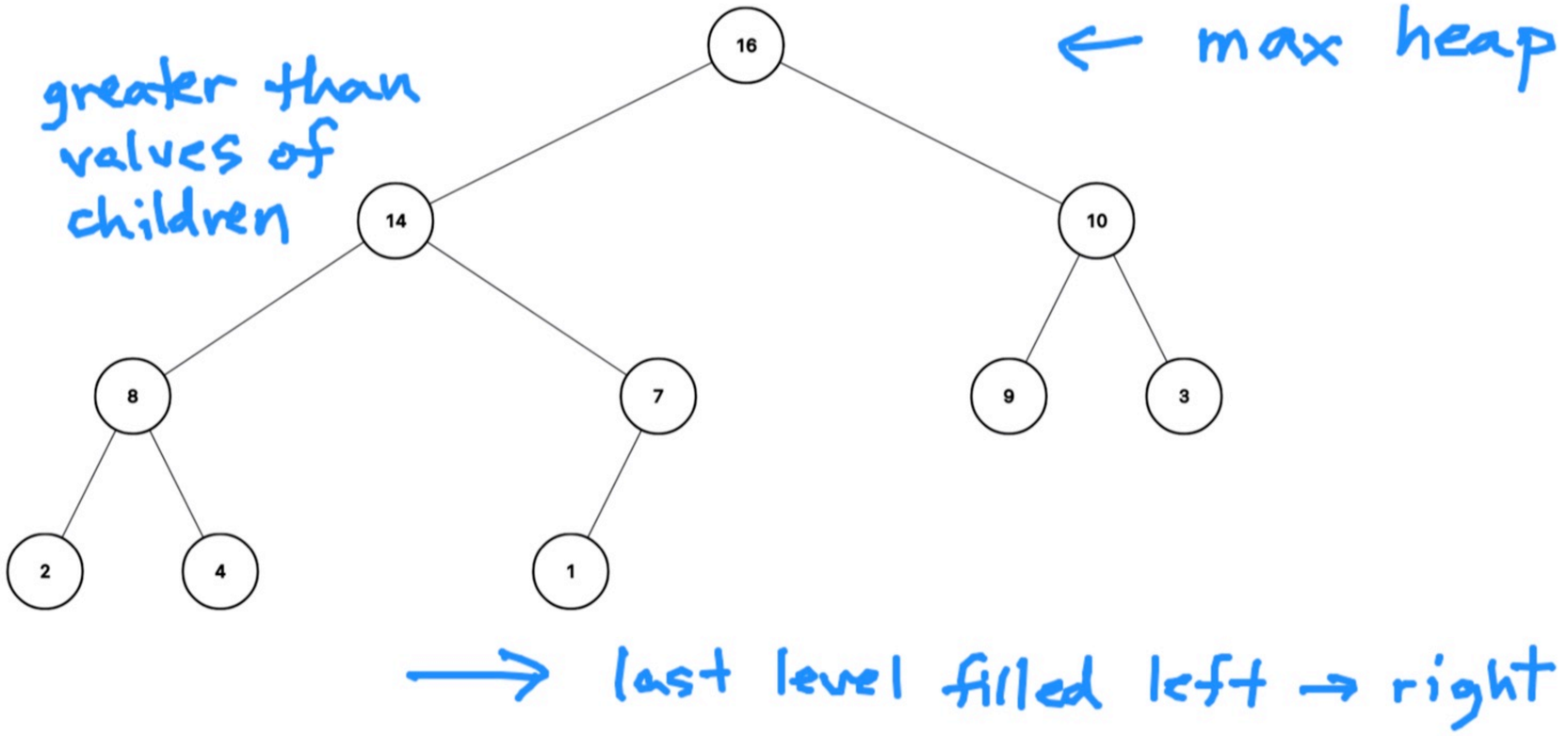
https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

Implementation note: this implementation provides O(log(n)) time for the enqueuing and dequeuing methods (`offer`, `poll`, `remove()` and `add`); linear time for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods (`peek`, `element`, and `size`).

add/offer, remove/poll are O(log n)

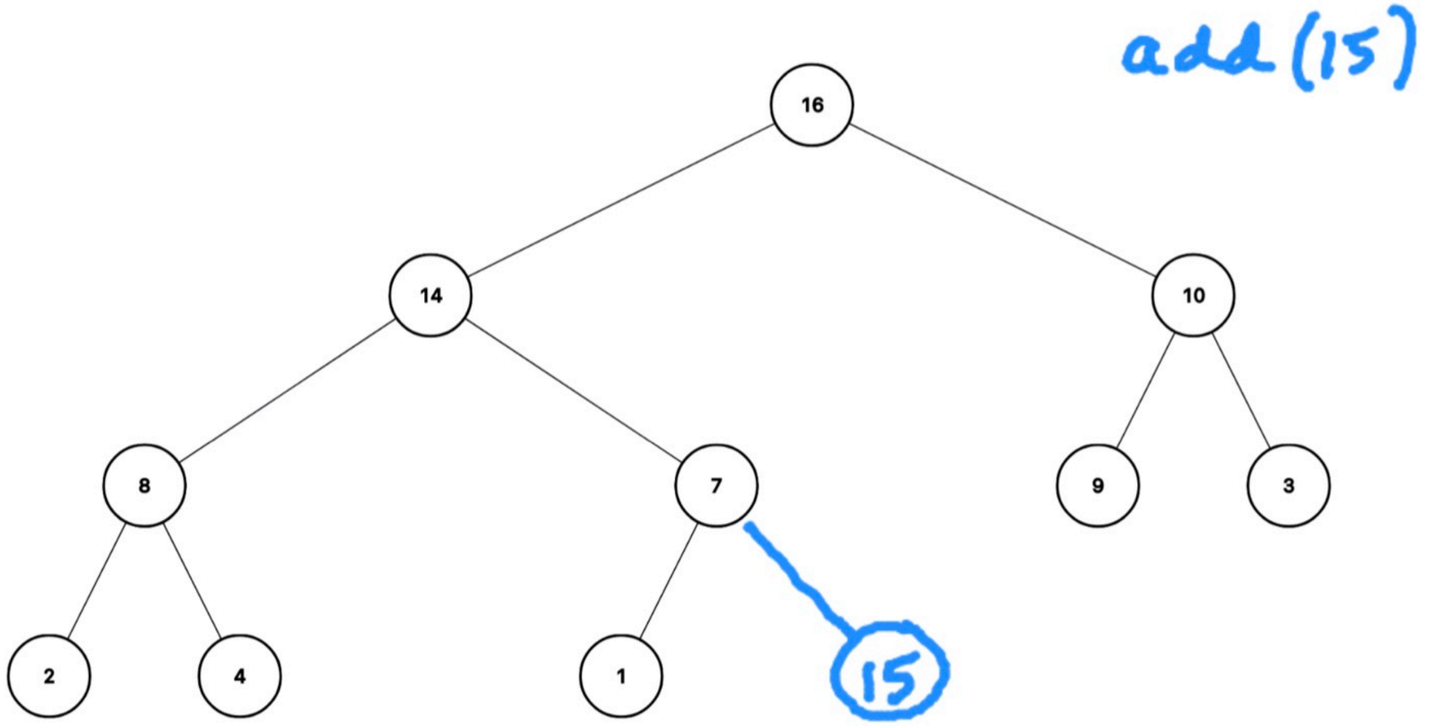
A *heap* is a binary tree with two extra properties.

1. It is complete (all levels are filled except possibly the last, which is filled left to right).
2. It satisfies a *heap property*:
    - **For a max-heap:** Every node value is *greater* than (or equal to) the values of its child nodes. So the root is the largest!
    - **For a min-heap:** Every node value is *less* than (or equal to) the values of its child nodes. So the root is the smallest!
    - We need to maintain this property when adding to (`add`) or removing from (`poll`) the heap.



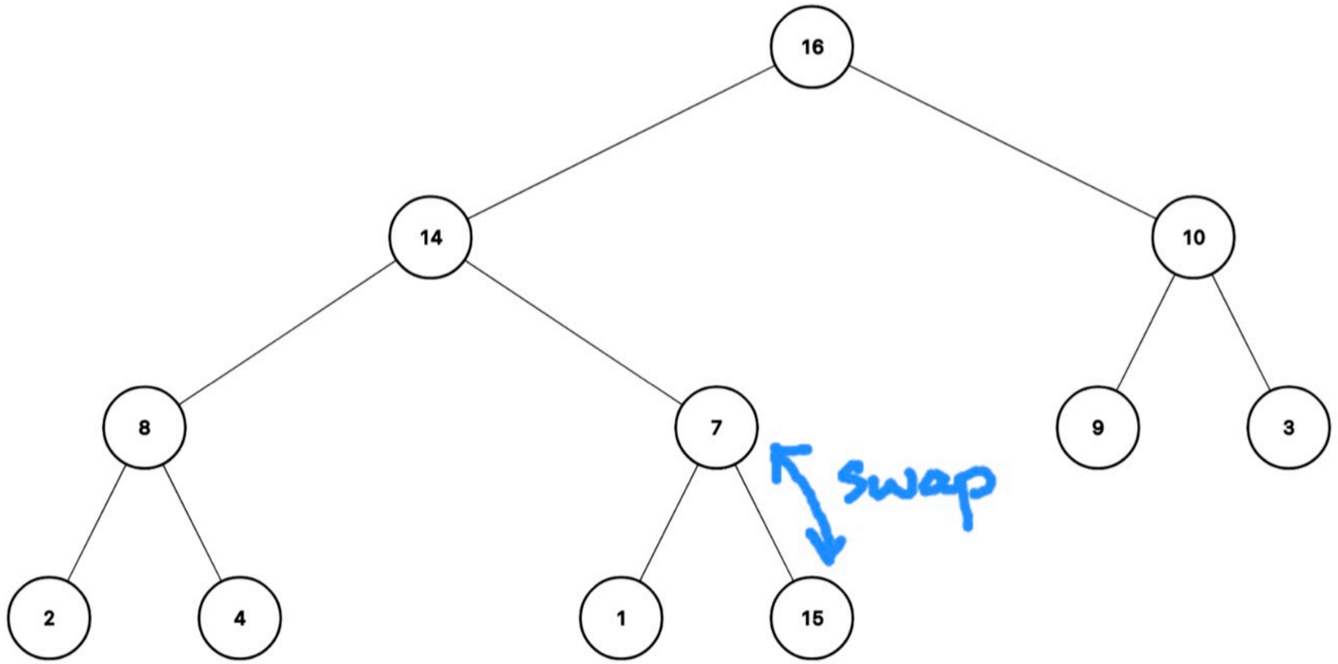
Adding (add) a value to the heap:

1. Make a new leaf node (maintaining a complete binary tree) to hold this value.
2. Set the current node to this new leaf node.
3. `while` heap property not satisfied:
    - Swap the values of the current node with the parent node.
    - Set the current node to the parent node.



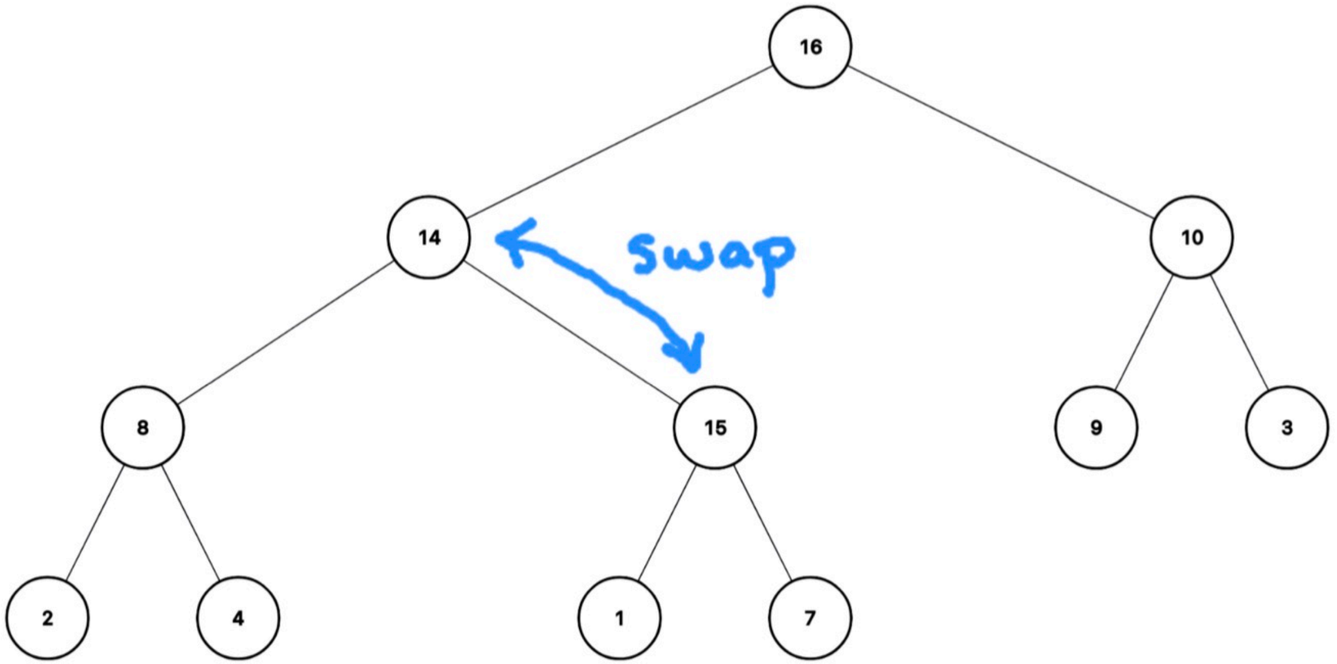
Adding (add) a value to the heap:

1. Make a new leaf node (maintaining a complete binary tree) to hold this value.
2. Set the current node to this new leaf node.
3. `while` heap property not satisfied:
    - Swap the values of the current node with the parent node.
    - Set the current node to the parent node.



Adding (add) a value to the heap:

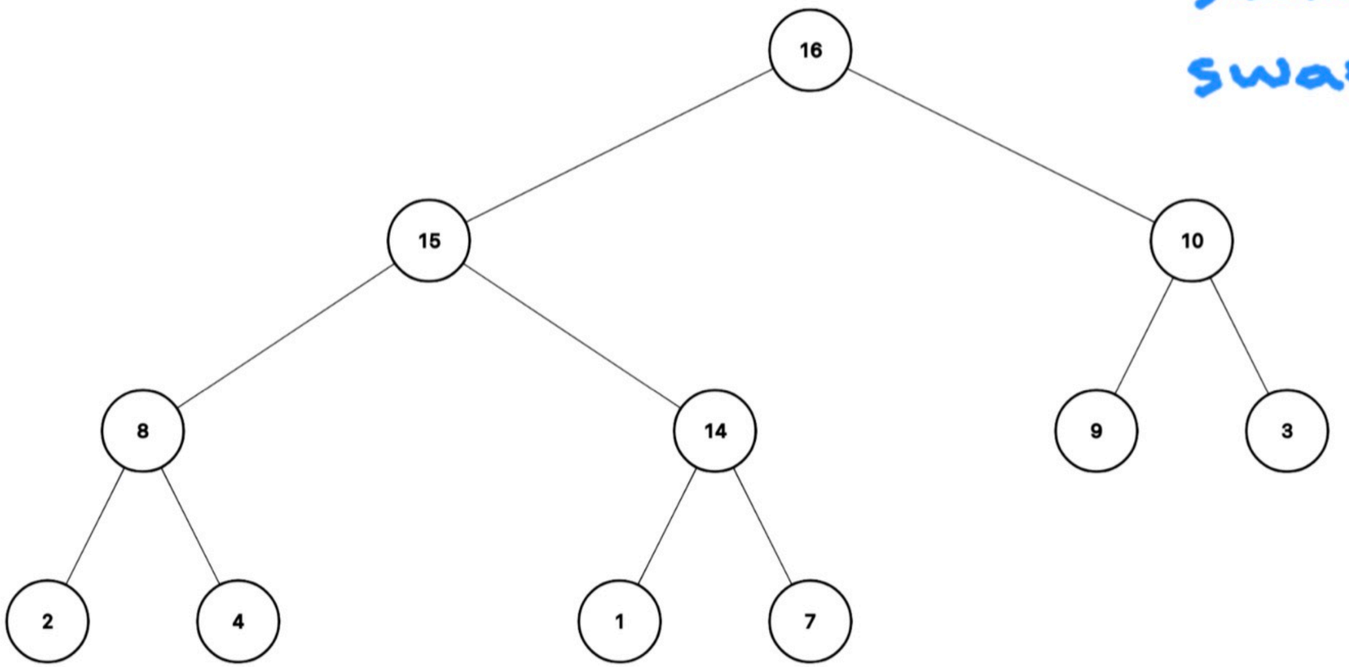
1. Make a new leaf node (maintaining a complete binary tree) to hold this value.
2. Set the current node to this new leaf node.
3. `while` heap property not satisfied:
    - Swap the values of the current node with the parent node.
    - Set the current node to the parent node.



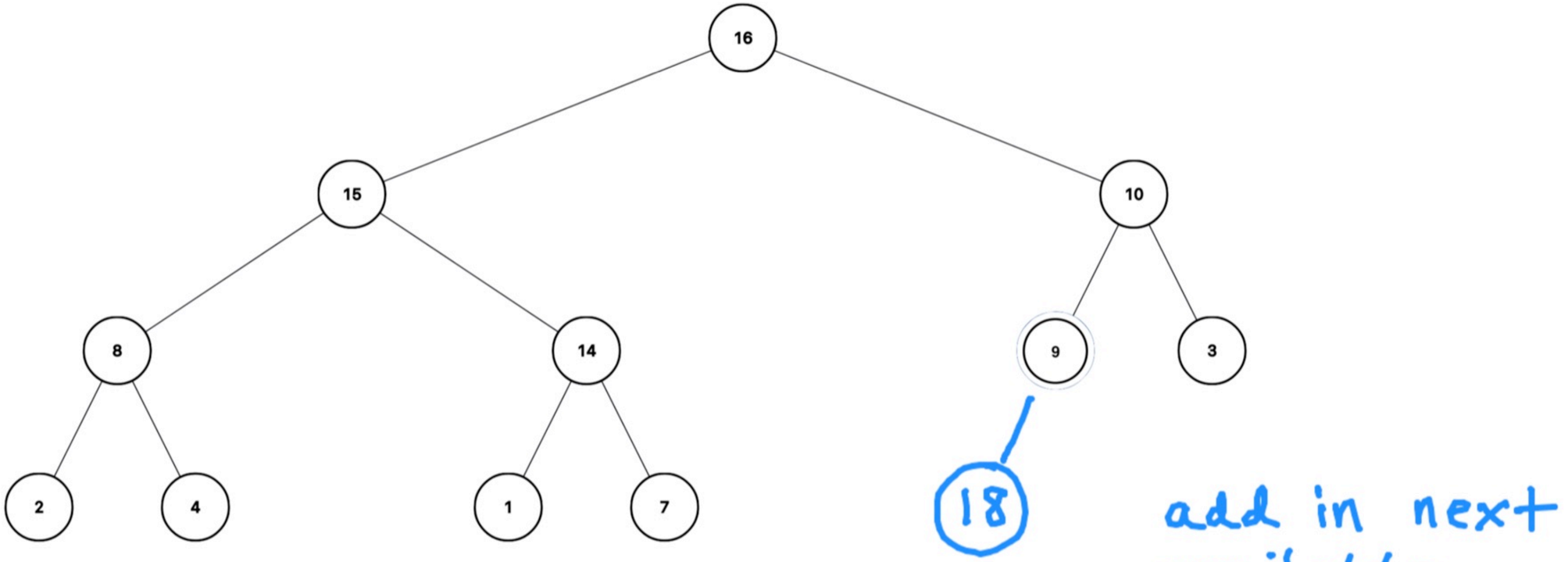
Adding (add) a value to the heap:

1. Make a new leaf node (maintaining a complete binary tree) to hold this value.
2. Set the current node to this new leaf node.
3. `while` heap property not satisfied:
    - Swap the values of the current node with the parent node.
    - Set the current node to the parent node.

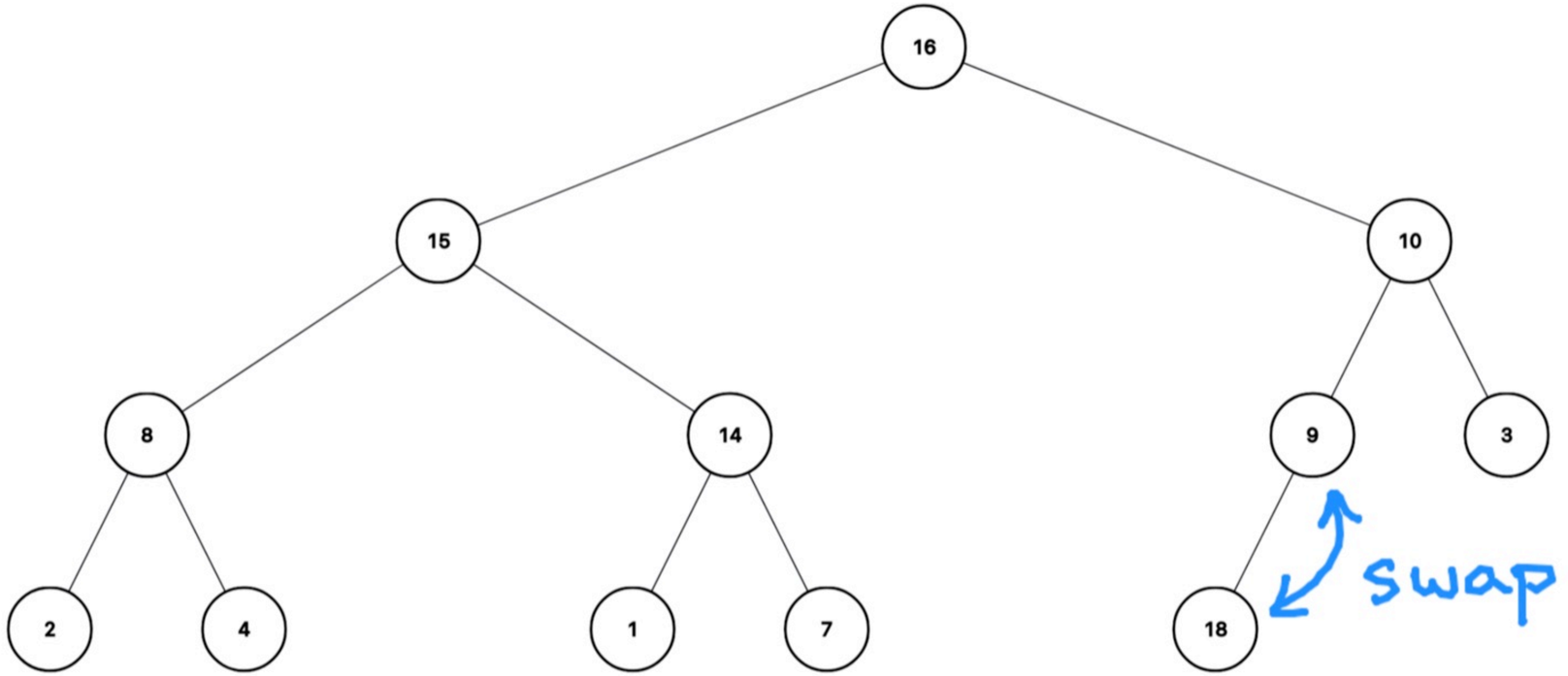
to add(15):
swap 7,15
swap 14,15
done



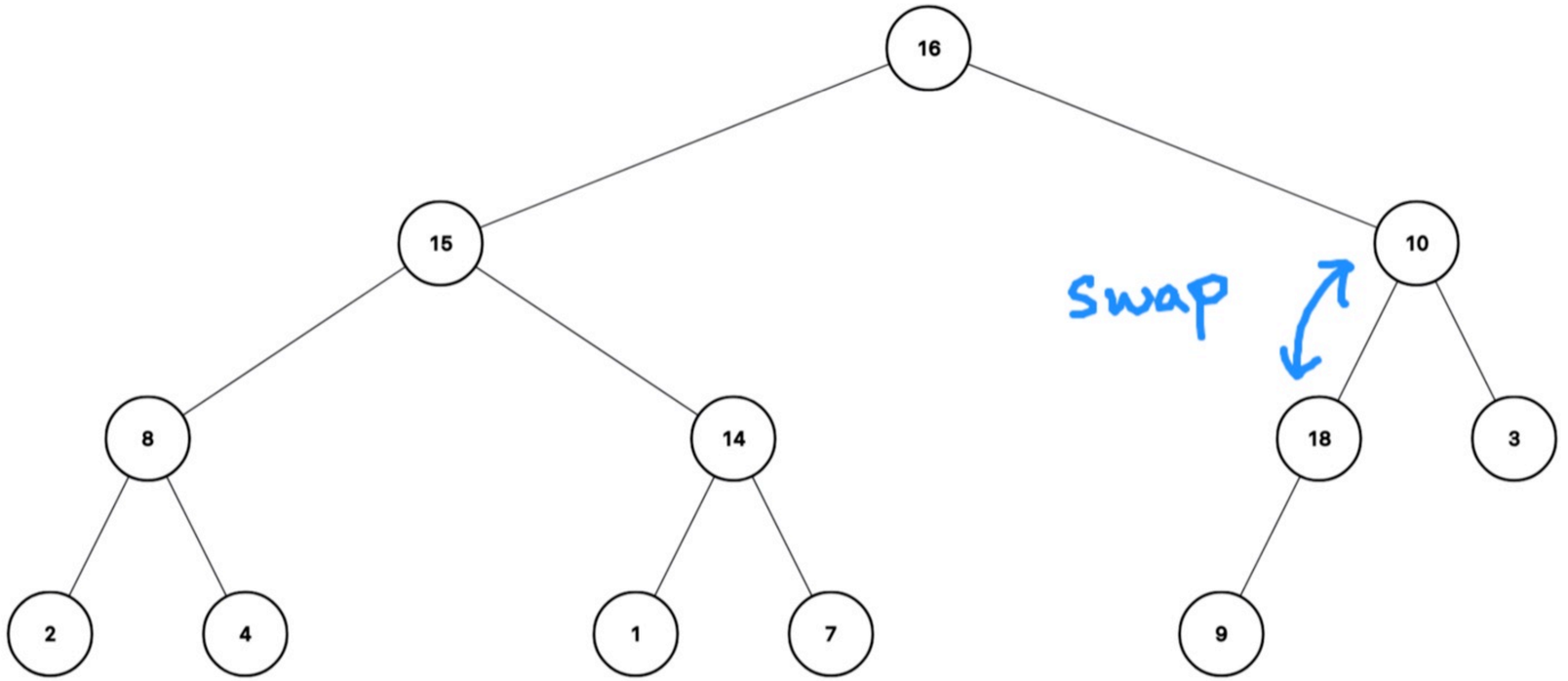
Exercise: add the value 18 to the heap, i.e. add(18).



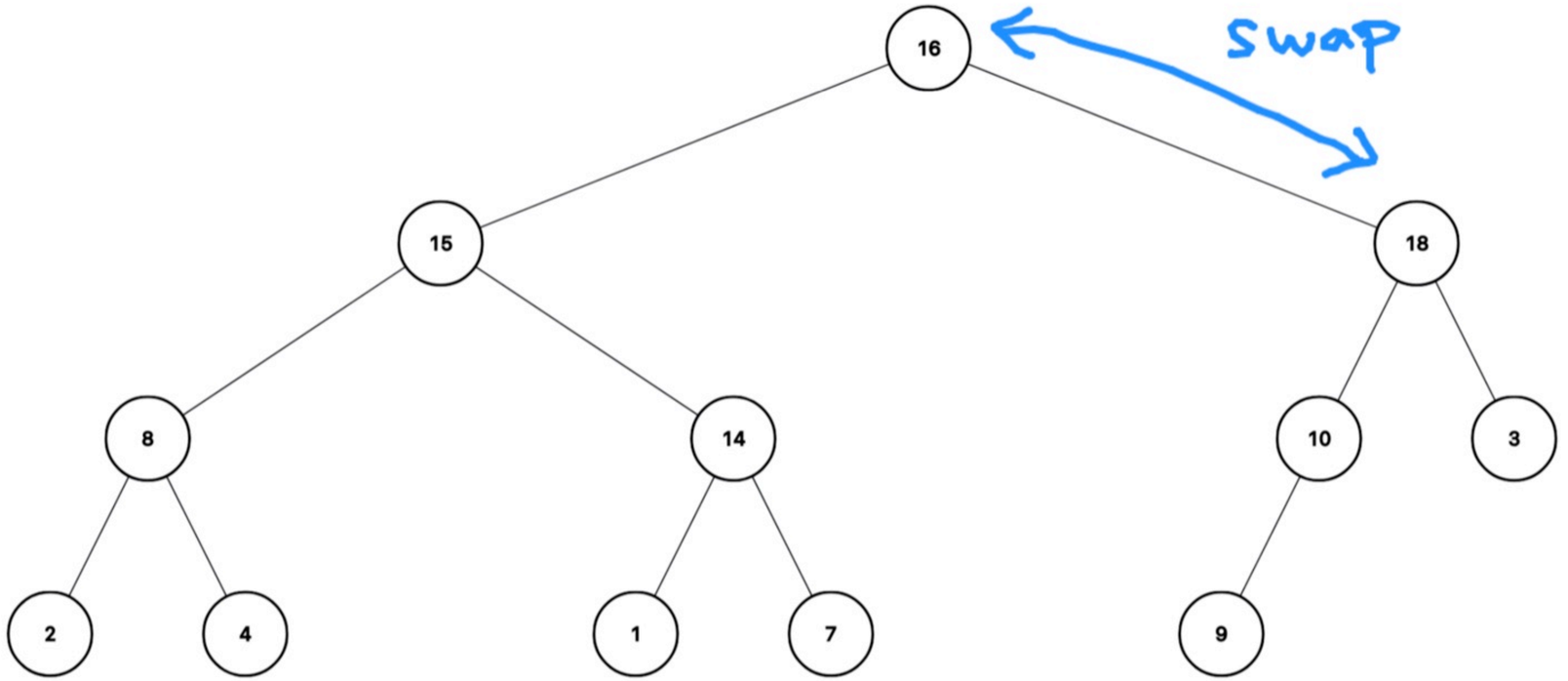
Exercise: add the value 18 to the heap, i.e. add(18).



Exercise: add the value 18 to the heap, i.e. add(18).

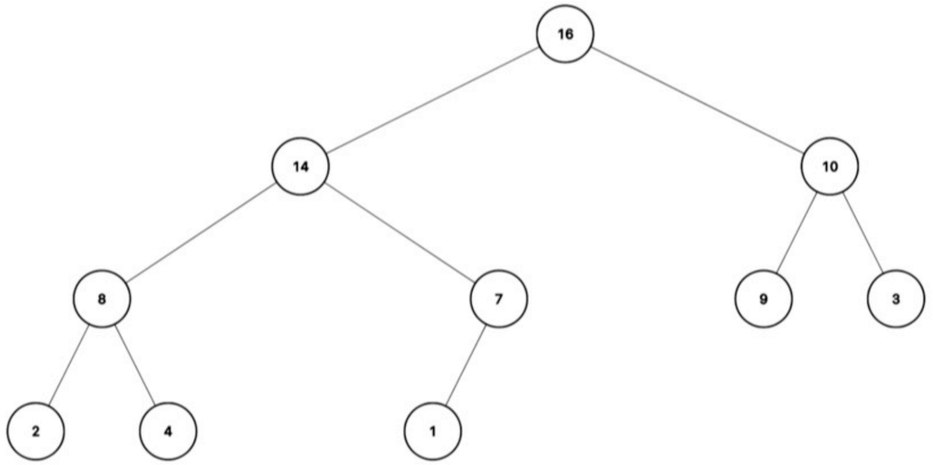


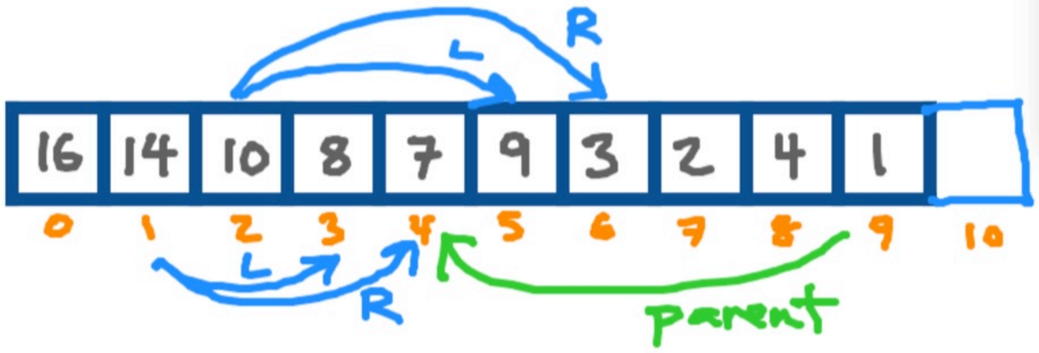
Exercise: add the value 18 to the heap, i.e. add(18).



Alternatively, complete binary trees can be represented nicely with an array.

Main idea: index nodes top-to-bottom and left-to-right within each level.





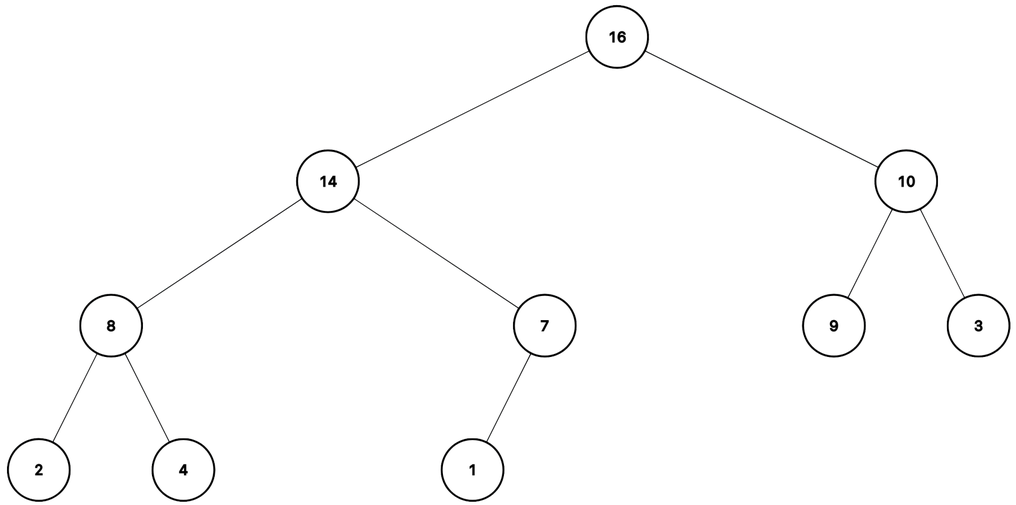

```java
class CompleteBinaryTre<E extends Comparable<E>> {
  private ArrayList<E> data;

  public CompleteBinaryTree() {
    data = new ArrayList<>();
  }

  public CompleteBinaryTree(E[] items) {
    data = new ArrayList<>();
    for (E value : items) {
      data.add(value);
    }
  }

  public static int left(int i) {
    return 2 * i + 1;
  }

  public static int right(int i) {
    return 2 * (i + 1);
  }

  public static int parent(int i) {
    return (i - 1) / 2;
  }
}
```

left child of i: $2i+1$
right child of i: $2i+2$
parent of i: $\lfloor \frac{i-1}{2} \rfloor$

Printing the tree using pre-order traversal with our `CompleteBinaryTree` representation.

```
1  public String toStringHelper(String padding, int index) {
2    if (index >= data.size()) return "";
3
4    String result = padding + "└──(" + data.get(index).toString() + ")\n";
5
6    padding += "│    ";
7    result += toStringHelper(padding, left(index));
8    result += toStringHelper(padding, right(index));
9
10   return result;
11 }
```

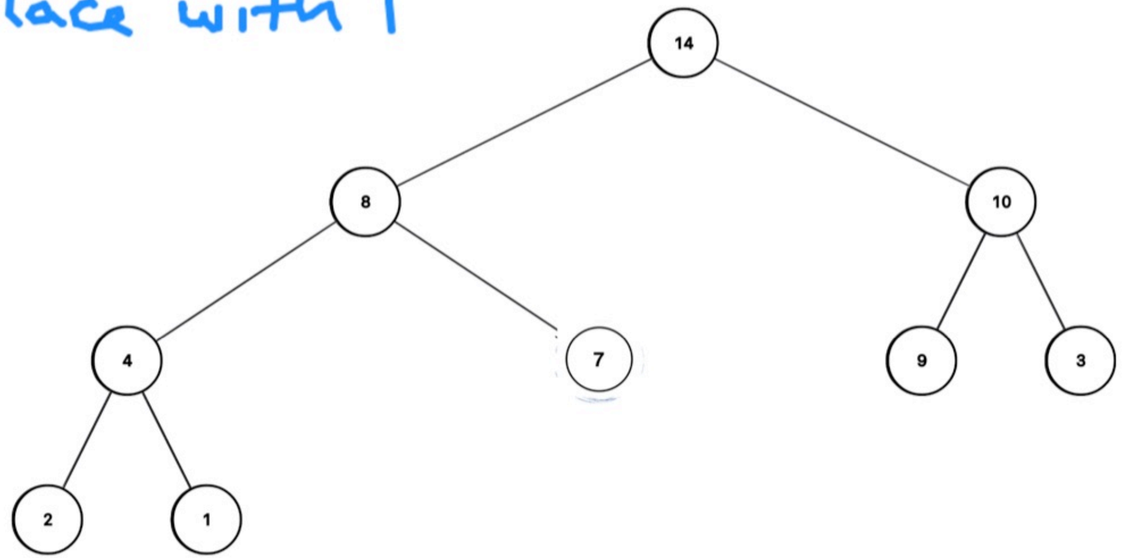
```
└──(16)
    └──(15)
        └──(8)
            └──(2)
            └──(4)
        └──(7)
            └──(1)
    └──(10)
        └──(9)
        └──(3)
```



What about removing the top (i.e. highest priority) item from the heap? Implementing the `poll` method.

1. Save the root node value (so we can return it later).
2. Set the root node value to the value of *last* node (a leaf).
3. Remove this leaf node.
4. Set the current node as the root node.
5. `while` heap property not satisfied:
    - a. Find index of which child (left or right) is largest (for max heap), or smallest (for min heap).
    - b. Swap current node value with value of index found in previous step (a).
    - c. Set current node as the child index found in step (a).

remove 16, replace with 1
swap 1, 14
swap 1, 8
swap 1, 4



to poll
heapify down

complexity
O(log n)

Consider the `isMaxHeap` method for the `CompleteBinaryTree` class.

Assume we are checking the **max-heap property** (node values $\geq$ child values).

- Loop through all nodes (entire `size()` of `data ArrayList`).
- Retrieve indices of left and right children and check heap property.

```java
public boolean isMaxHeap() {
  for (int i = 0; i < data.size(); i++) {
    int l = left(i);
    int r = right(i);
    E value = data.get(i);

    // return false if value < lValue

    // return false if value < rValue

    // return false if parent value < value

  }
  return true;
}

```

Consider the `isMaxHeap` method for the `CompleteBinaryTree` class.

Assume we are checking the **max-heap property** (node values $\geq$ child values).

- Loop through all nodes (entire `size()` of `data ArrayList`).
- Retrieve indices of left and right children and check heap property.

```java
public boolean isMaxHeap() {
  for (int i = 0; i < data.size(); i++) {
    int l = left(i);
    int r = right(i);
    E value = data.get(i);

    if (l < data.size()) {
      E lValue = data.get(l);
      if (value.compareTo(lValue) < 0) {
        // left child value is smaller than value
        return false;
      }
    }
    if (r < data.size()) {
      E rValue = data.get(r);
      if (value.compareTo(rValue) < 0) {
        // right child value is smaller than value
        return false;
      }
    }

    if (i > 0) {
      int p = parent(i);
      E pValue = data.get(p);
      if (pValue.compareTo(value) < 0) {
        // parent value is smaller than value
        return false;
      }
    }
  }
  return true;
}
```

x.compareTo(y):

x < y : < 0

x > y : > 0

x == y : 0

Huffman Coding using a priority queue

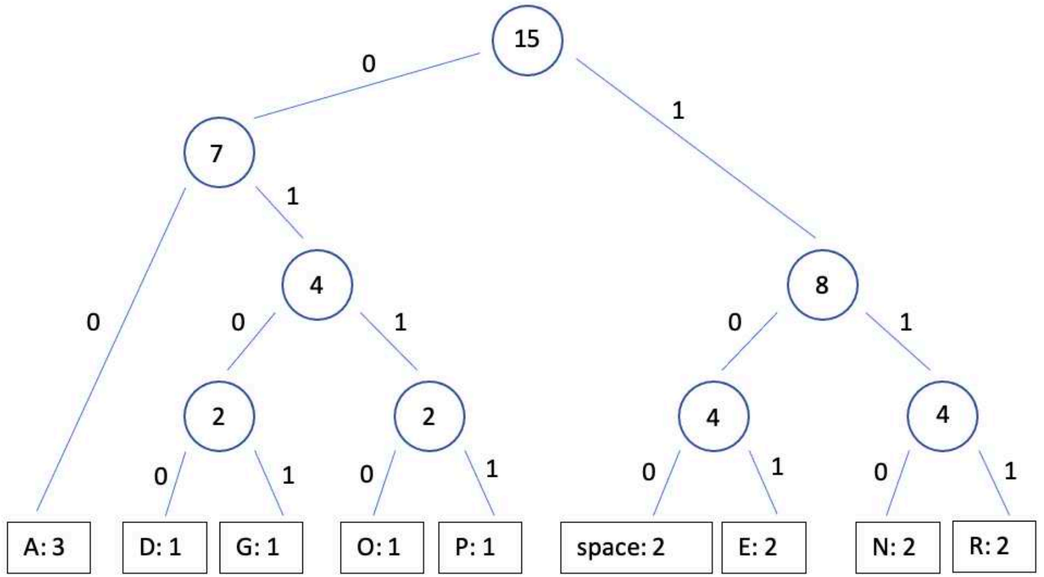
Representing data with fewer bits

- Zip

- Unicode

- JPEG

- MP3



What if we want to send the text bananabread?

11 characters

DEC	OCT	HEX	BIN	Symbol	HTML Number	HTML Name	Description
97	141	61	01100001	a	a		Lowercase a
98	142	62	01100010	b	b		Lowercase b
99	143	63	01100011	c	c		Lowercase c
100	144	64	01100100	d	d		Lowercase d
101	145	65	01100101	e	e		Lowercase e
102	146	66	01100110	f	f		Lowercase f
103	147	67	01100111	g	g		Lowercase g
104	150	68	01101000	h	h		Lowercase h
105	151	69	01101001	i	i		Lowercase i
106	152	6A	01101010	j	j		Lowercase j
107	153	6B	01101011	k	k		Lowercase k
108	154	6C	01101100	l	l		Lowercase l
109	155	6D	01101101	m	m		Lowercase m
110	156	6E	01101110	n	n		Lowercase n
111	157	6F	01101111	o	o		Lowercase o
112	160	70	01110000	p	p		Lowercase p
113	161	71	01110001	q	q		Lowercase q
114	162	72	01110010	r	r		Lowercase r
115	163	73	01110011	s	s		Lowercase s
116	164	74	01110100	t	t		Lowercase t
117	165	75	01110101	u	u		Lowercase u
118	166	76	01110110	v	v		Lowercase v
119	167	77	01110111	w	w		Lowercase w
120	170	78	01111000	x	x		Lowercase x
121	171	79	01111001	y	y		Lowercase y
122	172	7A	01111010	z	z		Lowercase z

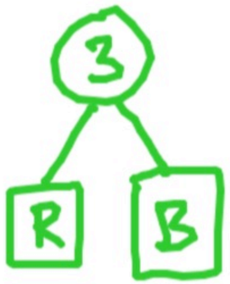
https://www.ascii-code.com/

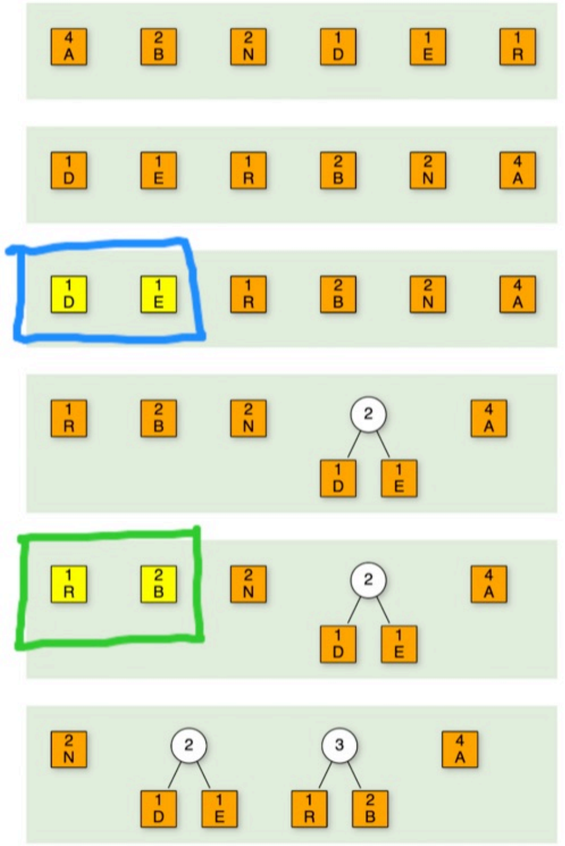
8 bits × 11 characters

b `01100010` a `01100001` ... encode string: 88 bits

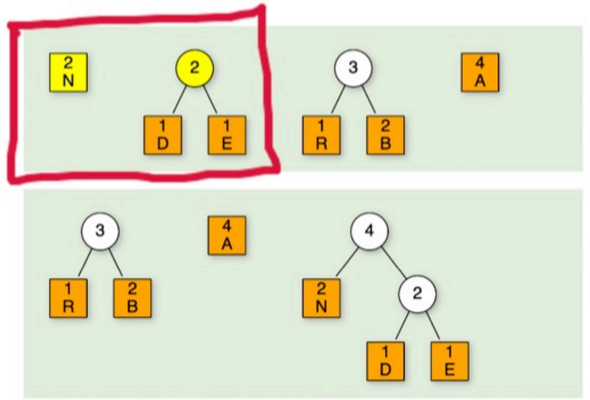
Instead, we can use a variable-length encoding: use a different number of bits for each character.

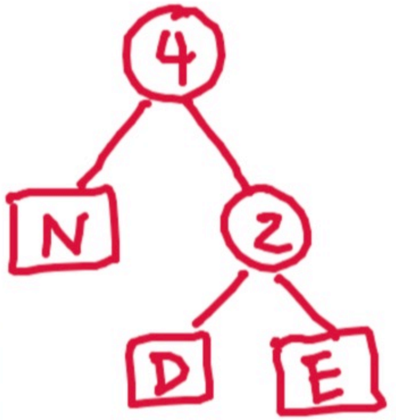
Huffman coding: encode less (more) frequent characters with more (less) bits.





1. Count frequency of each character.
2. Insert characters into **priority queue** (lower frequency has higher priority).
3. `while` priority queue is not empty:
    1. Extract (and remove) top **two** items from priority queue.
    2. Create a new internal node.
    3. Add value (frequency) of two items and assign to internal node.
    4. Make the left child of the new node the first (lower) item.
    5. Make the right child of the new node the second (higher) item.
    6. Insert new node into priority queue.

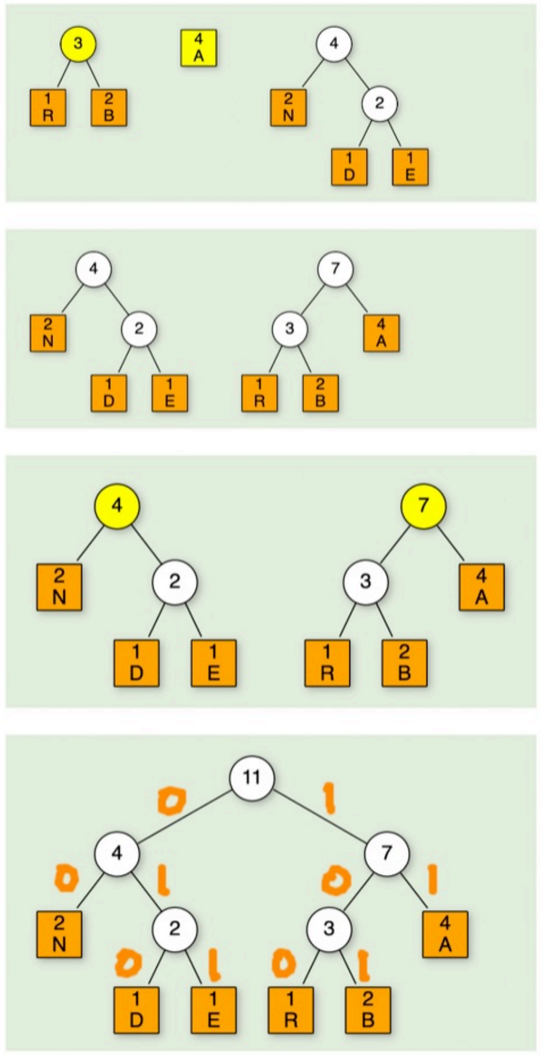




Completing the Huffman tree for bananabread.

Label left edges 0

Label right edges 1



Letter	Frequency	Code
a	4	11
b	2	101
n	2	00
d	1	010
e	1	011
r	1	100

← edge labels from root to leaf

Encoding:

b a n a n a b r e a d

101 11 00 11 00 11 101 100 011 11 010

27 bits!

Decoding: traverse tree to retrieve characters (until leaf)

Decoding practice:

1 1 1 0 0 1 0 1 0 a n d

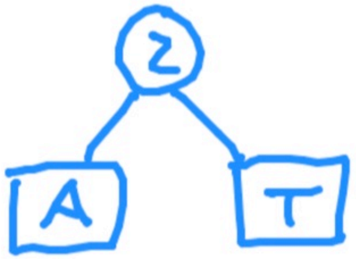
1 0 0 1 0 1 1 1 0 1 0 r e d

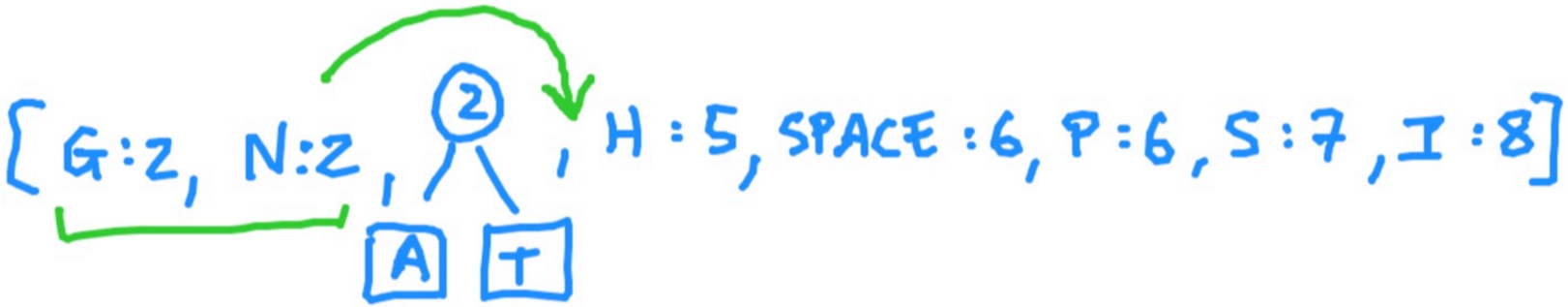
1 0 0 1 0 1 1 1 1 1 0 1 0 r e a d

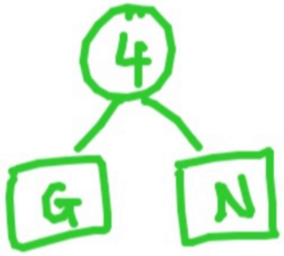
Consider this priority queue.

[A:1, T:1, G:2, N:2, H:5, SPACE:6, P:6, S:7, I:8]

A few steps in building the Huffman Coding Tree:

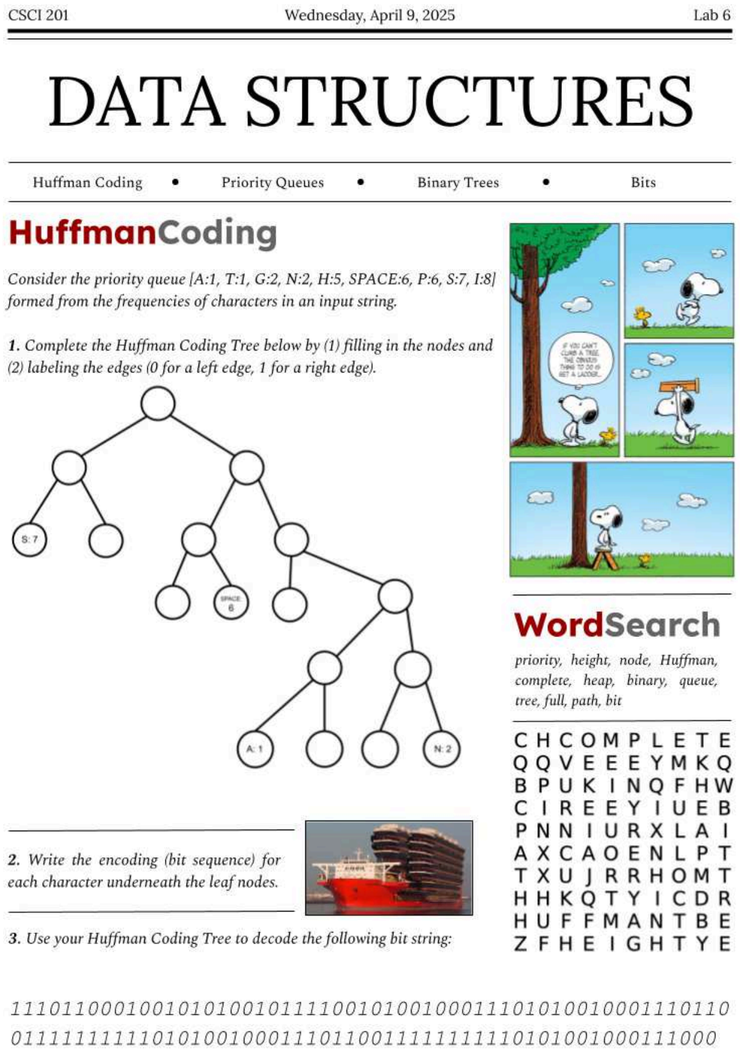






Lab 6 assignment:

- Complete the Lab 6 worksheet to build the Huffman tree for the given frequency of characters.
- Use your tree to decode a sequence of bits.
- Submit worksheet results to Canvas quiz (unlimited attempts).
- Nothing to submit to Gradescope for this lab.



CSCI 201 Wednesday, April 9, 2025 Lab 6

DATA STRUCTURES

Huffman Coding • Priority Queues • Binary Trees • Bits

HuffmanCoding

Consider the priority queue [A:1, T:1, G:2, N:2, H:5, SPACE:6, P:6, S:7, I:8] formed from the frequencies of characters in an input string.

1. Complete the Huffman Coding Tree below by (1) filling in the nodes and (2) labeling the edges (0 for a left edge, 1 for a right edge).

2. Write the encoding (bit sequence) for each character underneath the leaf nodes.

3. Use your Huffman Coding Tree to decode the following bit string:

11101100010010101001011110010100100011101010010001110110
01111111111010100100011101100111111111101010010001111000

WordSearch

priority, height, node, Huffman, complete, heap, binary, queue, tree, full, path, bit

C H C O M P L E T E
Q Q V E E E Y M K Q
B P U K I N Q F H W
C I R E E Y I U E B
P N N I U R X L A I
A X C A O E N L P T
T X U J R R H O M T
H H K Q T Y I C D R
H U F F M A N T B E
Z F H E I G H T Y E

Notes:

- Homework 6 due Thursday 4/10: implement a calculator (using a stack) & mid-semester check-in.
- Lab 6 due Monday 4/14: complete the Lab 6 worksheet on your own or with a study partner. Use a priority queue to encode messages efficiently!
- No lab meetings on Friday 4/11 -- attend the Spring Student Symposium.
- If you want to make your own trees, have a look at this app: https://tree-visualizer.netlify.app/ (trees for today's class were made with it).
- Reminder that Noah (go/noah) and Smith (go/smith) have office hours throughout the week and the 201 Course Assistants have drop-in hours in the late afternoons/evenings (go/cshelp).