CS 106 INTRODUCTION TO DATA STRUCTURES

SPRING 2020

PROF. SARA MATHIESON

HAVERFORD COLLEGE

ADMIN

- **Lab 5 due Sunday (email me if you would like to choose a new deadline)**
- **Lab 6 posted this weekend**
- **Questionnaire fill out if you haven't already!**
- **Email me (and cc partner(s)) if you want to work together in breakout rooms (or prefer individual)**

Remind me to RECORD

REVISED TA/OFFICE HOURS

Sunday 7-9pm (Juvia)

Monday 8-midnight (Steve)

Tuesday 11:30-12:30pm (Lizzie)

Tuesday 4:30-6pm (Sara)

Wednesday 8-midnight (Steve)

Today/Tomorrow

Thursday 11:30-12:30pm (Lizzie)

Thursday 9-11pm (Will)

Friday 8-10pm (Gareth)

Saturday 4-6pm (Will)

Saturday 8-10pm (Gareth)

QUESTIONNAIRE

Time zones: range from -4 to +13

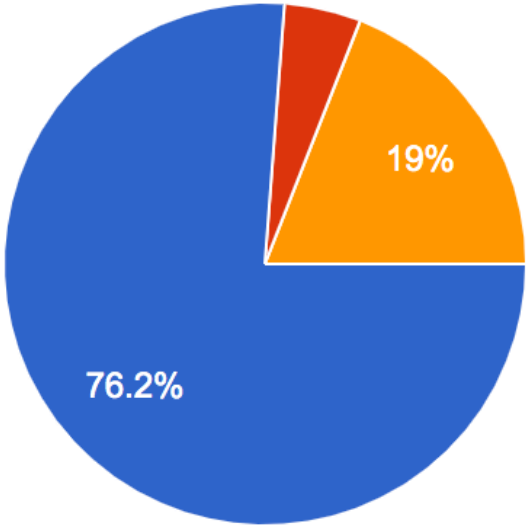
Breakout rooms and private chat: generally positive

- Requests to work with partners (will try today)
- Maybe more than 2 people

Major drawbacks (besides the generally difficult situation right now)

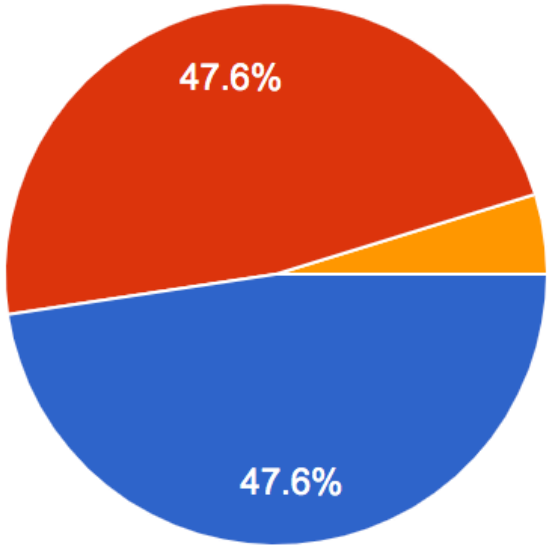
- Lack of peer collaboration opportunities
- Live coding not working so well right now
- More clarity in lab instructions
- Whiteboard is not the same

QUESTIONNAIRE



Synchronous (i.e. real-time Zoom meetings)
Asynchronous (videos + exercises on your own)
No preference

Synchronous vs. asynchronous



Printer

Yes
No
Maybe

QUESTIONNAIRE

Position to complete the course?

- Very mixed answers
- General stress and difficulty focusing is true for many of us
- Will continue to provide flexibility in terms of optional exercises and extra credit
- "Choose your own deadline" model
- I am re-envisioning the final project, as this seems to worry people most (it was very difficult last year and also done in pairs/groups, which is less practical now)
- There will be the chance to go deep with the final project but the required part will be only the core material

MAR 31 OUTLINE

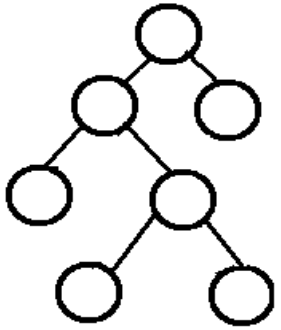
- Recap tree algorithms and operations
- Huffman encoding example
- Begin: priority queues and heaps
- Array-based implementation of a heap

MAR 31 OUTLINE

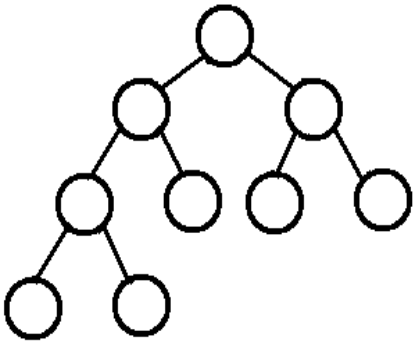
- **Recap tree algorithms and operations**
- **Huffman encoding example**
- **Begin: priority queues and heaps**
- **Array-based implementation of a heap**

TYPE OF BINARY TREES

A binary tree is **proper** (or **full**) if each node has zero or two children

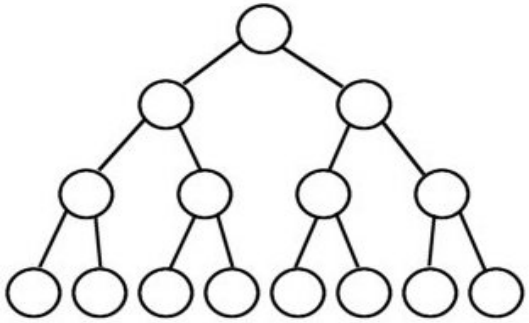


A binary tree is **complete** if every level (except possibly the last) is filled



Edit: if the last level is not filled, nodes should be as far left as possible!

If a complete binary tree is filled at every level, it is **perfect**

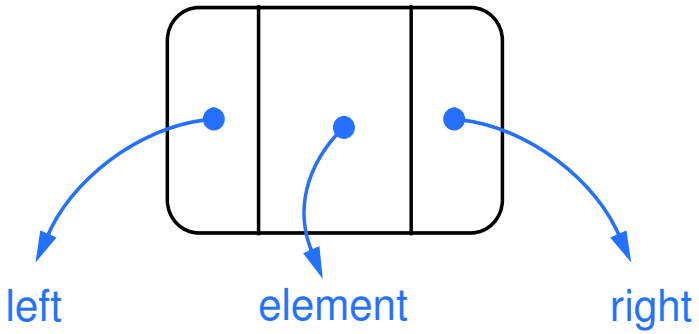


IMPLEMENTATION (LAB 5)

Fully recursive data structure!

A tree is kind of like a Node...

- *** data**
- *** left (also a BinaryTree)**
- *** right (also a BinaryTree)**



Think carefully about the constructor(s) you will need

LAB 5: EXAMPLE TESTING

```
BinaryTree<Character> letterTree = new LinkedBinaryTree<Character>();

letterTree.insert('A');
letterTree.insert('C');
letterTree.insert('G');
letterTree.insert('B');
letterTree.insert('D');
letterTree.insert('G'); // inserting again, should replace
letterTree.insert('F');
letterTree.insert('E');
letterTree.insert('H');
letterTree.insert('I');

System.out.println("size:" + letterTree.size());
System.out.println(letterTree);
```

BINARY TREE INTERFACE (LAB 5)

```
public interface BinaryTree<E extends
Comparable<E>> {

   E getRootElement();

   int size();

   boolean isEmpty();

   void insert(E element);

   boolean contains(E element);

   boolean remove(E element);

   String toStringInOrder();
   String toStringPreOrder();
   String toStringPostOrder();

}
```

Getter for the data of the root

Recommended way: recursive

Optional: extra challenge

REMOVE FOLLOW UP: TWO CHILDREN

Replace with in-order predecessor or in-order successor

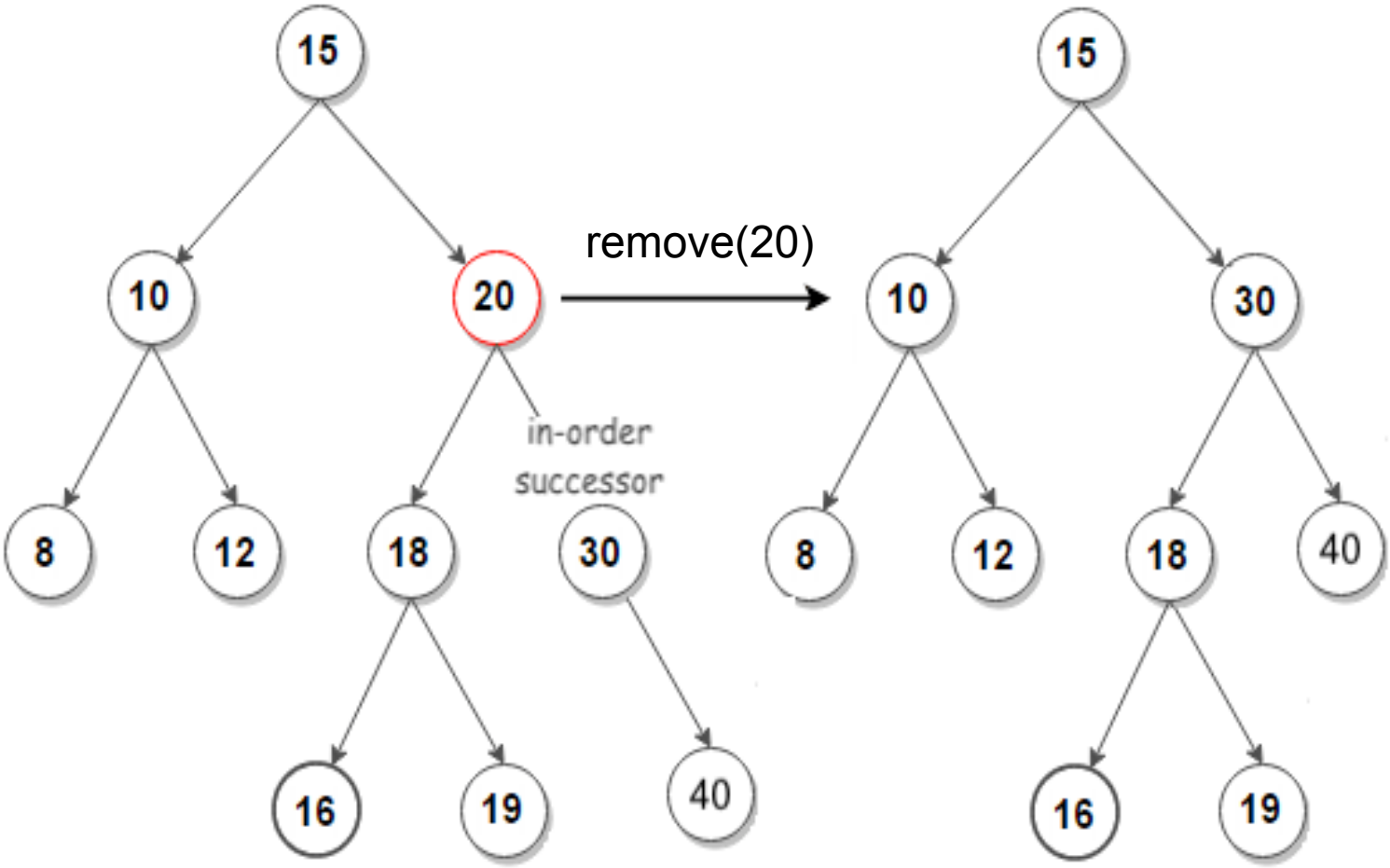
in-order predecessor

- rightmost child in left subtree
- max-value child in left subtree

in-order successor

- leftmost child in right subtree
- min-value child in right subtree

REPLACE WITH SUCCESSOR



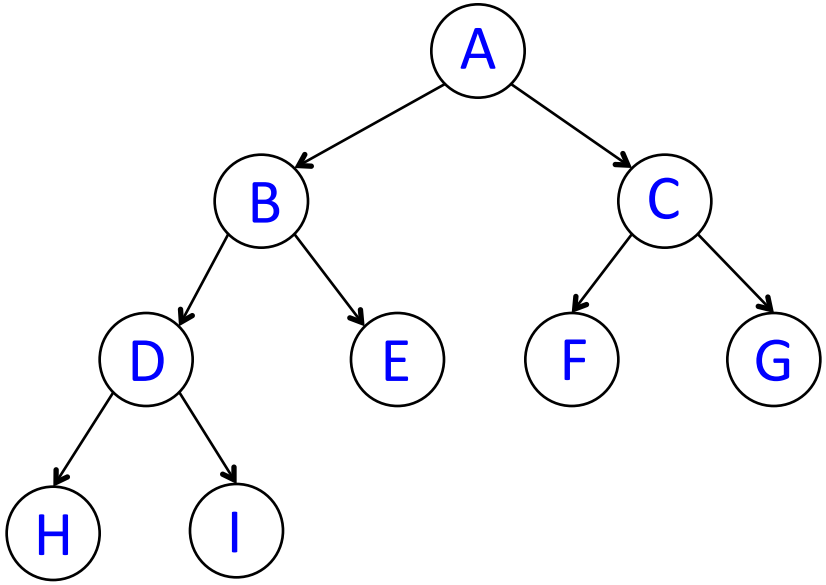
Note: this is a special case when successor is also child (replace entire node, just just data)

RUNTIME OF TRAVERSALS?

Exercise for outside of class (optional: write up in Lab 5 README)

Hint 1: all our traversals have the same runtime

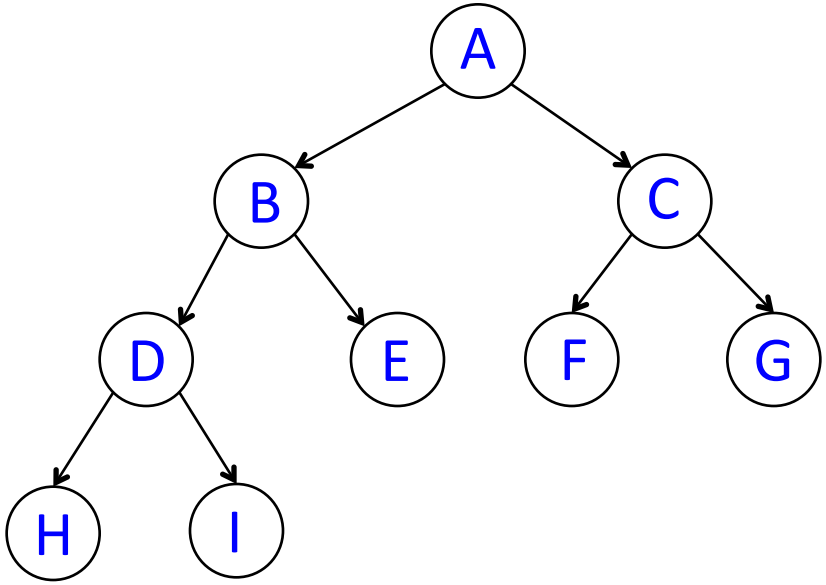
Hint 2: how many times is each node visited?



DEPTH-FIRST VS BREADTH-FIRST

All our traversals so far are depth-first
Pre-order is the most classic form of depth-first traversal/search

A B D H I E C F G



DEPTH-FIRST VS BREADTH-FIRST

All our traversals so far are depth-first
Pre-order is the most classic form of depth-first traversal/search

A B D H I E C F G

Breadth-first ("level order"):
Implement with a queue:
(add root to queue, process root, add all children to queue⋯)

A B C D E F G H I

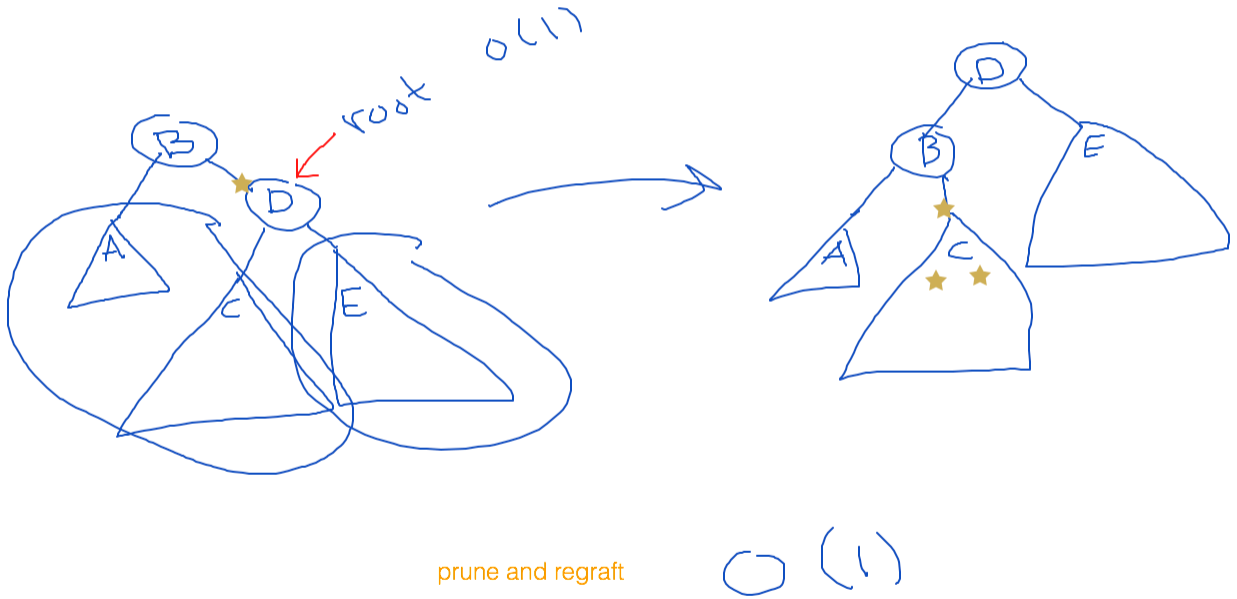
REBALANCING TREES

Many ways! Here is one (more info in link):

https://en.wikipedia.org/wiki/Tree_rotation

REBALANCING TREES

Many ways! Here is one:



Note: this maintains alphabetical order so sorting is fast, but it does change some parent/child relationships.

MAR 31 OUTLINE

- Recap tree algorithms and operations
- **Huffman encoding example**
- Begin: priority queues and heaps
- Array-based implementation of a heap

HUFFMAN CODING

Outline:

Step 1: sort by frequency and make a queue (low -> high)

Step 2: merge first two elements off the queue to form a root, then add root back into queue (respecting frequency)

Repeat until queue is empty.

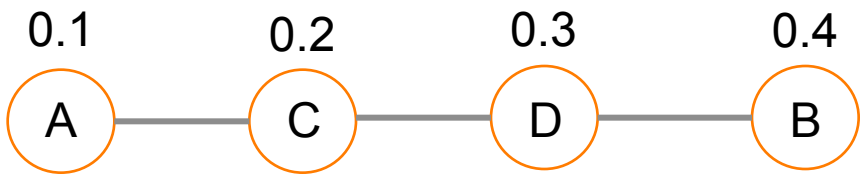
Step 3: label resulting branches of tree with 0's for left branches and 1's for right branches

HUFFMAN CODING EXAMPLE

letter	A	B	C	D
frequency	0.1	0.4	0.2	0.3
code				

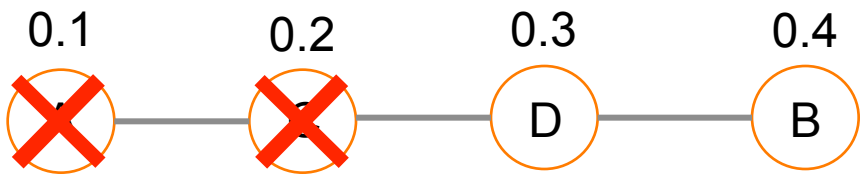
HUFFMAN CODING EXAMPLE

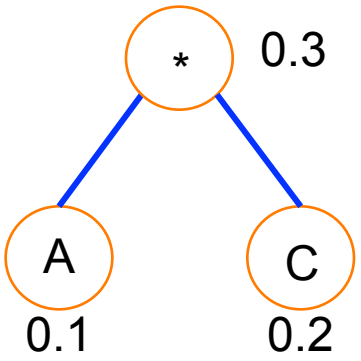
letter	A	B	C	D
frequency	0.1	0.4	0.2	0.3
code				



HUFFMAN CODING EXAMPLE

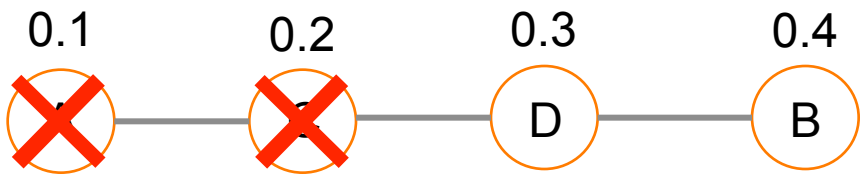
letter	A	B	C	D
frequency	0.1	0.4	0.2	0.3
code				

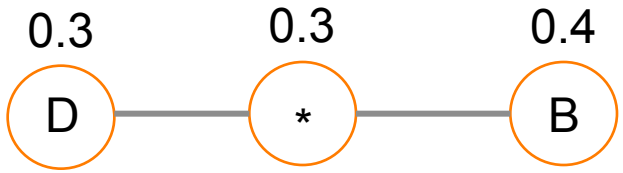


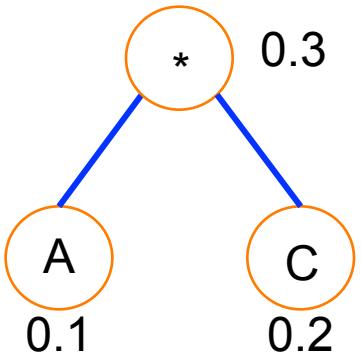


HUFFMAN CODING EXAMPLE

letter	A	B	C	D
frequency	0.1	0.4	0.2	0.3
code				

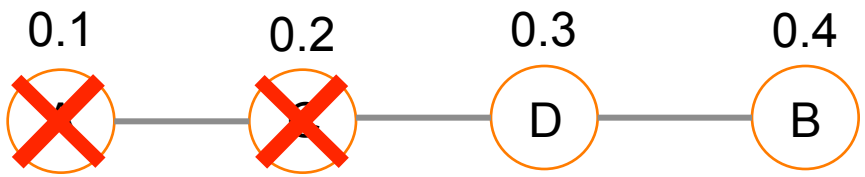


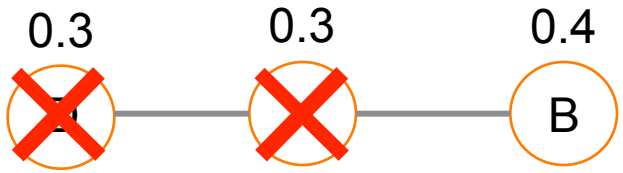


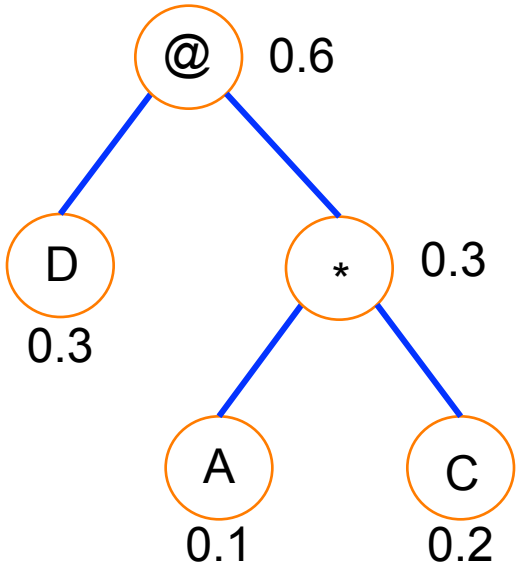


HUFFMAN CODING EXAMPLE

letter	A	B	C	D
frequency	0.1	0.4	0.2	0.3
code				

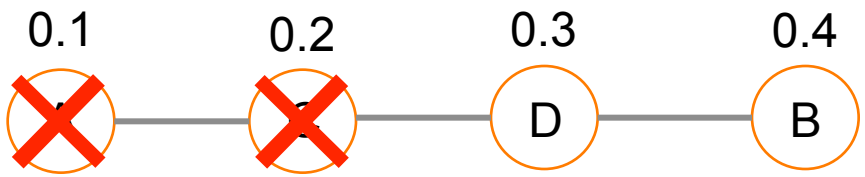


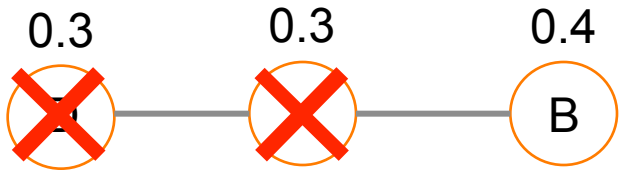


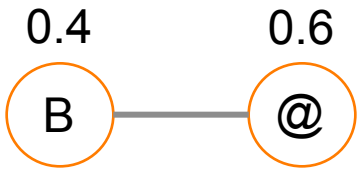


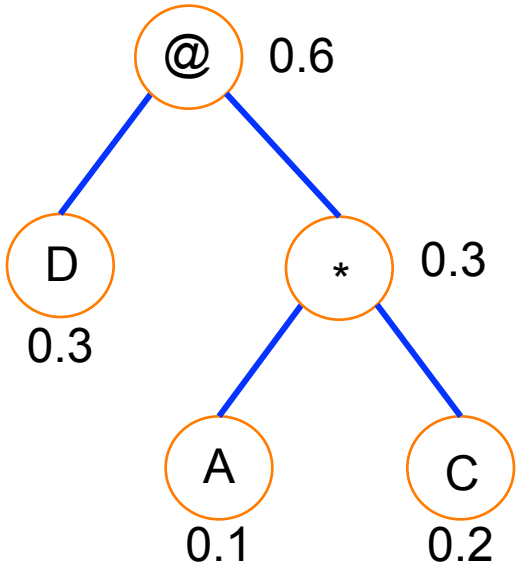
HUFFMAN CODING EXAMPLE

letter	A	B	C	D
frequency	0.1	0.4	0.2	0.3
code				



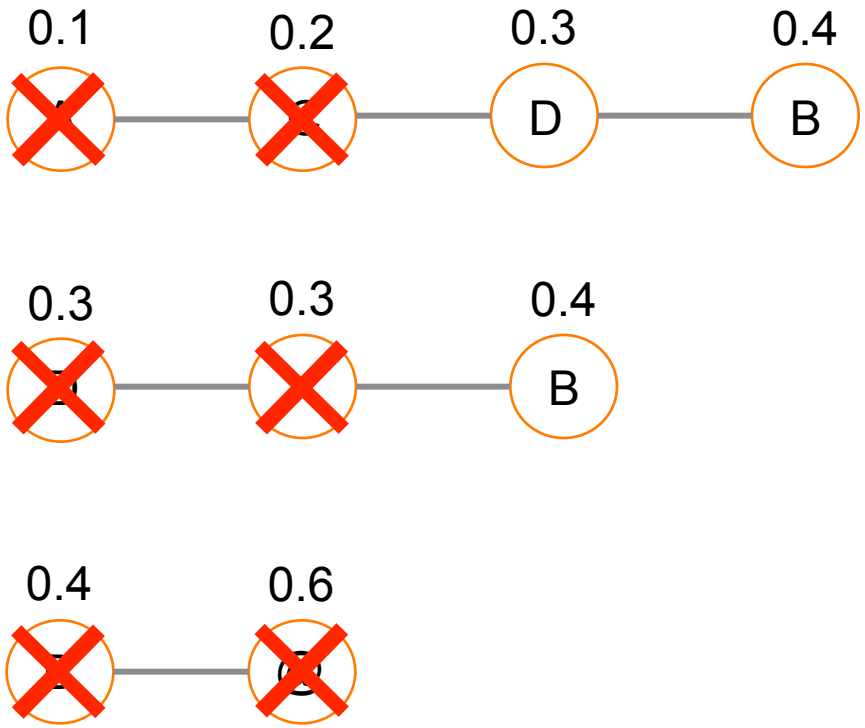


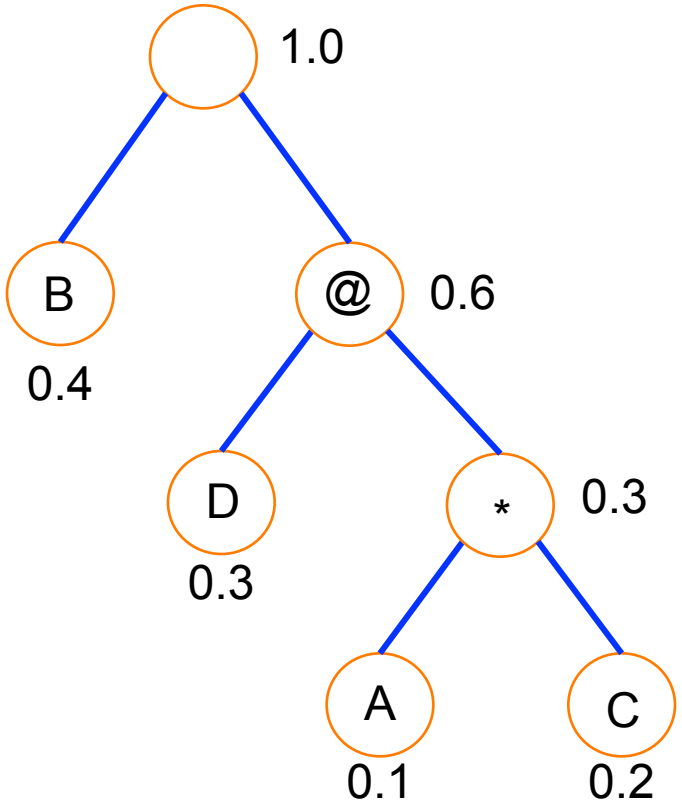




HUFFMAN CODING EXAMPLE

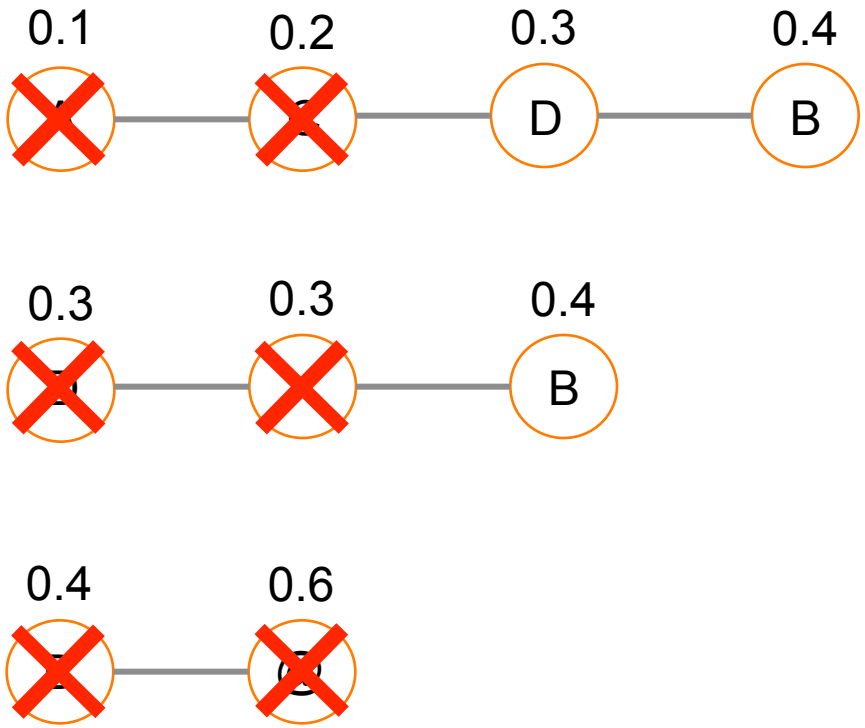
letter	A	B	C	D
frequency	0.1	0.4	0.2	0.3
code				

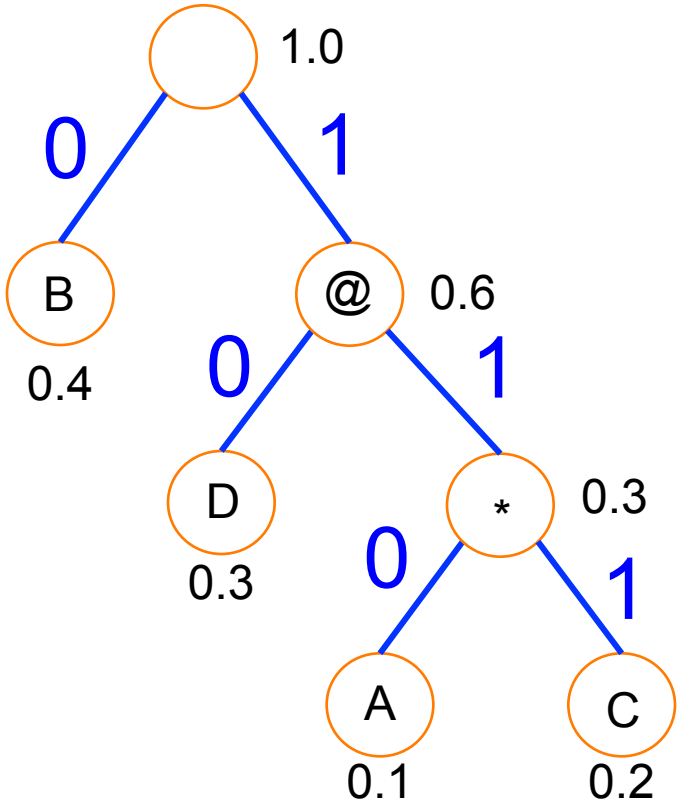




HUFFMAN CODING EXAMPLE

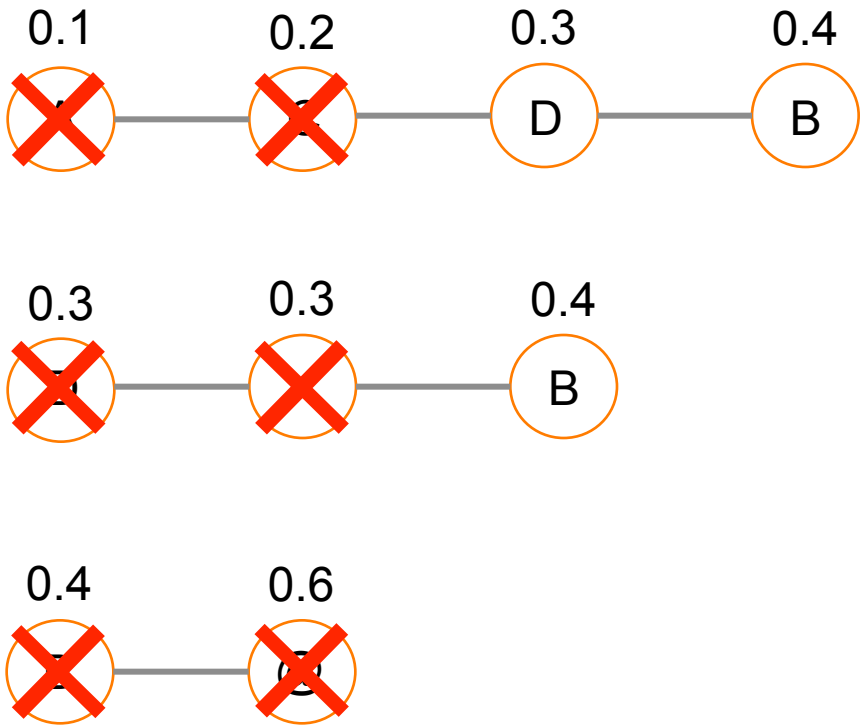
letter	A	B	C	D
frequency	0.1	0.4	0.2	0.3
code				

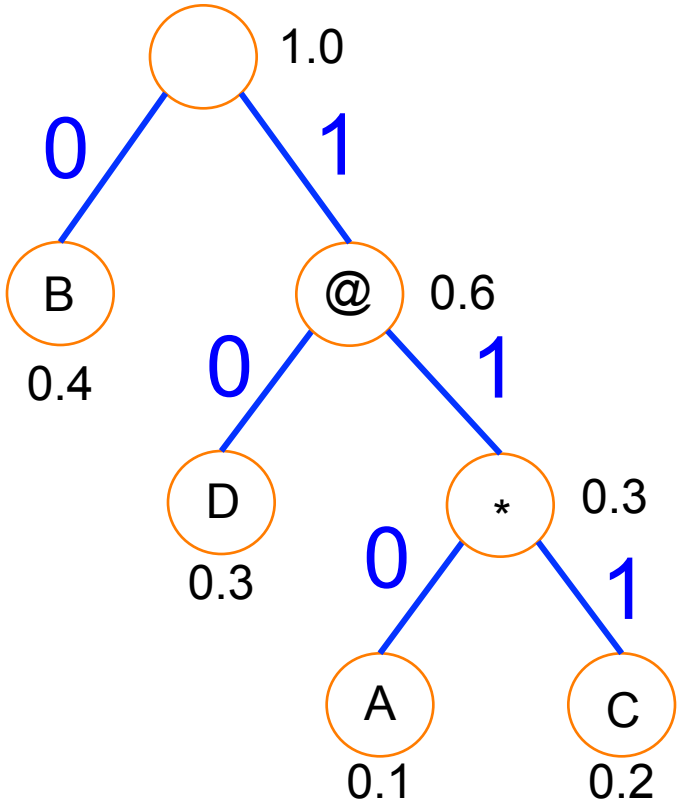




HUFFMAN CODING EXAMPLE

letter	A	B	C	D
frequency	0.1	0.4	0.2	0.3
code	110	0	111	10

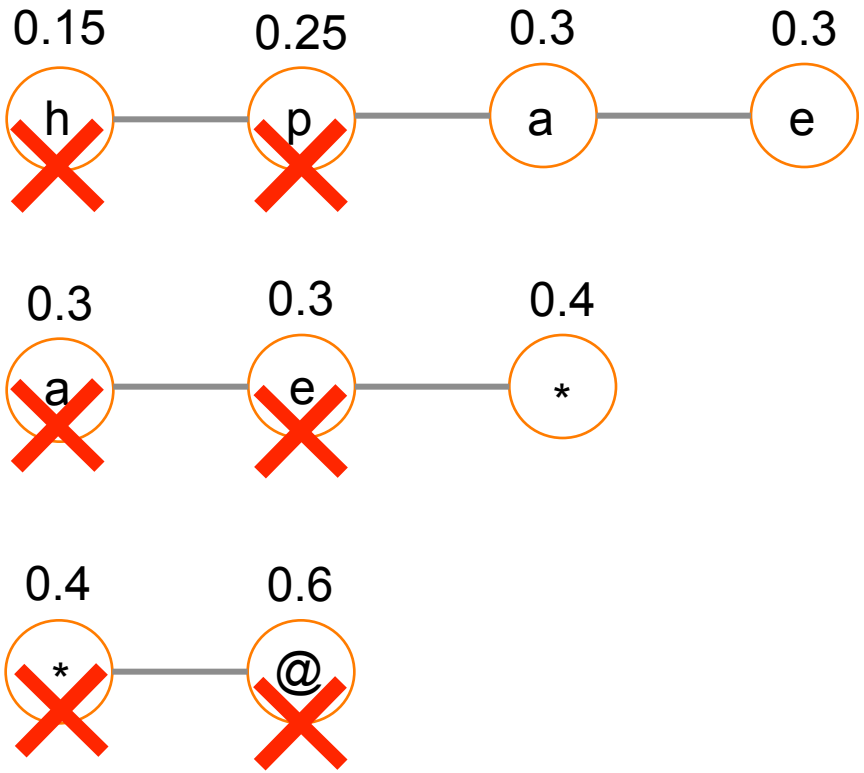




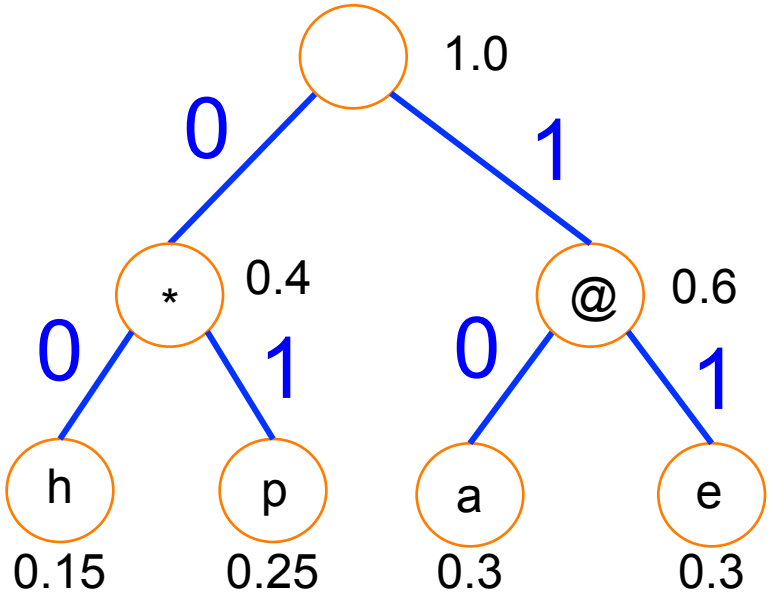
HUFFMAN: PAIR EXERCISE

letter	a	e	h	p
frequency	0.3	0.3	0.15	0.25
code	**10**	**11**	**00**	**01**

Handout 19 Solution



Mystery word: 00111001
= heap (!)



MAR 31 OUTLINE

- Recap tree algorithms and operations
- Huffman encoding example
- **Begin: priority queues and heaps**
- Array-based implementation of a heap

PRIORITY QUEUE

A queue that maintains the order of the elements according to some priority

- generally not FIFO
- some other order (although insertion time *could* be one criteria)

Removal order, not general order

- object with minkey/maxkey in front
- the rest **may or may not be sorted** (implementation dependent)

INTERFACE (NOT THE ONLY ONE)

```
public interface PriorityQueue<E extends
Comparable<E>> {

        void insert(E element);

        E min() or max();

        E removeMin() or removeMax();

        int size();

        boolean isEmpty();

}
```

EXAMPLE – MIN PRIORITY QUEUE

Say the number (i.e. 5) is the polling percentage, and the letter is the candidate's initial. We are prioritizing by min polling number.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }

Example 9.1

EXAMPLE – MIN PRIORITY QUEUE

Say the number (i.e. 5) is the polling percentage, and the letter is the candidate's initial. We are prioritizing by min polling number.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }

Example 9.1

EXAMPLE – MIN PRIORITY QUEUE

Say the number (i.e. 5) is the polling percentage, and the letter is the candidate's initial. We are prioritizing by min polling number.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }

Example 9.1

EXAMPLE – MIN PRIORITY QUEUE

Say the number (i.e. 5) is the polling percentage, and the letter is the candidate's initial. We are prioritizing by min polling number.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }

Example 9.1

EXAMPLE – MIN PRIORITY QUEUE

Say the number (i.e. 5) is the polling percentage, and the letter is the candidate's initial. We are prioritizing by min polling number.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }

Example 9.1

EXAMPLE – MIN PRIORITY QUEUE

Say the number (i.e. 5) is the polling percentage, and the letter is the candidate's initial. We are prioritizing by min polling number.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }

Example 9.1

EXAMPLE – MIN PRIORITY QUEUE

Say the number (i.e. 5) is the polling percentage, and the letter is the candidate's initial. We are prioritizing by min polling number.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }

Example 9.1

EXAMPLE – MIN PRIORITY QUEUE

Say the number (i.e. 5) is the polling percentage, and the letter is the candidate's initial. We are prioritizing by min polling number.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }

Example 9.1

EXAMPLE – MIN PRIORITY QUEUE

Say the number (i.e. 5) is the polling percentage, and the letter is the candidate's initial. We are prioritizing by min polling number.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }

Example 9.1

EXAMPLE – MIN PRIORITY QUEUE

Say the number (i.e. 5) is the polling percentage, and the letter is the candidate's initial. We are prioritizing by min polling number.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }

Example 9.1

EXAMPLE – MIN PRIORITY QUEUE

Say the number (i.e. 5) is the polling percentage, and the letter is the candidate's initial. We are prioritizing by min polling number.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

Example 9.1

SORTED VS. UNSORTED LIST

- **For unsorted:**
  - Insertion is fast, just add to end
  - But finding the min or max is slow

Method	Unsorted list	Sorted list
size		
insert		
min/max		
removeMin/Max		

SORTED VS. UNSORTED LIST

- **For unsorted:**
  - Insertion is fast, just add to end
  - But finding the min or max is slow

Method	Unsorted list	Sorted list
size	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min/max	$O(n)$	$O(1)$
removeMin/Max	$O(n)$	$O(1)$

ENTER: THE HUMBLE HEAP

Both sorted and unsorted lists have disadvantages

We do not want O(n) to insert or O(n) to remove

Need a semi-sorted data structure!

ENTER: THE HUMBLE HEAP

Both sorted and unsorted lists have disadvantages

We do not want O(n) to insert or O(n) to remove

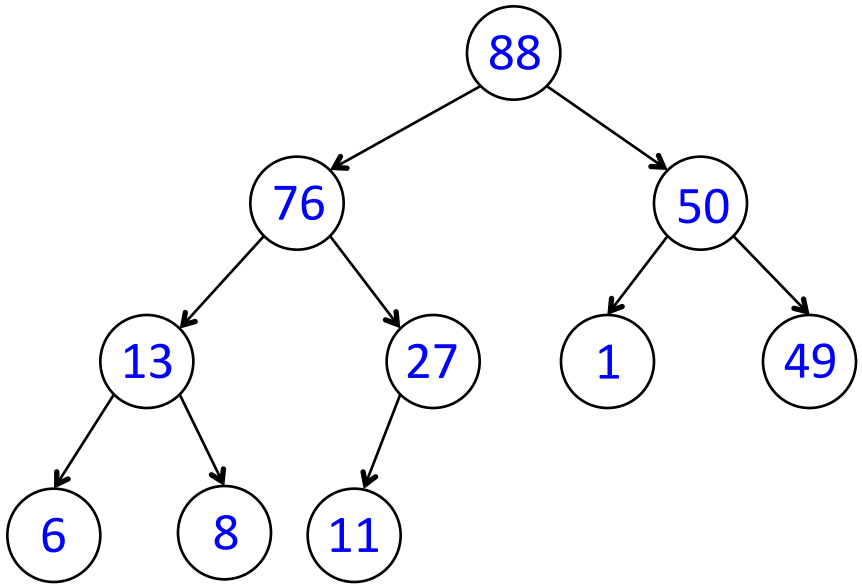
Need a semi-sorted data structure!

Heap: complete binary tree (every level filled except maybe the last, which is filled from the left)

Max heap: parent >= both children

Min heap: parent <= both children

Every subtree is also a heap



MAX HEAP: INSERT

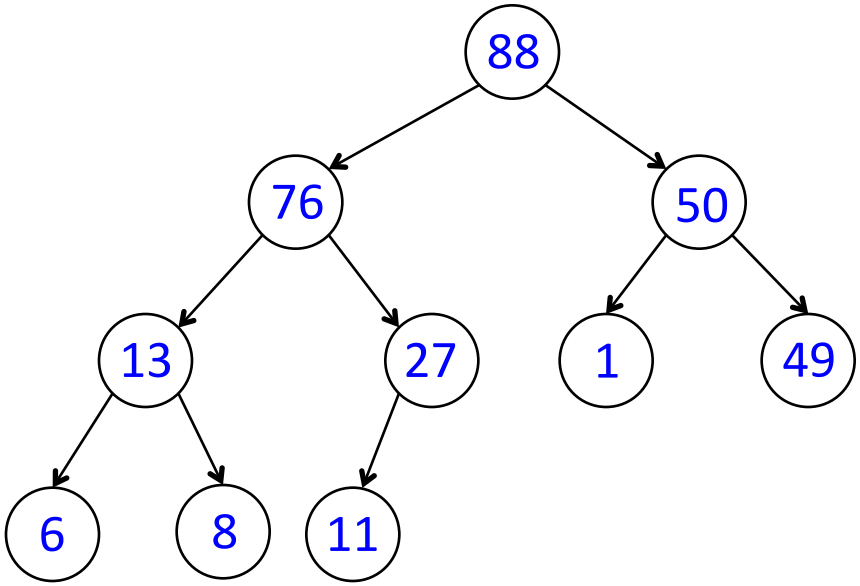
insert(x):

place x in first open spot on lowest level (or make a new level)

"bubble up" x until heap condition satisfied, i.e.:

while child > parent:

swap parent and child (Lab 6: write a swap helper method)



Runtime?

MAX HEAP: INSERT

insert(x):

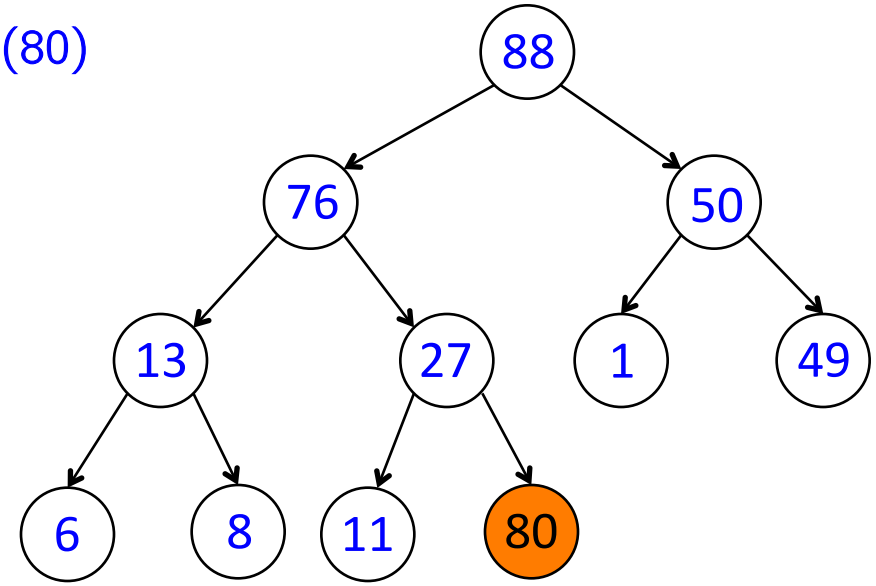
place x in first open spot on lowest level (or make a new level)

"bubble up" x until heap condition satisfied, i.e.:

while child > parent:

swap parent and child (Lab 6: write a swap helper method)

Example: insert(80)



Runtime?

MAX HEAP: INSERT

insert(x):

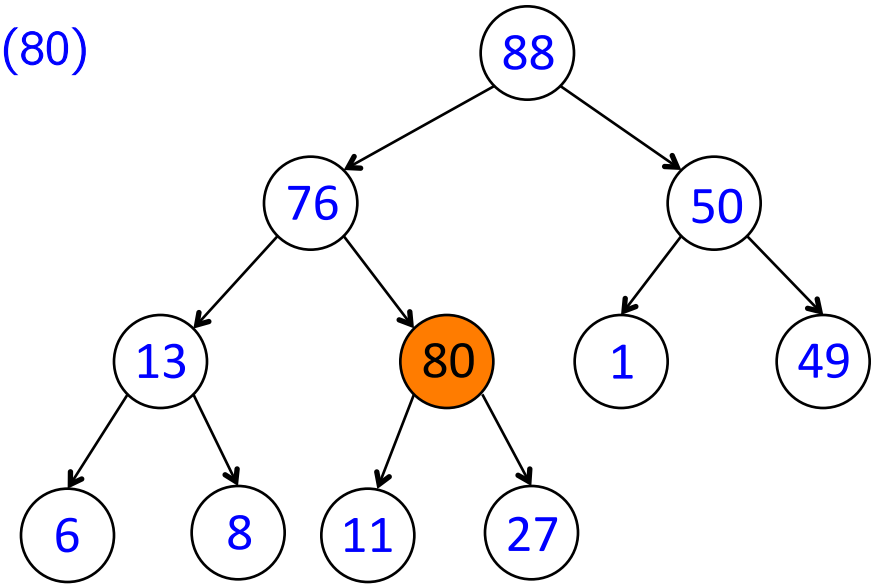
place x in first open spot on lowest level (or make a new level)

"bubble up" x until heap condition satisfied, i.e.:

while child > parent:

swap parent and child (Lab 6: write a swap helper method)

Example: insert(80)



Runtime?

MAX HEAP: INSERT

insert(x):

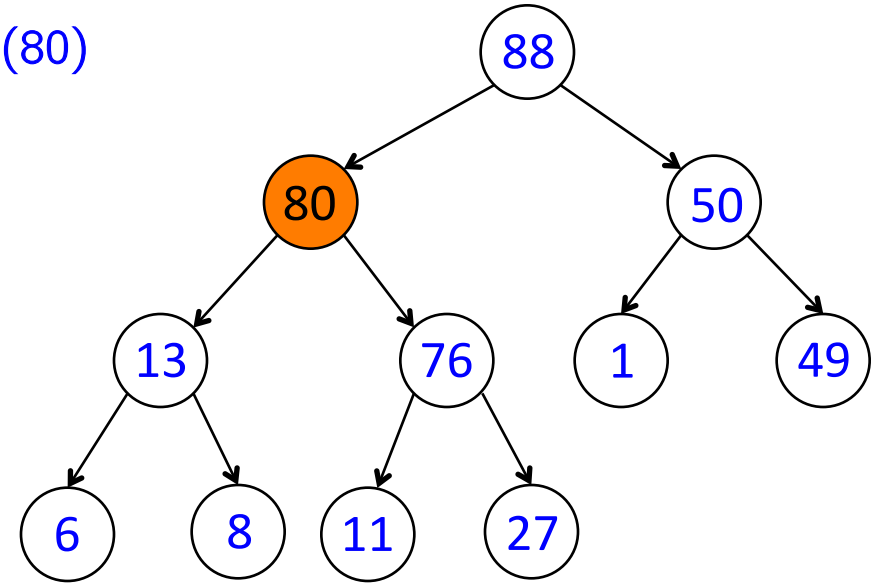
place x in first open spot on lowest level (or make a new level)

"bubble up" x until heap condition satisfied, i.e.:

while child > parent:

swap parent and child (Lab 6: write a swap helper method)

Example: insert(80)



Runtime?

MAX HEAP: INSERT

insert(x):

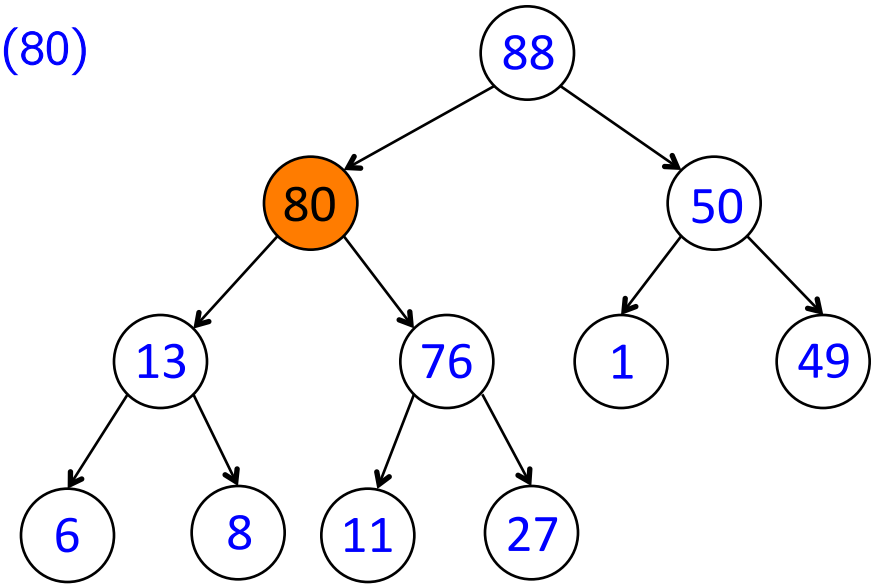
place x in first open spot on lowest level (or make a new level)

"bubble up" x until heap condition satisfied, i.e.:

while child > parent:

swap parent and child (Lab 6: write a swap helper method)

Example: insert(80)



Runtime: O(log(n)) !

MAX HEAP: REMOVE

removeMax():

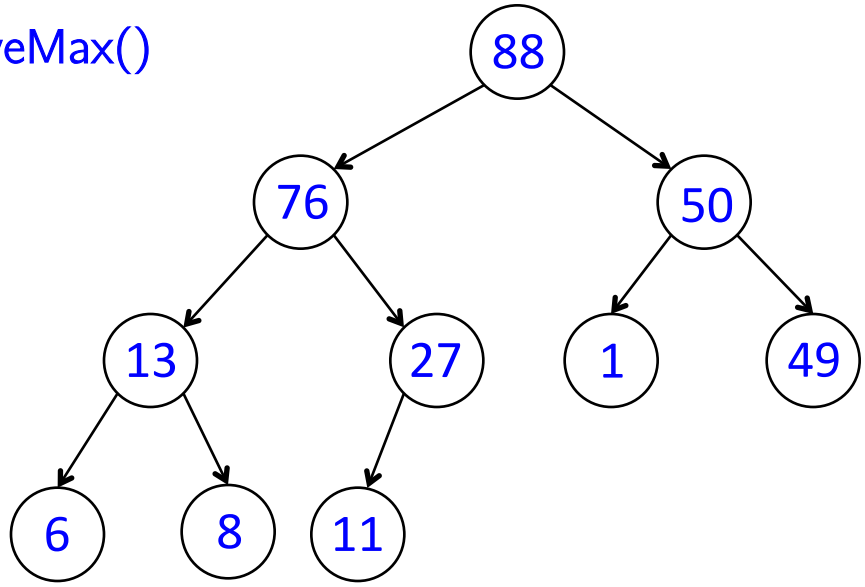
move last element to root

"bubble down" until heap condition satisfied, i.e.:

while parent < either child:

swap parent with largest child

Example: removeMax()



Runtime?

MAX HEAP: REMOVE

removeMax():

move last element to root

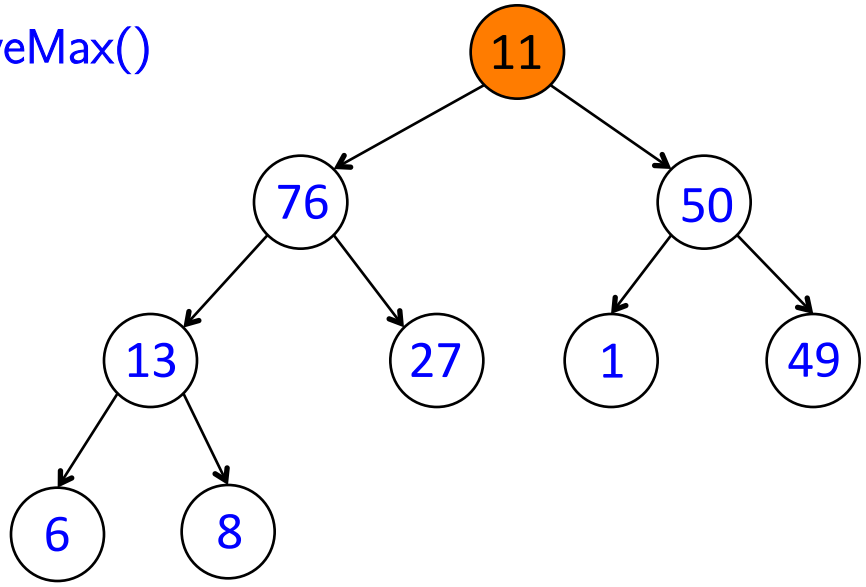
"bubble down" until heap condition satisfied, i.e.:

while parent < either child:

swap parent with largest child

Example: removeMax()

Return: 88



Runtime?

MAX HEAP: REMOVE

removeMax():

move last element to root

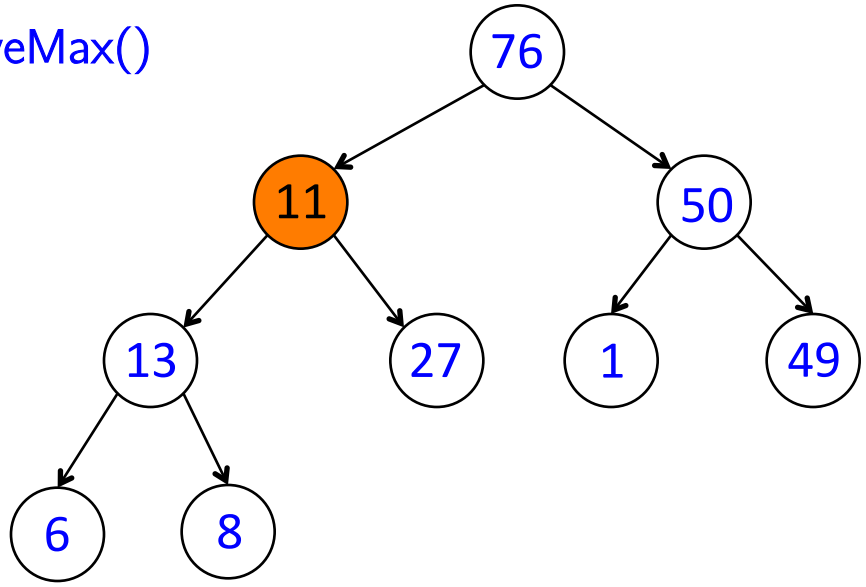
"bubble down" until heap condition satisfied, i.e.:

while parent < either child:

swap parent with largest child

Example: removeMax()

Return: 88



Runtime?

MAX HEAP: REMOVE

removeMax():

move last element to root

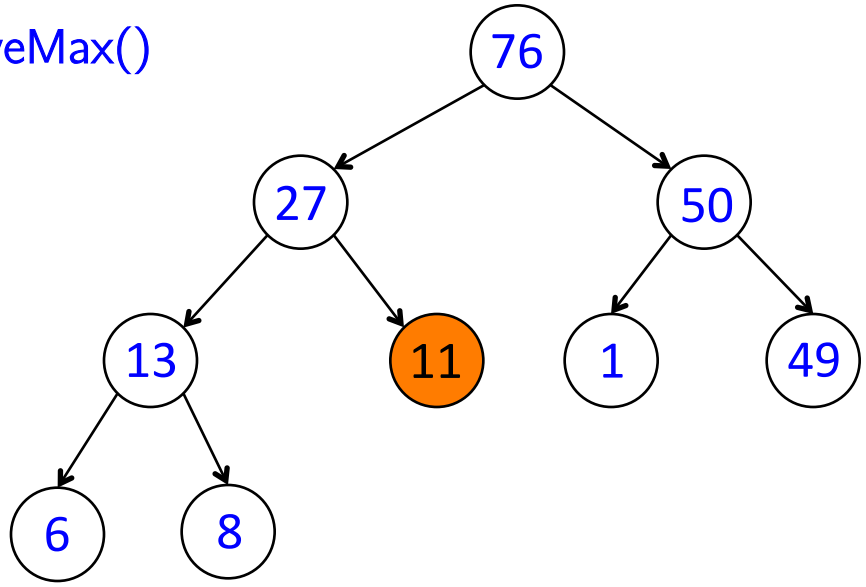
"bubble down" until heap condition satisfied, i.e.:

while parent < either child:

swap parent with largest child

Example: removeMax()

Return: 88



Runtime?

MAX HEAP: REMOVE

removeMax():

move last element to root

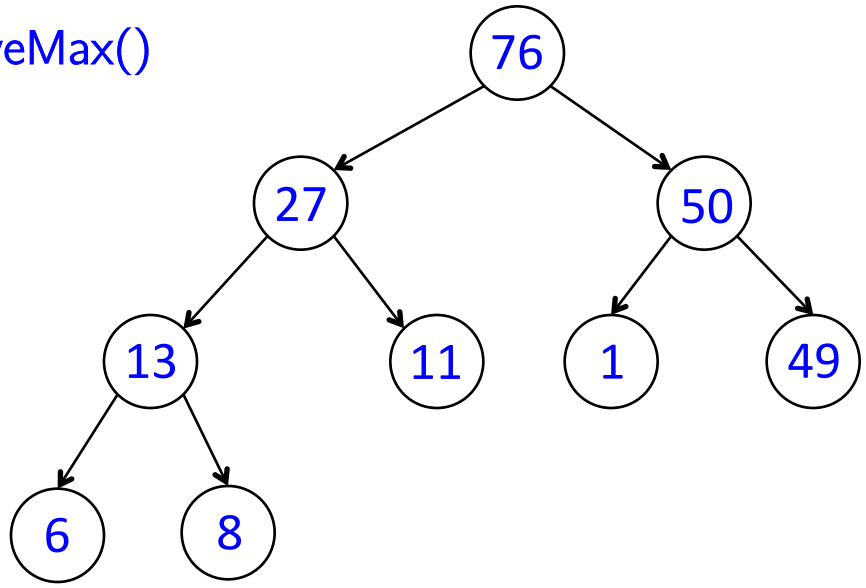
"bubble down" until heap condition satisfied, i.e.:

while parent < either child:

swap parent with largest child

Example: removeMax()

Return: 88



Runtime: O(log(n)) !

MAR 31 OUTLINE

- Recap tree algorithms and operations
- Huffman encoding example
- Begin: priority queues and heaps
- **Array-based implementation of a heap**

IMPLEMENTATION USING AN ARRAY

Order in array: breadth-first!