

CS 106

INTRODUCTION TO

DATA STRUCTURES

SPRING 2020

PROF. SARA MATHIESON

HAVERFORD COLLEGE

ADMIN

- **Lab 5** due Sunday
- **Lab tomorrow** same as last week (I will start at 9am)
 - Sign-in sheet + zoom to join the queue
- **Lab 6** posted TODAY
- Email me (and cc partner(s)) if you want to work together in **breakout rooms** (or prefer individual)

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)

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

Thursday 9-11pm (Will)

~~**Friday 8-10pm (Gareth)**~~

Saturday 4-6pm (Will)

Saturday 8-10pm (Gareth)

} *Today/Tomorrow*

LAB 5 MULTIPLE FILES

Edit: please add poll_data if you can!

```
poll_data/dempres_20190103_1.csv poll_data/dempres_20190202_1.csv
poll_data/dempres_20190302_1.csv
```

Edit: output from the above example

```
Tree:
Pre:   Bernard Sanders:21.1 Joseph R. Biden Jr.:37.0 Beto O'Rourke:5.0 Joseph
Kennedy III:9.0 Kamala D. Harris:9.0 Hillary Rodham Clinton:3.0 Cory A. Booker:5.9
Michael Bloomberg:1.9 Sherrod Brown:0.9 Steve Bullock:0.0 Julián Castro:0.2 Pete
Buttigieg:0.4 Kirsten E. Gillibrand:3.3 Andrew Cuomo:0.0 John K. Delaney:0.0 Eric
Garcetti:0.0 Tulsi Gabbard:1.5 John Hickenlooper:1.0 Jay Robert Inslee:0.0 Eric H.
Holder:0.0 John Kerry:1.0 Amy Klobuchar:0.9 Terry R. McAuliffe:0.0 Gavin Newsom:0.0
Richard Neece Ojeda:1.0 Elizabeth Warren:5.2 Tom Steyer:1.0 Howard Schultz:0.0 Eric
Swalwell:0.0
In:    Joseph R. Biden Jr.:37.0 Michael Bloomberg:1.9 Cory A. Booker:5.9 Sherrod
Brown:0.9 Steve Bullock:0.0 Pete Buttigieg:0.4 Julián Castro:0.2 Hillary Rodham
Clinton:3.0 Andrew Cuomo:0.0 John K. Delaney:0.0 Tulsi Gabbard:1.5 Eric
Garcetti:0.0 Kirsten E. Gillibrand:3.3 Kamala D. Harris:9.0 John Hickenlooper:1.0
Eric H. Holder:0.0 Jay Robert Inslee:0.0 Joseph Kennedy III:9.0 John Kerry:1.0 Amy
Klobuchar:0.9 Terry R. McAuliffe:0.0 Gavin Newsom:0.0 Beto O'Rourke:5.0 Richard
Neece Ojeda:1.0 Bernard Sanders:21.1 Howard Schultz:0.0 Tom Steyer:1.0 Eric
Swalwell:0.0 Elizabeth Warren:5.2
Post:  Michael Bloomberg:1.9 Pete Buttigieg:0.4 Julián Castro:0.2 Steve
Bullock:0.0 Sherrod Brown:0.9 Cory A. Booker:5.9 Tulsi Gabbard:1.5 Eric
Garcetti:0.0 John K. Delaney:0.0 Andrew Cuomo:0.0 Kirsten E. Gillibrand:3.3 Hillary
Rodham Clinton:3.0 Eric H. Holder:0.0 Jay Robert Inslee:0.0 John Hickenlooper:1.0
Kamala D. Harris:9.0 Gavin Newsom:0.0 Terry R. McAuliffe:0.0 Amy Klobuchar:0.9 John
Kerry:1.0 Joseph Kennedy III:9.0 Richard Neece Ojeda:1.0 Beto O'Rourke:5.0 Joseph
R. Biden Jr.:37.0 Howard Schultz:0.0 Eric Swalwell:0.0 Tom Steyer:1.0 Elizabeth
Warren:5.2 Bernard Sanders:21.1
```

LAB 5 NOTES

Try **NOT** to use helper methods in a “static” way (like below)

```
method inOrder():
```

```
    BinaryTree myTree = new LinkedBinaryTree(root)
    inOrderHelper(myTree)
```

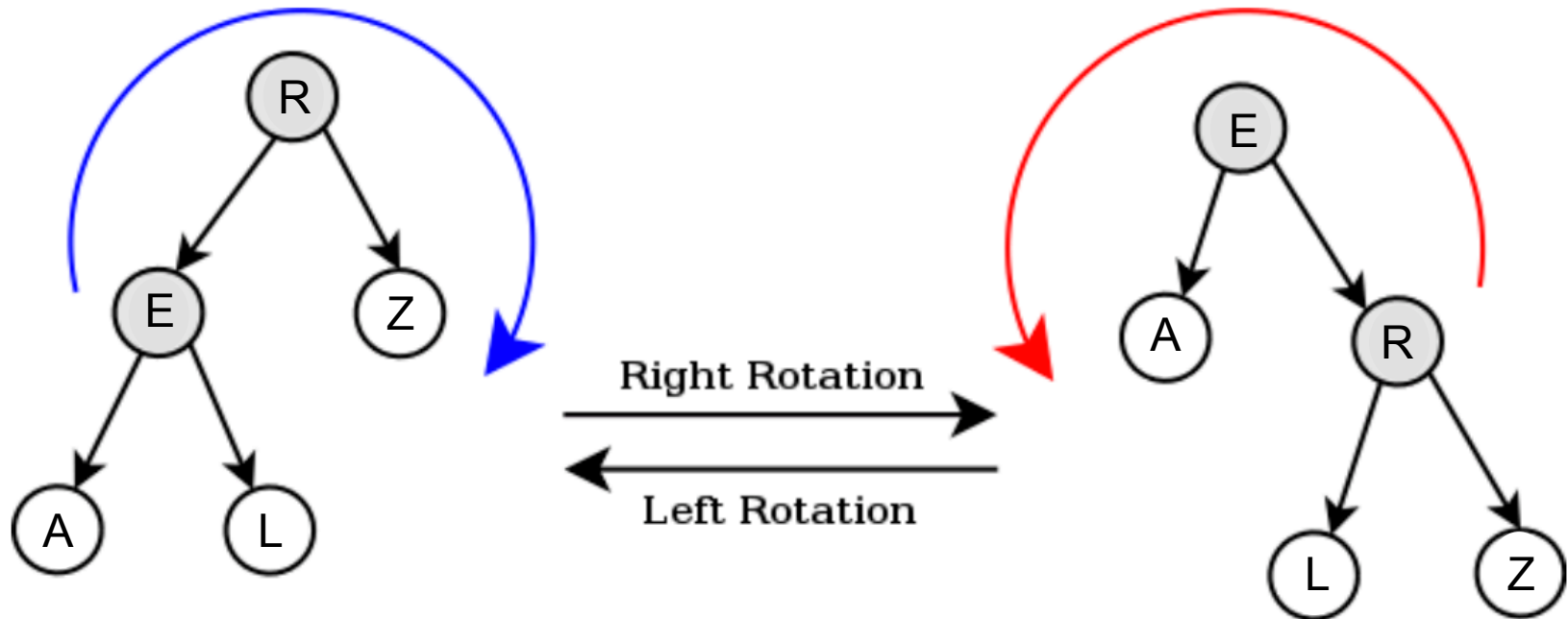
```
method inOrderHelper(BinaryTree tree):
```

```
    ...
```

REBALANCING TREES

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

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



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

Edit: this example is fixed now!

APR 2 OUTLINE

- **Recap priority queues and heaps**
- **Array-based implementation of a heap**
- **Heap sort**

APR 2 OUTLINE

- **Recap priority queues and heaps**
- Array-based implementation of a heap
- Heap sort

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)

HEAP DATA STRUCTURE

Sorted list: $O(n)$ to insert (enqueue)

Unsorted list: $O(n)$ to remove (dequeue)

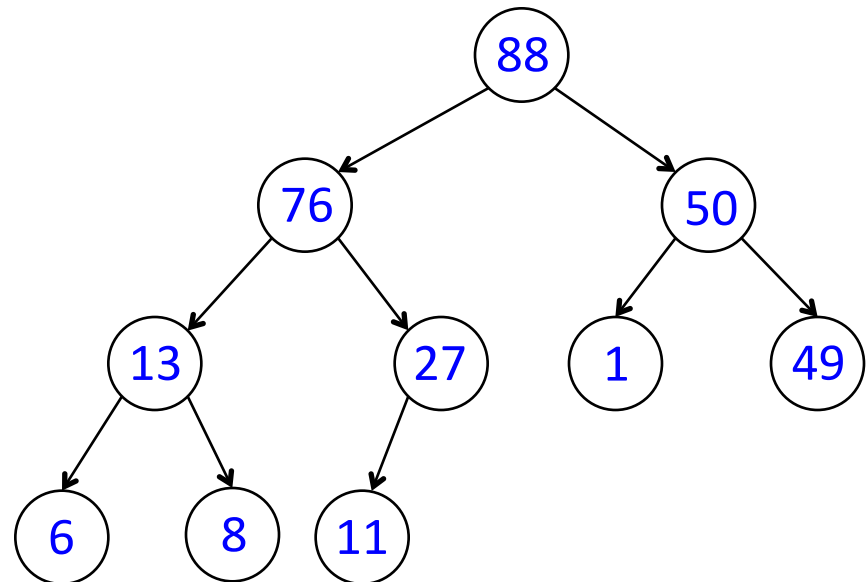
Need a semi-sorted data structure!

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

Max heap: parent \geq both children

Min heap: parent \leq both children

Every subtree is also a heap



MAX HEAP: INSERT

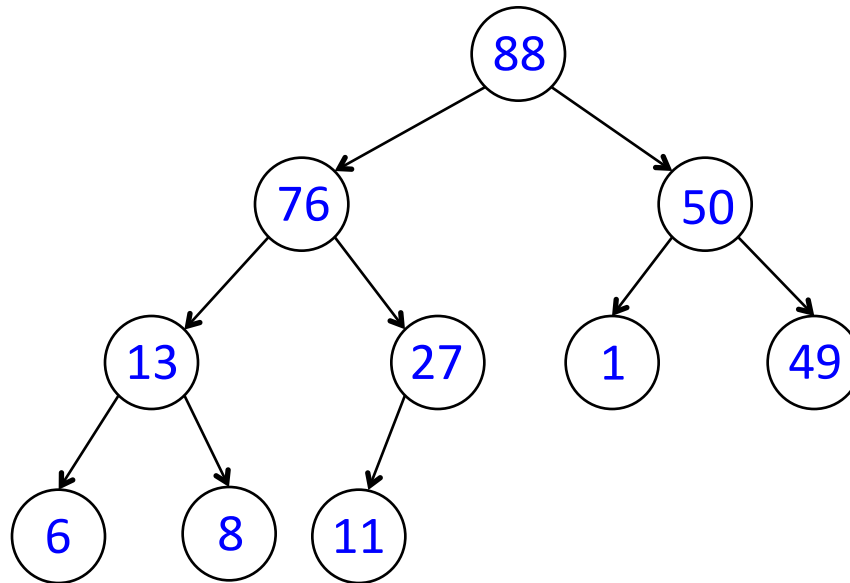
`insert(x)`:

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

“bubble up” `x` until heap condition satisfied, i.e.:

while $\text{child} > \text{parent}$:

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



Runtime?

MAX HEAP: INSERT

insert(x):

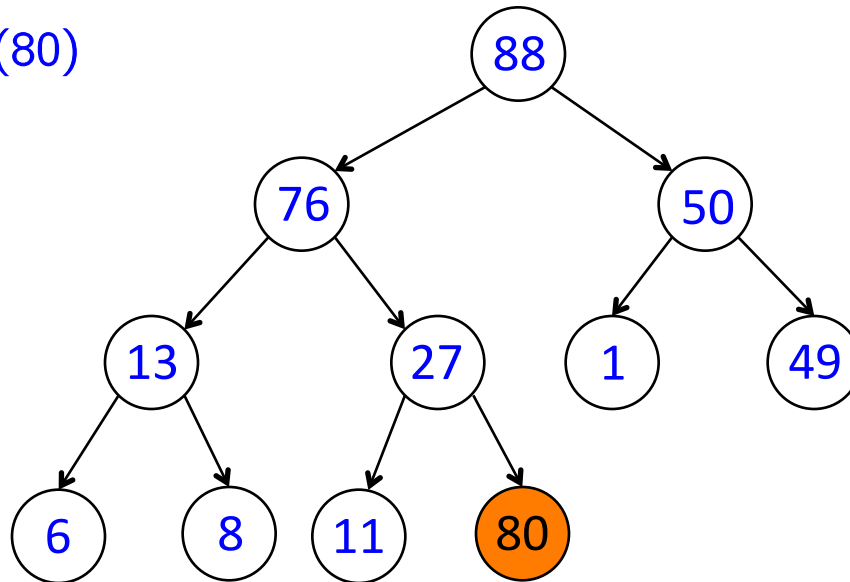
place **x** in first open spot on lowest level (or make a new level)

“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):

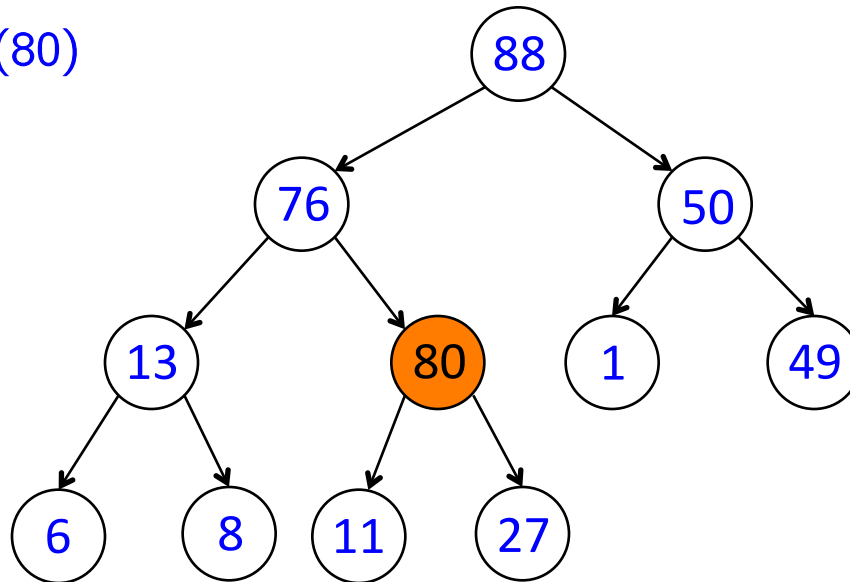
place **x** in first open spot on lowest level (or make a new level)

“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):

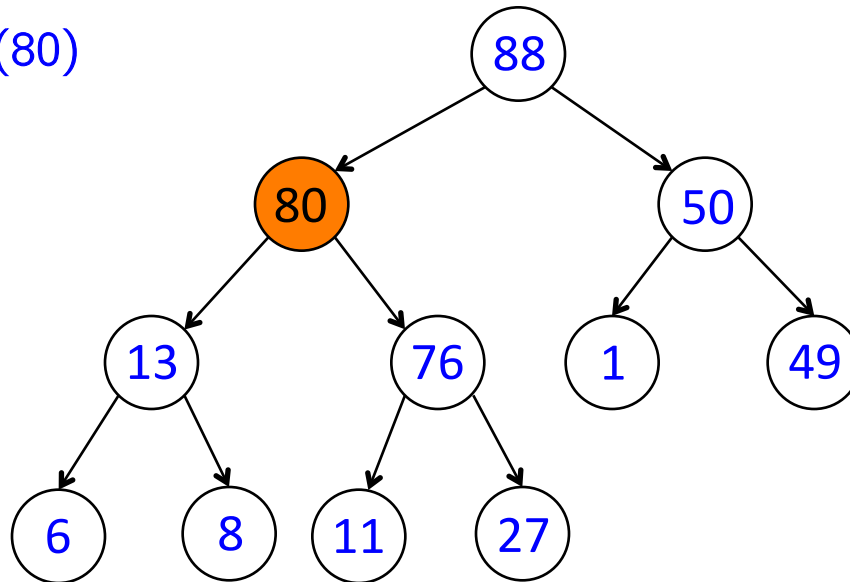
place **x** in first open spot on lowest level (or make a new level)

“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):

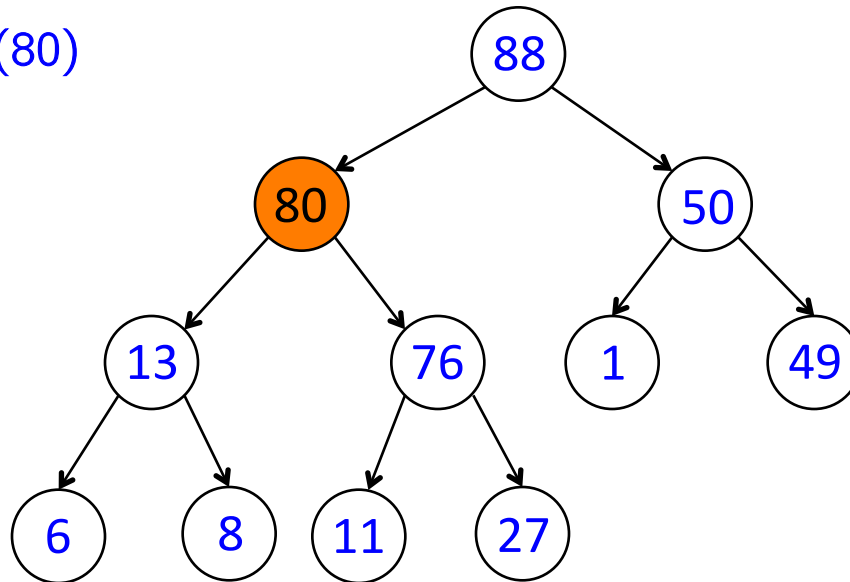
place **x** in first open spot on lowest level (or make a new level)

“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()`:

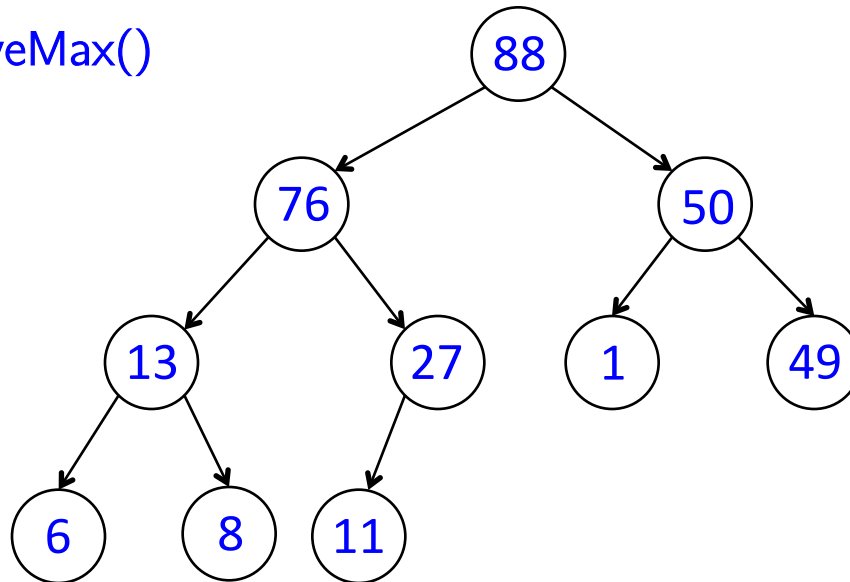
move last element to root

“bubble down” until heap condition satisfied, i.e.:

while parent < either child:

swap parent with largest child

Example: `removeMax()`



Runtime?

MAX HEAP: REMOVE

MAX HEAP: REMOVE

`removeMax()`:

move last element to root

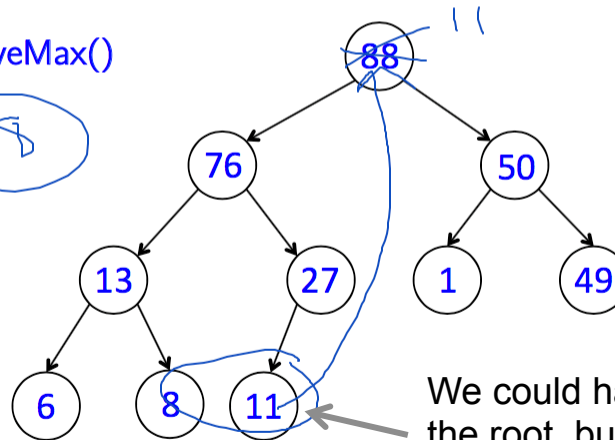
“bubble down” until heap condition satisfied, i.e.:

while parent < either child:

swap parent with largest child

Example: `removeMax()`

Save
to return later



We could have moved any leaf to the root, but we remove the “last” one to keep the tree balanced

Runtime?

MAX HEAP: REMOVE

`removeMax()`:

move last element to root

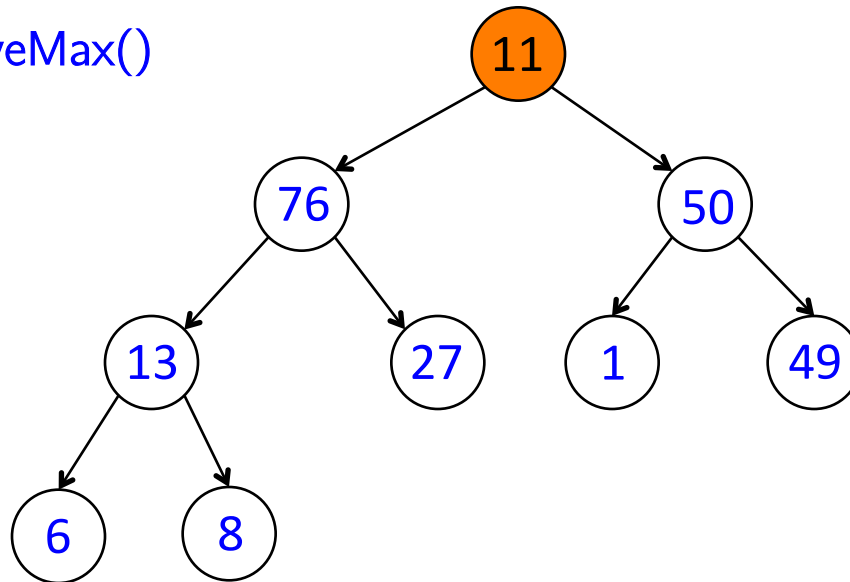
“bubble down” until heap condition satisfied, i.e.:

while parent < either child:

swap parent with largest child

Example: `removeMax()`

Return: 88



Runtime?

MAX HEAP: REMOVE

`removeMax()`:

move last element to root

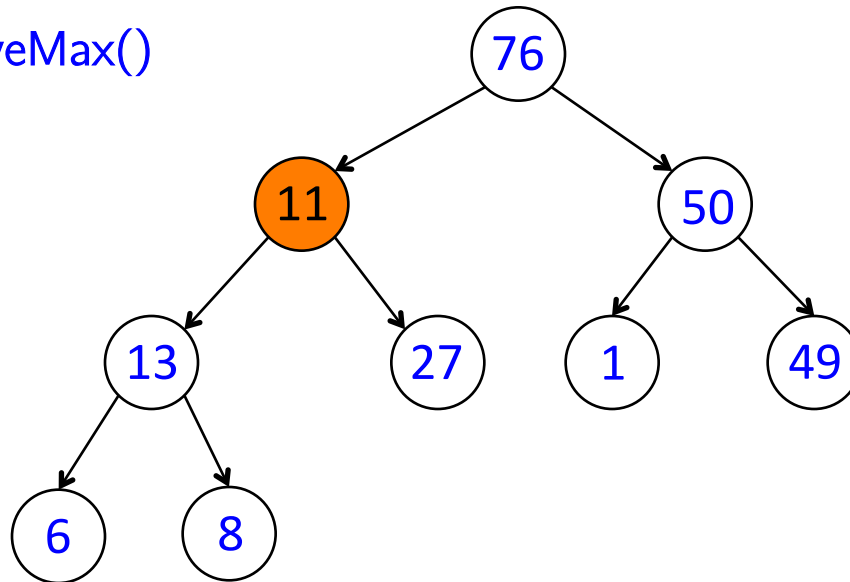
“bubble down” until heap condition satisfied, i.e.:

while parent < either child:

swap parent with largest child

Example: `removeMax()`

Return: 88



Runtime?

MAX HEAP: REMOVE

`removeMax()`:

move last element to root

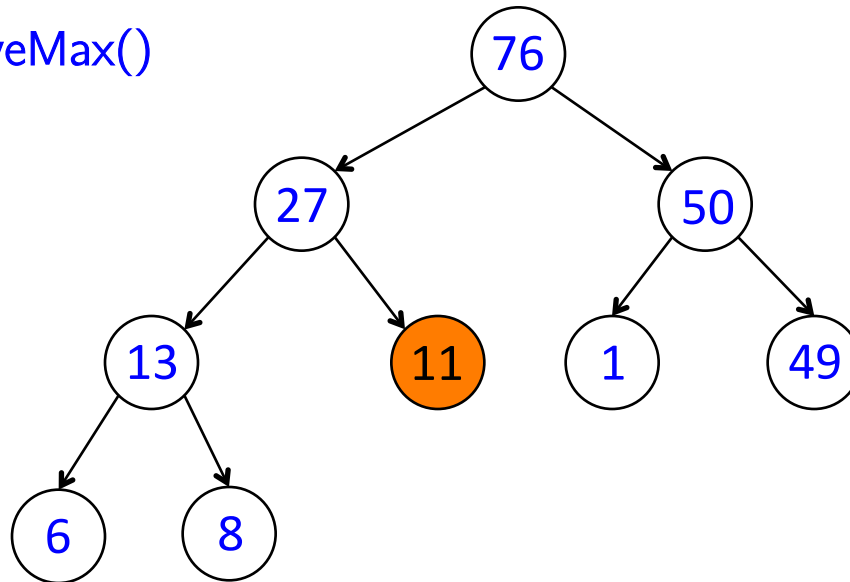
“bubble down” until heap condition satisfied, i.e.:

while parent < either child:

swap parent with largest child

Example: `removeMax()`

Return: 88



Runtime?

MAX HEAP: REMOVE

`removeMax()`:

move last element to root

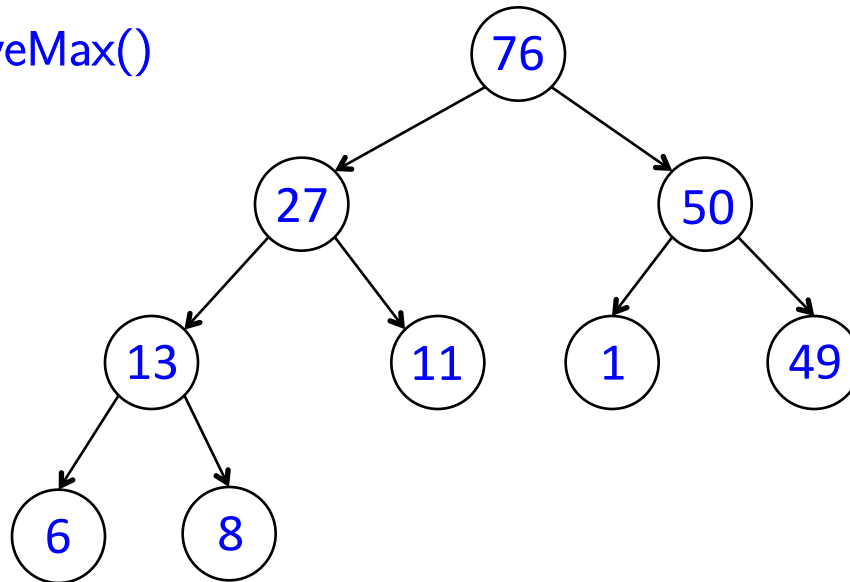
“bubble down” until heap condition satisfied, i.e.:

while parent < either child:

swap parent with largest child

Example: `removeMax()`

Return: 88



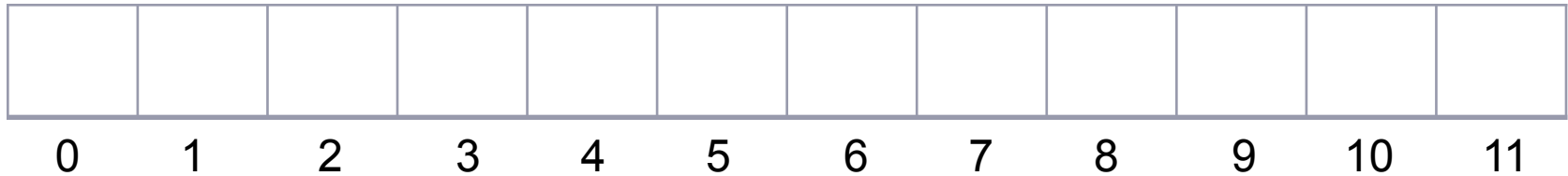
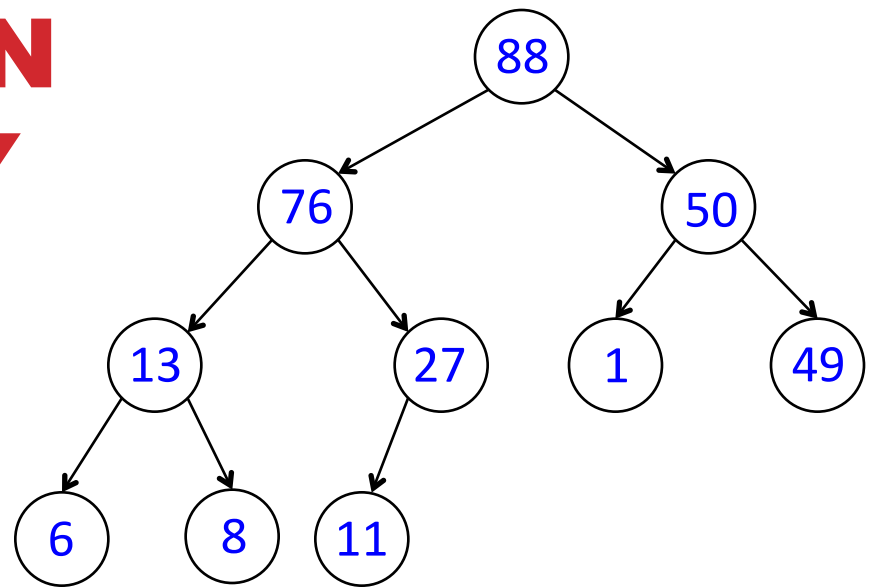
Runtime: $O(\log(n))$!

APR 2 OUTLINE

- Recap priority queues and heaps
- **Array-based implementation of a heap**
- Heap sort

IMPLEMENTATION USING AN ARRAY

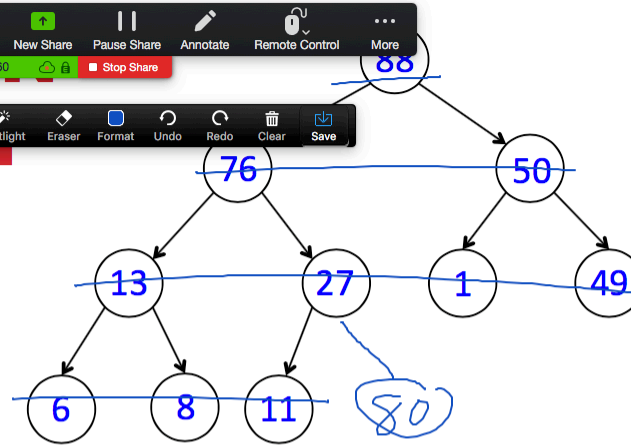
Order in array: breadth-first!



<i>i</i>	parent(<i>i</i>)	left(<i>i</i>)	right(<i>i</i>)
0			
1			
2			
3			
4			

IMPLEMENTATION USING AN ARRAY

Order in array: breadth-first!



88	76	50	13	27	1	49	6	8	11	80		
0	1	2	3	4	5	6	7	8	9	10	11	

row 0: 0, 1, 2
row 1: 3, 4, 5, 6, 7, 8, 9, 10
row 2: 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000

i	parent(i)	left(i)	right(i)
0	none	1	2
1	0	3	4
2	0	5	6
3	1	7	8
4	1	9	10

PARENT/CHILD RELATIONSHIPS (PAIR EXERCISE)

PARENT/CHILD RELATIONSHIPS (PAIR EXERCISE)

parent(i)

$$= \left\lfloor \frac{i-1}{2} \right\rfloor$$

$$\left\lfloor \frac{3-1}{2} \right\rfloor = 1$$

$$\left\lfloor \frac{4-1}{2} \right\rfloor = 1$$

left(i)

$$= 2i + 1$$

right(i)

$$= 2i + 2$$

$$i = 2^0 + 2^1 + 2^2 + \dots + 2^k + x$$

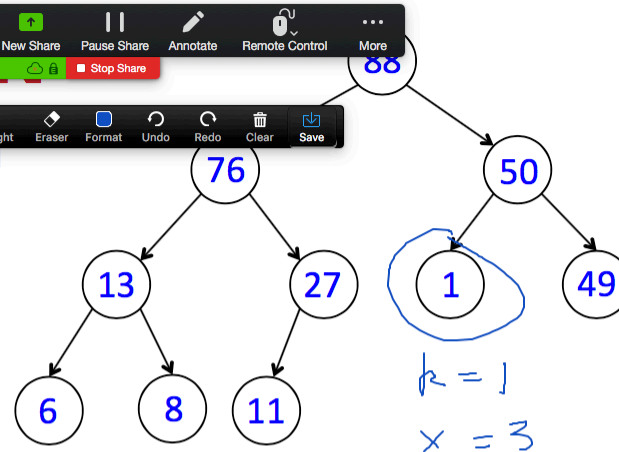
$$ch \rightarrow d(i) = \dots + 2^{k+1} + 2x$$

HINTS FOR PROVING FORMULAS

IMPLEMENTATION USING AN ARRAY

Order in array: breadth-first!

$$5 = 2^0 \times 2^1 + 3 - 1$$



					1						
0	1	2	3	4	5	6	7	8	9	10	11

i	$\text{parent}(i)$	$\text{left}(i)$	$\text{right}(i)$
0			
1			
2			
3			
4			

INSERT EXAMPLE

INSERT EXAMPLE

80

76 80

27

27

88	76	50	13	27	1	49	6	8	11	80	
0	1	2	3	4	5	6	7	8	9	10	11

bubble up

① $\text{parent}(10) = \left\lfloor \frac{10-1}{2} \right\rfloor = 4$

$80 > 27 ? \checkmark$

$\text{swap}(4, 10)$

② $\text{parent}(4) = \left\lfloor \frac{4-1}{2} \right\rfloor = 1$

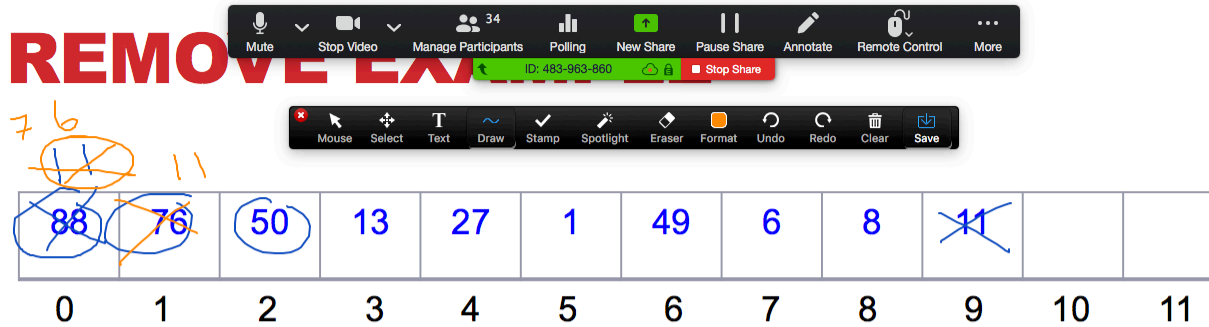
$80 > 76 ? \checkmark$

$\text{swap}(1, 4)$

③ $\text{parent}(1) = \left\lfloor \frac{1-1}{2} \right\rfloor = 0$

$80 > 88 ? \underline{\text{no}}$

REMOVE EXAMPLE



save 88

pos = 0

$$\textcircled{1} \quad \text{left}(0) = 2 \cdot 0 + 1 = 1$$
$$\text{right}(0) = 2 \cdot 0 + 2 = 2$$

swap(0, 1)

② pos = 1

11 > 76 ? no

11 > 50 ? no

76 > 50 ✓

APR 2 OUTLINE

- Recap priority queues and heaps
- Array-based implementation of a heap
- **Heap sort**

IN-PLACE SORTING

In-place sorting algorithm: we do not create a new data structure, we instead sort the elements within their existing data structure

- **Cons:** destroys the original order, which may have been important
- **Pros:** very efficient in terms of space

IN-PLACE SORTING

In-place sorting algorithm: we do not create a new data structure, we instead sort the elements within their existing data structure

- **Cons:** destroys the original order, which may have been important
- **Pros:** very efficient in terms of space

Out-of-place sorting algorithm: returns a new data structure with the original data sorted

- **Cons:** space inefficient
- **Pros:** preserves original order

IN-PLACE SORTING

In-place sorting algorithm: we do not create a new data structure, we instead sort the elements within their existing data structure

- **Cons:** destroys the original order, which may have been important
- **Pros:** very efficient in terms of space

Out-of-place sorting algorithm: returns a new data structure with the original data sorted

- **Cons:** space inefficient
- **Pros:** preserves original order

Heap Sort can be implemented either way, but we will cover the in-place version now

HEAP SORT (IN PLACE WITH ARRAY)

Phase I: unsorted array \rightarrow heap

for $i = 0, 1, \dots, n-1$:

 bubble up element at index i until $\text{arr}[0\dots i]$ form a heap

HEAP SORT (IN PLACE WITH ARRAY)

Phase I: unsorted array \rightarrow heap

for $i = 0, 1, \dots, n-1$:

 bubble up element at index i until $\text{arr}[0\dots i]$ form a heap

Phase II: heap \rightarrow sorted array

for $i = n-1, n-2, \dots, 0$:

$\text{swap}(0, i)$ // 0 is the root index

 bubble down so $\text{arr}[0\dots i]$ are still a heap

HEAP SORT RUNTIME?

PAIR EXERCISE

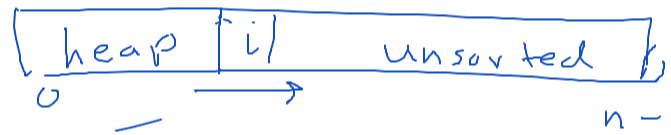
HEAP SORT (IN PLACE WITH ARRAY)

Phase I: unsorted array \rightarrow heap

for $i = 0, 1, \dots, n-1$:

bubble up element at index i until $\text{arr}[0\dots i]$ form a heap

$$O(n \log(n))$$

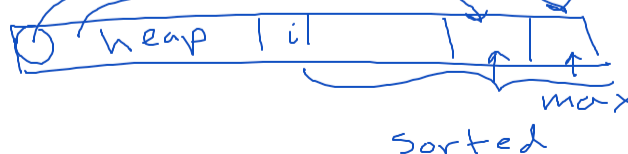


Phase II: heap \rightarrow sorted array

for $i = n-1, n-2, \dots, 0$:

swap($0, i$) // 0 is the root index

bubble down so $\text{arr}[0\dots i]$ are still a heap



runtime?

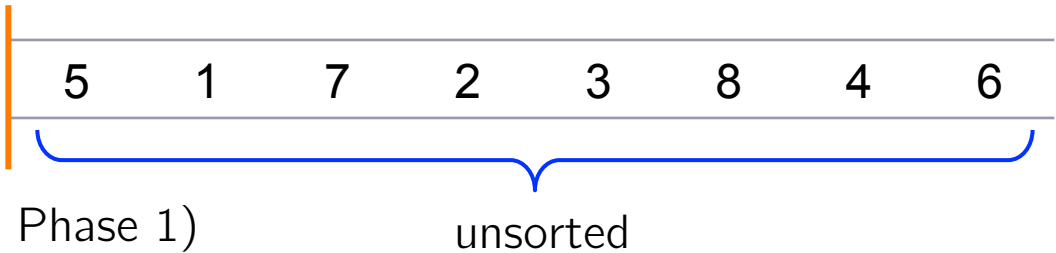
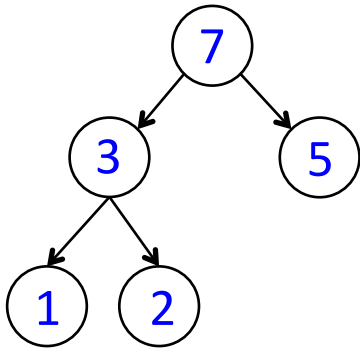
low \rightarrow high

HEAP SORT EXAMPLE: PHASE I

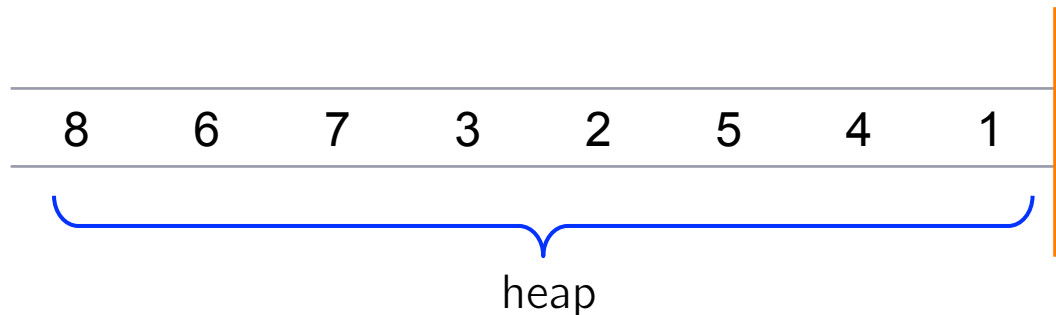
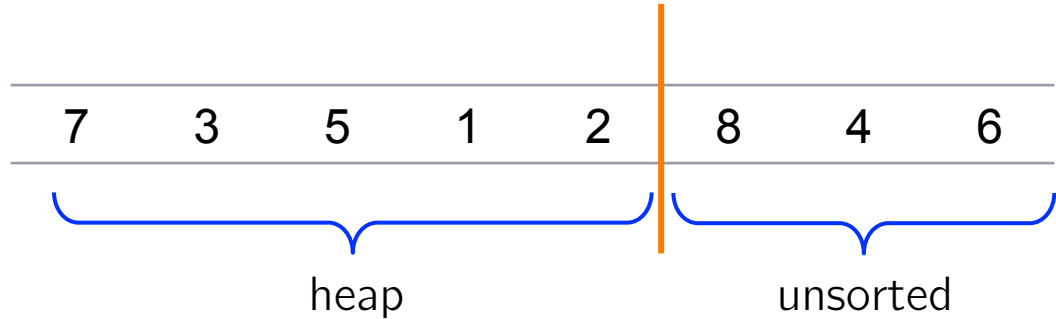
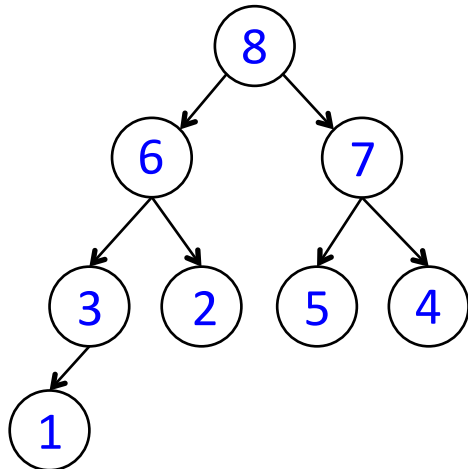
Phase I: unsorted array \rightarrow heap

(Below are two different stages in Phase 1)

After processing $i=4$:



After processing $i=n-1$ (end of Phase I):



HEAP SORT EXAMPLE: PHASE II

Phase II: heap \rightarrow sorted array

5	1	7	2	3	8	4	6
---	---	---	---	---	---	---	---

Next time!