Heaps and Priority Queues

Computer Science E-22
Harvard Extension School

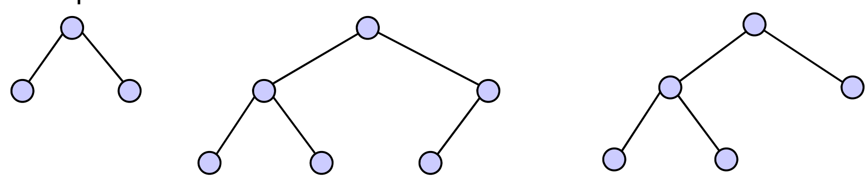
David G. Sullivan, Ph.D.

Priority Queue

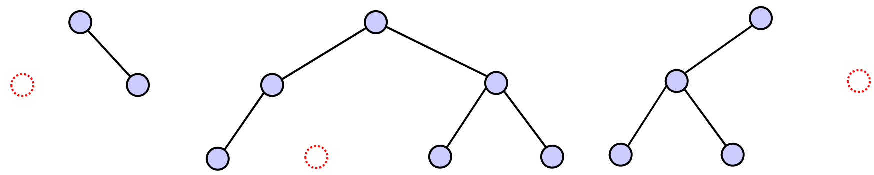
- A *priority queue* is a collection in which each item in the collection has an associated number known as a *priority*.
  - ("Henry Leitner", 10), ("Drew Faust", 15), ("Dave Sullivan", 5)
  - use a higher priority for items that are "more important"
- Example: scheduling a shared resource like the CPU
  - give some processes/applications a higher priority, so that they will be scheduled first and/or more often
- Key operations:
  - *insert:* add an item to the priority queue, positioning it according to its priority
  - *remove:* remove the item with the highest priority
- How can we efficiently implement a priority queue?
  - use a type of binary tree known as a *heap*

Complete Binary Trees

- A binary tree of height *h* is *complete* if:
  - levels 0 through *h* – 1 are fully occupied
  - there are no “gaps” to the left of a node in level *h*
- Complete:

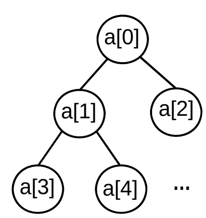


- Not complete (= missing node):

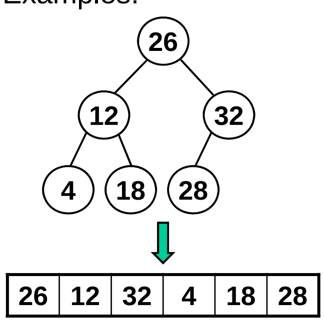


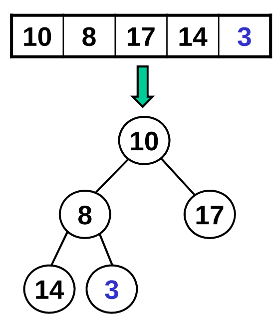
Representing a Complete Binary Tree

- A complete binary tree has a simple array representation.
- The nodes of the tree are stored in the array in the order in which they would be visited by a level-order traversal (i.e., top to bottom, left to right).



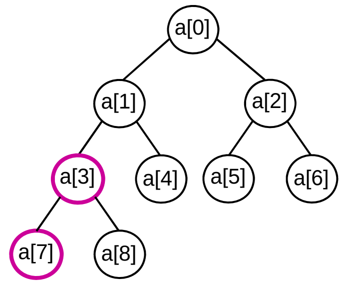
- Examples:





Navigating a Complete Binary Tree in Array Form

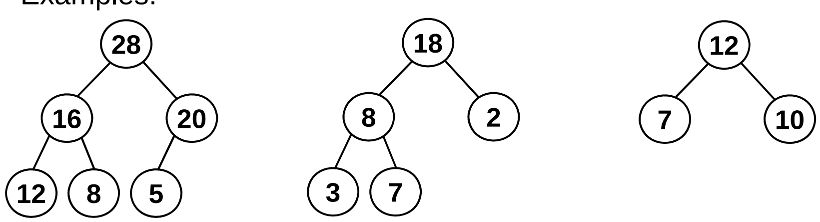
- The root node is in `a[0]`
- Given the node in `a[i]`:
  - its left child is in `a[2*i + 1]`
  - its right child is in `a[2*i + 2]`
  - its parent is in `a[(i – 1)/2]` (using integer division)



- Examples:
  - the left child of the node in `a[1]` is in `a[2*1 + 1]` = `a[3]`
  - the right child of the node in `a[3]` is in `a[2*3 + 2]` = `a[8]`
  - the parent of the node in `a[4]` is in `a[(4-1)/2]` = `a[1]`
  - the parent of the node in `a[7]` is in `a[(7-1)/2]` = `a[3]`

Heaps

- Heap: a complete binary tree in which each interior node is greater than or equal to its children
- Examples:



- The largest value is always at the root of the tree.
- The smallest value can be in *any* leaf node – there's no guarantee about which one it will be.
- Strictly speaking, the heaps that we will use are *max-at-top* heaps. You can also define a *min-at-top* heap, in which every interior node is less than or equal to its children.

How to Compare Objects

- We need to be able to compare items in the heap.
- If those items are objects, we can't just do something like this:

```
if (item1 < item2)
```

  Why not?

- Instead, we need to use a method to compare them.

An Interface for Objects That Can Be Compared

- The `Comparable` interface is a built-in generic Java interface:

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- It is used when defining a class of objects that can be ordered.
- Examples from the built-in Java classes:

```
public class String implements Comparable<String> {
    ...
    public int compareTo(String other) {
        ...
}
public class Integer implements Comparable<Integer> {
    ...
    public int compareTo(Integer other) {
        ...
}
```

An Interface for Objects That Can Be Compared (cont.)

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- `item1.compareTo(item2)` should return:
  - a negative integer if `item1` "comes before" `item2`
  - a positive integer if `item1` "comes after" `item2`
  - 0 if `item1` and `item2` are equivalent in the ordering
- These conventions make it easy to construct appropriate method calls:

numeric comparison	comparison using compareTo
`item1 < item2`	`item1.compareTo(item2) < 0`
`item1 > item2`	`item1.compareTo(item2) > 0`
`item1 == item2`	`item1.compareTo(item2) == 0`

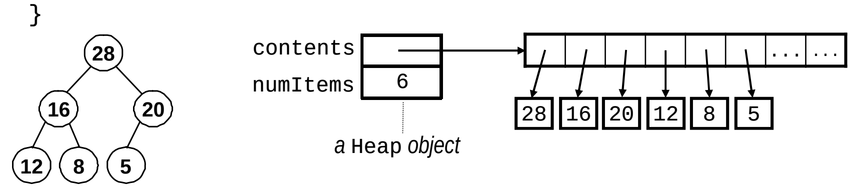
A Class for Items in a Priority Queue

```
public class PQItem implements Comparable<PQItem> {
    // group an arbitrary object with a priority
    private Object data;
    private int priority;
    ...

    public int compareTo(PQItem other) {
        // error-checking goes here…
        return (priority – other.priority);
    }
}
```

- Its `compareTo()` compares `PQItems` based on their priorities.
- `item1.compareTo(item2)` returns:
  - a negative integer if `item1` has a lower priority than `item2`
  - a positive integer if `item1` has a higher priority than `item2`
  - 0 if they have the same priority

Heap Implementation

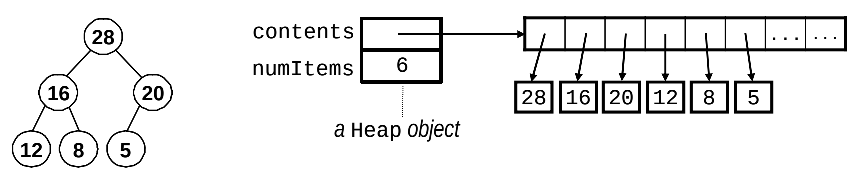
```
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;

    public Heap(int maxSize) {
        contents = (T[])new Comparable[maxSize];
        numItems = 0;
    }
    …
}
```



- `Heap` is another example of a generic collection class.
  - as usual, `T` is the type of the elements
  - `extends Comparable<T>` specifies `T` must implement `Comparable<T>`
  - must use `Comparable` (not `Object`) when creating the array

Heap Implementation (cont.)

```
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;

    …
}
```



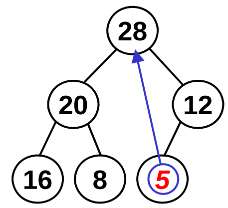
- The picture above is a heap of integers:

  `Heap<Integer> myHeap = new Heap<Integer>(20);`
  - works because `Integer` implements `Comparable<Integer>`
  - could also use `String` or `Double`
- For a priority queue, we can use objects of our `PQItem` class:

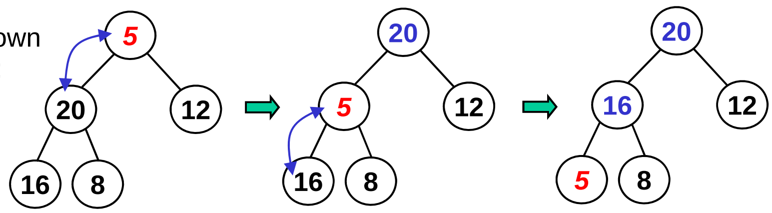
  `Heap<PQItem> pqueue = new Heap<PQItem>(50);`

Removing the Largest Item from a Heap

- Remove and return the item in the root node.
- In addition, we need to move the largest remaining item to the root, while maintaining a complete tree with each node >= children
- Algorithm:
  1. make a copy of the largest item
  2. move the last item in the heap to the root
  3. “sift down” the new root item until it is >= its children (or it’s a leaf)
  4. return the largest item



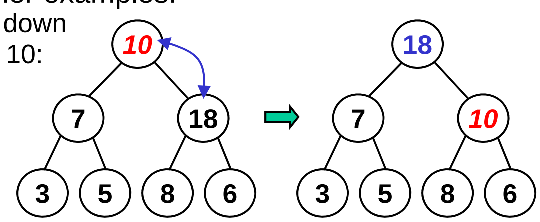
sift down the 5:



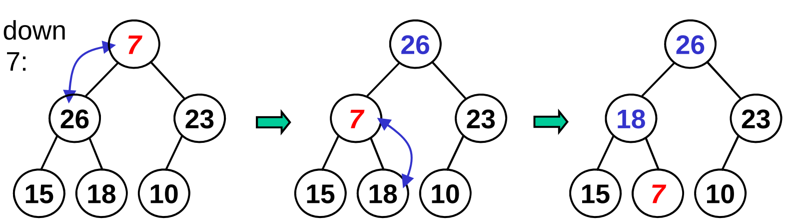
Sifting Down an Item

- To sift down item *x* (i.e., the item whose key is *x*):
  1. compare *x* with the larger of the item’s children, *y*
  2. if *x* < *y*, swap *x* and *y* and repeat
- Other examples:

sift down the 10:



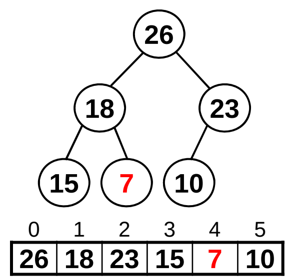
sift down the 7:



`siftDown()` Method

```
private void siftDown(int i) {
    T toSift = contents[i];

    int parent = i;
    int child = 2 * parent + 1;
    while (child < numItems) {
        // If the right child is bigger, compare with it.
        if (child < numItems - 1  &&
          contents[child].compareTo(contents[child + 1]) < 0)
            child = child + 1;

        if (toSift.compareTo(contents[child]) >= 0)
            break;  // we're done

        // Move child up and move down one level in the tree.
        contents[parent] = contents[child];
        parent = child;
        child = 2 * parent + 1;
    }

    contents[parent] = toSift;
}
```

- We don't actually swap items. We wait until the end to put the sifted item in place.

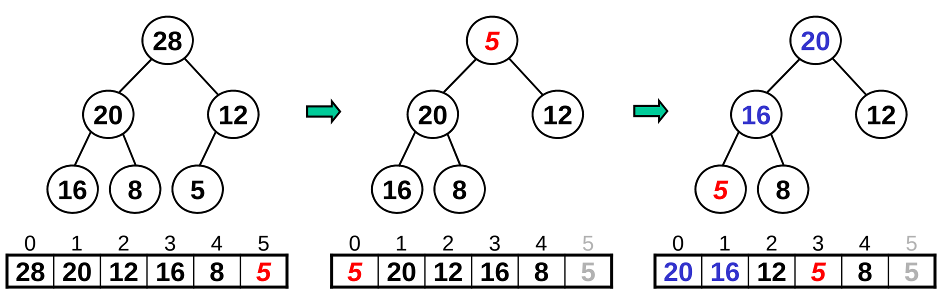


toSift: **7**

parent	child
0	1
1	3
1	4
4	9

`remove()` Method

```
public T remove() {
    T toRemove = contents[0];

    contents[0] = contents[numItems - 1];
    numItems--;
    siftDown(0);

    return toRemove;
}
```



numItems: **6**
toRemove: **28**

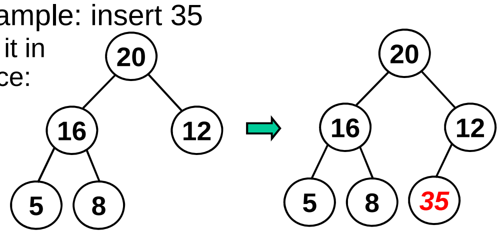
numItems: **5**
toRemove: **28**

numItems: **5**
toRemove: **28**

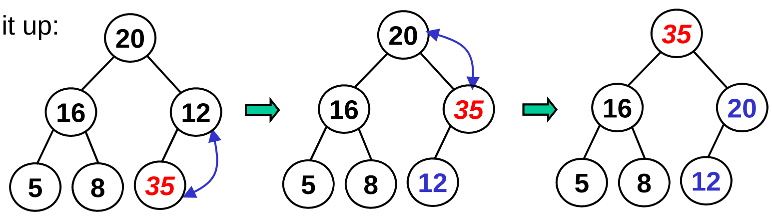
Inserting an Item in a Heap

- Algorithm:
  1. put the item in the next available slot (grow array if needed)
  2. “sift up” the new item
     until it is <= its parent (or it becomes the root item)
- Example: insert 35

put it in place:

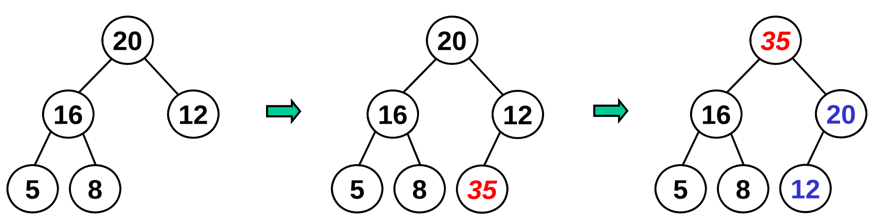


sift it up:



`insert()` Method

```
public void insert(T item) {
    if (numItems == contents.length) {
        // code to grow the array goes here…
    }
    contents[numItems] = item;
    siftUp(numItems);
    numItems++;
}
```



0	1	2	3	4	5
20	16	12	5	8	

numItems: **5**
item: **35**

0	1	2	3	4	5
20	16	12	5	8	35

numItems: **5**
item: **35**

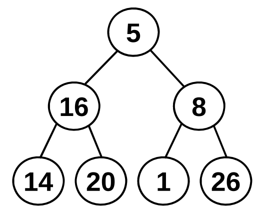
0	1	2	3	4	5
35	16	20	5	8	12

numItems: **6**

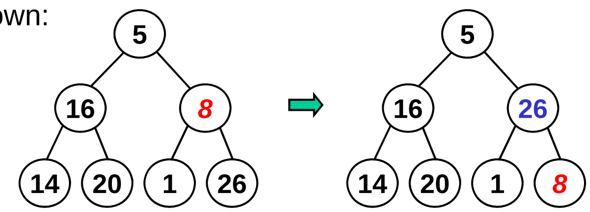
Converting an Arbitrary Array to a Heap

- Algorithm to convert an array with n items to a heap:
  1. start with the parent of the last element:
     `contents[i]`, where `i = ((n – 1) – 1)/2 = (n – 2)/2`
  2. sift down `contents[i]` and all elements to its left
- Example:

0	1	2	3	4	5	6
5	16	8	14	20	1	26

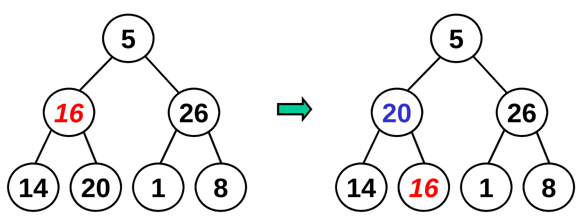


- Last element's parent = `contents[(7 – 2)/2]` = `contents[2]`.
  Sift it down:

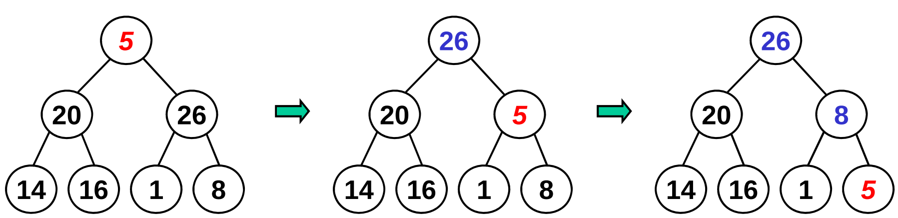


Converting an Array to a Heap (cont.)

- Next, sift down `contents[1]`:



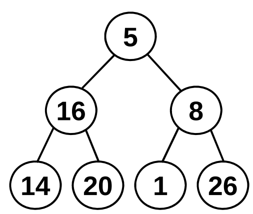
- Finally, sift down `contents[0]`:



Creating a Heap from an Array

```
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;
    ...
    public Heap(T[] arr) {
        // Note that we don't copy the array!
        contents = arr;
        numItems = arr.length;
        makeHeap();
    }

    private void makeHeap() {
        int last = contents.length - 1;
        int parentOfLast = (last - 1)/2;
        for (int i = parentOfLast; i >= 0; i--)
            siftDown(i);
    }
    ...
}
```

Time Complexity of a Heap



- A heap containing n items has a height <= $\log_2 n$.
- Thus, removal and insertion are both $O(\log n)$.
  - remove: go down at most $\log_2 n$ levels when sifting down from the root, and do a constant number of operations per level
  - insert: go up at most $\log_2 n$ levels when sifting up to the root, and do a constant number of operations per level
- This means we can use a heap for a $O(\log n)$-time priority queue.
- Time complexity of creating a heap from an array?

Using a Heap to Sort an Array

- Recall selection sort: it repeatedly finds the smallest remaining element and swaps it into place:

0	1	2	3	4	5	6
5	16	8	14	20	1	26

0	1	2	3	4	5	6
1	16	8	14	20	5	26

0	1	2	3	4	5	6
1	*5*	8	14	20	16	26

…

- It isn't efficient ($O(n^2)$), because it performs a linear scan to find the smallest remaining element ($O(n)$ steps per scan).
- Heapsort is a sorting algorithm that repeatedly finds the *largest* remaining element and puts it in place.
- It *is* efficient ($O(n \log n)$), because it turns the array into a heap, which means that it can find and remove the largest remaining element in $O(\log n)$ steps.

Heapsort

```
public class HeapSort {
    public static <T extends Comparable<T>> void
    heapSort(T[] arr) {
        // Turn the array into a max-at-top heap.
        Heap<T> heap = new Heap<T>(arr);

        int endUnsorted = arr.length - 1;
        while (endUnsorted > 0) {
            // Get the largest remaining element and put it
            // at the end of the unsorted portion of the array.
            T largestRemaining = heap.remove();
            arr[endUnsorted] = largestRemaining;

            endUnsorted--;
        }
    }
}
```

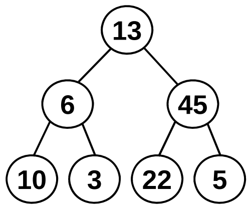
- We define a *generic method*, with a type variable in the method header. It goes right before the method's return type.
- `T` is a placeholder for the type of the array.
  - can be any type `T` that implements `Comparable<T>`.

Heapsort Example

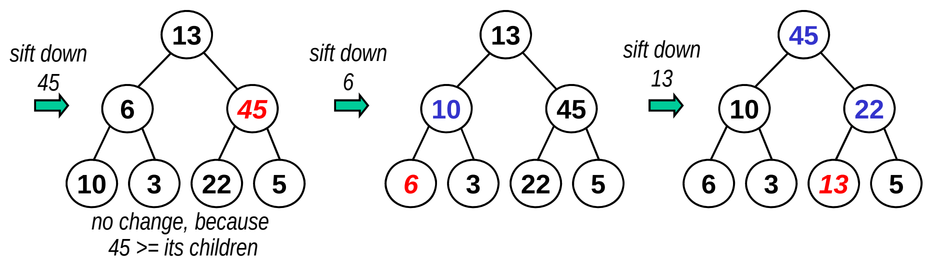
- Sort the following array:

0	1	2	3	4	5	6
13	**6**	**45**	**10**	**3**	**22**	**5**

- Here's the corresponding complete tree:

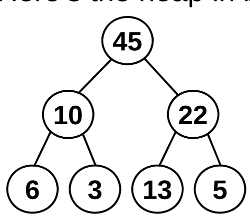


- Begin by converting it to a heap:



Heapsort Example (cont.)

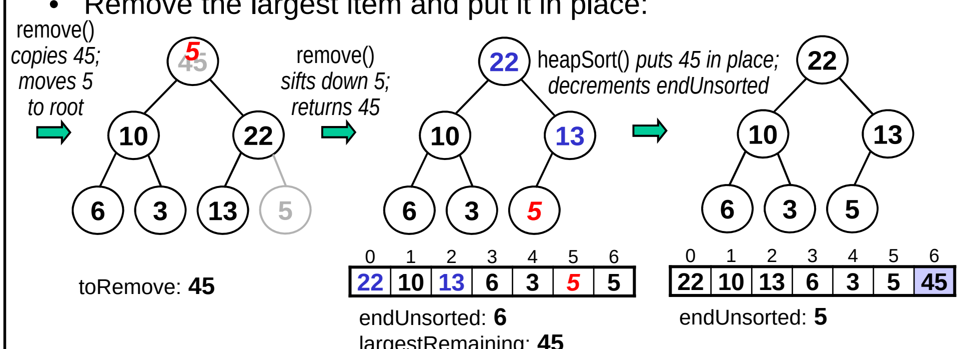
- Here's the heap in both tree and array forms:



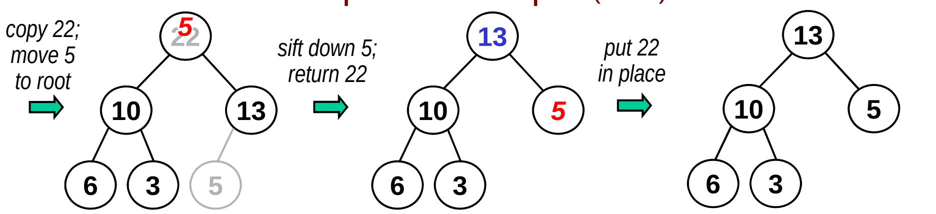
0	1	2	3	4	5	6
45	**10**	**22**	**6**	**3**	**13**	**5**

endUnsorted: **6**

- Remove the largest item and put it in place:



Heapsort Example (cont.)



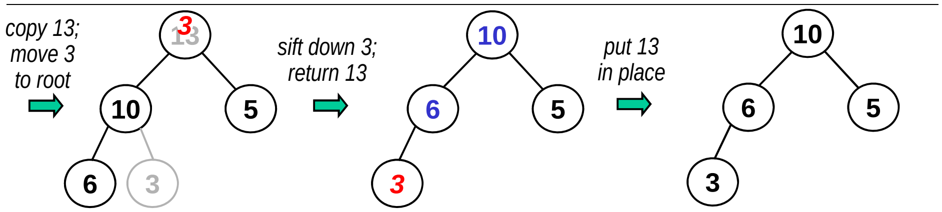
toRemove: **22**

0	1	2	3	4	5	6
13	10	5	6	3	5	45

endUnsorted: **5**
largestRemaining: **22**

0	1	2	3	4	5	6
13	10	5	6	3	22	45

endUnsorted: **4**



toRemove: **13**

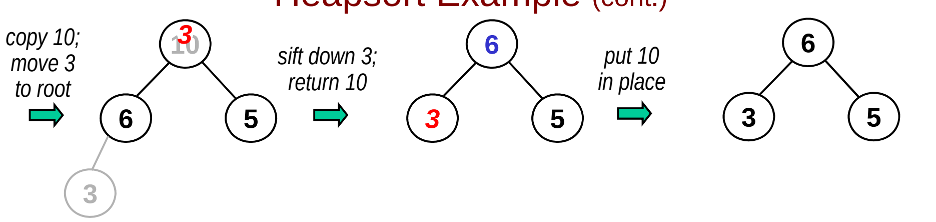
0	1	2	3	4	5	6
10	6	5	3	3	22	45

endUnsorted: **4**
largestRemaining: **13**

0	1	2	3	4	5	6
10	6	5	3	13	22	45

endUnsorted: **3**

Heapsort Example (cont.)



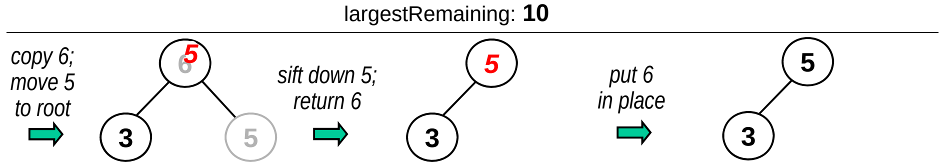
toRemove: **10**

0	1	2	3	4	5	6
6	3	5	3	13	22	45

endUnsorted: **3**
largestRemaining: **10**

0	1	2	3	4	5	6
6	3	5	10	13	22	45

endUnsorted: **2**



toRemove: **6**

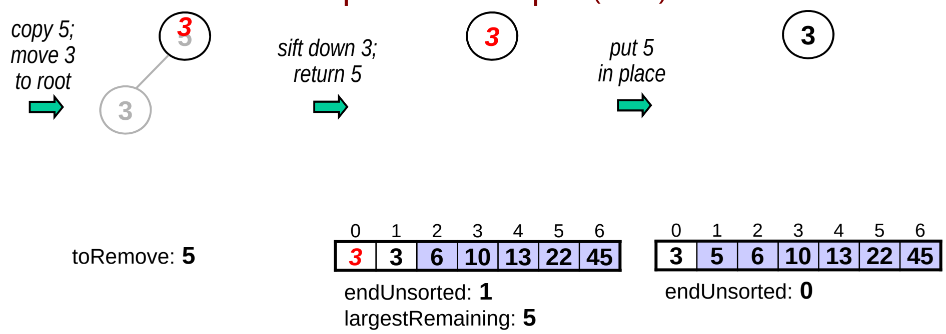
0	1	2	3	4	5	6
5	3	5	10	13	22	45

endUnsorted: **2**
largestRemaining: **6**

0	1	2	3	4	5	6
5	3	6	10	13	22	45

endUnsorted: **1**

Heapsort Example (cont.)



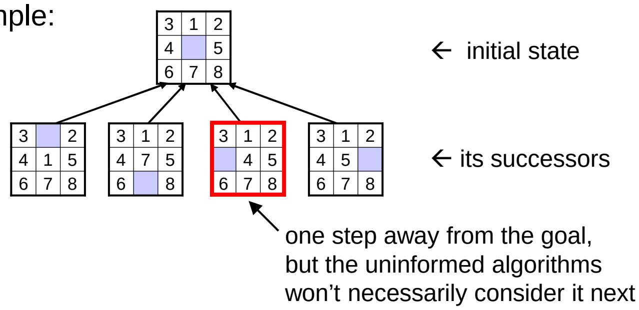
How Does Heapsort Compare?

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell sort	$O(n \log n)$	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(1)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

- Heapsort matches mergesort for the best worst-case time complexity, but it has better space complexity.
- Insertion sort is still best for arrays that are almost sorted.
  - heapsort will scramble an almost sorted array before sorting it
- Quicksort is still typically fastest in the average case.

State-Space Search Revisited

- Earlier, we considered three algorithms for state-space search:
  - breadth-first search (BFS)
  - depth-first search (DFS)
  - iterative-deepening search (IDS)
- These are all *uninformed* search algorithms.
  - always consider the states in a certain order
  - do not consider how close a given state is to the goal
- 8 Puzzle example:



Informed State-Space Search

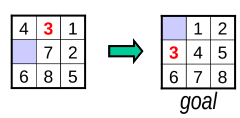
- *Informed* search algorithms attempt to consider more promising states first.
- These algorithms associate a *priority* with each successor state that is generated.
  - base priority on an estimate of nearness to a goal state
  - when choosing the next state to consider, select the one with the highest priority
- Use a priority queue to store the yet-to-be-considered search nodes.

State-Space Search: Estimating the Remaining Cost

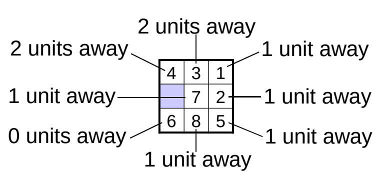
- The priority of a state is based on the *remaining cost* – i.e., the cost of getting from the state to the closest goal state.
  - for the 8 puzzle, remaining cost = # of steps to closest goal
- For most problems, we can't determine the exact remaining cost.
  - if we could, we wouldn't need to search!
- Instead, we estimate the remaining cost using a *heuristic function* h(x) that takes a state x and computes a cost estimate for it.
  - heuristic = rule of thumb
- To find optimal solutions, we need an *admissable* heuristic – one that never overestimates the remaining cost.

Heuristic Function for the Eight Puzzle

- Manhattan distance = horizontal distance + vertical distance
  - example: For the board at right, the Manhattan distance of the 3 tile from its position in the goal state = 1 column + 1 row = 2



- Use h(x) = sum of the Manhattan distances of the tiles in x from their positions in the goal state
  - for our example:

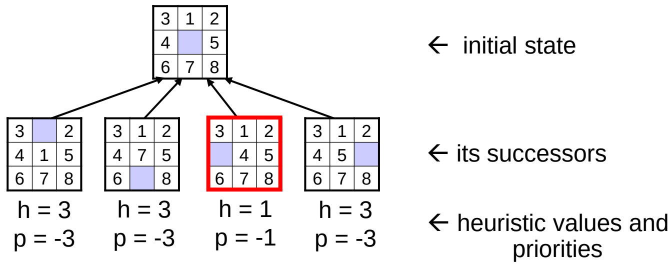


h(x) = 1 + 1 + 2 + 2
+ 1 + 0 + 1 + 1 = 9

- This heuristic is admissible because each of the operators (move blank up, move blank down, etc.) moves a single tile a distance of 1, so it will take at least h(x) steps to reach the goal.

Greedy Search

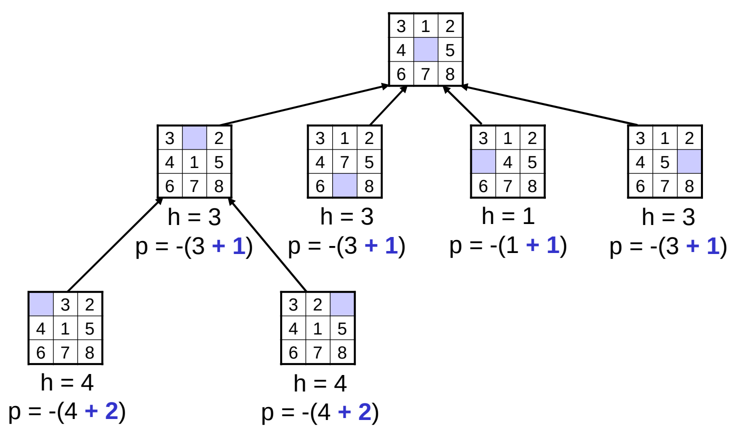
- Priority of state x, p(x) = -1 * h(x)
  - mult. by -1 so states closer to the goal have higher priorities



- Greedy search would consider the highlighted successor before the other successors, because it has the highest priority.
- Greedy search is:
  - incomplete: it may not find a solution
    - it could end up going down an infinite path
  - not optimal: the solution it finds may not have the lowest cost
    - it fails to consider the cost of getting *to* the current state

A* Search

- Priority of state x, p(x) = -1 * (h(x) + g(x))
  where g(x) = the cost of getting from the initial state to x



- Incorporating g(x) allows A* to find an optimal solution – one with the minimal *total* cost.

Characteristics of A*

- It is complete and optimal.
  - provided that h(x) is admissable, and that g(x) increases or stays the same as the depth increases
- Time and space complexity are still typically exponential in the solution depth, d – i.e., the complexity is $O(b^d)$ for some value b.
- However, A* typically visits far fewer states than other optimal state-space search algorithms.

solution depth	iterative deepening	A* w/ Manhattan dist. heuristic
4	112	12
8	6384	25
12	364404	73
16	did not complete	211
20	did not complete	676

Source: Russell & Norvig, *Artificial Intelligence: A Modern Approach*, Chap. 4.

The numbers shown are the average number of search nodes visited in 100 randomly generated problems for each solution depth.

The searches do *not* appear to have excluded previously seen states.

- Memory usage can be a problem, but it's possible to address it.

Implementing Informed Search

- Add new subclasses of the abstract `Searcher` class.
- For example:

```
public class GreedySearcher extends Searcher {
    private Heap<PQItem> nodePQueue;

    public void addNode(SearchNode node) {
        nodePQueue.insert(
          new PQItem(node, -1 * node.getCostToGoal()));
    }
    …
```