CSCI 2270 – CS 2: Data Structures

University of Colorado
Boulder

Reminders

Topics

- Heaps
- Priority Queues

Heaps

- What is a heap?
  - A list in which each element contains a key, such that the key in the element at position k in the list is at least as large as the key in the element at position 2k + 1 (if it exists) and 2k + 2 (if it exists).

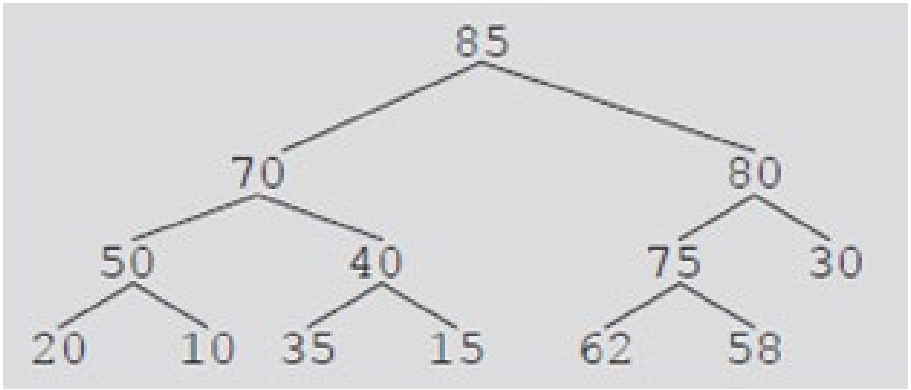
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
85	70	80	50	40	75	30	20	10	35	15	62	58

  - The (binary) heap is a tree data structure, where the data is stored as a complete binary tree.
- A **complete** binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. The difference in height between two branches is at most 1.

Heaps

- Example of a complete binary tree

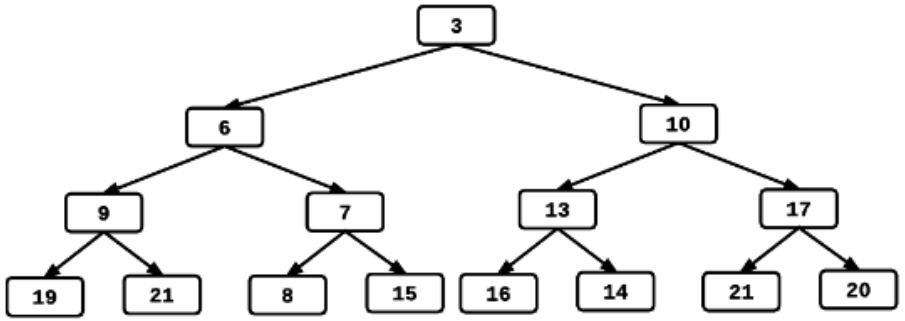
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
85	70	80	50	40	75	30	20	10	35	15	62	58



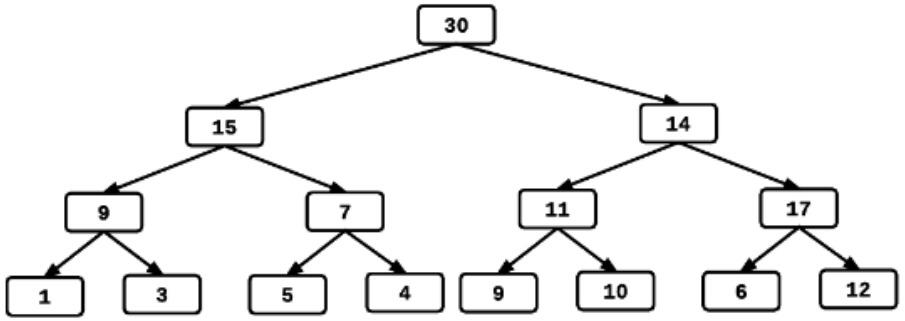
- parent(x) = floor((x-1)/2)
- leftChild(x) = 2x+1
- rightChild(x) = 2x+2

Heaps

- There are 2 types of heaps
  - Min-heap
  - Max-heap



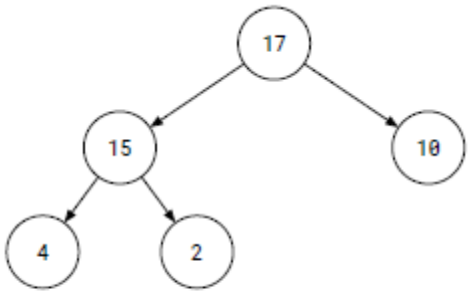
In a min-heap, the value of a node is less than the value of either of its children.
The minimum value in the tree is the root node.



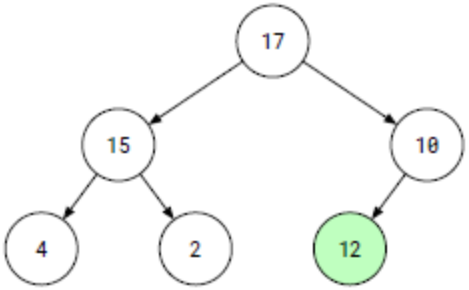
In a max-heap, the value of a node is greater than the value of either of its children.
The root of the tree is the maximum value in the tree.

Heaps – Adding Elements

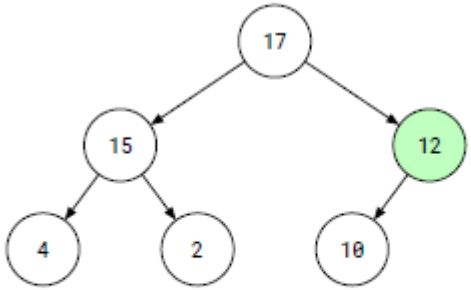
- Let's look at an example of inserting 12 into the heap.



starting heap



add 12 to first empty slot

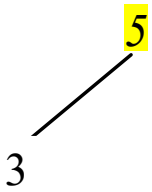


swap with parents until invariant OK
(and we're now OK)

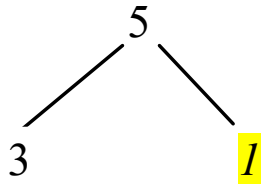
Heaps – Adding Elements

- Adding 3, 5, 1, 19, 11, and 22 to a heap, initially empty

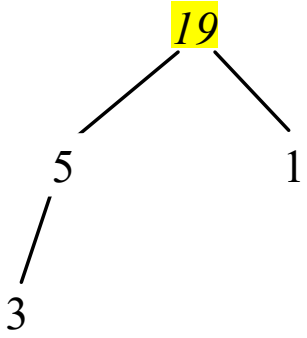
(a) After adding 3



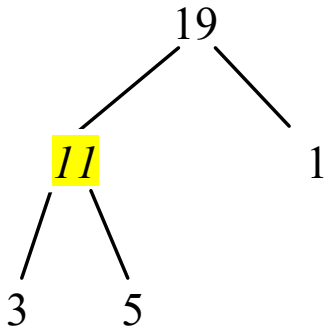
(b) After adding 5



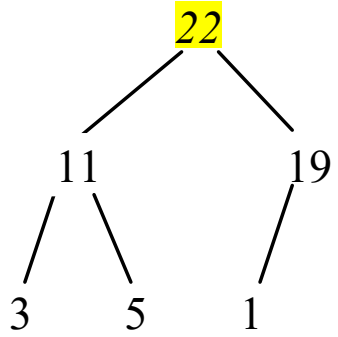
(c) After adding 1



(d) After adding 19



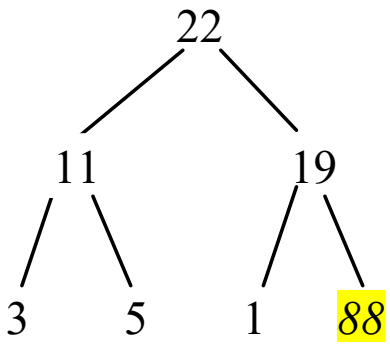
(e) After adding 11



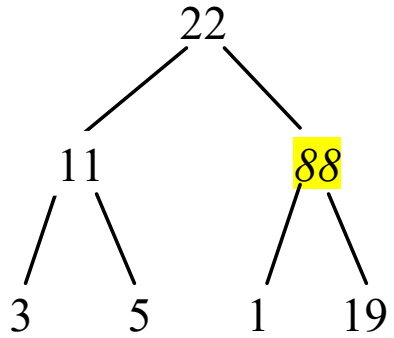
(f) After adding 22

Heaps – Adding Elements

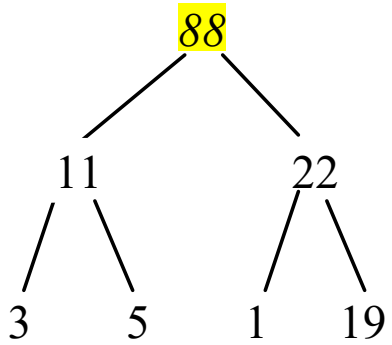
- Rebuild the heap after adding a new node
  - Adding 88 to the heap



(a) Add 88 to a heap



(b) After swapping 88 with 19



(b) After swapping 88 with 22

Min-Heap ADT

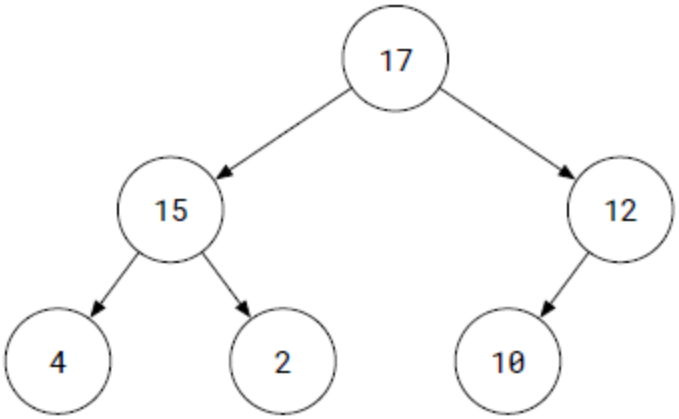
```
class MinHeap{
   public:
      MinHeap(int capacity);
      ~MinHeap();
      void push(int k); // insert
      int pop(); // extracMin
      int peek() { return heap[0]; }
      void printHeap();
   private:
      int *heap; // pointer to array of elements in heap
      int capacity; // maximum possible size of min heap
      int currentSize; // Current number of elements in min heap
      void MinHeapify(int index);
      int parent(int i) { return (i-1)/2; }
      int left(int i) { return (2*i+1); }
      int right(int i) { return (2*i + 2); }
      void swap(int &x, int &y);
};
```

When are Heaps useful?

- Heaps are used when the highest or lowest order/priority element needs to be removed. They allow quick access to this item in O(1) time. One use of a heap is to implement a priority queue.
- Binary heaps are usually implemented using arrays, which save overhead cost of storing pointers to child nodes.

Heap Sort

- Goal: produce an ordered sequence of elements based on some input. Let's use heap sort.

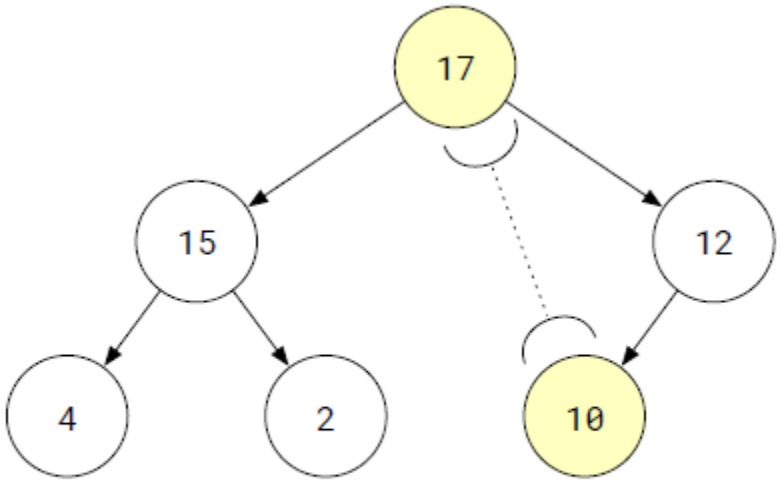


- Build a heap out of the data, just inserting values one at a time.

Heap Sort Algorithm

1. Swap the top of the heap with the last element.
2. Remove the last element, add it to the sorted list.
3. Repair the heap: large numbers above smaller ones.
   - Opposite for min-heap
4. Repeat until heap has nothing left in it.

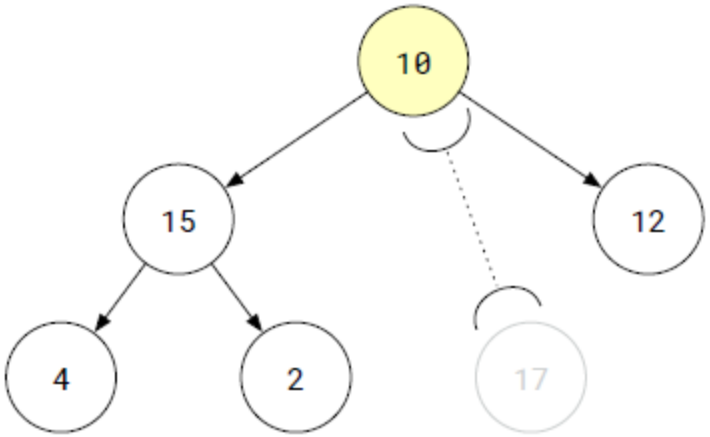
Heap Sort Max-Heap E.g.




```
// Swap top node (17) with last (10).
```

```
// Return list: [ ] (empty)
```

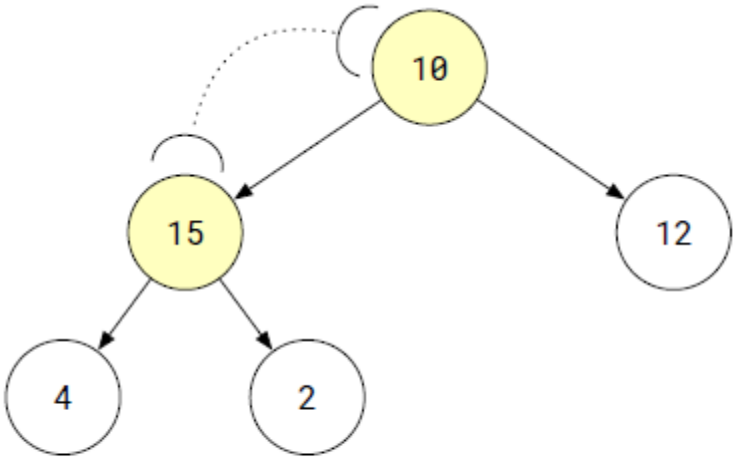
Heap Sort Max-Heap E.g.




```
// Now remove the 17, add it to the return list.
```

```
// Return list: [ 17 ]
```

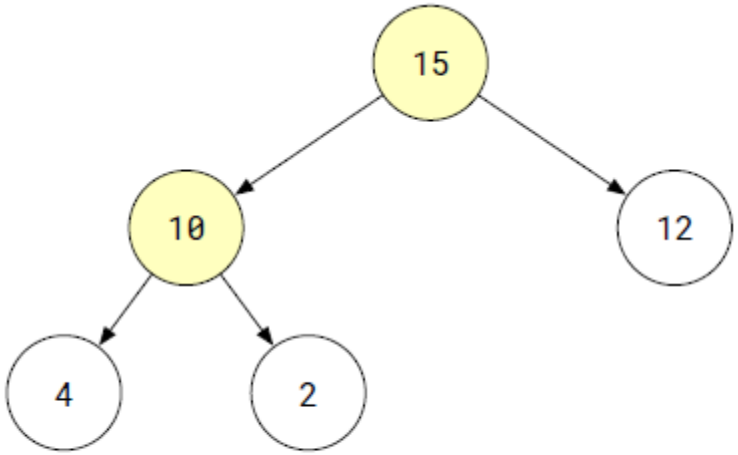
Heap Sort Max-Heap E.g.




```
// Fix broken max heap invariant. Swap 10 with
// the larger of the two children.
```

```
// Return list: [ 17 ]
```

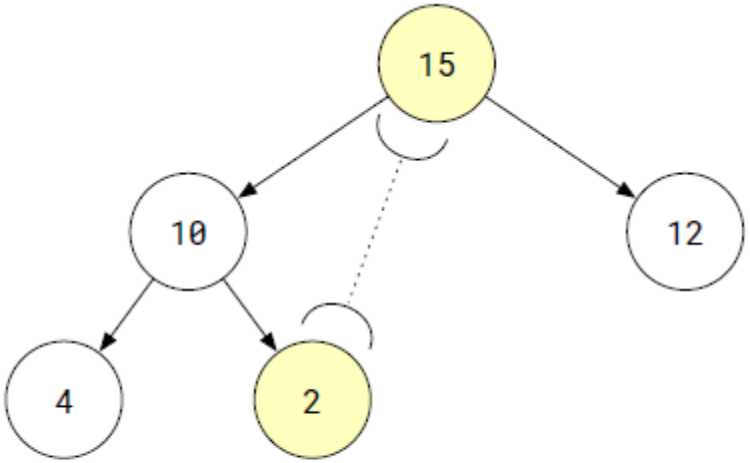
Heap Sort Max-Heap E.g.




```
// Heap is repaired. Can now continue sorting.
```

```
// Return list: [ 17 ]
```

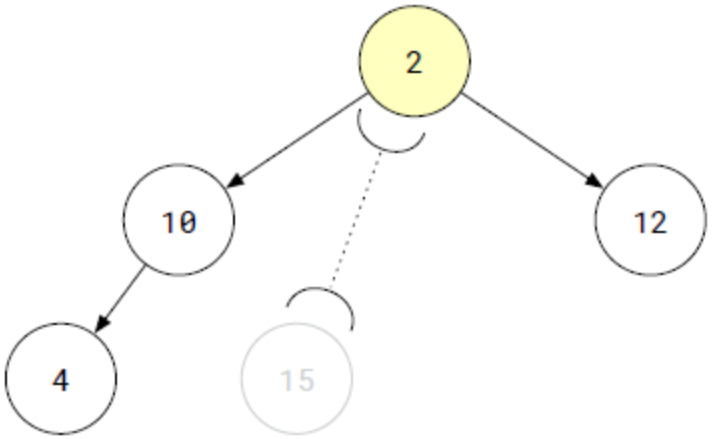
Heap Sort Max-Heap E.g.




```
// Step 1 again. Swap root with last element.
```

```
// Return list: [ 17 ]
```

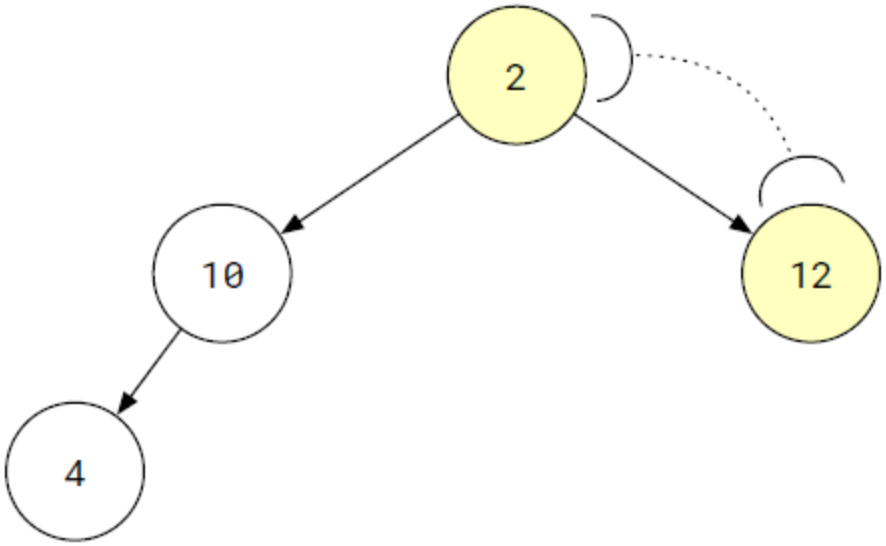
Heap Sort Max-Heap E.g.




```
// 15 added to sorted list, but heap needs fixing.
```

```
// Return list: [ 17, 15 ]
```

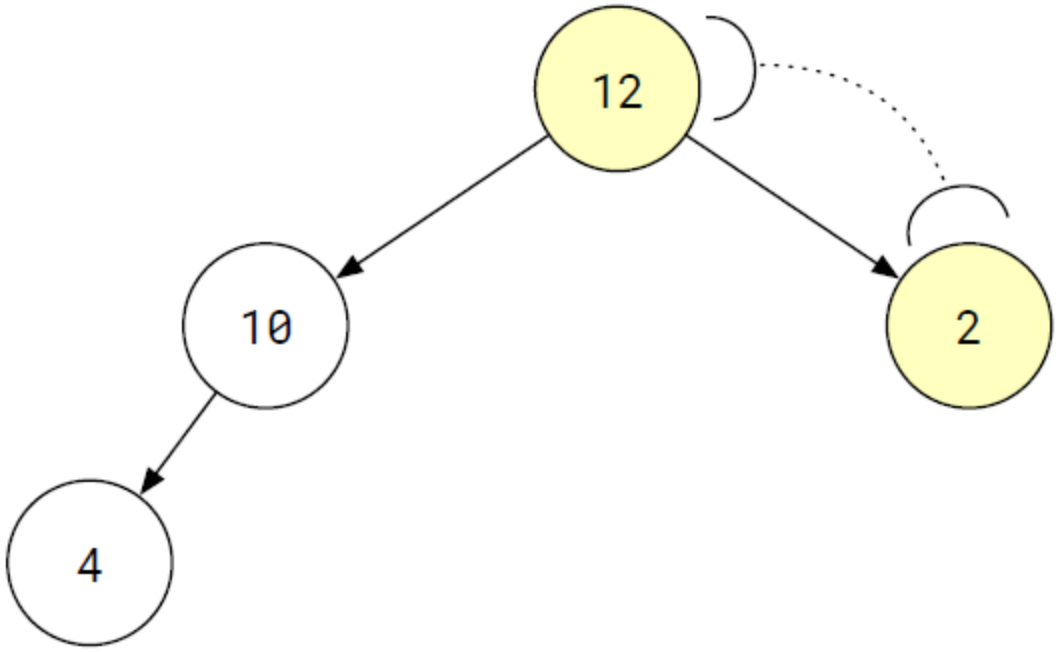
Heap Sort Max-Heap E.g.




```
// Swap 2 with the larger child
```

```
// Return list: [ 17, 15 ]
```

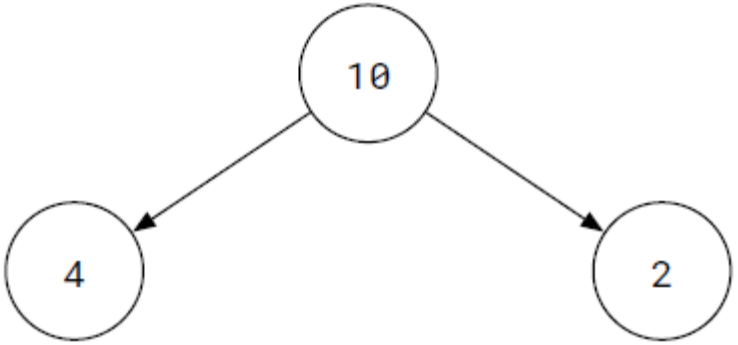
Heap Sort Max-Heap E.g.




```
// All good
```

```
// Return list: [ 17, 15 ]
```

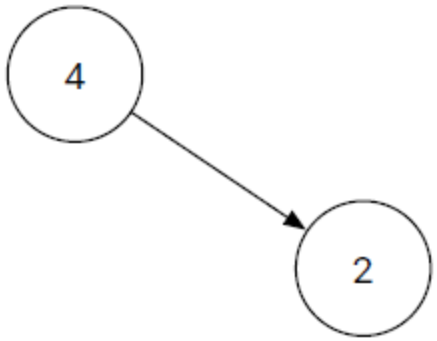
Heap Sort Max-Heap E.g.




```
// After removing 12 and fixing

// Return list: [ 17, 15, 12 ]
```

Heap Sort Max-Heap E.g.




```
// After removing 10 and fixing
```

```
// Return list: [ 17, 15, 12, 10 ]
```

Heap Sort Max-Heap E.g.


```
// After removing 4
```

```
// Return list: [ 17, 15, 12, 10, 4 ]
```

Heap Sort Max-Heap E.g.

```
                                        // After removing 2 - heap empty. All done!
// Return list: [ 17, 15, 12, 10, 4 ]
```

Priority Queues

- A common use of a heap is to implement a priority queue.
- In a heap, the highest (or lowest) priority element is always stored at the root.
- In a priority queue, the element/customer/job, etc. with the higher priority are pushed to the front of the queue.
  - Elements are assigned priorities
  - When accessing elements, the elements with the highest priority is removed first.

Priority Queues: Application

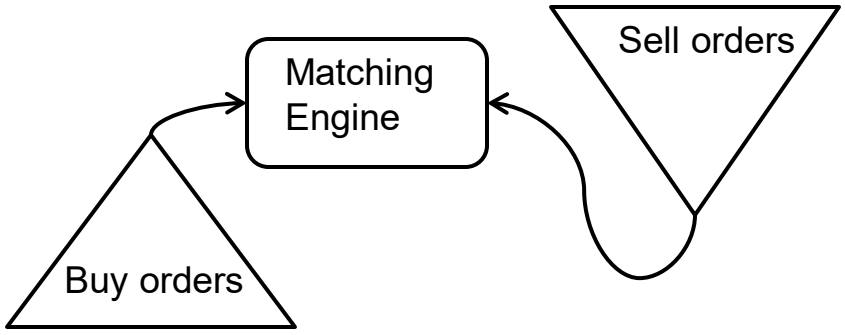
- Stock Matching Engine
  - At the heart of modern stock trading systems are highly reliable systems known as **matching engines**, which match the stock trades of buyers and sellers.
  - A simplification of how such a system works is in terms of a **continuous limit order book**, where buyers post bids to buy a number of shares in a given stock at or below a specified price and sellers post offers to sell a number of shares of a given stock at or above a specified price.

STOCK: EXAMPLE.COM

Buy Orders			Sell Orders		
Shares	Price	Time	Shares	Price	Time
1000	4.05	20 s	500	4.06	13 s
100	4.05	6 s	2000	4.07	46 s
2100	4.03	20 s	400	4.07	22 s
1000	4.02	3 s	3000	4.10	54 s
2500	4.01	81 s	500	4.12	2 s
			3000	4.20	58 s
			800	4.25	33 s
			100	4.50	92 s

Priority Queues: Application

- Buy and sell orders are organized according to a **price-time priority**:
  - "price" has highest priority and if there are multiple orders for the same price, then ones that have been in the order book the longest have higher priority.
- When a new order is entered, the matching engine determines if a trade can be immediately executed and if so, then it performs the appropriate matches according to price-time priority.
- This amount to **two** instances of the data structure we discuss here—the **priority queue**—one for buy orders and one for sell orders.
- This data structure performs element removals based on priorities assigned to elements when they are inserted.



STOCK: EXAMPLE.COM

Buy Orders			Sell Orders		
Shares	Price	Time	Shares	Price	Time
1000	4.05	20 s	500	4.06	13 s
100	4.05	6 s	2000	4.07	46 s
2100	4.03	20 s	400	4.07	22 s
1000	4.02	3 s	3000	4.10	54 s
2500	4.01	81 s	500	4.12	2 s
			3000	4.20	58 s
			800	4.25	33 s
			100	4.50	92 s

Priority Queue is an ADT

- We can use the following to implement a priority queue:
  - Unsorted list
  - Sorted list
  - A heap
    - ***Items are stored in a heap data structure (binary trees and arrays)***
  - Other
    - ***As long as it does what a priority queue entails, it is acceptable***

Priority Queue Interface

- Operations
  - insert(pq, something, priority)
    - ***adds something (e.g. string) to queue with given priority***
    - ***Aka enqueue; push***
  - remove(pq)
    - ***Removes something from queue with highest priority***
    - ***Aka dequeue; pop***
  - peek(pq)
    - ***Reads (doesn't remove) something with the highest priority***
    - ***Aka front; head***

Priority Queue Operations

```
void insert(pq*&queue, string &data, float priority);

string remove(pq*&queue);

string peek(pq*&queue)
```

Priority Queue Summary

- Priority queues can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among the aforementioned data structures, the heap data structure provides an efficient implementation of priority queues.

	peek	insert	delete
Linked List	O(1)	O(n)	O(1)
Binary Heap	O(1)	O(log n)	O(log n)
BST	O(1)	O(log n)	O(log n)

Heap Code Examples

- Let's look at a few examples - minHeapify.cpp, stlPriorityQueue.cpp

Questions?