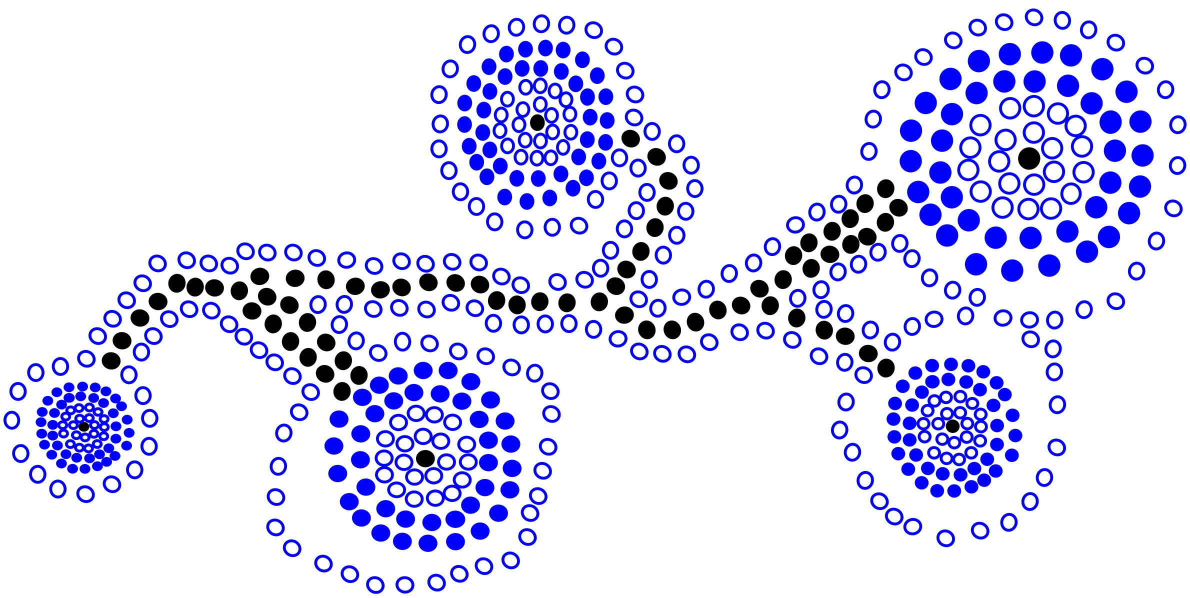
PRIORITY QUEUES

- Stock trading (motivation)
- The priority queue ADT
- Implementing a priority queue with a sequence
- Elementary sorting
- Issues in sorting



Stock Trading

- We focus on the trading of a single security, say Akamai Technologies, founded in 1998 by CS professors and students at MIT (200 employees, $20B market cap)
- Investors place ***orders*** consisting of three items (***action***, ***price***, ***size***), where ***action*** is either ***buy*** or ***sell***, ***price*** is the worst price you are willing to pay for the purchase or get from your sale, and ***size*** is the number of shares
- At equilibrium, all the buy orders (***bids***) have prices lower than all the sell orders (***asks***)
- A ***level 1 quote*** gives the highest bid and lowest ask (as provided by popular financial sites, and e-brokers for the naive public)
- A ***level 2 quote*** gives all the bids and asks for several price steps (Island ECN on the Web and quote subscriptions for professional traders)
- A ***trade*** occurs whenever a new order can be matched with one or more existing orders, which results in a series of ***removal*** transactions
- Orders may be ***canceled*** at any time

Data Structures for the Stock Market

- For each security, we keep two structures, one for the buy orders (bids), and the other for the sell orders (asks)
- Operations that need to be supported

Action	***Ask Structure***	***Bid Structure***
place an order	***insert***(price, size)	***insert***(price, size)
get level 1 quote	***min***()	***max***()
trade	***removeMin***()	***removeMax***()
cancel	***remove***(order)	***remove***(order)

- These data structures are called ***priority queues***.
- The NASDAQ priority queues support an average daily trading volume of 1B shares ($50B)

Keys and Total Order Relations

- A Priority Queue ranks its elements by ***key*** with a ***total order*** relation

- Keys:
  - Every element has its own key
  - Keys are not necessarily unique

- Total Order Relation
  - Denoted by $\leq$
  - **Reflexive:** $k \leq k$
  - **Antisymetric:** if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 \leq k_2$
  - **Transitive:** if $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$

- A Priority Queue supports these fundamental methods on key-element pairs:
  - min()
  - insertItem(k, e)
  - removeMin()

Sorting with a Priority Queue

- A Priority Queue P can be used for sorting a sequence S by:
  - inserting the elements of S into P with a series of insertItem(e, e) operations
  - removing the elements from P in increasing order and putting them back into S with a series of removeMin() operations

Algorithm PriorityQueueSort(S, P):

Input: A sequence S storing n elements, on which a total order relation is defined, and a Priority Queue P that compares keys with the same relation

Output: The Sequence S sorted by the total order relation

```
while !S.isEmpty() do
    e ← S.removeFirst()
    P.insertItem(e, e)
while P is not empty do
    e ← P.removeMin()
    S.insertLast(e)
```

The Priority Queue ADT

- A prioriy queue *P* supports the following methods:

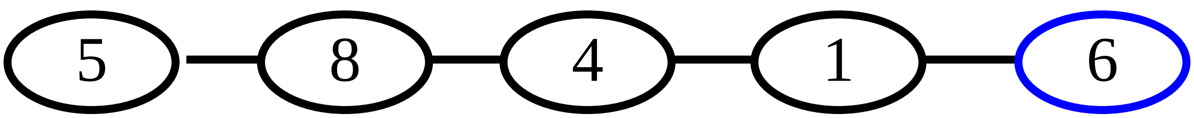
  - size():
    Return the number of elements in *P*
  - isEmpty():
    Test whether *P* is empty
  - insertItem(*k*,*e*):
    Insert a new element *e* with key *k* into *P*
  - minElement():
    Return (but don't remove) an element of *P* with smallest key; an error occurs if *P* is empty.
  - minKey():
    Return the smallest key in *P*; an error occurs if *P* is empty
  - removeMin():
    Remove from *P* and return an element with the smallest key; an error condidtion occurs if *P* is empty.

Comparators

- The most general and reusable form of a priority queue makes use of **comparator** objects.
- Comparator objects are external to the keys that are to be compared and compare two objects.
- When the priority queue needs to compare two keys, it uses the comparator it was given to do the comparison.
- Thus a priority queue can be general enough to store any object.
- The comparator ADT includes:
  - isLessThan(*a*, *b*)
  - isLessThanOrEqualTo(*a*,*b*)
  - isEqualTo(*a*, *b*)
  - isGreaterThan(*a*,*b*)
  - isGreaterThanOrEqualTo(*a*,*b*)
  - isComparable(*a*)

Implementation with an Unsorted Sequence

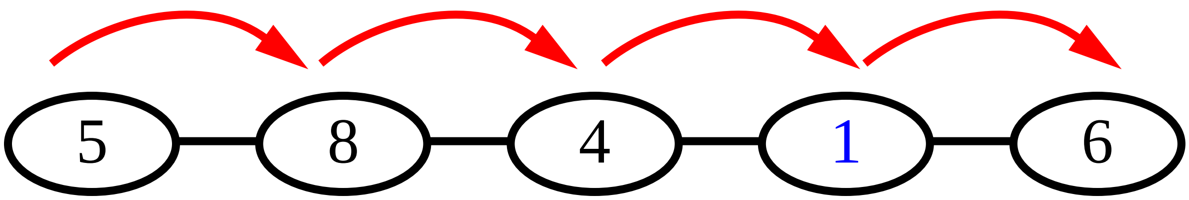
- Let's try to implement a priority queue with an unsorted sequence *S*.
- The elements of *S* are a composition of two elements, *k*, the key, and *e*, the element.
- We can implement insertItem() by using insertLast() on the sequence. This takes $O(1)$ time.



- However, because we always insert at the end, irrespectively of the key value, our sequence is not ordered.

Implementation with an Unsorted Sequence (contd.)

- Thus, for methods such as minElement(), minKey(), and removeMin(), we need to ***look at all the elements*** of *S*. The worst case time complexity for these methods is ***O***(*n*).



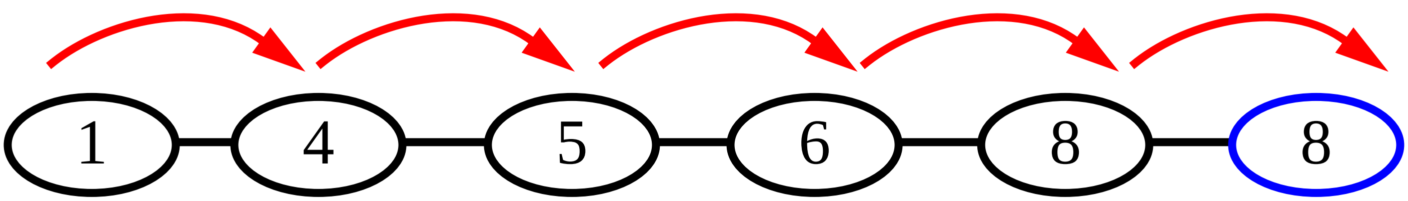
- Performance summary

insertItem	***O*(1)**
minKey, minElement	***O*(*n*)**
removeMin	***O*(*n*)**

Implementation with a Sorted Sequence

- Another implementation uses a sequence *S*, sorted by increasing keys
- minElement(), minKey(), and removeMin() take ***O***(1) time

- However, to implement insertItem(), we must now scan through the entire sequence ***in the worst case***. Thus, insertItem() runs in $O(n)$ time



- Performance summary

insertItem	***O*(*n*)**
minKey, minElement	***O*(1)**
removeMin	***O*(1)**

Implementation with a Sorted Sequence(contd.)

```
public class SequenceSimplePriorityQueue
implements SimplePriorityQueue {
  //Implementation of a priority queue
  using a sorted sequence
 protected Sequence seq = new NodeSequence();
 protected Comparator comp;


// auxiliary methods
 protected Object key (Position pos) {
    return ((Item)pos.element()).key();
  } // note casting here


 protected Object element (Position pos) {
    return ((Item)pos.element()).element();
  } // casting here too


  // methods of the SimplePriorityQueue ADT
 public SequenceSimplePriorityQueue (Comparator c) {
    comp = c; }
 public int size () {return seq.size(); }
```

...Continued on next page...

Implementation with a Sorted Sequence(contd.)

```
public void insertItem (Object k, Object e) throws
InvalidKeyException {
  if (!comp.isComparable(k)) {
    throw new InvalidKeyException("The key is not valid");
  }
  else {
    if (seq.isEmpty()) {
      //if the sequence is empty, this is the
      seq.insertFirst(new Item(k,e));//first item
    }
    else { //check if it fits right at the end
      if (comp.isGreaterThan(k,key(seq.last()))) {
        seq.insertAfter(seq.last(),new Item(k,e));
      }
      else {
        //we have to find the right place for k.
        Position curr = seq.first();
        while (comp.isGreaterThan(k,key(curr))) {
          curr = seq.after(curr);
        }
        seq.insertBefore(curr,new Item(k,e));
      }
    }
  }
```

...Continued...

Implementation with a Sorted Sequence(contd.)

```
public Object minElement () throws
EmptyContainerException {
if (seq.isEmpty()) {
    throw new EmptyContainerException("The priority
    queue is empty");
}
else {
    return element(seq.first());
}



public boolean isEmpty () {
    return seq.isEmpty();
}
}
```

Selection Sort

- Selection Sort is a variation of PriorityQueueSort that uses an ***unsorted sequence*** to implement the priority queue *P*.
- **Phase 1**, the insertion of an item into *P* takes ***O***(1) time
- **Phase 2**, removing an item from *P* takes time proportional to the current number of elements in *P*

	Sequence *S*	Priority Queue *P*
Input	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(7, 4)
...	...	...
(g)	()	(7, 4, 8, 2, 5, 3, ,9)
Phase 2:		
(a)	(2)	(7, 4, 8, 5, 3, 9)
(b)	(2, 3)	(7, 4, 8, 5, 9)
(c)	(2, 3, 4)	(7, 8, 5, 9)
(d)	(2, 3, 4, 5)	(7, 8, 9)
(e)	(2, 3, 4, 5, 7)	(8, 9)
(f)	(2, 3, 4, 5, 7, 8)	(9)
(g)	(2, 3, 4, 5, 7, 8, 9)	()

Selection Sort (cont.)

- As you can tell, a bottleneck occurs in Phase 2. The first removeMinElement operation take $\boldsymbol{O}(n)$, the second $\boldsymbol{O}(n-1)$, etc. until the last removal takes only $\boldsymbol{O}(1)$ time.
- The total time needed for phase 2 is:

$$O(n + (n-1) + \ldots + 2 + 1) \equiv O\left(\sum_{i=1}^{n} i\right)$$

- By a well-known fact:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

- The total time complexity of phase 2 is then $\boldsymbol{O}(n^2)$. Thus, the time complexity of the algorithm is $\boldsymbol{O}(n^2)$.

Insertion Sort

- Insertion sort is the sort that results when we perform a PriorityQueueSort implementing the priority queue with a ***sorted sequence***.

	Sequence *S*	Priority Queue *P*
Input	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(4, 7)
(c)	(2, 5, 3, 9)	(4, 7, 8)
(d)	(5, 3, 9)	(2, 4, 7, 8)
(e)	(3, 9)	(2, 4, 5, 7, 8)
(f)	(9)	(2, 3, 4, 5, 7, 8)
(g)	()	(2, 3, 4, 5, 7, 8, 9)
Phase 2:		
(a)	(2)	(3, 4, 5, 7, 8, 9)
(b)	(2, 3)	(4, 5, 7, 8, 9)
...	...	...
(g)	(2, 3, 4, 5, 7, 8, 9)	()

Insertion Sort(cont.)

- We improve phase 2 to $\boldsymbol{O}(n)$.
- However, phase 1 now becomes the bottleneck for the running time. The first insertItem takes $\boldsymbol{O}(1)$ time, the second one $\boldsymbol{O}(2)$, until the last opertation takes $\boldsymbol{O}(n)$ time, for a total of $\boldsymbol{O}(n^2)$ time
- Selection-sort and insertion-sort both take $\boldsymbol{O}(n^2)$ time
- Selection-sort will ***always*** executs a number of operations proportional to n^2, no matter what is the input sequence.
- The running time of insertion sort varies depending on the input sequence.
- Neither is a good sorting method, except for small sequences
- We have yet to see the ultimate priority queue....

Sorting

- By now, you've seen a little bit of sorting, so let us tell you a little more about it.
- Sorting is essential because efficient ***searching*** in a database can be performed only if the records are sorted
- It is estimated that about 20% of all the computing time worldwide is devoted to sorting
- We shall see that there is a trade-off between the "simplicity" and efficiency of sorting algorithms:
- The elementary sorting algorithms you've just seen, though easy to understand and implement, take $\boldsymbol{O}(n^2)$ time (unusable for large values of n)
- more sophisticated algorithms take $\boldsymbol{O}(n\log n)$ time
- Comparison of Keys: *do we base comparison upon the entire key or upon parts of the key?*
- Space Efficiency: ***in-place*** *sorting vs. use of auxiliary structures*
- Stability: *a* ***stable*** *sorting algorithm preserves the initial relative order of equal keys*