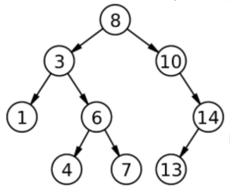
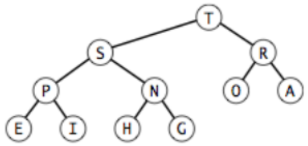
Heap Data Structures
BST: left node less than parent, right node greater than parent

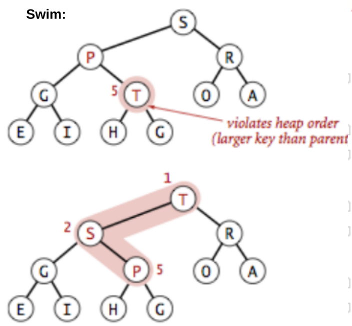




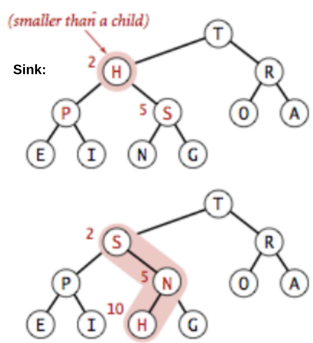
A heap-ordered complete binary tree

BST "Heap-Ordered"
each node is larger than or equal to the keys in that node's two children
compares up/down **O(log n)**
Parent: floor(k/2) --> Node = 6, then 6/2 = 3. Parent node is at a[3]
Left Child: 2*k --> Node = 6, then 6*2 = 12. First child node is at a[12]
Right Child: (2*k) + 1 --> Node = 6, then 6*2+1 = 13. Second child node is at a[13]
Represented as a Binary Tree, **Structured** as an array

Swim:



Sink:



PQ
Can solve what is the largest or smallest in a set
insert(5): add something new to collection
delMax(): removes element that is largest and returns
Unordered (Lazy Approach)
Big-O performance for **Unordered Array**
Insert: O(1)
delMax: O(n)
Ordered (Eager Approach):
Big-O performance for **Ordered Array**
Insert: O(n)
delMax:
Small → Large order: O(1)
Large → Small order: O(n)

MaxPQ(): create a priority queue
PaxPQ(int max): create a PQ of initial max capacity
MaxPQ(Key[ ] a): creat PQ from keys in a [ ]
void insert(Key v): insert a key into the PQ
Key max(): return largest key
Key delMax(): return and remove largest key
boolean isEmpty(): is PQ empty?
int size(): number of keys in PQ

What is the difference between a (max) heap and a priority queue?
A Max-Heap is a complete binary tree in which the value in each internal node is greater than or equal to the values in the children of that node. The Priority Queue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

any sorting algorithm needs at least nlogn compares, so: Ω(NlgN).
As long as compares are the optimal way to sort, one cannot build any computer or any write any algorithm to do general sorting in less than nlogn time.

```java
public class MaxPQ<Key extends Comparable<Key>> {
    private Key[] pq;
    private int N = 0;
    public MaxPQ(int maxN) {
        pq = (Key[]) new Comparable[maxN + 1];
    }
    public boolean isEmpty() {
        return N == 0;
    }
    public int size() {
        return N;
    }
    public void insert(Key v) {
        pq[++N] = v;
        swim(N);
    }
    public Key delMax() {
        Key max = pq[1];
        exch( i: 1, N--);
        pq[N + 1] = null;
        sink( k: 1);
        return max;
    }
    private void swim(int k) {
        while (k > 1 && less( i: k / 2, k)) {
            exch(k,  j: k / 2);
            k = k / 2;
        }
    }
    private void sink(int k) {
        while (2 * k <= N) {
            int j = 2 * k;
            if (j < N && less(j,  j: j + 1)) j++;
            if (!less(k, j)) break;
            exch(k, j);
            k = j;
        }
    }
    private boolean less(int i, int j) {
        return ((Comparable<Key>) pq[i])
                .compareTo(pq[j]) < 0;
    }
    private void exch(int i, int j) {
        Key swap = pq[i];
        pq[i] = pq[j];
        pq[j] = swap;
    }
}
```

HeapSort nlogn

```java
public void sort(Comparable[] a) {
    int N = a.length;
    for (int k = N/2; k >= 1; k--)
        sink(a, k, N);
    while (N > 1) {
        exch(a, 1, N--);
        sink(a, 1, N);
    }
}
```

- **Selection Sort:**
  Comparisons: $\sim\frac{n^2}{2}$
  Exchanges: $\sim n$
- **Insertion Sort:**
  Comparisons: $\sim\frac{n^2}{4}$
  Exchanges: $\sim\frac{n^2}{4}$
- **Shellsort:**
  Comparisons: $O(n^{\frac{3}{2}}), \Omega(nlogn)$

Selection Sort
PROPERTIES

For a (randomly ordered) input, the *average* case is:
- Comparisons: $\sim\frac{n^2}{4}$
- Exchanges: $\sim\frac{n^2}{4}$

For a (reverse input), the worse case is:
- Comparisons: $\sim\frac{n^2}{2}$
- Exchanges: $\sim\frac{n^2}{2}$

For a (sorted) input, the best case is:
- Comparisons: $n - 1$
- Exchanges: 0

```java
public static
  int N = a
  for (int
```

Symbol Tables
ST(): create symbol table
void put(Key key, Value val): put key/value pair in table, remove if null
Value get(Key key): value paired with key, null if absent
void delete(Key key): remove key(and value) from table
boolean contains(Key key): is value paired with key?
boolean isEmpty(): is table empty?
int size(): number of key-value pairs
Key min(): smallest key
Key max(): largest key
Key floor(Key key): largest key <= to key
Key ceiling(Key key): smallest key >= to key
int rank(Key key): # of keys < key (start at 0)
Key select(int k): key of rank K
void deleteMin(): delete smallest key
void deleteMax(): delete largest key
int size(Key lo, Key hi): # of keys in [lo...hi]
Iterable<Key> keys(Key lo, Key hi): keys in[lo...hi] in sorted order
Iterable<Key> keys(): all keys in table
Comparisons: worst case (unordered list):
Unsuccessful: O(n), Successful: O(n), Insert: O(n), Average for successful search: O(1/2n)

Keys: More than 1 val? No
Keys allow values to nulls? No (means deleted)
Iterator order of keys: No order
Keys: immutable
Values: either
Ordered: Comparisons:
Unsuccessful: O(logn), Successful: O(logn)
Insert: Exists? O(logn), Doesn't? O(n)
Hash: Avg Search: O(logn), Avg Insert: O(logn). Doesn't efficiently support ordered operations

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		efficiently support ordered operations?
	search	insert	search hit	insert	
sequential search (unordered linked list)	N	N	$N/2$	N	no
binary search (ordered array)	$\lg N$	$2N$	$\lg N$	N	yes

method	order of growth of running time
put()	N
get()	$\log N$
delete()	N
contains()	$\log N$
size()	1
min()	1
max()	1
floor()	$\log N$
ceiling()	$\log N$
rank()	$\log N$
select()	1
deleteMin()	N
deleteMax()	1

ALGORITHM 3.2 Binary search (in an ordered array)

```java
public class BinarySearchST<Key extends Comparable<Key>, Value>
{
   private Key[] keys;
   private Value[] vals;
   private int N;

   public BinarySearchST(int capacity)
   {  // See Algorithm 1.1 for standard array-resizing code.
      keys = (Key[]) new Comparable[capacity];
      vals = (Value[]) new Object[capacity];
   }

   public int size()
   {  return N;  }

   public Value get(Key key)
   {
      if (isEmpty()) return null;
      int i = rank(key);
      if (i < N && keys[i].compareTo(key) == 0) return vals[i];
      else                                      return null;
   }

   public int rank(Key key)
   // See page 381.

   public void put(Key key, Value val)
   {  // Search for key. Update value if found; grow table if new.
      int i = rank(key);
      if (i < N && keys[i].compareTo(key) == 0)
      {  vals[i] = val; return;  }
      for (int j = N; j > i; j--)
      {  keys[j] = keys[j-1]; vals[j] = vals[j-1];  }
      keys[i] = key; vals[i] = val;
      N++;
   }

   public void delete(Key key)
   // See Exercise 3.1.16 for this method.
}
```

ALGORITHM 3.2 (continued) Binary search in an ordered array (iterative)

```java
   public int rank(Key key)
   {
      int lo = 0, hi = N-1;
      while (lo <= hi)
      {
         int mid = lo + (hi - lo) / 2;
         int cmp = key.compareTo(keys[mid]);
         if      (cmp < 0) hi = mid - 1;
         else if (cmp > 0) lo = mid + 1;
         else return mid;
      }
      return lo;
   }
```

ALGORITHM 3.2 (continued) Ordered symbol-table operat

```java
   public Key min()
   {  return keys[0];  }

   public Key max()
   {  return keys[N-1];  }

   public Key select(int k)
   {  return keys[k];  }

   public Key ceiling(Key key)
   {
      int i = rank(key);
      return keys[i];
   }

   public Key floor(Key key)
   // See Exercise 3.1.17.

   public Key delete(Key key)
   // See Exercise 3.1.16.

   public Iterable<Key> keys(Key lo, Key hi)
   {
      Queue<Key> q = new Queue<Key>();
      for (int i = rank(lo); i < rank(hi); i++)
      q.enqueue(keys[i]);
      if (contains(hi))
         q.enqueue(keys[rank(hi)]);
      return q;
   }
```

```java
Recorder<Item>
{void add(Item item)
boolean contains(Item item)}
class RecorderImp<Item>implements Recorder<Item>
private Stack<Item>stack;
public RecorderImp(){stack = new Stack<>();}
@override public void add(Item item){stack.push(item);}
@override public boolean containst(Item item){return stack.contains(item)}
@override public void undo(){stack.pop();}
```

ALGORITHM 3.1 Sequential search (in an unordered linked list)

```java
public class SequentialSearchST<Key, Value>
{
   private Node first;        // first node in the linked list

   private class Node
   {  // linked-list node
      Key key;
      Value val;
      Node next;
      public Node(Key key, Value val, Node next)
      {
         this.key  = key;
         this.val  = val;
         this.next = next;
      }
   }

   public Value get(Key key)
   {  // Search for key, return associated value.
      for (Node x = first; x != null; x = x.next)
         if (key.equals(x.key))
            return x.val;    // search hit
      return null;           // search miss
   }

   public void put(Key key, Value val)
   {  // Search for key. Update value if found; grow table if new.
      for (Node x = first; x != null; x = x.next)
         if (key.equals(x.key))
         {  x.val = val; return;  }       // Search hit: update val.
      first = new Node(key, val, first); // Search miss: add new node.
   }
}
```

a SymbolTable vs an OrderedSymbolTable (effic implimentation)
ordered symbol table would be better because you could use it to do binary search in an array or to create a BST.

why must there be at least N! leaves
-each node represents a decision, worst case is depth of tree, depth d so max leaves = 2^d, size of algo is n, so there are n! permu of n inputs. Any possible permu can end up as an output so every permu must be different leaf nodes

What is the difference between a (max) heap and a priority queue?A Max-Heap is a complete binary tree in which the value in each internal node is greater than or equal to the values in the children of that node. The Priority Queue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.