Solutions to Assignment 7

Verify validity of an AVL tree: Recursive method verify() returns a 4-tuple: a boolean indicating whether the tree has the structure and order of a valid AVL tree, its minimum element, its maximum element, and its height. If the tree is not a valid AVL tree, then the min and max elements, and the height are arbitrary.

```
verify(): // bottom up algorithm. RT = O(n)
   if size() = 0 then return true
   ( flag, min, max, height ) ← verify( root )
   return flag

(boolean, T, T, int) verify( ent ):
   cur ← ent.element
   lmin ← cur,   lh ← −1
   rmax ← cur,  rh ← −1
   if ent.left != null then
         ( flag, lmin, lmax, lh ) ← verify ( ent.left )
         if not flag or lmax >= cur then
            return ( false, lmin, lmax , 1+lh)
   if ent.right != null then
         ( flag, rmin, rmax, rh ) ← verify ( ent.right )
         if not flag or cur >= rmin then
            return ( false, lmin, rmax, 1+rh )
   if ent.height != 1 + max(lh, rh) or |lh-rh| > 1 then
         return ( false, lmin, rmax, ent.height )
   else
         return ( true, lmin, rmax, ent.height )
```

Solutions to Assignment 8

```
List<T> mostOften(T[ ] arr) {
   Map<T,Integer> map = new HashMap<>();
   List<T> result = new LinkedList<>();
   max = 0;
   for (e: arr) {
         c = map.get(e);
         c = c == null ? 1 :  c+1;
         map.put(e, c);
          if (c > max) max = c;
   }
   // Go through array again and create output
   for (e: arr) {
         if (map.get(e) == max) {
               result.add(e);
               map.put(e,0);
         }
   }
   return result;
}
```

Solutions to Assignment 9

Worst-fit heuristic for bin packing: use a priority queue for the bins, with priority equal to how much of the bin has been utilized by items. The algorithm places the next item in the bin which has been least used, if it fits, and otherwise starts a new bin for the item. We will assume that the size of every item is less than or equal to C.

```
int worstFit ( int[ ] size, int C ) {  // RT = O(n logn)
   if ( size.length == 0 ) return 0;
   Queue<Integer> q = new PriorityQueue<>();
   q.add(0);  // Add bin 1, its utilization = 0
   for ( int x: size ) {
         if ( q.peek() + x <= C )  // Item fits
               q.add ( q.remove() + x );
         else  // Start a new bin for x
               q.add ( x );
   }
   return q.size();
}
```

Best-fit heuristic: use a TreeSet, where the elements are bins ordered by residual capacities.

```
class Bin implements Comparable<Bin> {
   int name, cap;  // residual capacity
   Bin(int n, int C) { name = n;  cap = C; }
   void placeItem(int item) { cap = cap - item; }
   int compareTo(Bin other) {
         if (this.cap < other.cap) return -1;
         else if (this.cap > other.cap) return 1;
         else if (this.name < other.name) return -1;
         else if (this.name > other.name) return 1;
         else return 0;
   }
}
int bestFit ( int[ ] size, int C ) {  // RT = O(n logn)
   if ( size.length == 0 ) return 0;
   TreeSet<Bin> s = new TreeSet<>();
   s.add ( new Bin(1, C) );
   for ( int x: size ) {
         Bin b = s.ceiling( new Bin(0, x) );
         if ( b == null )   // start new bin
            s.add ( new Bin(1+s.size(), C−x ) );
          else {  // Place x in b
            s.remove(b);
            b.placeItem(x);
            s.add(b);
         }
   }
   return s.size();
} //Try solution with TreeMap<Integer,Integer>
```

Selection problem:

Input: Array $A[1..n]$, integer k ↳ unsorted — Internal version. n fits in memory.

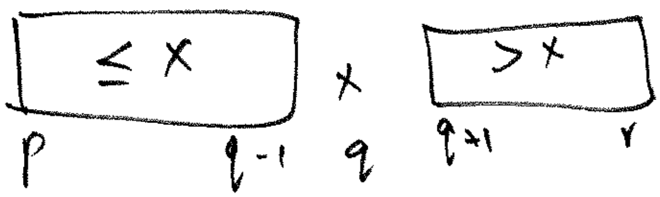
Output: k^{th} largest element of A, or k largest elements of A.

Of specific interest: Median: $k = \lceil n/2 \rceil$.

Naive algorithms: $O(n \log n)$ time.

Idea: Select(A, p, r, k): // index of k^{th} largest of $A[p..r]$

$q \leftarrow$ randomizedPartition(A, p, r)



$lsize \leftarrow q - p$ $\quad rsize \leftarrow r - q$

Case 1: if $rsize \geq k$ then
return Select$(A, q+1, r, k)$

Case 2: else if $rsize == k-1$ then
return q

Case 3: else return Select$(A, p, q-1, k-(rsize+1))$

RT = $O(n)$, expected time.
Analysis similar to QuickSort

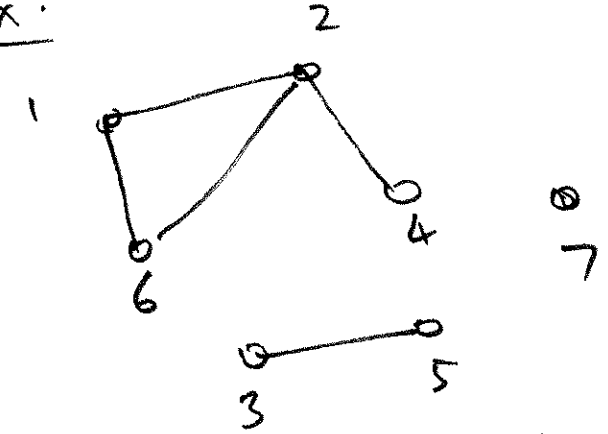
k largest: Select(A, k): $q \leftarrow$ Select$(A, 1, n, k)$
return $A[n-k+1 \dots n]$.

Graphs: $G = (V, E)$

V = Vertex set (nodes)

E = Edge set $E \subseteq V \times V$

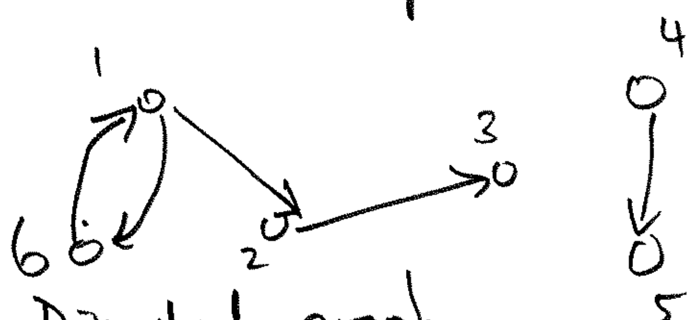
Ex:



$V = \{1, 2, 3, 4, 5, 6, 7\}$

$E = \{(1,2), (2,4), (6,2), (3,5), (1,6)\}$

Undirected graph. Edge $(3,5) \equiv (5,3)$.



Directed graph

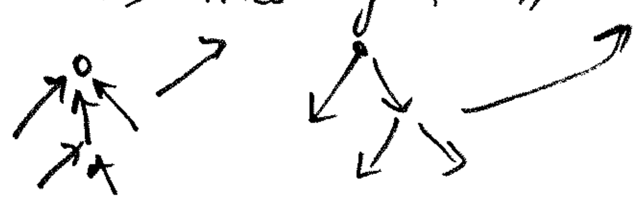
$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{(6,1), (1,6), (1,2), (2,3), (4,5)\}$

Subclasses of graphs

1. Tree = connected, undirected graph
2. Rooted tree, Arborescence, Branching = directed tree
   directed graph that is a tree.

   ↳ incoming trees, outgoing trees.



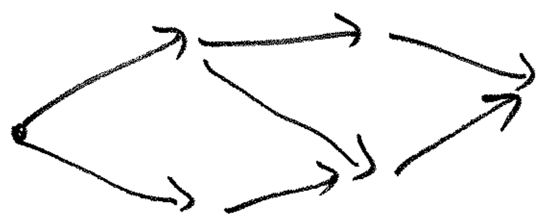
3. Simple graphs: (directed or undirected):

- no edges of the form (u, u). — self loop not allowed
- disallow more than one edge (u, v). — parallel edges not allowed.
- Default.

4. Directed, acylic graph (DAG)

- directed graph with no directed cycles.

<u>Ex:</u>



<u>Representation of graphs</u>

① Adjacency matrix: $V \times V$ matrix

- infeasible for moderate sized graphs.

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	0	0	1	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	1	0
5	0	0	0	0	0	0
6	1	0	0	0	0	0

② Adjacency list:

1	→ 2, 6
2	→ 3
3	→ ≡
4	→ 5
5	→ ≡
6	→ 1

→ G is stored as an array of lists.
For each $u \in V$, there is a list $\mathrm{adj}(u)$.

Finding the k largest elements of an unsorted array A of size n:
Naive algorithm 1: sort A and take $A[\, n-k \cdots n-1 \,]$. RT = O(nlog(n)).
Naive algorithm 2: insert $A[\, 0 \cdots n-1 \,]$ into a max heap (priority queue). Repeat k times: Delete max.

The following algorithm runs in expected O(n) time:

```
Select ( A, k ):  // Find the k largest elements of unsorted array A
   n ← A.length
   if k ≤ 0 then return empty list
   if k > n then return A
   Select(A, 0, n−1, k)
   return A[ n−k···n−1 ]   // Output is not in sorted order

Select( A, p, r, k ):   // Find kth largest element of A[ p..r ].  Precondition k ≤ n = r−p+1.
   if r−p+1 < T then
      insertionSort(A, p, r)
      return A[ r−k+1 ]
   else
      q ← randomizedPartition( p, r )
      left ← q−p
      right ← r−q
      if right ≥ k then     // kth largest element of A[ p..r ] is also kth largest of A[ q+1..r ]
         return Select( A, q+1, r, k )
      else if  right+1 = k then
         return A[ q ]       // Pivot element happens to be kth largest element
      else                   // kth largest in A[ p .. r ] is [k−(right+1)]th largest in A[ p..q−1 ]
         return Select( A, p, q−1, k−(right+1) )
```

The above algorithm is not suitable when A is a stream or an array stored on disk that is too big to be stored in memory. This version of the problem can be solved in O(n log k) time by using a priority queue:

```
Select( A, k ):   // Find the k largest elements of a stream A
   it ← A.iterator()
   q ← new Priority Queue (min heap)   // for storing the k largest elements seen
   for i ← 1 to k do
      if it.hasNext() then
         q.add( it.next() )
      else
         return q
   while it.hasNext() do
      x ← it.next()
      if q.peek() < x then    // This step is more efficiently done with our own heap as q.replace(x).
          q.remove()
          q.add( x )
   return q
```