Look up runtimes

- Sorts:
  - sort in $O(n+k)$ time (Feld)(Counting sort)
  - sort in $O(n)$ (List)
    sort alphabetical (bucketsort lexicograpical sort)
- Graphen:
  - reversen Graph (j,i to i,j)
  - Komplement des Graph (Adjazenzmatrix 1 to 0, 0 to 1)
  - 2014.5
  - Baum-, Rückwärts-, Vorwärts-, oder Querkante

1 Sorts

1.1 Heapsort

1.1.1 Aufbauphase

```
for i=⌊n/2⌋downto 1
  Sink(i,n)

```

1.1.2 Heapify

```
Sink(i,r){
  x = A[i];
  j = 2*i;
  while (j≤ r){
    if (j+1 ≤ r){
      if (A[j+1]>A[j]){
        j = j+1;
      }
    }
    if (x≥ A[j]){
      break;
    }
    A[i] = A[j];
    i = j;
    j = 2*i;
  }
  A[i] = x;
}
```

1.1.3 Selektionsphase

```
r = n
while (r > 1){
  A[1] <=> A[r];
  r = r-1;
  Heapify(1,r)
}
```

1.2 Quicksort

```
Quicksort(l,r){
  if (L≥r)
    return;
  x = A[l]      pivot
  i = l+1
  j = r
  repeat
    while i≤r && A[i]<x
      i++0
    while j≥l+1 && A[j]≥x
      j--
    if i<j
      A[i]<=>A[j]
  until i>j
  A[l]<=>A[j]
  Quicksort(l,j-1)
  Quicksort(j+1,r)
}
```

2 Bäume

2.1 Aufbau

```
BaueBaum(A,l,r){
  if (l>r)
    return 0;
  m = ⌊(l+r)/2⌋;
  p = new bintree_node();
  p.key = A[m];
  p.left = BaueBaum(l,m-1);
  p.right = BaueBaum(m+1,r);
  return p;
}
```

2.2 Lookup

```
Lookup(x)
  if (p = null)
    return null;
  while (p ist kein Blatt)
    if x

```

2.3 Insert

```
Insert(Node root, int key, int value){
  if (root = null)
    root = new Node(key, value);
  else if (key < root.key)
    root.left = insert(root.left, key, value);
  else  // key >= root.key
    root.right = insert(root.right, key, value);
  return root;
}
```

2.4 Delete

```
Delete(x){
  v = lookup(x);
  (Sei p vater von v)
  Fall 1: (v ist Blatt){
    if (v = p.left)
      p.left = null;
    else
      p.right = null;
  }
  Fall 2: (v hat ein Kind w){
    if (v = p.left)
      p.left = w;
    else
      p.right = w;
  }
  Fall 3: (v hat 2 Kinder)
    u = v.left;
    while (u.right!=null)
      u = u.right;
    v.key = u.key;
}
```

2.5 Print

```
Print(node n){
  if node.left != null
    print(node.left)
  schreibe(node.key)
  if node.right != null
    print(node.right)
}
```

2.6 Max/Min Key

```
Max(){
  u = rootNode
  while u.right != null
    u = u.right
  return u;
}


Min(){
  u = rootNode
  while u.left != null
    u = u.left
  return u;
}

```

2.7 Rotate

2.7.1 Single

```
RotateRight(node u){
  w = u.parent
  v = u.right
  b = v.left
  v.parent = w
  if w = null
    v = root
  else if u = w.left
    w.left = v
  else
    w.right = v
  u.parent = v
  v.left = u
  u.right = b
  if b!=null
    b.parent = u
}

```

2.7.2 Double

```
DoubleRR(node u){
  v = u.left
  rotateLeft(v)
  rotateRight(u)
}

DoubleRL(node u){
v = u.right
rotateRight(v)
rotateLeft(u)
}
```

2.8 Depth-First Search

```
DFS(G, v){
  v als beschriften
  for all adjacent nodes w to v do
    if node is not labeled discovered
      DFS(G,w)
}

```

2.9 Breadth-First Search

```
BFS(G,v){
  Q:= queue initialized with root
  while Q.notEmpty()
    current = Q.dequeue()
    if current = goal
      return current
    for each node n that is adjacent to current
      if n is not discovered
        label n as discovered
        n.parent = current
        Q.enqueue(n)
}
```

3 Graphen

4 Definitionen

4.1 Topologische Sortierung

Eine Abbildung $ord : V \to \{1, ..., n\}$
Es gilt zudem:

- ord ist injektiv
- $\forall (v, w) \in E : ord(v) < ord(w)$