Week 1

Arrays:

To insert a point at position `pos`, where $0 \leq pos \leq size$:

```
maxsize = 100
Point[] locations = new Point[maxsize];
int size = 0;     // number of points currently stored

void insert(int pos, Point pt) {
  if (size == maxsize) {
      throw new ArrayFullException("locations_array");
  }
  for (int i=size-1; i >= pos; i--) {
    // Copy entry in pos i one pos towards the end
    locations[i+1] = locations[i];
  }
  locations[pos] = pt;
  size++;
}
```

Linked Lists:

Node

```
class Node {
  int val;
  Node next;
}
Node list = END;
```

Inserting at beginning of linked list

```
void insert_beg(Node list, int number) {
  newNode = new Node();
  newNode.val = number
  newNode.next = list;
  list = newNode;
}
```

Deleting at the beginning

```
Boolean is_empty(Node list) {
  return (list == END);
}
```

```
void delete_begin(Node list) {
  if is_empty(list) {
    throw new EmptyListException("delete_begin");
  }
  list = list.next;
}
```

Linked List Lookup

```
int value_at(Node list, int index) {
  int i = 0;
  Node nextnode = list;
  while (true) {
    if (nextnode == END) {
      throw new OutOfBoundsException();
    }
    if (i == index) {
      break;
    }
    nextnode = nextnode.next;
    i++;
  }
  return nextnode.val;
}
```

Insert at the End

```
void insert_end(Node list, int number) {
  newblock = new Node();
  newblock.val = number;
  newblock.next = END;
  if (list == END) {
    list == newblock;
  }
  else
  {
    cursor = list;
    while (cursor.next != END){
      cursor = cursor.next;
    }
    cursor.next = newblock;
  }
}
```

Circular Queue – Array Implementation:

```
// Initialize empty queue:
queue = new int[MAXQUEUE];
front  = 0;
size = 0;
```

```
boolean is_empty () {
  return size == 0;
}
```

```
boolean is_full () {
  return size == MAXQUEUE;
}
```

```
void enqueue (int val) {
  if (size == MAXQUEUE) { throw QueueFullException; }
  queue[(front+size) mod MAXQUEUE] = val;
  size ++;
}
```

```
int dequeue () {
  int val;
  if (size == 0) { throw QueueEmptyException; }
  val = queue[front];
  front = (front+1) mod MAXQUEUE
  size --;
  return val;
}
```

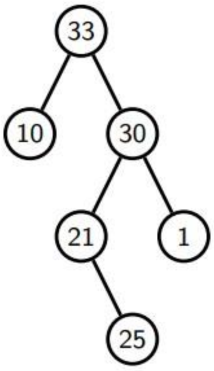
Trees:

ADT

- Constructors:
  - `EmptyTree` : returns an empty tree
  - `MakeTree(v, l, r)` : returns a new tree where the root node has value `v`, left subtree `l` and right subtree `r`
- Accessors:
  - `isEmpty(t)` : return true if `t` is the empty tree, otherwise returns false
  - `root(t)` : returns the value of the root node of the tree `t`[1]
  - `left(t)` : returns the left subtree of the tree `t`[2]
  - `right(t)` : returns the right subtree of the tree `t`[2]
- Convenience Constructor:
  - `Leaf(v)` = `MakeTree(v, EmptyTree, EmptyTree)`

Construction

```
    MakeTree(33,
             Leaf(10),
             MakeTree(30,
                      MakeTree(21,
                               EmptyTree,
                               Leaf(25)),
                      Leaf(1)))
```



Reversing a Tree

```
reverseTree(t) {
  if ( isEmpty(t) )
     return (t)
  else
    return (MakeTree(root(t),
                reverseTree(right(t)),
                reverseTree(left(t))))
```

Flatten a Tree into a List

```
flatten(t) {
  if Tree.isEmpty(t)
    return EmptyList
  else
   return append(flatten(left(t)),
                 MakeList(root(t), flatten(right(t)))))
}
```

Quad Tree:

ADT

- Constructors:
  - `baseQT` : returns a single, leaf node quad tree with a value
  - `MakeQT(luqt, ruqt, llqt, rlqt)` : returns a new quad tree built from four sub-quad trees.
- Accessors:
  - `isValue(qt)` : return true if `qt` is a value node quad tree, otherwise returns false
  - `lu(qt)` : returns the left upper sub-quad tree of `qt` [2]
  - `ru(qt)` : returns the right upper sub-quad tree of `qt` [2]
  - `ll(qt)` : returns the left lower sub-quad tree of `qt` [2]
  - `rl(qt)` : returns the right lower sub-quad tree of `qt` [2]

Rotation

```
rotate(qt) {
  if ( isValue(qt) )
    return qt
  else
    return makeQT( rotate(rl(qt)),
                   rotate(ll(qt)),
                   rotate(ru(qt)),
                   rotate(lu(qt)) )
```

Binary Search Trees:

Insertion

```
insert(v, bst) {
  if ( isEmpty(bst) )
    return MakeTree(v, EmptyTree, EmptyTree)
  elseif ( v < root(bst) )
    return MakeTree(root(bst),
                    insert(v, left(bst)),
                    right(bst))
  elseif ( v > root(bst) )
    return MakeTree(root(bst),
                     left(bst),
                     insert(v, right(bst)))
  else error("Error:␣value␣already␣in␣tree")
}
```

Insertion in Java

```java
public class BSTTree {
    private BSTNode tree = null ;

    priavte static class Node {
        private int val;
        private Node left, right;

        public BSTNode(int val, Node left, Node right){
            this.val=val, this.left=left, this.right=right;
    }
    public void insert(int v){
        if (tree == null) tree = new  Node(v, null, null);
        else insert (v, tree)
        }
    private void insert(int v, Node ptr){
        if (v < ptr.val){
            if(ptr.left == null)
                ptr.left = new Node(v, null, null);
            else insert(v, ptr.left);
        }
        else if (v > ptr.val){
            if (ptr.right == null)
                ptr.right = New node(v, null, null);
            else insert (v, ptr.right);
        }
        else throw new Error("Value already in tree.")
    }
}
```

Searching BSTs

Recursively

```
isIn(value v, tree t) {
    if ( isEmpty(t) )
        return false
    elseif ( v == root(t) )
        return true
    elseif ( v < root(t) )
        return isIn(v, left(t))
    else
        return isIn(v, right(t))
}
```

Iteratively

```
isIn(value v, tree t) {
    while ( (not isEmpty(t)) and (v != root(t)) )
        if (v < root(t) )
            t = left(t)
        else
            t = right(t)
    return ( not isEmpty(t) )
}
```

Sorting using BSTs

```
printInOrder(tree t) {
  if ( not isEmpty(t) ) {
    printInOrder(left(t))
    print(root(t))
    printInOrder(right(t))
  }
}

sort(array a of size n) {
  t = EmptyTree
  for i = 0,1,...,n-1
    t = insert(a[i],t)
  printInOrder(t)
}
```

BST Check

```
isbst(tree t) {
   if ( isEmpty(t) )
     return true
   else
     return ( allsmaller(left(t),root(t)) and
              isbst(left(t)) and
              allbigger(right(t),root(t)) and
              isbst(right(t)) )
}
allsmaller(tree t, value v) {
   if ( isEmpty(t) )
      return true
   else
      return ( (root(t) < v) and
               allsmaller(left(t),v) and
               allsmaller(right(t),v) )
}

allbigger(tree t, value v) {
  if ( isEmpty(t) )
    return true
  else
    return ( (root(t) > v) and
             allbigger(left(t),v) and
             allbigger(right(t),v) )
}
```

Deleting from a BST in Java

```
Java
delete(value v, tree t){
    if ( isEmpty(t))
        error("Error: given item is not in the tree")
    else
        if ( v < root(t))
            return MakeTree(root(t), delete(v, left(t)), right(t))
        else if ( v > root(t))
            return MakeTree(root(t), left(t), delete(v, right(t)))
        else
            if ( isEmpty(left(t)) )
                return right(t)
            elseif ( isEmpty(right(t)) )
                return left(t)
            else return
                MakeTree(smallestNode(right(t)), left(t),
                removeSmallestNode(right(t)))
}
smallestNode(tree t){
    if ( isEmpty(left(t)))
        return root(t)
    else
        return smallestNode(left(t))
}
removeSmallestNode(tree t){
    if ( isEmpty(left(t)))
        return root(t)
    else
        return MakeTree(root(t), removeSmallestNode(left(t)), right(t))
}
```

Binary Heaps:

Heap Tree ADT

```java
public class PriorityHeap{
  private int MAX = 100;
  private int heap[MAX+1];
  private int n = 0;

  public int value(int i){
    if (i < 1 or i > n)
     throw IndexOutOfBoundsException;
    return heap[i];
  }
  public boolean isRoot(int i) { return i == 1; }
  public int level(int i)      { return log(i); }
  public int parent(int i)     { return i / 2; }
  public int left(int i)       { return 2 * i; }
  public int right(int i)      { return 2 * i + 1; }
  // More methods to be added here
}
```

```java
  public boolean isEmpty() {
    return n == 0;
  }

  public int root() {
    if ( heapEmpty() )
       throw HeapEmptyException;
     else return heap[1]
  }

  public int lastLeaf()) {
     if ( heapEmpty() )
       throw HeapEmptyException;
     else return heap[n]
  }
```

Update

```java
public void update(int i, int priority) {
  if (n < 1)
    throw EmptyHeapException;
  if (i < 1 or i > n)
    throw IndexOutOfBoundsException;

  heap[i] = priority;
  bubbleUp(i)
  bubbleDown(i);
}
```

Insert

```java
public void insert(int p) {
  if (n == MAXSIZE)
    throw HeapFullException;
  n = n + 1;
  heap[n] = p;  // insert the new value as the last
                // node of the last level
  bubbleUp(n);  // and bubble it up
}
```

```java
private void bubbleUp(int i) {
  if (i == 1) return;  // i is the root

  if (heap[i] > heap[parent(i)]) {
    swap heap[i] and heap[parent(i)];
    bubbleUp(parent(i));
  }
}
```

Delete

```java
public void delete(int i) {
  if (n < 1)
    throw EmptyHeapException;
  if (i < 1 or i > n)
    throw IndexOutOfBoundsException;

  heap[i] = heap[n];
  n = n-1;
  bubbleUp(i)
  bubbleDown(i);
}
```

```java
private void bubbleDown(int i) {
   if ( left(i) > n )                  // no children
     return;
   else if ( right(i) > n )            // only left child
     if ( heap[i] < heap[left(i)] )
       swap heap[i] and heap[left(i)]
   else                                // two children
     if ( heap[left(i)] > heap[right(i)] and
          heap[i] < heap[left(i)] ) {
       swap heap[i] and heap[left(i)]
       bubbleDown(left(i),heap,n)
     }
   else if ( heap[i] < heap[right(i)] ) {
       swap heap[i] and heap[right(i)]
       bubbleDown(right(i),heap,n)
     }
   }
}
```

Heapify

```java
public void heapify() {
   for( int i = n/2 ; i > 0 ; i— )
      bubbleDown(i)
}
```

Sorting:

Bubble Sort in Java

```
bubblesort(a, n){
    for(i=1 ; i<n ; i++){
        for(j=n-1 ; j>= i ; j--){
            if(a[j] < a[j-1]){
                swap a[j] and a[j-1]
}}}}
```

Insertion Sort

Pseudocode

```
insertionsort(a, n) {
  for ( i = 1 ; i < n ; i++ ) {
     j = i
     t = a[j]
     while ( j > 0 && t < a[j-1] ) {
        a[j] = a[j-1]
        j--
     }
     a[j] = t
  }
}
```

Example

1. 5 | 12 , 6, 3, 11, 8, 4
2. 5, 12 | 6 , 3, 11, 8, 4
3. 5, 6, 12, | 3 , 11, 8, 4
4. 3, 5, 6, 12 | 11 , 8, 4
5. 3, 5, 6, 11, 12 | 8 , 4
6. 3, 5, 6, 8, 11, 12 | 4
7. 3, 4, 5, 6, 8, 11, 12 |

Selection Sort

Psuedocode

```
selectionsort(a, n){
   for ( i = 0 ; i < n-1 ; i++ ) {
      k = i
      for ( j = i+1 ; j < n ; j++ )
         if ( a[j] < a[k] )
            k = j
      swap a[i] and a[k]
   }
}
```

Example

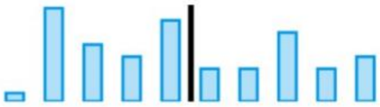
1. | 5, 12, 6, 3 , 11, 8, 4
2. 3 | 12, 6, 5, 11, 8, 4
3. 3, 4 | 6, 5 , 11, 8, 12
4. 3, 4, 5 | 6 , 11, 8, 12
5. 3, 4, 5, 6 | 11, 8 , 12
6. 3, 4, 5, 6, 8 | 11 , 12
7. 3, 4, 5, 6, 8, 11 | 12
8. 3, 4, 5, 6, 8, 11, 12 |

Heap Sort

```
heapSort(array a, int n) {
   heapify(a,n)
   for( j = n ; j > 1 ; j-- ) {
      swap a[1] and a[j]
      bubbleDown(1,a,j-1)
   }
}
```

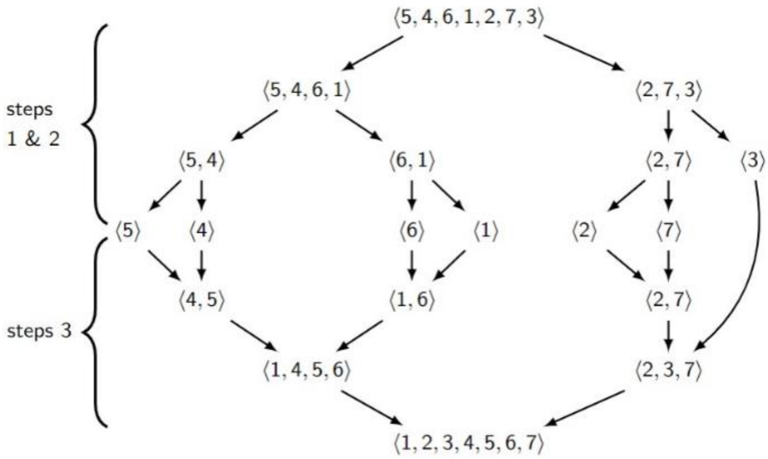
Merge Sort

1. Split the array into two halves:



2. Sort each of them recursively:

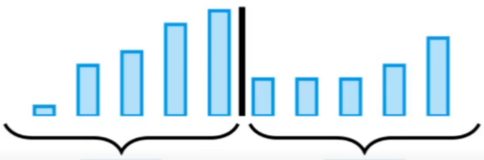
3. Merge the sorted parts:



Idea: In variables `i` and `j` we store the current positions in `a[-]` and `b[-]`, respectively (starting from `i=0` and `j=0`). Then:

1. Allocate a *temporary* array `tmp[-]`, for the result.
2. If `a[i] <= b[j]` then copy `a[i]` to `tmp[i+j]` and `i++`,
3. Otherwise, copy `b[j]` to `tmp[i+j]` and `j++`.

Repeat 2./3. until `i` or `j` reaches the end of `a[-]` or `b[-]`, respectively, and then copy the rest from the other array.




```
mergesort(a, n) {
    mergesort_run(a, 0, n−1)
}

void mergesort_run(a, left, right) {
    if (left < right){
      mid = (left + right) div 2

      mergesort_run(a, left, mid)
      mergesort_run(a, mid+1, right)

      merge(a, left, mid, right)
    }
}
```

```
merge(array a, int left, int mid, int right) {
   create new array b of size right−left+1
   bcount = 0
   lcount = left
   rcount = mid+1
   while ( (lcount <= mid) and (rcount <= right) ) {
      if ( a[lcount] <= a[rcount] )
         b[bcount++] = a[lcount++]
      else
         b[bcount++] = a[rcount++]
   }
   if ( lcount > mid )
      while ( rcount <= right )
         b[bcount++] = a[rcount++]
   else
      while ( lcount <= mid )
         b[bcount++] = a[lcount++]
   for ( bcount = 0 ; bcount < right−left+1 ; bcount++ )
      a[left+bcount] = b[bcount]
}
```

Quick Sort

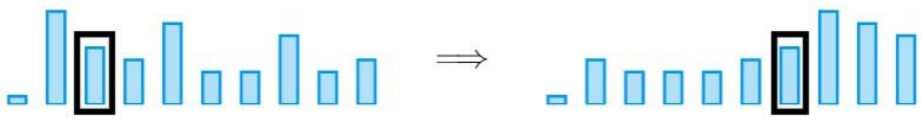
```
void quicksort(a, n){
    quicksort_run(a, 0, n-1)
}

quicksort_run(a, left, right) {
  if ( left < right ) {
    pivotindex = partition(a, left, right)
    quicksort_run(a, left, pivotindex-1)
    quicksort_run(a, pivotindex+1, right)
  }
}
```

Where `partition` rearranges the array so that

- the small entries are stored on positions `left, left+1, left+2, ..., pivot_index-1`,
- pivot is stored on position `pivot_index` and
- the large entries are stored on `pivot_index+1, pivot_index+2, ..., right`.

1. Choose a pivot `p` from `a`.
2. Allocate two temporary arrays: `tmpLE` and `tmpG`.
3. Store all elements *less than or equal to* `p` to `tmpLE`.
4. Store all elements *greater than* `p` to `tmpG`.
5. Copy the arrays `tmpLE` and `tmpG` back to `a` and return the index of `p` in `a`.

The time complexity of partitioning is $O(n)$.



Partitioning array `a` in-place (unstable)

```
partition(array a, int left, int right) {
  pivotindex = choosePivot(a, left, right)
  pivot = a[pivotindex]
  swap a[pivotindex] and a[right]
  leftmark = left
  rightmark = right - 1
  while (leftmark <= rightmark) {
    while (leftmark <= rightmark and
           a[leftmark] <= pivot)
      leftmark++
    while (leftmark <= rightmark and
           a[rightmark] >= pivot)
      rightmark--
    if (leftmark < rightmark)
      swap a[leftmark++] and a[rightmark--]
  }
  swap a[leftmark] and a[right]
  return leftmark
}
```

Partitioning array `a`, using temporary storage (stable)

```
partition(array a, int left, int right) {
  create new array b of size right-left+1
  pivotindex = choosePivot(a, left, right)
  pivot = a[pivotindex]
  acount = left
  bcount = 1
  for ( i = left ; i <= right ; i++ ) {
    if ( i == pivotindex )
      b[0] = a[i]
    else if ( a[i] < pivot ||
              (a[i] == pivot && i < pivotindex) )
      a[acount++] = a[i]
    else
      b[bcount++] = a[i]
  }
  for ( i = 0 ; i < bcount ; i++ )
    a[acount++] = b[i]
  return right-bcount+1
}
```

Pigeonhole Sort

```
pigeonhole_sort(a, n){
    create array b of size n
    for (i=0; i<n; i++)
        b[a[i]] = a[i]
    copy array b into array a
}
```

We can alternatively pigeonhole sort in-place.

```
pigeonhole_sort_inplace(a, n){
    for (i=0; i<n; i++)
        while( a[i] != i)
            swap a[a[i]] and a[i]
}
```

Graph / Pathing Algorithms:

Dijkstra's algorithm (pseudocode with adjacency matrix)

```
dijkstra_with_matrix(int[][] G, int v, int z) {
    n = G.length;
    d = new int[n]; p = new int[n]; f = new bool[n];

    for (int w = 0; w < n; w++) {
        d[w] = infty;   p[w] = w;   f[w] = false;
    }
    d[v] = 0;

    while (true) {
        w = min_unfinished(d, f);
        if (w == -1)
            break;

        for (int u = 0; u < n; u++)
            update(w, u, d, p);

        f[w] = true;
    }
    // compute results in desired form
    return compute_result(v, z, G, d, p);
}
```

```
int min_unfinished(int[] d, bool[] f) {
    int min = infty;
    int idx = -1;

    for (int i=0; i < d.length; i++) {
        if ( (not f[i]) && d[i] < min) {
            idx = i;
            min = d[i]
        }
    }

    return idx;
}
```

```
void update(w, u, G, d, p) {
    if (d[w] + G[w][u] < d[u]) {
        d[u] = d[w] + G[w][u];
        p[u] = w;
    }
}
```

Dijkstra's algorithm (pseudocode with adjacency lists)

```
dijkstra_with_lists(List<Edge>[] N, int v, int z) {
    n = G.length;
    d = new int[n];     p = new int[n];
    Q = new MinPriorityQueue();

    for (int w = 0; w < n; w++) {
        d[w] = infty;     p[w] = w;
        Q.add(w, d[w]);
    }
    d[v] = 0;
    Q.update(v, 0);

    while (Q.notEmpty()) {
        w = Q.deleteMin()

        for (Edge e : N[w]) { // iterate over edges to neighbours
            u = e.target;
            if (d[w] + e.weight < d[u]) { // should we update?
                d[u] = d[w] + e.weight;
                p[u] = w;
                Q.update(u, d[u]);
            }
        }
    }
    return compute_result(v, z, G, d, p);
}
```

```
class Edge {
  // target node
  int target;

  int weight;
}
```

Kruskal's Algorithm

```
let result be a new empty list of edges
for each node n in G
  makeSet(n)
let E be a list of edges in G sorted by increasing weights
for each edge e = (u,v) in E in order
{
  if (find(u) != find(v))
  {
    result.add(e)
    union(find(u), find(v))
  }
}
return result
```

Union Find (Based on Size)

```
void makeSet(int v) {
    parent[v] = v;
    size[v] = 1;
}

int find(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find(parent[v]);
}

void union(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b]
    }
}
```

Union Find (Based on Rank)

```
void makeSet(int v) {
    parent[v] = v;
    rank[v] = 0;
}

int find(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find(parent[v]);
}

void union(int a, int b) {
    a = find(a);
    b = find(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++
    }
}
```