KDTree library class configurability:

- Number of points in each node (`'nodeSize'`)
- Number of levels of neighbor-nodes to explore(`'adjacentLevels'`)
- Function pointer to decide on which dimension to divide dataset (`'dimensionSelFn'`)

Additionally 'KDTree' has following:

- '*numDimension*' : number of dimensions
- `'dataPoints'`: All data points as vector of `'Point'` class objects.

Features supported:

1. KDTree construction using two options
2. KDTree destruction using 'destroy' method. Destructor also calls 'destroy'.
3. By default, dimension is chosen as per the given specification. But, there is an option to extend this by providing function pointer used to select dimension
4. Novel (to the best of my knowledge) and efficient approach for finding nearest neighbor
5. KDTree is templated class
6. Generated tree can be stored to disk and loaded from disk

KDTree instance generation:

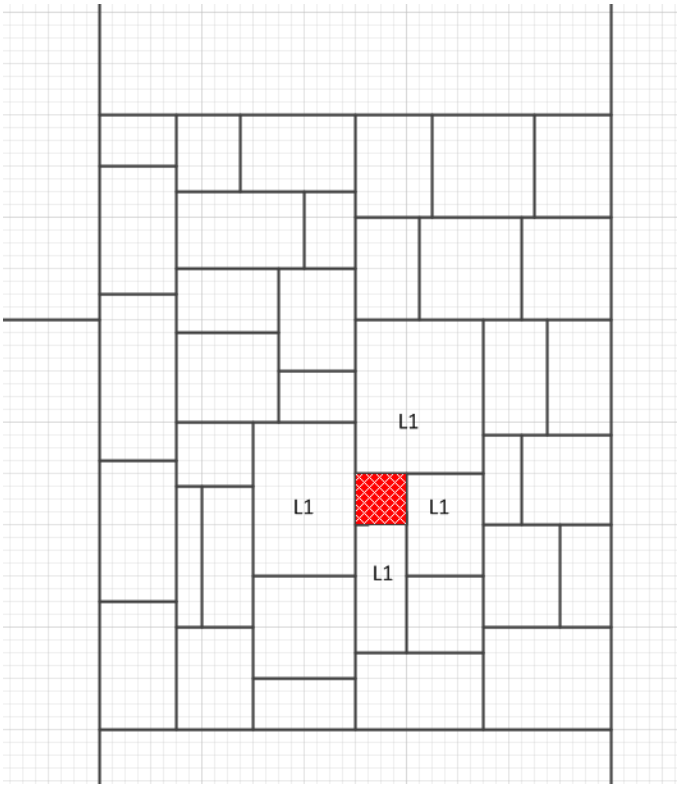
1. Option-1: Build using constructor, which takes k-dimensional point vector, number of nodes in each node, number of levels and function pointer.
2. Option-2: Build using empty constructor and then call 'build' method using above four arguments

Fourth argument (function pointer) is optional. By default, KDTree will divide data set based on maximum range dimension as given in specification.

Efficient Nearest Neighbor Search Algorithm:

- Find leaf node using traditional binary search tree traversal based on median value of selected dimension at each non-leaf node
- Once leaf node is located, find nearest point in that leaf node.
- '*adjacentLevels*'specifies how many levels of neighbor nodes to explore. '*adjacentLevels'==1* means first immediate neighbor nodes. '*adjacentLevels'==2* means first level as well as neighbor of neighbor at 2nd level.
- If 'adjacentLevels' is more than 0, then that many levels of neighbor nodes are explored.
- Example:
    - As shown in following figure of 2DTree example, query point is found to lie in red-square.
    - That node is inserted into 'set'.

- In present example '*adjacentLevels*' is 1. So, all nodes marked in 'L1' (level-1) are inserted to set.
- All nodes in 'set' are searched for finding 'nearest' neighbor.
- If '*adjacentLevels*' was greater than 1, then next level of neighbors will be added using 'Breadth First Search' traversal



To support above algorithm each node ('KDTreeNode') has following:

- '*numDimension*' : number of dimensions
- '*dimBnd*': Vector ('`DimensionBondaryVec<T>`') of pairs for each dimension. Each 'pair' represent min and max of dimension
- '`adjNodes`': Set of nodes who are adjacent to current node. This set contains first level of neighbors. Next levels of neighbors can be found recursively on the fly. This will empty for non-leaf node.

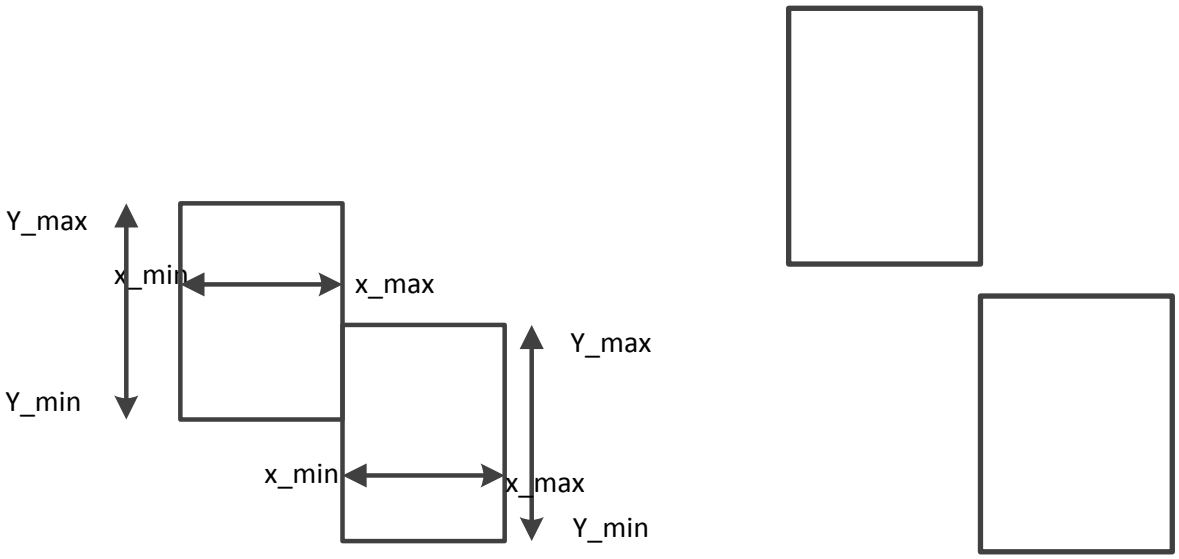
Tree Build Algorithm:

- At every traversal, 'buildRecursive' method checks if remaining number of data point are greater than nodeSize or not. If it is less than or equal to nodeSize, then leaf node is created and datapoints range is marked using '`pointIndexRange`.
- If current node is bigger than nodeSize, dimension is selected using pointer function ('`dimensionSelFn`') provided. Also, '`adjNodes`' is updated for child nodes.
- All datapoints are sorted on selected dimension and then median point is used to divide current node.

- Once left and right nodes are generated, they are assigned to each other as ‘adjacent’ nodes. This way ‘adjacency’ information is generated on the fly.
- One caveat in this approach is once tree is generated, node might have some adjacent nodes which are non-leaf nodes.
- So, after ‘tree’ is generated all leaf nodes’ adjacent nodes are checked. If adjacent node is non-leaf then that non-leaf node is recursively checked to find any leaf node that might be adjacent to current leaf node.
- Another alternative was to get the list of all leaf nodes in tree. And, for each leaf node find all nodes that are adjacent. This will yield O(n2) time complexity where n is number of nodes in tree. While current approach of on the fly generation yields O(a*n*k) time complexity where a<n (‘a’ is the number non-leaf adjacent nodes and ‘k’ is number of dimensions).

How to decide if nodes are adjacent or not:

- Nodes are adjacent if they touch each other in one dimension **and** intersect in every other dimension
- Touch condition: When one dimension’s min or max equals min or max of another node
- Intersect condition: when min or max of one node falls into range of [min, max] of another node
- First figure on left shows two nodes are adjacent to each other while figure on right shows non-adjacent nodes
    - First node’s x_max equals second node’s x_min **and**
    - First node’s [y_min, y_max] range intersects with second node’s [y_min, y_max]
- This node adjacency condition is scalable to higher dimensional data as well.
- For K-dimnsional nodes, it will yield O(k) time complexity. Because, only one of the dimension can satisfy ‘touch’ condition. And if ‘touch’ condition is satisfied, all other dimensions are checked for ‘intersect’ condition.



Time/Space Complexity Analysis:

1. Tree building:
    a. Tree is created using recursively creating nodes. So, time complexity will be O(m) and space complexity will be O(log m). Here 'm' is number of nodes created.
    b. Each leaf node has k-dimensional vector of pairs representing [min, max] range of each dimension. So, that will add O(k) space complexity
    c. Each leaf node also keep set of neighbor nodes pointers, so it will add O(a) space complexity, where 'a' is number of adjacent nodes.
    d. Once tree is built, each leaf node's `'adjNodes'` is checked for non-leaf node and each of this non-leaf node is searched for leaf nodes which are adjacent to current leaf node. So, this will add O(a*n*k) time complexity as mentioned earlier. Here, n is total leaf nodes and 'a' is number of non-leaf node which are adjacent to current node. Once actual leaf nodes are found, non-leaf adjacent nodes are removed from 'set'.

    Total time complexity = O(m) + O(a*n*k) = O(a*n*k) // 'a' is number of adjacent nodes and 'n' is total nodes and 'k' is number of dimensions

2. Nearest Neighbor Search
    a. First leaf node containing query point is searched. For this, time complexity will be O(log n) where 'n' is total number of nodes.
    b. Depending upon 'adjcentLevels' all adjacent nodes are gathered into set. This time complexity will be O('a'), where 'a' is total adjacent nodes
    c. For each node in this 'set' nearest point is searched. This time complexity will be O(a * p), where 'p' is nodeSize configured (number of points in each node)

    Total time complexity = O(log n) + O(a * p) = O(a * p) // where 'a' is total adjacent nodes and 'p' is number of points in each node

How to extend to support different ways of choosing the splitting axis:

- As mentioned earlier, this can be achieved by supplying optional fourth argument to constructor/'build'.
- This supplied function must have four arguments in following order:
    - Reference to vector of 'Point' representing whole training data set.
    - 'start' index in above vector
    - 'end' index in above vector
    - Number of dimensions
- Intension is this function will decide which dimension is selected to split using points which are in between 'start' and 'end' indices.
- Example:
    - By default, 'getCutDimension' function from 'KDTreeExtended.h' is used for splting axis.
    - 'KDTreeExtended.h' file also has getCutDimensionBad function.
    - This function is passed as fourth argument when constructing as shown in 'KDTreeApplication' class commented line.

KDTree library interface:

1. ‘build’ method: To construct tree (internally constructor call this method)
2. ‘destroy’ method: to destruct tree (internally destructor call this method)
3. ‘getNearestNeighbor’ method: To search for nearest neighbor. It returns a ‘pair’ of index and distance.
4. ‘saveToDisk’ method: To save tree to disk
5. ‘loadFromDisk’ static method: To load stored tree