Author: Chaviaras Michalis

This is a computational geometry library and the reason for developing such a library is to implement what I am studying about computational geometry. From this development I want to face programming decision problems, to learn how one can maintain a library and extend its capabilities. Of course this maintenance could exists with good documentation, so the present pdf exists for this reason.

With the time complexity we mean the asymptotic time complexity of a function or a block. With the space complexity we mean the asymptotic space complexity of the memory that is allocated dynamically or not in a function or a block, i.e all the space that is allocated.

# Contents

# Chapter 1

# Source

## 1.1 Point2d class

Purpose: It is a class that was created for trivial reasons to represent a point in the plane.
**Private Members:**

- `double myx;`

  The x coordinate of the point

- `double myy;`

  The y coordinate of the point

- `static double TOL_OF_EQUALITY_OF_POINTS2D = 1e-9;`

  If two points are distant less than this tolerance then these points are considered equal. This constant can change during the running of the program. But its usage is dubious because there is no clear argument about whether to use it. Currently I don't use it.

**Public Members:**

- `Point2d();`

  The reason for the existence of this default constructor is referred in the dicussion 1. You can find it in the `programming_decisions.pdf`

- `Point2d(const Point2d& otherPoint);`

  The classic copy constructor.

- `Point2d(const double x, const double y);`

  If you want to make a Point2d you have to provide the two coordinates, otherwise you can't make it.

- `double GetX() const;`

  If you want to get access of the x coordinate. Notice that you can't change it.

- `double GetY() const;`

  If you want to get access of the y coordinate. Notice that you can't change it.

- `bool operator==(const Point2d &other_point) const;`

  Overloaded operator of the class. It returns true if two points are equal, otherwise returns false. The equality of two points holds if the following holds :
  `this->myx == other.myx && this->myy == other.myy`

- `static double GetToleranceOfEquality() const;`

  Returns the above constant if you want to know how much is it.

- `static void SetToleranceOfEquality(double new_tol);`

  With this you can change the tolerance.

- `Point2d operator+() const;`

  Is the classic unary + operator, returns a Point2d with the same sign in its coordinates of the *(this)

- `Point2d operator-() const;`

  Is the classic unary - operator, returns a Point2d with the opposite sign in its coordinates of the *(this)

- `Point2d operator+(const Point2d& p1) const;`

  Is the binary addition operator, returns a Point2d which is the addition of *(this) and p1, i.e *(this)+p1.

- `Point2d operator-(const Point2d& p1) const;`

  Is the binary subtraction operator, returns a Point2d which is the addition of *(this) and p1, i.e *(this)-p1.

- `static double Distanceof2dPoints(const Point2d p1, const Point2d p2);`

  Returns the distance between the points p1 and p2. The distance is calculating based on the classic euclidean distance.

- `friend std::ostream& operator<<(std::ostream& output,const Point2d& z);`

  It is the classic operator that allows us to print a `Point2d` with the << operator.

## 1.2 Edge2d class

Purpose: To provide a class that represents an edge constitute of 2d points i.e the class Point2d.
**Private Members:**

- `std::vector<Point2d> mypoints;`

  A vector contains the two points(Point2d) of the edge. It is important I think to remain a vector because if I have pointer to which I had to allocate memory then I would have problem to return the two points because I had to return the pointer and I don't want to return the pointer. If I return the pointer then I will have problem that one can change the content of the points.

**Public Members:**

- `Edge2d(const Edge2d& other_edge);`

  It is the classic copy constructor.

- `Edge2d(const Point2d& p1, const Point2d& p2);`

  If you want to make an edge you have to provide two points, otherwise you can't.

- `std::vector<Point2d> GetPoints() const;`

  Returns a copy of the vector of the edge that contains the points, i.e a copy of mypoints. Notice the const;

- `bool IsNeighbour(const Edge2d& other_edge);`

  **O(1)**. Returns true if *(this) edge is neighbouring with `other_edge`, false otherwise. What does neighbouring mean? It means that one point from *this edge and one point from `other_edge` must be equal. The meaning of equality that operator== (of the class Point2d) provides.

- `friend std::ostream& operator<<(std::ostream& output,const Edge2d& z);`

  It is the operator that allows us to print the `Edge2d` with the operator <<.

## 1.3 Pred class

Purpose: To implement basic predicates such as the orientation predicate and others.
**Public Members:**

- `static double Orient(const Point2d& p_1, const Point2d& p_2, ...`

  `const Point2d& p_3);`

  **O(1)**. It is the classic orientation predicate. There are two vectors one that starts at the point `p_2` and ends at point `p_1` and the other vector starts at point `p_2` and ends at point `p_3`. So it returns $> 0$ if we have an anti clockwise rotation from the first vector to the second, $< 0$ if we have a clockwise rotation, $= 0$ if the vectors are collinear.

- `static bool Edg_Inters(const Edge2d& ed_1, const Edge2d& ed_2);`

  **O(1)**. It returns true if the `ed_1` and `ed_2` are intersecting and false otherwise.

- `static bool Edg_Inters(const Point2d& p_1, const Point2d& p_2, ...`

  `const Point2d& p_3, const Point2d& p_4);`

  **O(1)**. It returns true if the first edge and the second intersect. The first edge consists of the `p_1` and `p_2` point and the second edge of `p_3` and `p_4`.

- `static bool Edg_Inters(const double p1_x, const double p1_y, ...`

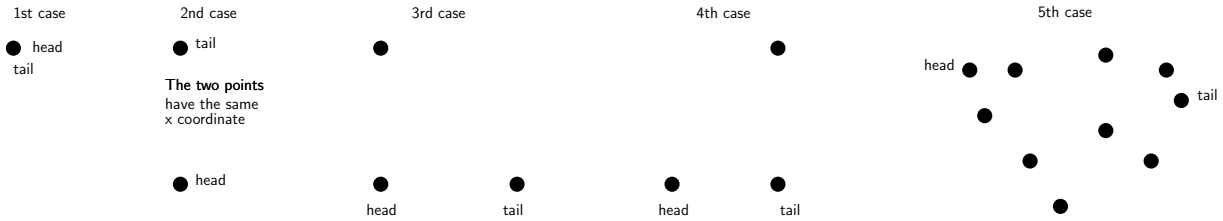
  `const double p2_x, const double p2_y, const double p3_x, ...`

  `const double p3_y, const double p4_x, const double p4_y);`

  **O(1)**. It returns true if the first edge and the second intersect. `p1_x` is the x-coordinate of the first point, `p_1y` is the y-coordinate of the first point ... and so on. So the first and the second point make the first edge.

## 1.4 CH2d_dlclist class

This class was made to represent a convex hull of 2d point on the plane. It implements a doubly linked cyclic list. The head is pointed to the `Point2d` with the smallest coordinate `x` and if there are more than one then the head is pointed to the one with the smallest `y` coordinate(note that you can have maximum 2 points with the same `x` coordinate in a convex hull(2d)). The tail is pointed to the `Point2d` with the biggest `x` and if there are more than one then the tail is pointed to the one with the smallest `y` coordinate. I defined the head and the tail like this because it helps me algorithmically and because with this way the points that correspond to head and to the tail will be surely in the convex hull even if the convex hull haven't constructed yet. Below you can see some cases of where the head and the tail are pointed to.




The usage of the head and tail is that, if you go from head to tail clockwise then you get the upper hull of the convex hull. If you go from tail to head clockwise then you get the lower hull. We have to define what we mean by the size of an object of the "`CH2d_dlclist`"class. That is the number of the vertices that contained in the convex hull(2d), i.e the number of the elements inside the d(oubly)l(inked)c(yclic) list.

**Members:**

- ```
  struct Node{
  Point2d data;
  Node* back;
  Node* front;
  };
  ```

  This is a classic node of double linked cyclic list.

The "back"and "front"have also another meaning beyond the iteration within the doubly linked(and cyclic) list. It has the meaning of the clockwise and counter clockwise iteration. So if we iterate "front"we are iterating the convex hull(2d) clockwise and with "back"we are iterating counter clockwise.

**Private Members:**

- `Node* head;`

  A pointer which points to a node that has as data point the head as we defined it before.

- `Node* tail;`

  A pointer which points to a node that has as data point the tail as we defined it before.
  ```

- `unsigned int my_size;`

  It holds the size of the list, i.e the number of the elements in the list. Of course it is also the number of the points in the convex hull(2d).

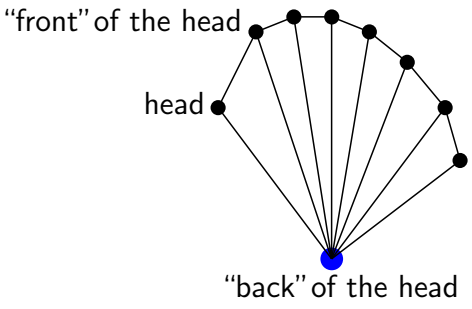
- `double my_area;`

  This is the area that the convex hull covers.

- `double area_of_triangle(const Point2d& p1, const Point2d& p2,... const Point2d& p3)`
  The time complexity is **O(** $1$ **)**.
  The space complexity is **O(** $1$ **)**.
  This is an auxiliary function that computes the area of a specific triangle. It is private because it is used from the function "`void notify_area()`"which is also a private member and because we don't want a user of this class to be able to call it.

- `void notify_area ()`

  The time complexity is **O(** $n$ **)**, $n =$ the size of "verb!this!"convex hull(2d).
  The space complexity of this function is **O(** $1$ **)**.
  It iterates the convex hull(2d) and finds its area. It is private because at the moment we don't want the user of this class to be able to call it whenever he wants. At the moment is not so costly about the time complexity. This function create intuitively a triangulation of the convex hull which is a very specific triangulation that you can see it below. As we




  can see all the triangles have as vertex the one with the blue color.

**Public Members:**

The purpose and the usage of the below iterator class is to provide basic iterator functions to get access of the convex hull(2d). If we increase the iterator then we iterate the convex hull(2d) clockwise, if we decrease it we iterate counter clockwise. Notice that the overloading operator "`Point2d operator*() const`"doesn't provide a pointer or a reference to a point but only a point. The reason for this is we don't want to change the basic ingredient of the convex hull(2d), which are the points, from the iterator.

- ```
  class ch_iterator{
  private:
      const Node* p;
  public:
      ch_iterator()
      ch_iterator(Node* x)
      ch_iterator(const ch_iterator& other_it)
      ch_iterator& operator++()
      ch_iterator operator++(int)
      ch_iterator& operator--()
      ch_iterator operator--(int)
      bool operator==(const ch_iterator& other_it) const
      bool operator!=(const ch_iterator& other_it) const
      ch_iterator& operator=(const ch_iterator& other_ch_it)
      Point2d operator*() const
      ch_iterator operator+(const unsigned int num) const
      ch_iterator operator-(const unsigned int num) const
  };
  ```

  - `const Node* p;`
    This is the Node that the iterator points to and since we don't want the iterator to change the list we declare it as a pointer to a constant Node.

  - `ch_iterator()`
    It is the default constructor of the iterator. It sets pointer the equal to zero.

  - `ch_iterator(Node* x)`
    The time complexity is **O(** 1 **)**.
    The space complexity is **O(** 1 **)**.
    It is the constructor which make the p points to a specific Node, i.e to which the x points to.

  - `ch_iterator(const ch_iterator& other_it)`
    The time complexity is **O(** 1 **)**.
    The space complexity is **O(** 1 **)**.
    The classic copy constructor, it makes an iterator points to the same Node as the `other_it`.

  - `ch_iterator& operator++()`
    The time complexity is **O(** 1 **)**.
    The space complexity is **O(** 1 **)**.
    The overloading of the postfix operator. It makes the iterator to point to the next element of the list, and specifically makes the iterator to point to "front" Node of the current that points to.
  ```

- `ch_iterator operator++(int)`
  The time complexity is **O( 1 )**.
  The space complexity is **O( 1 )**.
  The overloading of the prefix operator. The (int) is because the operator wants it to differentiate from postfix. It does what the prefix ++ does but is a prefix and the priority changes.

- `ch_iterator& operator--()`
  The time complexity is **O( 1 )**.
  The space complexity is **O( 1 )**.
  The overloading of the postfix operator. It makes the iterator to point to the previous element of the list, and specifically makes the iterator to point to "back" Node of the current that points to.

- `ch_iterator operator--(int)`
  The time complexity is **O( 1 )**.
  The space complexity is **O( 1 )**.
  The overloading of the prefix operator. The (int) is because the operator wants it to differentiate from postfix. It does what the prefix -- does but is a prefix and the priority changes.

- `bool operator==(const ch_iterator& other_it) const`
  The time complexity is **O( 1 )**.
  The space complexity is **O( 1 )**.
  It is the equality check operator, checks if two iterators of this class is equal, i.e if they point to the same Node.

- `bool operator!=(const ch_iterator& other_it) const`
  The time complexity is **O( 1 )**.
  The space complexity is **O( 1 )**.
  Similarly with the equality check operator, checks if two iterators of this class is different, i.e if they point to different Nodes.

- `ch_iterator& operator=(const ch_iterator& other_ch_it)`
  The time complexity is **O( 1 )**.
  The space complexity is **O( 1 )**.
  It is the assignment operator and it makes the iterator to point where the `other_ch_it` points.

- `Point2d operator*() const`
  The time complexity is **O( 1 )**.
  The space complexity is **O( 1 )**.

It is the most important operator because it returns the data of the Node that the iterator points to. Notice that the return type is (`Point2d`) and not (`Point2d&`) because I don't want to be able someone to change the content of where the iterator points.

- `ch_iterator operator+(const unsigned int num) const`
  The time complexity is **O(** `num` **)**.
  The space complexity is **O(** 1 **)**.
  If I add 1 to an iterator then it points to the next Node, i.e from a specific Node after the addition it points to the Node->front. If we move $+1$ every time, we are going clockwise.

- `ch_iterator operator-(const unsigned int num) const`
  The time complexity is **O(** `num` **)**.
  The space complexity is **O(** 1 **)**.
  If I subtract 1 to an iterator then it points to the previous Node, i.e from a specific Node after the addition it points to the Node->back. If we move -1 every time, we are going counter clockwise.

- `CH2d_dlclist()`

  The time complexity is **O(** 1 **)**.
  The space complexity is **O(** 1 **)**.
  It is the default constructor and make the head and tail equals to zero. The size is equal to zero and also the area.

- `CH2d_dlclist(const CH2d_dlclist& other_ch)`

  The time complexity is **O(** $n$ **)**, $n$= size of `other_ch`.
  The space complexity is **O(** $n$ **)**, $n$= size of `other_ch`.
  It is the copy constructor. It makes a new list which is a copy of the `other_ch` and isn't pointing to the `other_ch`.

- `CH2d_dlclist(const std::vector<Point2d>& points, ...`
  `std::string algorithm = "Jarvis")`

  The time complexity of "Jarvis"or "Gift-Wrapping"algorithm is **O(** $n * h$ **)**, $n$ = size of vector `points`, $h$ = size of the resulted convex hull(2d)(output sensitive).
  The space complexity of "Jarvis"or "Gift-Wrapping"algorithm is **O(** $h$ **)**, $h$ = the size of the resulted convex hull(2d)(output sensitive).
  `@param points`: Can be any vector without duplicate points.
  `@param algorithm`: Can be any supported by the program algorithm.
  This is the most important constructor, it makes the convex hull(2d) given a set of point and the name of the algorithm which will be used in order to construct the convex hull(2d).
  The "`std::vector<Point2d>& points`"is the set of the points, note that I considered a

std::set but I can't avoid duplicate points as an input because the algorithm will rejects them(and this is its work). "std::string algorithm"is the name of the algorithm. At this time the supported algorithm is the famous "Jarvis" or "Jarvis-March" or "Gift-Wrapping" algorithm.
Firstly it is checked if the size of the set of the points is "trivial", i.e if the size is equal to $0, 1, 2$. After that it is checked what algorithm the caller of the constructor wants. In the "Jarvis"algorithm first of all I find the position of the head and the tail in the given std::vector<Point2d>. After, the basic loop starts, we have the position of the last added point and one pointer to the node of the last point added to the list. The basic loop refresh the best point that will be added to the convex hull(2d). Assume we have a current candidate point that will be added to the hull and there is a new candidate point considered, if the new candidate has a better angle with the last point added(to the hull) than the current candidate has with the last point added, then we change the current candidate value with the new one. If the new one is collinear and is more distant from the last point added than the current candidate is distant from the last point added, then we again change the current candidate with the new one. After we considered all the new candidates(with the loop), we have as current candidate the best candidate and this will be added to the list(i.e to the convex hull(2d)). After, we allocate memory and we check if the last point which added to the list is the tail to assign the tail pointer to the node.

- ~CH2d_dlclist()

  The time complexity is **O(** $n$ **)**, $n =$ size of "this"convex hull(2d).
  The space complexity is **O(** 1 **)**.
  This is the destructor, we have to provide a destructor since we allocate dynamically memory with "new"(dynamically) when we make a list.

- CH2d_dlclist& operator=(const CH2d_dlclist& other_ch)

  The time complexity is **O(** n **)**, $n =$ size of other_ch convex hull(2d).
  The space complexity is **O(** n **)**, $n =$ size of other_ch convex hull(2d).
  It is the assignment operator. We have to check for auto-reference, we have to delete the "*this" list and after that we make a copy of other_ch with dynamically allocation of memory. Notice that even if the "other_ch"is "const"I prefer to access it by an iterator and not directly by using pointer to "Node"for safety reason so as not to change the "other_ch".

- unsigned int size() const

  The time complexity is **O(** 1 **)**.
  The space complexity is **O(** 1 **)**.
  It returns the size of the list, i.e the number of vertices of the convex hull(2d).

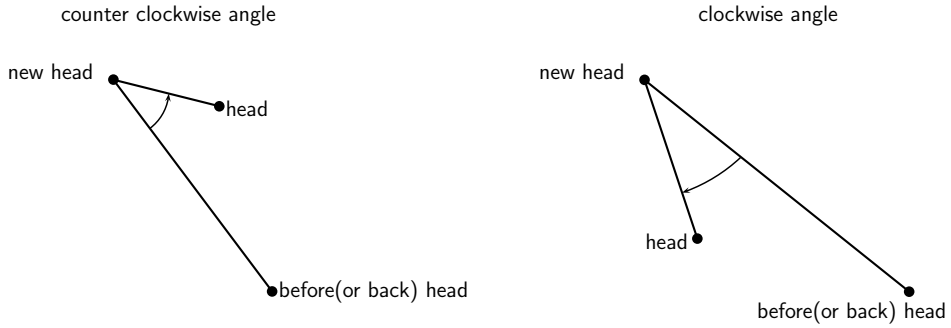
- int push(const Point2d& query_po)

  The time complexity is **O(** $n$ **)**, $n =$ size of "this".
  The space complexity is **O(** $n$ **)**, $n =$ size of "this".

`@param query_po`: Can be any point on the plane, even one that is equal to any point to the current convex hull(2d).
`@returns` : 1 if the "`query_po`" added to the convex hull(2d) and -1 if not.
This function takes as an argument a query point and adds it in the hull if it is not interior point. Returns $1$ if the query point added to the convex hull(2d) and $-1$ if it wasn't added to the convex hull(2d). First of all the routine checks the "trivial" cases where the current(before the addition or not of the query point) size of the convex hull(2d) is $0, 1, 2$. If the current size is $0$ then the query point will be added to the hull surely. If the current size is $1$ the query point is added if it is different than the one that is already in the hull. At this case there is different cases about which is the head and which is the tail of the list. There is 4 fundamental cases, for the different current sizes

- size $= 0$. Then the new point is added certainly. The "`head`" and the "`tail`" points to this point, the "`my_size`" becomes 1 and the "`my_area`" is equal to 0.
- size $= 1$. Then the new point is checked if it is equal with the one that exists in the list. If it is equal then it is returned -1. Otherwise the new point is added to the list and the program checks which one is the head and which is the tail. The "`my_size`" is increasing and the "`my_area`" remains 0.
- size $= 2$. First the new point is checked if it is equal with the one or the other points from the two that are in the list(i.e the head and the tail), if this holds then it is returned -1.
  After the new point is checked if it is collinear and between the two existant points that are in the list. If it is so then it is returned -1.
  If the new point is collinear with the two existant points in the list but it isn't between them then it will be added but one of the two existant points must be deleted and so the "`head`" or the "`tail`" will change, the "`my_size`" will remain unchanged and the same holds for "`my_area`".
  The other case is when the new point is not collinear to the two existant points nor equal to anyone of the them, then the new point will certainly be added to the list(we will have a triangle if we consider the new point). It is checked whether or not the "`head`" or the "`tail`" or none of them must change, the "`void notify_area()`" is called because the area must be computed(we have a triangle now) and the "`my_size`" is increased.
- size $\geq 3$. In this the procedure is split to two stages, the first is the one in which the new point is inserted at the list if it is not interior to the convex hull(2d) or lying inside an edge of the convex hull(2d)(if any of these two holds it is returned -1). The second procedure is after the insertion of the new point, if it was done, it consists of a lower and upper hull iteration and the deletion of points that make a "bad" angle. Let's analyze this two procedure starting from the first one.
  - The new point becomes "head". The new point lies where the head of the convex hull(2d) must be, so we assign the new point as head. There are two

cases about how to connect to the current list the new point, and you can see one representative of each case below :




The connection in the first case must be: `before head`, `new head`, `head` in clockwise order. In the second case: `before head`, `head`, `new head` in clockwise order

* The new point becomes "tail". The new point lies where the tail of the convex hull(2d) must be, so we assign the new point as tail. There are two cases similar with the previous and are not described.
* The new point is checked if: it is equal to any point in convex hull(2d), if it lies above the upper hull or below the lower hull. In the last two cases the new point is inserted to the list and later to the next stage the unnecessary points(the ones that are the middle points of a non counter clockwise turn) will be deleted. There is also an exceptional case where there are two points with the same x coordinate in the convex hull(2d) before the insertion of the new point and the new point lies in the line segment between the two points. Then the new point is not added and it is returned -1.

The next procedure is to delete the unnecessary points, we are using a part from the "Andrew's algorithm "or "monotone chain". Starting from the head and iterates the upper hull we delete the middle point of every non counter clockwise turn. The same holds for the lower hull iteration. All the points that are deleted are inserted to a deletion stack and are figuratively deleted when the upper and lower hull iteration are terminated. There is also a check of whether or not the new point is deleted to decide if the function will return -1 or 1.

- `unsigned int size() const`

  The time complexity is **O(** 1 **)**.
  The space complexity is **O(** 1 **)**.
  It returns the size of the list, i.e the number of the vertices of the convex hull(2d).

- `ch_iterator begin() const`

  The time complexity is **O(** 1 **)**.
  The space complexity is **O(** 1 **)**.
  It returns the iterator that points where the "head"points.

- `ch_iterator end() const`

  The time complexity is **O(** 1 **)**.
  The space complexity is **O(** 1 **)**.
  It returns the iterator that points where the "tail" points.

- `double area() const`

  The time complexity is **O(** 1 **)**.
  The space complexity is **O(** 1 **)**.
  Returns the area that is held at member "`my_area`".

## 1.5 CompGeomLibrary class

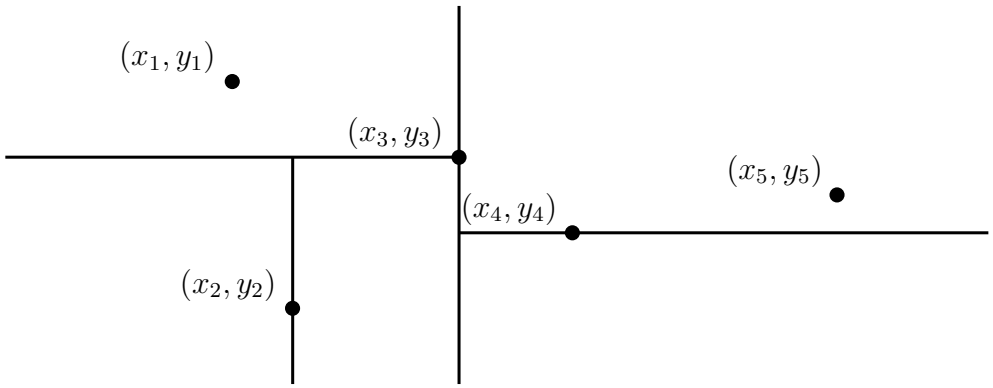
It is the main class I wanted to make.
Purpose: To implement algorithms of computational geometry, like algorithms that find the convex hull for a given set of 2d points and many others.

- `static std::list<Point2d> ComposeConvexHull2d(std::list<Edge2d> ... list_of_edges);`

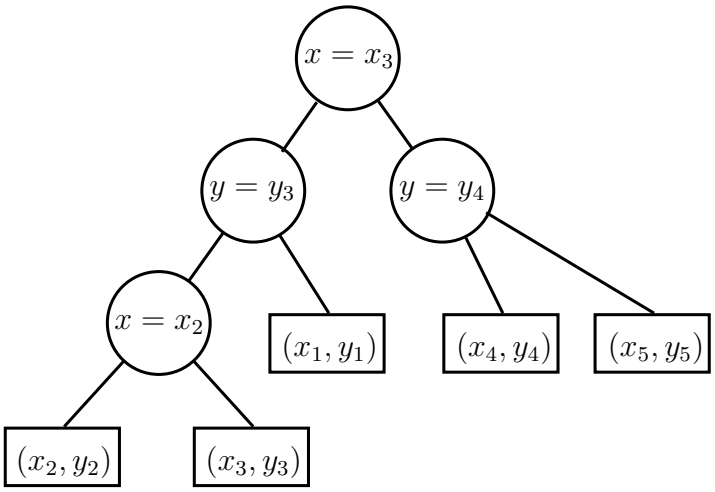
  Returns a list of points that contains the vertices of the convex hull in a clock wise order. The parameter `list_of_edges` contains the unsorted edges of the convex hull.

## 1.6 K2d_tree class

This class implements Kd-tree which is balanced binary search tree. It is useful for range searching and other areas. Suppose you have a set of points, the idea behind the Kd-tree is to split the space in subspaces using a specific coordinate each time until there is only one point in each subspace. You can see below how we construct a K2d-tree below :




First we split the space into two subspaces, how that is done? We choose the x coordinate and we find the median of the x coordinates of the set of points(the 5 points). The median is $x_3$, so we take the line $x = x_3$ as the first line that splits the space to two subspaces. The two sets that are formed are $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$ and $\{(x_4, y_4), (x_5, y_5)\}$. After that we take the two sets and we split it via finding the median of the y coordinates of the points in each set. The median of first set by the y coordinate is $y_3$ so we take the line $y = y_3$ as a splitter. The median of the second set is $y_4$. The procedure continues in this way while there is only one point in each segment of space. Note that the point on the split line belongs to the left or down subspace, it depends on if the split line is vertical or horizontal. Below you can see the tree that is created according to the aforementioned procedure:




**Members:**

- `struct Node{`
  ```
  Node* left;
  Node* right;
  int split_coord;
  double split_val;
  bool is_leaf;
  Point2d data_leaf;
  };
  ```
  The node above contains all the information a node of the tree must contain. If the node is internal node then "`left`"or "`right`"is non-zero, "`split_coord`"is 1 for x coordinate and 2 for y coordinate, "`split_val`"contains the median, "`is_leaf`"is false, "`data_leaf`"contains the `Point2d(0,0)`. If the node represents a leaf then "`left`"= "`right`"= "`split_coord`"= "`split_val`"= 0, "`is_leaf`"is true and "`data_leaf`"contains the unique point that is contained to the segment of the space.

**Private Members:**

- `Node* root;`

  The pointer "`root`"is pointing to the top node, that corresponds to the first split. It can be also a leaf when the set that we want to split contains only one point.

- `std::vector<Point2d> sort_by_x;`

  In this vector we hold all the points(all the leaves) but sorted with respect to x coordinate. It is needed to keep to a vector the points sorted because firstly the function "`Node* BuildTree(int, std::vector<Point2d>, std::vector<Point2d>)`"wants the points sorted and secondly because if we want to add a point to the set of points and reconstruct the tree then it is more easy to keep stored the points in a vector.

- `std::vector<Point2d> sort_by_y;`

  In this vector we hold all the points(all the leaves) but sorted with respect to x coordinate. It is needed to keep to a vector the points sorted because firstly the function "`Node* BuildTree(int, std::vector<Point2d>, std::vector<Point2d>)`"wants the points sorted and secondly because if we want to add a point to the set of points and reconstruct the tree then it is more easy to keep stored the points in a vector.

- `unsigned int my_size;`

  In this variable we hold the number of the leaves of the tree, i.e all the points that was given for the tree to be constructed.

- `Node* BuildTree(int split, std::vector<Point2d> points_by_x, std::vector<Point2d> points_by_y)`

  `@param split`: It is referred to the coordinate according to which we split the fragment of space, if it is even we split by the x coordinate, otherwise by the y coordinate.
  `@param points_by_x`: It is the current set of points that we will split it by finding the media, but sorted with respect to x coordinate. Currently this function works properly only for set of points that there is no pair of points that have equal x or y coordinates.
  `@param points_by_y`: It is the current set of points that we will split it by finding the media, but sorted with respect to y coordinate. Currently this function works properly only for set of points that there is no pair of points that have equal x or y coordinates.
  `@returns`: A pointer to the first root(node) of the sub-tree that corresponds to the current set of points.
  This is a recursive function that builds the whole tree. If the current call have a subset of size 1, i.e it is a leaf then memory is allocated to store the leaf.

- `std::vector<Node*> getAllNodes(Node* node)`

**Public Members:**

- ```
  class tree_iterator{
  private:
      const Node* p;
  public:
      tree_iterator()
      tree_iterator(Node* x)
      tree_iterator(const tree_iterator& other_it)
      bool operator==(const tree_iterator& other_it) const
      bool operator!=(const tree_iterator& other_it) const
      tree_iterator& operator=(const tree_iterator& other_ch_it)
      double getSplitValue()
      int getSplitCoord() const
      bool hasRightChild() const
  ```
  ```

```
        bool hasLeftChild() const
        bool hasChildren() const
        tree_iterator getLeftChild() const
        tree_iterator getRightChild() const
        void goRightChild()
        void goLeftChild()
        bool isInternalNode() const
        Point2d getLeafPoint() const
    };
```

- `static std::vector<Point2d> getAllPoints(tree_iterator it_node)`
- `K2d_tree(std::vector<Point2d> points)`
- `~K2d_tree()`
- `tree_iterator begin() const`
- `void addPoint(const Point2d poi)`
- `unsigned int size() const`

# Chapter 2

# Tests

The tests that I have implemented must be tracked in order to know what tests and what cases for each algorithm I have implemented.

## 2.1 CH2d_dlclist class

### 2.1.1 unit_test_CH2d_dlclist.cpp

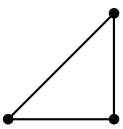
- `void test_parameter_constructor1(std::ostream& write_str)`

case 1(Jarvis)

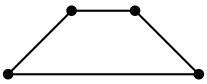


case 2(Jarvis)

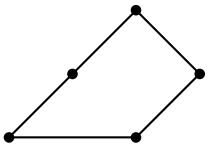
case 3(Jarvis)



case 4(Jarvis)



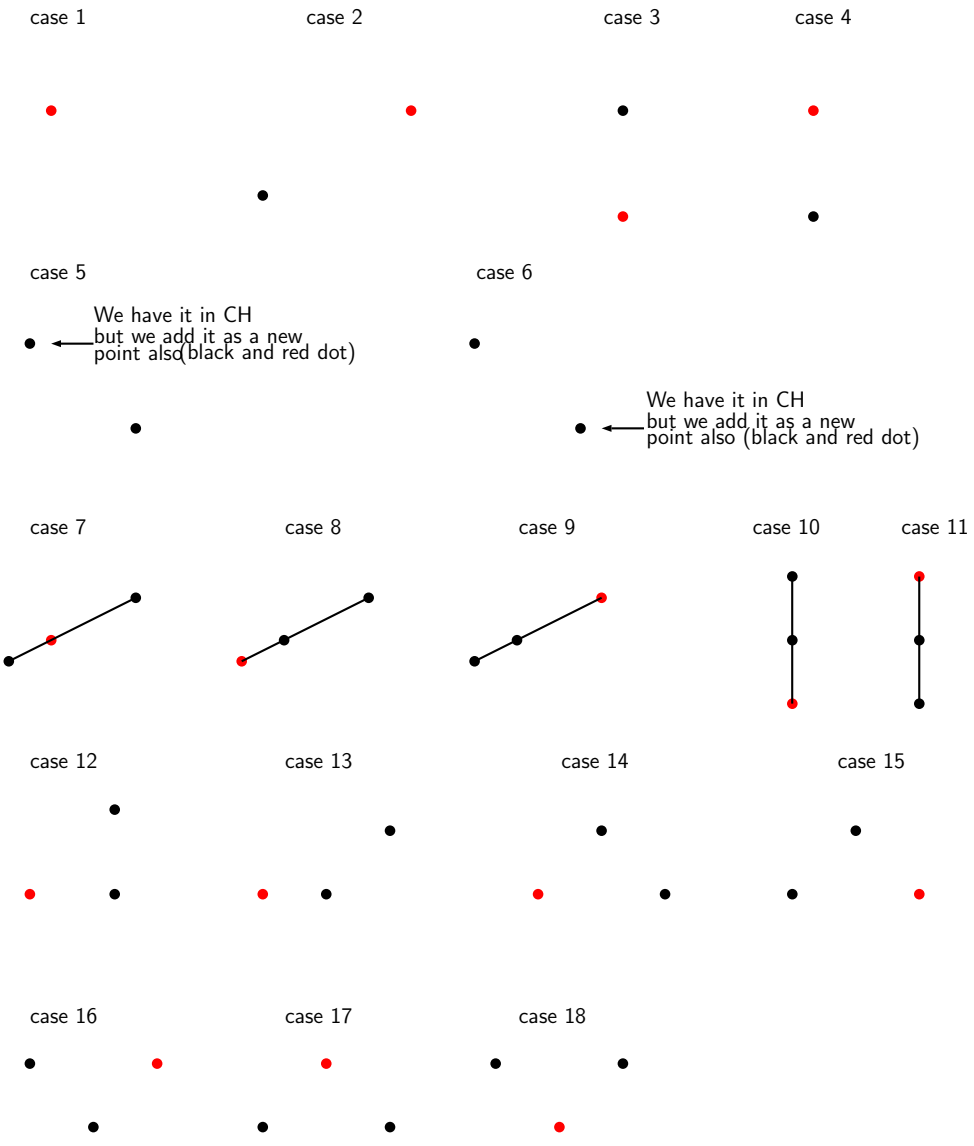
case 5(Jarvis)



case 6(Jarvis)

A vector with no points.

- `void test_push(std::ostream& write_str)`


case 1
case 2
case 3
case 4
case 5
We have it in CH
but we add it as a new
point also(black and red dot)
case 6
We have it in CH
but we add it as a new
point also (black and red dot)
case 7
case 8
case 9
case 10
case 11
case 12
case 13
case 14
case 15
case 16
case 17
case 18