Partitioning into Monotone Mountains

Warning: I ♥ templates

Outline

- C++ Preliminaries
- Linked Lists
- Sweep Line Algorithm
- Trapezoidalization
- Partitioning into Monotone Mountains

C++ Preliminaries

C++ supports defining (local) Lambda functions:

```
auto f = []( arguments ){ function-body };
```

Or, if you would like `function body` to have access to local values *by reference*:

```
auto f = [&]( arguments ){ function-body };
```

C++ Preliminaries

As the type of a Lambda is gnarly, C++ has the templated `std::function` class for wrapping functions:

```
function< return-type ( arguments ) > f;
f =[&]( arguments ){ function-body };
```

C++ Preliminaries

When class names get to be a mouthful, C++ has the typedef keyword for renaming them:

```
typedef really-long-name rln;
```

If the class is templated, C++ has the using keyword to support aliasing:

```
template< params >
using rln = really-long-name< params >;
```

Linked Lists

Useful to have a generic (template) class for linked lists and then instantiate with content type.

Linked Lists

```
template< typename Data >
struct CircularLinkedListElement
{
        typedef CircularLinkedListElement CLLE;
        Data data;
         CLLE *prev , *next;
        CLLE( void ){ prev = next = this; }
        CLLE( const Data &d ) : data(d) { prev = next = this; }
        ~CircularLinkedListElement( void );
        unsigned int size( void ) const;
        CLLE *addAfter( const Data &data );
        static CLLE *RemoveAndReturnPrevious(CLLE *e );
        …
```

Linked Lists

```
	…
	void process ( function< void ( CLLE * ) > pf );
	void process ( function< void ( const CLLE * ) > pf ) const;
	static pair< CLLE * , CLLE * > Split( pair< CLLE * , CLLE * > e );
};
```

Linked Lists

```
        ...
        void process ( function< void ( CLLE * ) > pf );
        void process ( function< void ( const CLLE * ) > pf ) const;
        static pair< CLLE * , CLLE * > Split( pair< CLLE * , CLLE * > e );
};
```

General-purpose functionality for iterating over the entries of the list and doing something.

Linked Lists

```
template< typename Data >
void CircularLinkedListElement< Data >::process( function< void ( CLLE * ) > pf )
{
	for( CLLE *v=this ; ; v=v->next )
	{
		pf(v);
		if( v->next==this ) break;
	}
}
```

Linked Lists

Example Usage:

```
template< typename Data >
void CircularLinkedListElement< Data >::size( void ) const
{
        unsigned int s = 0;
        process( [&]( const CLLE *v ){ s++; } );
        return s;
}
```

Linked Lists

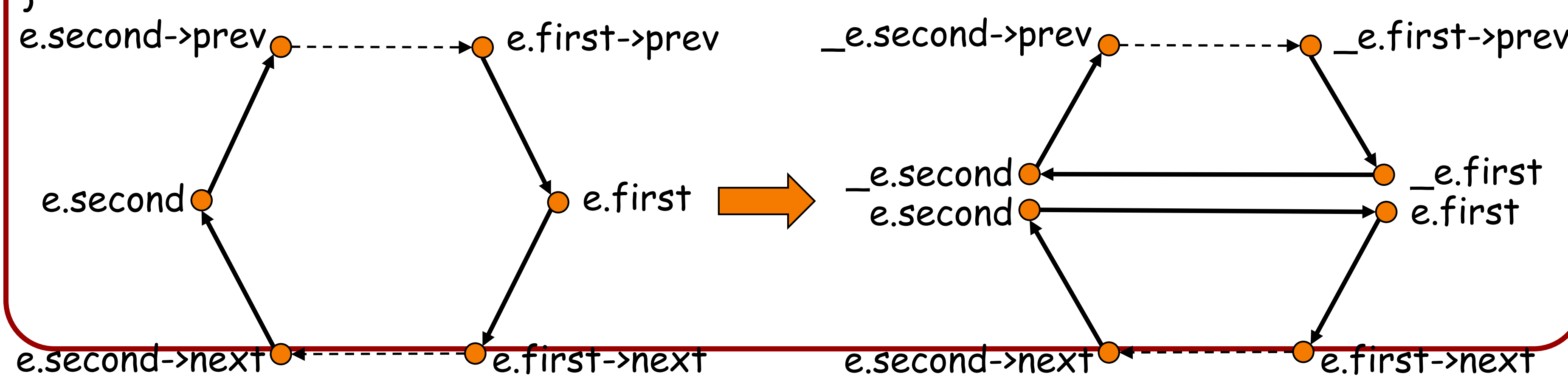
Example Usage:

```
template< typename Data > using CLLE = CircularLinkedListElement< Data >;


template< typename Data >
ostream &operator << ( ostream &os , const CLLE< Data > &v )
{
	v.process( [&]( const CLLE< Data > *v ){ os << v->data << endl; } );
	return os;
}
```

Linked Lists

```
template< typename Data >
pair< CLLE * , CLLE * > CLLE< Data >::Split( pair< CLLE * , CLLE * > e )
{
        pair< CLLE * , CLLE * > _e;
        _e.first = new CLLE( e.first->data );
         _ e.second = new CLLE( e.second->data );
        // Adjust the pointers here
        return _e;
}
```



Linked Lists

Since we will be using the circular linked list to represent polygons, we can create a specialized instance called PVertex:

```
typedef CircularLinkedListElement< unsigned int > PVertex;
```

The assumption here is that we have a (possibly const) array of vertices representing the set of vertex positions.

So instead of storing the actual vertex position in the list, we store the index of the vertex within the list.

Sweep Line Algorithm

Represent the active edge list using the `std::map` class from the standard template library. Requires:

- Key type
- Value type
- Comparator type

Sweep Line Algorithm

Key Type:

Something that represents the geometry of an edge stored in the active edge list:

```
struct EKey
{
        Point2i v1 , v2;
        EKey( void ){}
        EKey( Point2i _v1 , Point2i _v2 ) : v1(_v1) , v2(_v2){}
};
```

Sweep Line Algorithm

Value Type:

Something that represent the auxiliary information stored with each edge in the active edge list:

```
struct EValue
{
        // Some data
};
```

Sweep Line Algorithm

Comparator Type:

A function that compares two `EKeys` and returns `true` if the first is smaller than the second.

```
float sweepHeight;
typedef function< bool ( const EKey & , const EKey & ) > EComparator;
EComparator eComparator = [&]( const EKey &k1 , const EKey &k2 )
{
        // Compare the keys using the current value of sweepHeight
};
```

Note that because we declare the Lambda with [&], the function body has access to `sweepHeight`.

Sweep Line Algorithm

Comparator Type:

For this to work, we have to ensure that whenever we change the value of **sweepHeight**, the values in the **std::map** remain sorted with respect to the new value.

A

if

```
float sweepHeight;
typedef function< bool ( const EKey & , const EKey & ) > EComparator;
EComparator eComparator = [&]( const EKey &k1 , const EKey &k2 )
{
        // Compare the keys using the current value of sweepHeight
};
```

Note that because we declare the Lambda with [&], the function body has access to *sweepHeight*.

Sweep Line Algorithm

Using these we can declare an empty active edge list using our comparator to order the elements:

```
map< EKey , EValue , EComparator > activeEdges( eComparator );
```

Sweep Line Algorithm

To perform the sweep we need to sort the vertices by height, traverse the sorted vertices, and update the state of the active edge list.

Sweep Line Algorithm

```
void DoSweepLine( const vector< Point2i > &verts )
{
        // Turn the array of vertices into a linked-list (polygon)
        PVertex *poly = GetPolygon( (unsigned int)verts.size() );

        // Sort the vertices by height
        vector< PVertex * > sVertices;
        {
                poly->process ( [&]( PVertex * v ){ sVertices.push_back( v ); } );
                sort( sVertices.begin() , sVertices.end() , <sort functor> );
        }
        …
```

Sweep Line Algorithm

```
void DoSweepLine( const vector< Point2i > &verts )
{
        …
        float currentHeight;
        typedef function< bool ( const EKey & , const EKey & ) > EComparator;
        EComparator eCompararator = [&]( const EKey &k1 , const EKey &k2 )
        {
                // Compare the keys using the current value of sweepHeight
        };
        map< EKey , EValue , EComparator > activeEdges( eCompararator );
        …
```

Sweep Line Algorithm

```
void DoSweepLine( const vector< Point2i > &verts )
{
        …
        for( int i=0 ; i< sVertices.size() ; i++ )
        {
                PVertex *c = sVertices[i];
                PVertex *p = c->prev , *n = c->next;
                currentHeight = vertices[ c->data ][1];
                bool pAbove = verts[ p->data ][1]>currentHeight;
                bool nAbove = verts[ n->data ][1]>currentHeight;
                …
        }
}
```

Sweep Line Algorithm

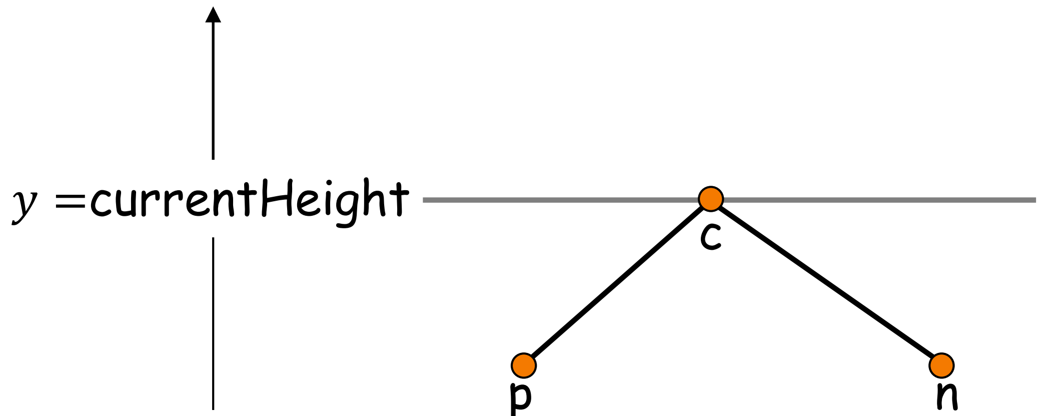
```
void DoSweepLine( const vector< Point2i > &verts )
{
	…
		// Get the edge keys  for the incoming and outgoing edges
		EKey pKey , nKey;
		if( pAbove ) pKey = EKey( verts[c->data] , verts[p->data] );
		else         pKey = EKey( verts[p->data] , verts[c->data] );
		if( nAbove ) pKey = EKey( verts[c->data] , verts[n->data] );
		else         pKey = EKey( verts[n->data] , verts[c->data] );
	…
}
```

For simplicity, vertices in EKey are sorted by height.

Sweep Line Algorithm

Case (!pAbove && !nAbove):

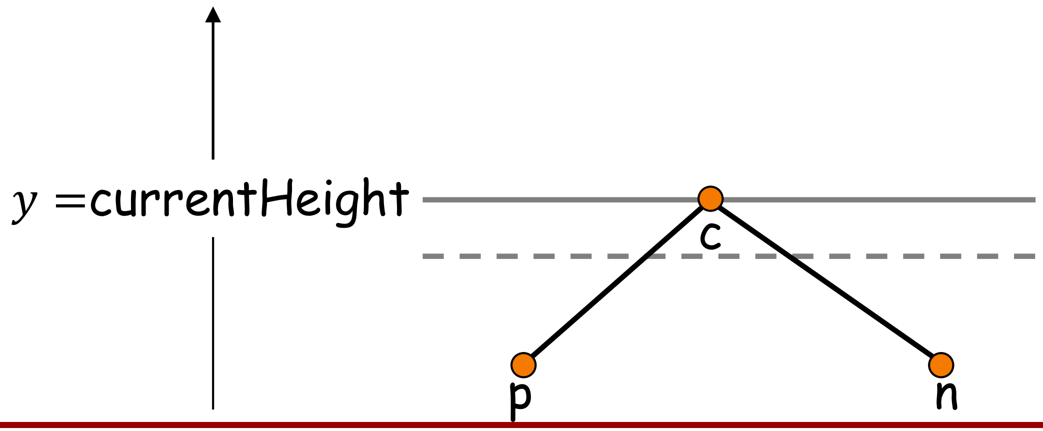
// Remove edges (p,c) and (n,c)



Sweep Line Algorithm

Case (!pAbove && !nAbove):

```
// Remove edges (p,c) and (n,c)
currentHeight -= 0.5f;
activeEdges.remove( pKey );
activeEdges.remove( nKey );
```

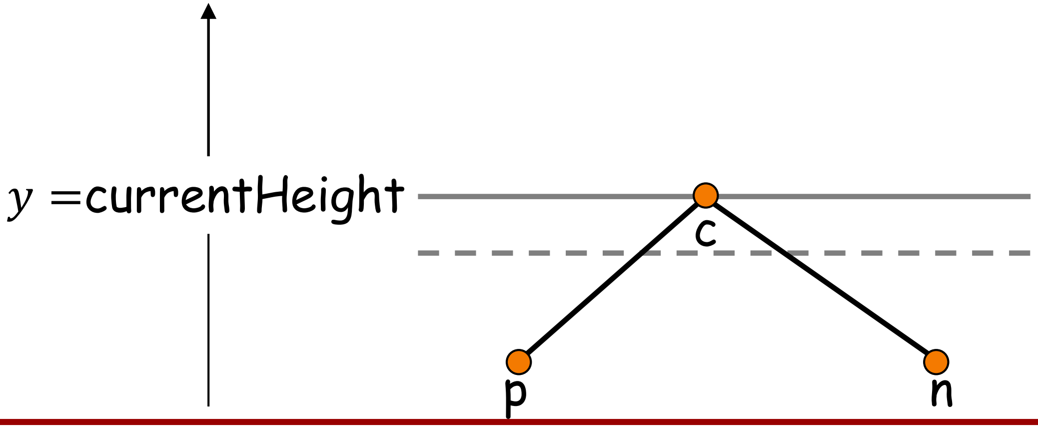


Sweep Line Algorithm

Case (!pAbove && !nAbove):

```
// Remove edges (p,c) and (n,c)
currentHeight -= 0.5f;
activeEdges.remove( pKey );
activeEdges.remove( nKey );
```

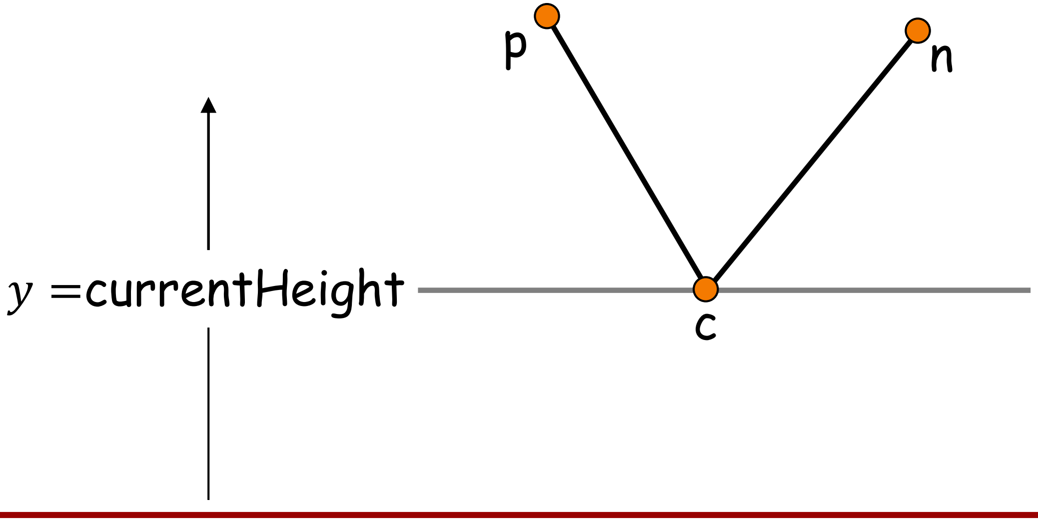
Note that we reduce currentHeight so the edges associated to pKey and nKey are correctly sorted in the active edge list.



Sweep Line Algorithm

Case (pAbove && nAbove):

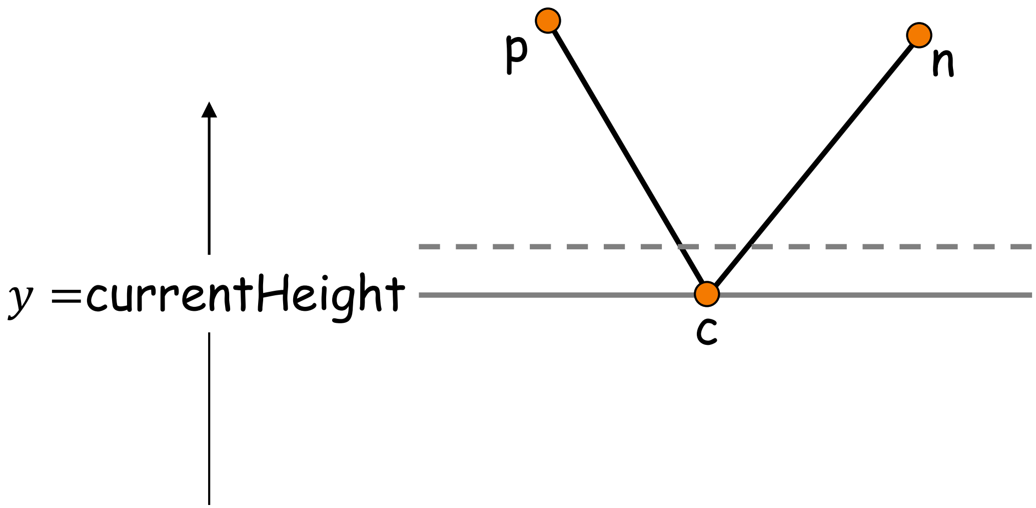
// Insert edges (p,c) and (n,c)



Sweep Line Algorithm

Case (pAbove && nAbove):

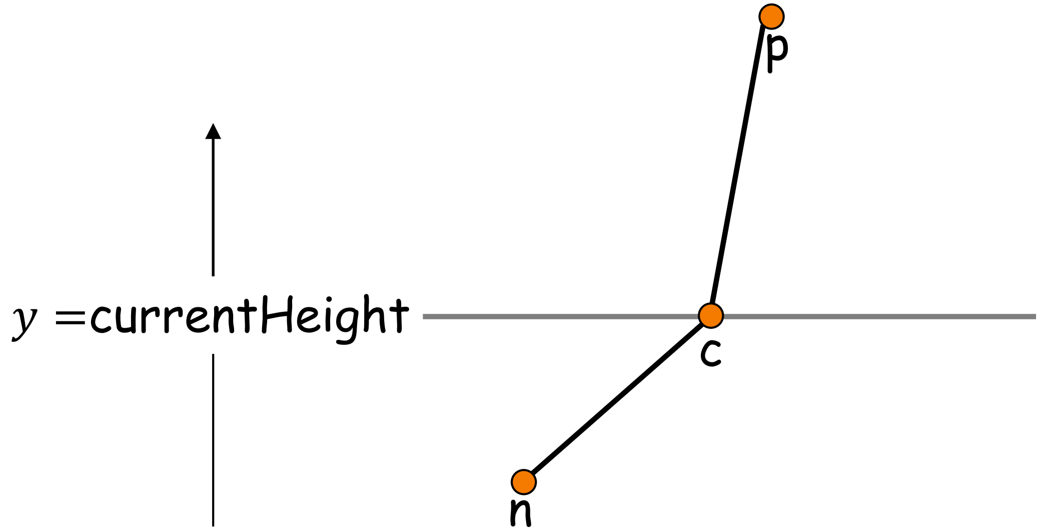
```
// Insert edges (p,c) and (n,c)
currentHeight += 0.5f;
activeEdges[pKey] = EValue();
activeEdges[nKey] = EValue();
```



Sweep Line Algorithm

<u>Case (pAbove && !nAbove)</u>:

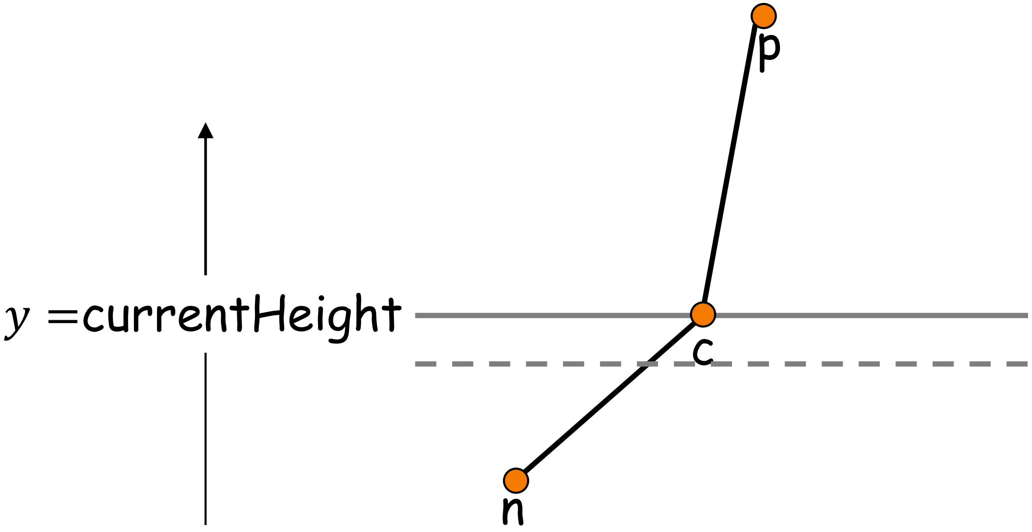
// Remove edge (n,c) and insert edge (p,c)



Sweep Line Algorithm

Case (pAbove && !nAbove):

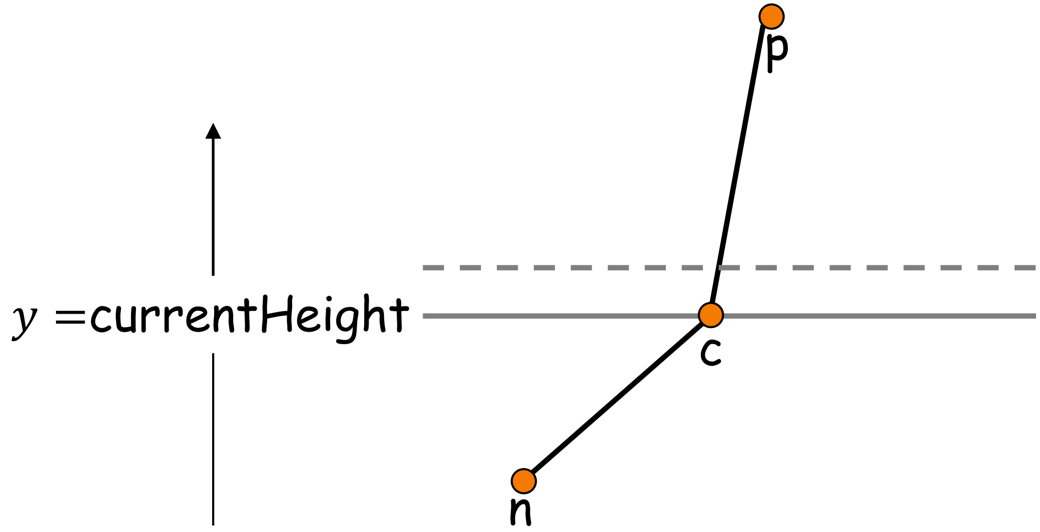
```
// Remove edge (n,c) and insert edge (p,c)
currentHeight -= 0.5f;
activeEdges.remove( nKey );
```



Sweep Line Algorithm

Case (pAbove && !nAbove):

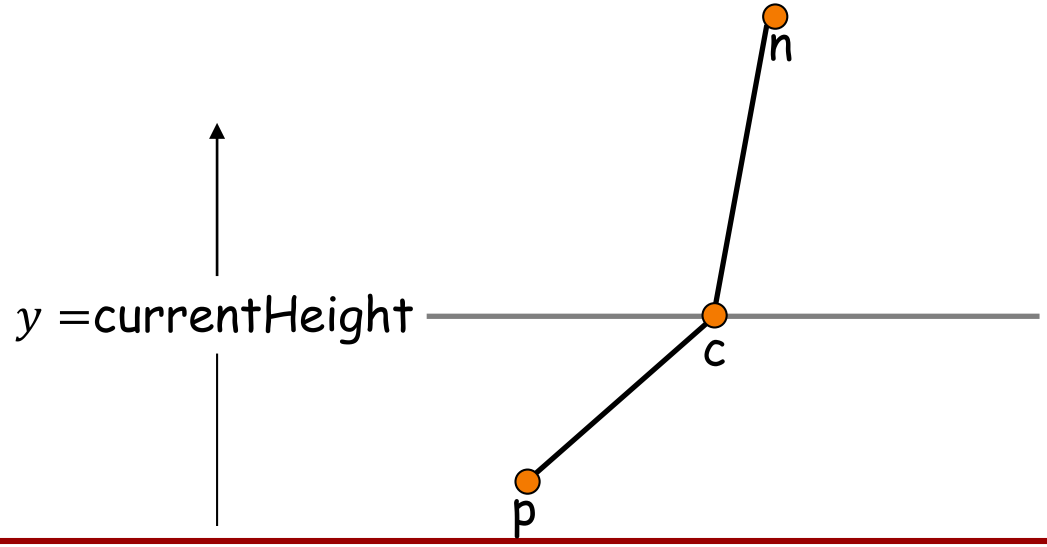
```
// Remove edge (n,c) and insert edge (p,c)
currentHeight -= 0.5f;
activeEdges.remove( nKey );
currentHeight += 1.0f;
activeEdges[pKey] = EValue();
```



Sweep Line Algorithm

Case (!pAbove && nAbove):

```
// Remove edge (p,c) and insert edge (n,c)
currentHeight -= 0.5f;
activeEdges.remove( pKey );
currentHeight += 1.0f;
activeEdges[nKey] = EValue();
```



Sweep Line Algorithm

Notes:

The algorithm can be implemented using only integer arithmetic.

- For `currentHeight`, scale the vertex coordinates by a factor of 2 and use odd height values.
  - » Since the vertex array passed in `const`, will need to make a copy.
- For the implementation of `eComparator`, compute the (fractional) positions of the intersections with the horizontal line and then multiply both sides by the product of denominators.
  - » Make sure to track the sign of the product for the comparison.

Trapezoidalization

Need to track enough information with the active edges to be able to reconstruct trapezoids. This includes:

- Actual vertices in the linked lost
- Information about the bottom supporting vertex

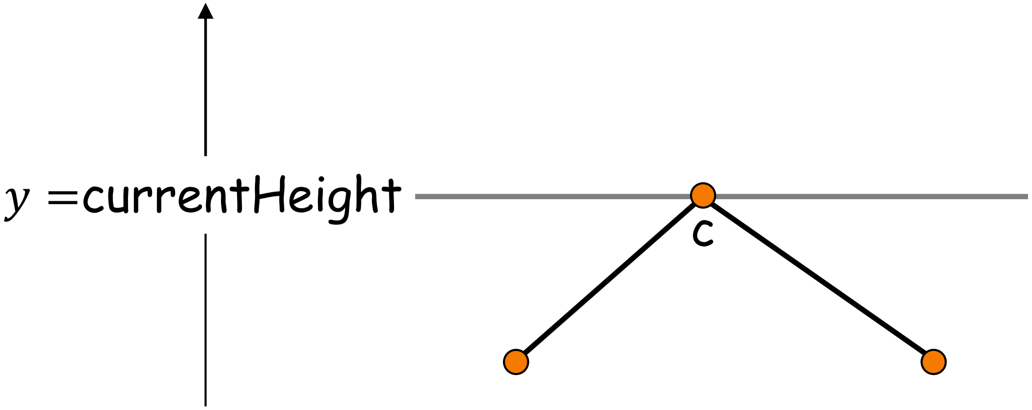
```
struct EValue
{
        PVertex *v1 , *v2 , *tSupport;
        EValue( PVertex *_v1=NULL , PVertex *_v2=NULL , Pvertex *_t=NULL )
                : v1(_v1) , v2(_v2) , tSupport(_t){}
};
```

Trapezoidalization

```
void DoTrapezoidalization( const vector< Point2i > &verts )
{
        …
                // Get the edge keys  for the incoming and outgoing edges
                …
                // Get the edge values  for the incoming and outgoing edges
                EValue pValue , nValue;
                if( pAbove ) pValue = EValue( c , p , c );
                else         pValue = EValue ( p , c , c );
                if( nAbove ) pValue = EValue ( c , n , c );
                else         pValue = EValue ( n , c , c );
        …
}
```

Trapezoidalization

Case (!pAbove && !nAbove):

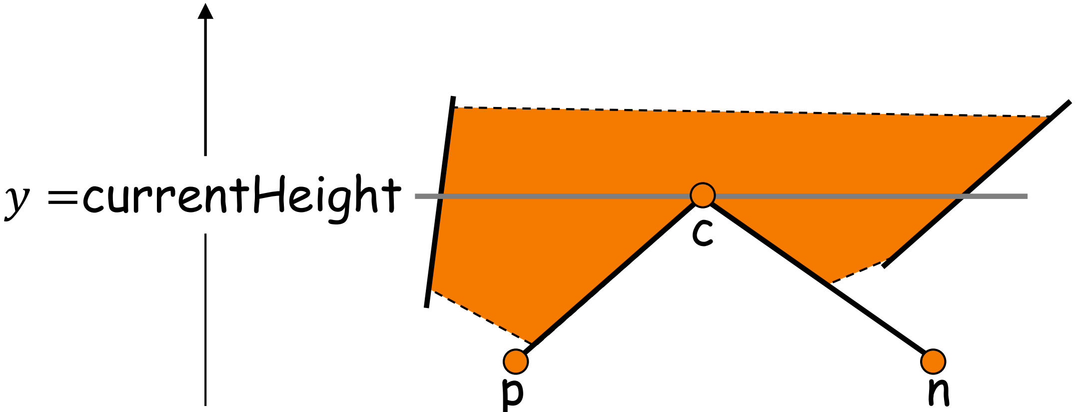


Trapezoidalization

Case (!pAbove && !nAbove && (p left of n)):

// Two trapezoids merge

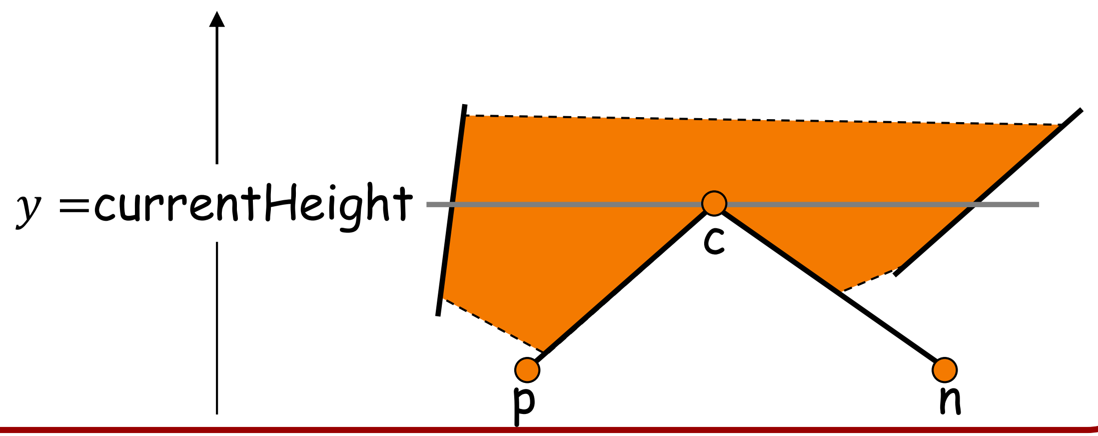
// Change support of active edges left of p-c and right of n-c



Trapezoidalization

<u>Case (!pAbove && !nAbove && (p left of n))</u>:

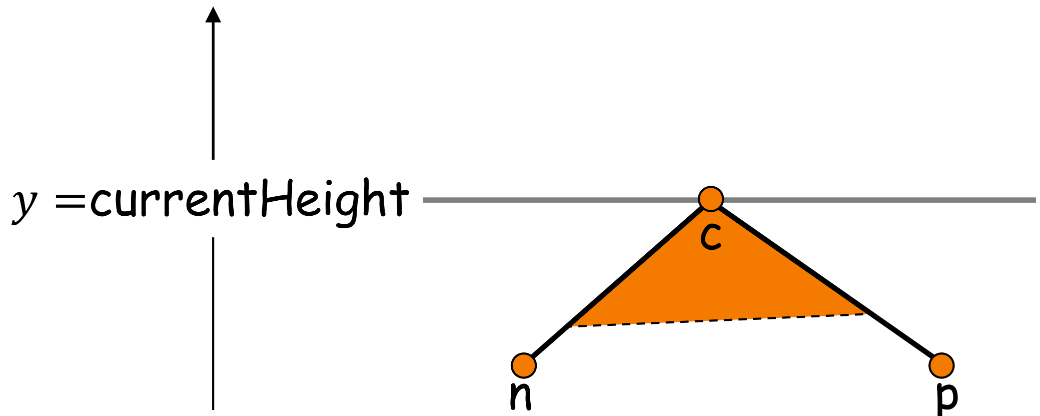
```
// Two trapezoids merge
// Change support of active edges left of p-c and right of n-c
auto iter = activeEdges.find( pKey );
iter--;
iter->second->tSupport = c;
…
```



Trapezoidalization

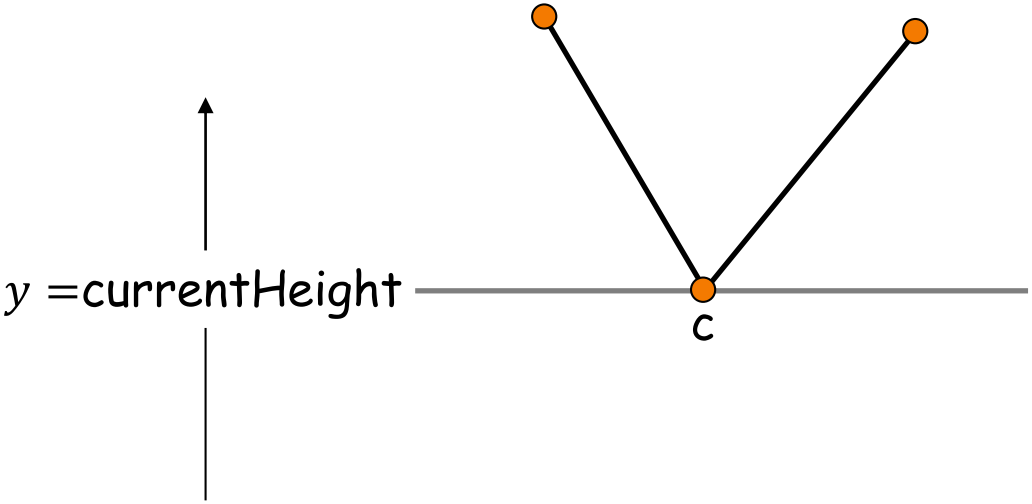
Case (!pAbove && !nAbove && (p not left of n)):

// A trapezoid disappears



Trapezoidalization

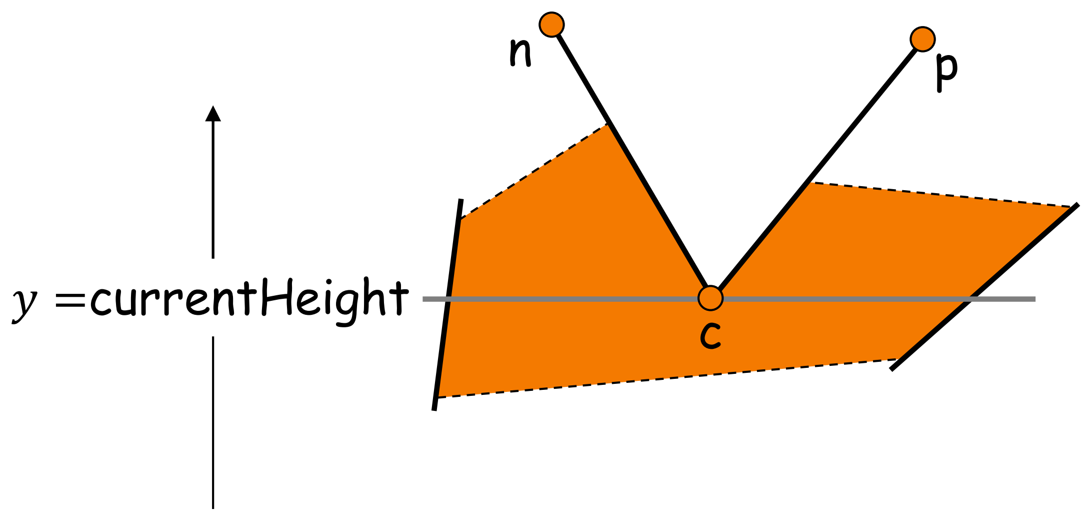
Case (pAbove && nAbove):



Trapezoidalization

<u>Case (pAbove && nAbove && (p not left of n))</u>:

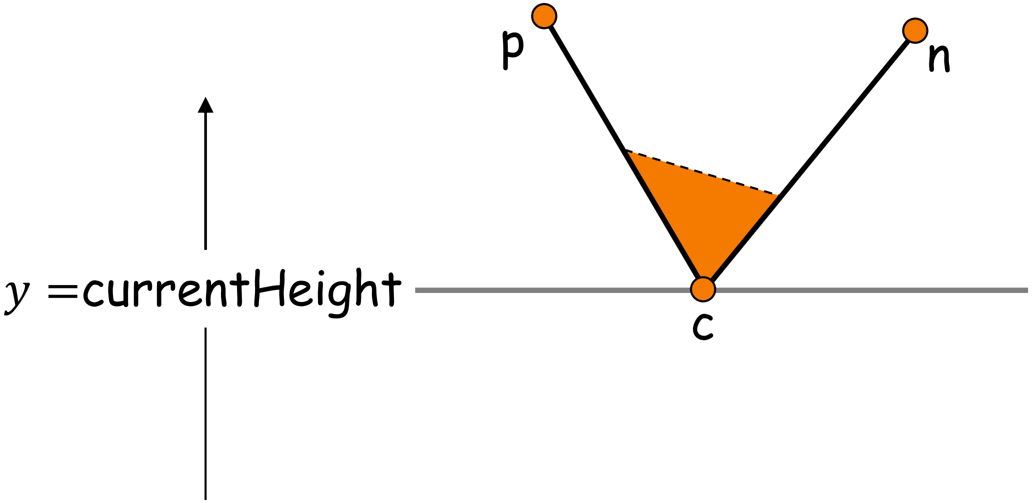
```
// A trapezoid splits in two
// Change support of active edges right of p-c and left of n-c
auto iter = activeEdges.find( pKey );
iter++;
iter->second->tSupport = c;
…
```



Trapezoidalization

Case (pAbove && nAbove && (p left of n)):

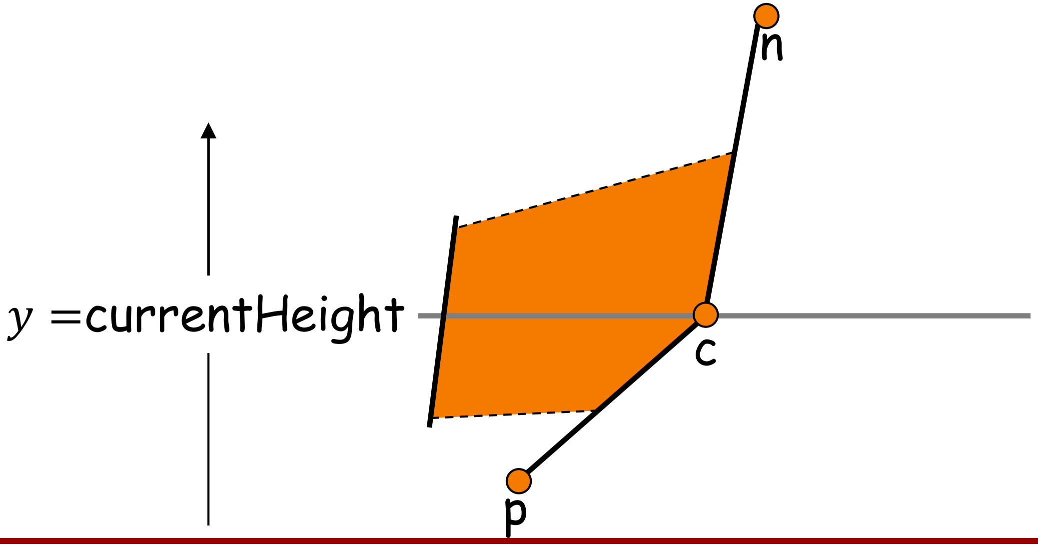
// A new trapezoid is created



Trapezoidalization

Case (!pAbove && nAbove):

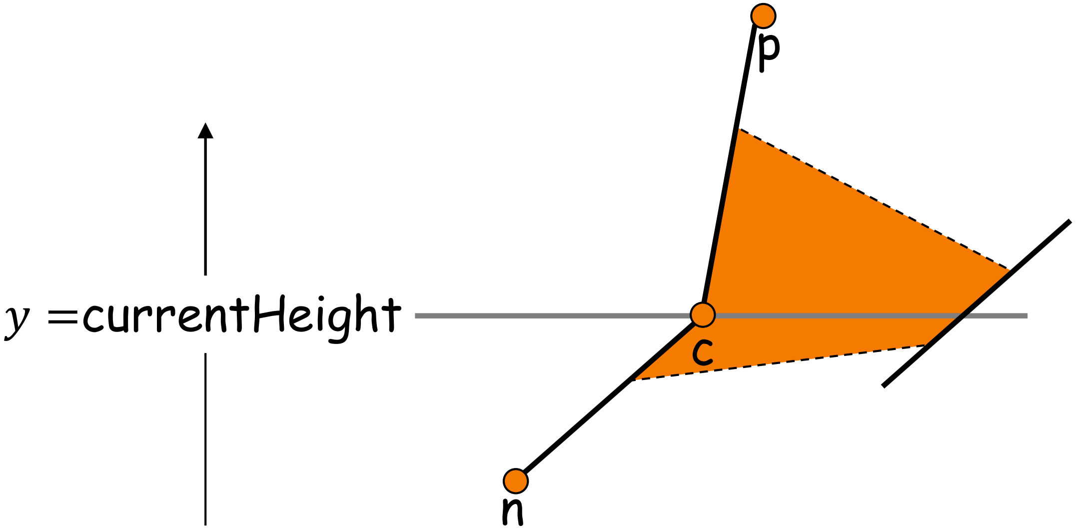
```
// Change support of active edge left of c-n
auto iter = activeEdges.find( nKey );
iter--;
iter->second->tSupport = c;
```



Trapezoidalization

Case (pAbove && !nAbove):

```
// Change support of active edge right of p-c
auto iter = activeEdges.find( nKey );
iter++;
iter->second->tSupport = c;
```



Monotone Mountains Partitioning

At sweep events, may need to split polygon into diagonals:

- Need to track multiple polygons
- May need to change pointers to vertices in the polygons

Key Idea:

Track the list of polygons by adding a monotone mountain when it is completed.

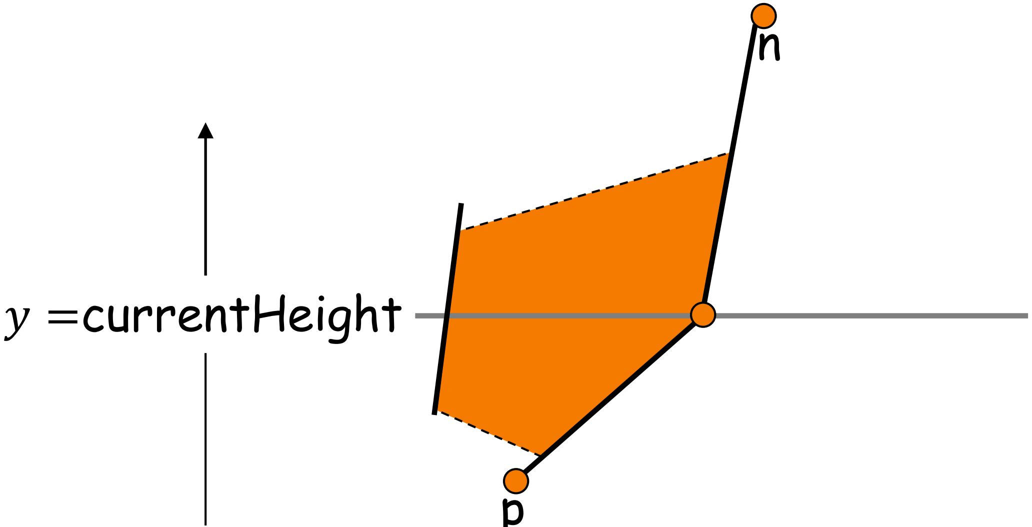
Monotone Mountains Partitioning

```
vector< PVertex * > GetMonotoneMountains( const vector< Point2i > &verts )
{
        vector< Pvertex * > mountains;
        …
        return mountains;
}
```

Monotone Mountains Partitioning

Case (!pAbove && nAbove):

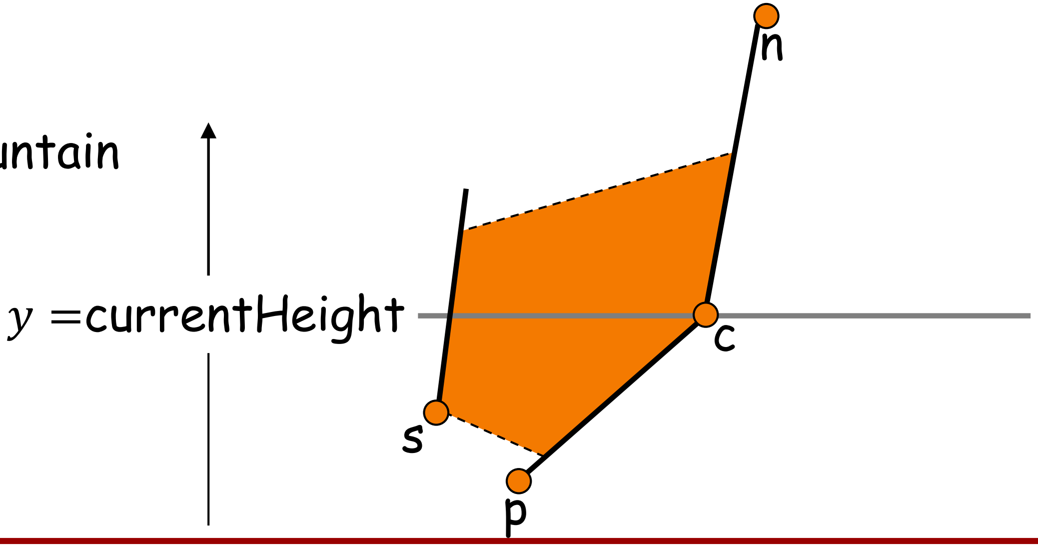
// Check if we need to add a diagonal (before updating supports)



Monotone Mountains Partitioning

Case (!pAbove && nAbove):

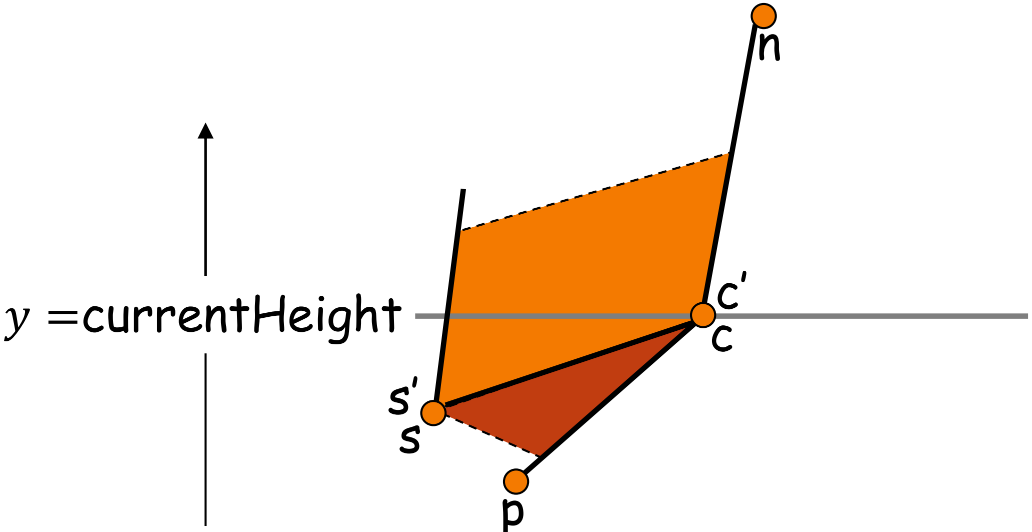
```
// Check if we need to add a diagonal (before updating supports)
auto iter = activeEdges.find( pKey );
PVertex *s = iter->second->tSupport;
if( s!=c->prev )
{
        // Split
        // Update
        // Output mountain
}
```



Monotone Mountains Partitioning

Case (!pAbove && nAbove):

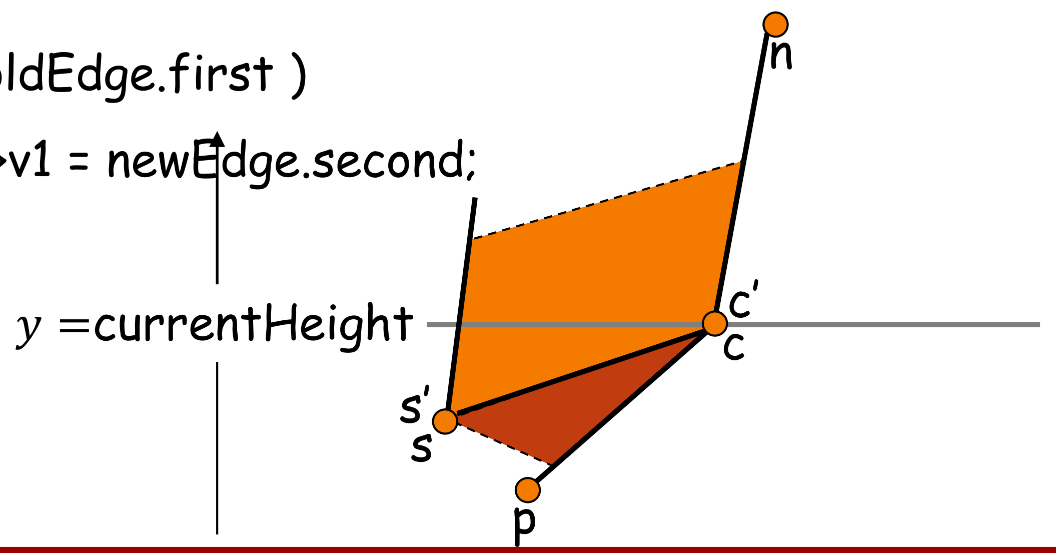
```
// Split
pair< PVertex * , PVertex * > oldEdge( c , s );
pair< PVertex * , PVertex * > newEdge = PVertex::Split( oldEdge );
```



Monotone Mountains Partitioning

Case (!pAbove && nAbove):

```
PVertex *oldC = c;
// Update the current vertex
c = newEdge.first;
// Update the bottom vertex of the edge left of p-c
auto iter = activeEdges.find( pKey );
iter--;
if( iter->second->v1==oldEdge.first )
        iter->second->v1 = newEdge.second;
```



Monotone Mountains Partitioning

Case (!pAbove && nAbove):

// Output the bottom mountain

```
mountains.push_back( oldC );
```