EECS 440 System Design of a Search Engine

Winter 2021

Lecture 12: The constraint solver

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Agenda

1. Course details.
2. The constraint solver.

Agenda

1. Course details.
2. The constraint solver.

Midterm

1. Midterm Monday March 8, 3:00 pm to 5:00 pm as promised in the course description. If you need an alternate time or other accommodations send mail to eecs440staff@umich.edu.

2. Exam will be online at https://crabster.eecs.umich.edu/ You will need to login with your Umich ID. Do not open multiple windows and do not use an incognito window.

3. Format will be 25 short answer questions, e.g., asking you to explain a concept or why one design approach might be better than another.

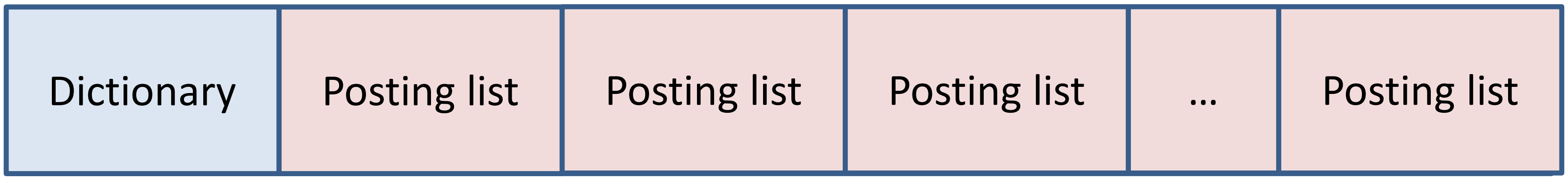
4. Open everything except collaboration, including posting questions anywhere, and attempts to seek out or use previous exams.

5. If you have questions about the exam, post privately on Piazza. We can look at your exam on Crabster and see it exactly as you see it.

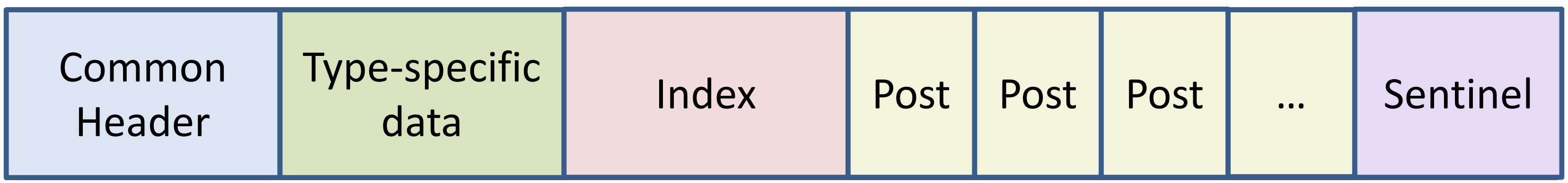
Agenda

1. Course details.
2. The constraint solver.

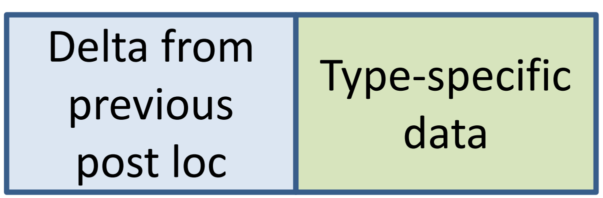
The inverted word index within a chunk.



A posting list



An individual post



We look up a search word in the dictionary, which takes us to a posting list. The index lets us jump to a location in the list without having to start from the beginning, adding up all the deltas.

An Index Stream Reader (ISR) is the abstraction we'll use for the seeking and reading posts in a posting list.

Index functions

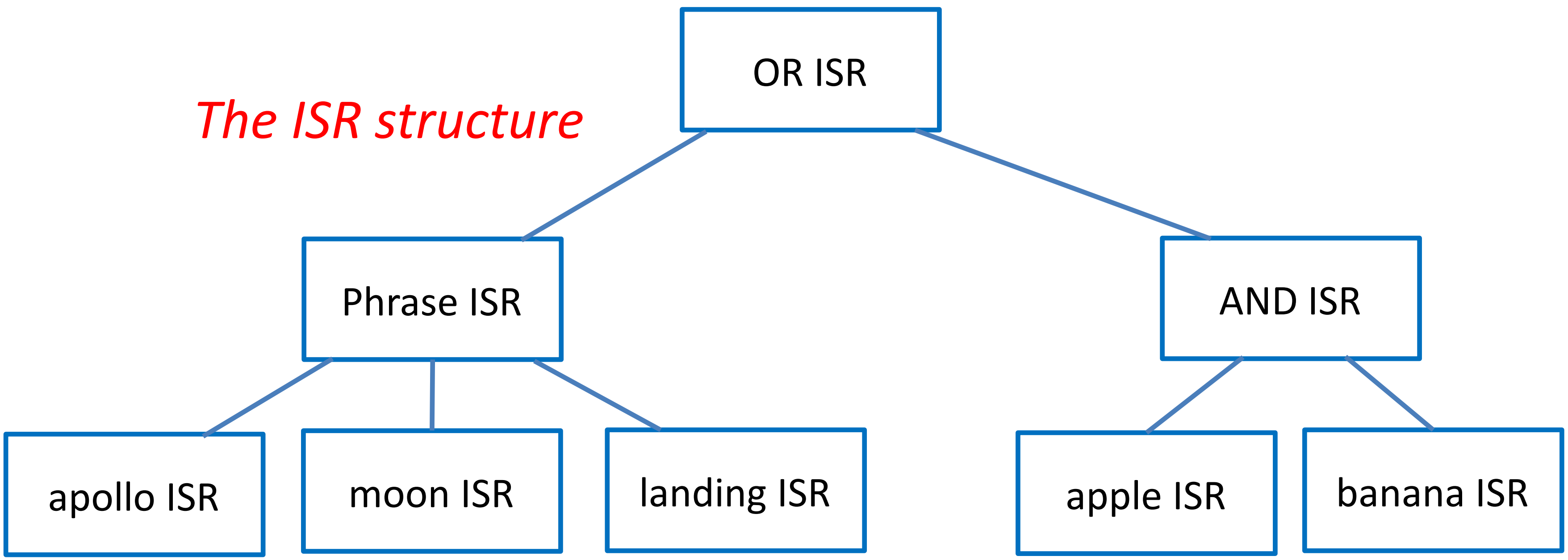
Index stream readers (ISRs)

`first( t )`	returns the first position at which t occurs.
`last( t )`	returns the last position at which t occurs.
`next( t, current )`	returns the next position where t occurs after the current position.
`prev( t, current )`	returns the last position where t occurs before the current position. *But slow and usually omitted.*

Basic idea to create ISR structures that match the query constraints.
Each ISR can find the next occurrence of whatever it's looking for.

"apollo moon landing" | (apple banana)

The ISR structure



Index Stream Reader (ISR)

Finds the next occurrence of the desired token or combination of child ISRs.

`ISRWord`	Find occurrences of individual words.
`ISREndDoc`	Find occurrences of document ends.
`ISROr`	Find occurrences of any child ISR.
`ISRAnd`	Find occurrences of all child ISRs within a single document.
ISRPhrase	Find occurrences of all child ISRs as a phrase.
`ISRContainer`	Find occurrences of contained ISRs in a single document not containing any excluded ISRs.

Index Stream Reader (ISR)

Two Basic ISRs to actual posts in the index.

`ISRWord`	Find occurrences of individual words.
`ISREndDoc`	Find occurrences of document ends.

Index Stream Reader (ISR)

Four abstract ISRs that combine sub-ISRs.

`ISROr`	Find occurrences of any child ISR.
`ISRAnd`	Find occurrences of all child ISRs within a single document.
ISRPhrase	Find occurrences of all child ISRs as a phrase.
`ISRContainer`	Find occurrences of contained ISRs in a single document not containing any excluded ISRs.

```cpp
typedef size_t Location;      // Location 0 is the null location.

typedef union Attributes
   {
   WordAttributes Word;
   DocumentAttributes Document;
   };


class Post
   {
   public:
      virtual Location GetStartLocation( );
      virtual Location GetEndLocation( );
      virtual Attributes GetAttributes( );
   };
```

```cpp
class Dictionary
  {
   public:
      ISRWord *OpenISRWord( char *word );
      ISREndDoc *OpenISREndDoc( );
      Location GetNumberOfWords( );
      Location GetNumberOfUniqueWords( );
      Location GetNumberOfDocuments( );
   };

class ISR
   {
   public:
      virtual Post *Next( );
      virtual Post *NextEndDoc( );
      virtual Post *Seek( Location target );
      virtual Location GetStartLocation( );
      virtual Location GetEndLocation( );
   };

class ISRWord : public ISR
   {
   public:
      unsigned GetDocumentCount( );
      unsigned GetNumberOfOccurrences( );
      virtual Post *GetCurrentPost( );
   };
```

```cpp
class ISREndDoc : public ISRWord
   {
   public:
      unsigned GetDocumentLength( );
      unsigned GetTitleLength( );
      unsigned GetUrlLength( );
   };
```

Consider these posting lists

quick	10 27 105 513 518 520
brown	28 50 62 70 514 790
fox	87 106 515 550 1200
#DocEnd	112 570 1006 1704

To read and merge these lists, we need to move from one entry to the next.

We'll do that with an ISR (index stream reader).

The ISR for each token has to be able to report its current location and attributes, and it needs Next( ) and Seek( ) functions.

OR'ing streams

quick	10 27 105 513 518 520
brown	28 50 62 70 514 790
fox	87 106 515 550 1200
#DocEnd	112 570 1006 1704
quick | fox	10 27 87 105 106 513 515 518 520 550 1200

An OR ISR simply merges the streams.

No need to pay attention to document boundaries. Each post is in whichever posting list and whatever document it happens to be.

```
class ISROr : public ISR
   {
   public:
      ISR **Terms;
      unsigned NumberOfTerms;

      Location GetStartLocation( )
         {
         return nearestStartLocation;
         }

      Location GetEndLocation( )
         {
         return nearestEndLocation;
         }

      Post *Seek( Location target )
         {
         // Seek all the ISRs to the first occurrence beginning at
         // the target location. Return null if there is no match.
         // The document is the document containing the nearest term.
         }

      Post *Next( )
         {
         // Do a next on the nearest term, then return
         // the new nearest match.
         }
```

```
   Post *NextDocument( )
      {
      // Seek all the ISRs to the first occurrence just past
      // the end of this document.
      return Seek( DocumentEnd->GetEndLocation( ) + 1 );
      }


private:
   unsigned nearestTerm;
   // nearStartLocation and nearestEndLocation are
   // the start and end of the nearestTerm.
   Location nearestStartLocation, nearestEndLocation;
};
```

AND'ing streams

quick	10 27 105 513 518 520
brown	28 50 62 70 514 790
fox	87 106 515 550 1200
#DocEnd	112 570 1006 1704
quick fox	*?*

AND'ing of terms should find occurrences of all the terms within a single document.

Should it return every possible combination, every combination only changing the nearest ISR or the first match in each matching document?

AND'ing streams

Easier to consider if we show the document boundaries.

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

quick fox *?*

To determine what document a post falls within, we advance a #DocEnd ISR to the next document end, where we can retrieve information about the document, including its length.

This tells us the start and end points of the document and whether all the word ISRs point within the same document.

AND'ing streams

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

quick fox *How many possible combinations?*
Can you reach all of them in a single pass, all ISRs only moving forward?

AND'ing of terms should find occurrences of all the terms within a single document.

Should it return every possible combination, every combination only changing the nearest ISR or the first match in each matching document?

AND'ing streams

quick	10 27 105 513 518 520
brown	28 50 62 70 514 790
fox	87 106 515 550 1200
#DocEnd	112 570 1006 1704

quick fox *How many possible combinations? 6*

Can you reach all of them in a single pass, all ISRs only moving forward? No.

Should it return every possible combination, every combination only changing the nearest ISR or the first match in each matching document?

AND'ing streams

quick	10 27 105 513 518 520
brown	28 50 62 70 514 790
fox	87 106 515 550 1200
#DocEnd	112 570 1006 1704

quick fox *How many possible combinations? 6*
Can you reach all of them in a single pass, all ISRs only moving forward? No.

Should it return every possible combination, every combination only changing the nearest ISR or the first match in each matching document?

The point of the constraint solver is to find matching pages. Once any match on the page has been found, it's the ranker's job to figure out what to do next

AND'ing streams

quick	10 27 105 513 518 520
brown	28 50 62 70 514 790
fox	87 106 515 550 1200
#DocEnd	112 570 1006 1704

quick fox *How many possible combinations? 6*
Can you reach all of them in a single pass, all ISRs only moving forward? No.

You probably want both:

Next( ) Advance the nearest ISR and look for the first match.

NextDocument( ) Seeks all the ISRs past the end of the document then looks for the first match.

AND'ing streams

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

quick fox *NextDocument( ) matches*

Next( ) Advance the nearest ISR and look for the first match.

returns (10 87) (27 87) (105 87) (105 106) (513 515) (518 515) (518 550) (520 550)

NextDocument( ) Seeks all the ISRs past the end of the document then looks for the first match.

returns (10 87) (513 515)

AND'ing streams

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

quick fox *NextDocument( ) matches*

To look for a new match, your objective is to skip forward through the index as fast as possible.

If a match is to be made including any of the present set of ISR positions, it must include whatever post is furthest down the index.

So there's no point in considering posts on the other lists that occur before the beginning of the document containing that furthest post.

AND'ing streams

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

quick fox *NextDocument( ) matches*

To look for a match:

1. Advance the #EndDoc ISR to just past the furthest ISR to get the length of the document.
2. Advance the other ISRs to their first matches starting at the beginning of the document.
3. If any ISR is past the end of document, you pick the new furthest and continue searching.

```cpp
class ISRAnd : public ISR
   {
   public:
      ISR **Terms;
      unsigned NumberOfTerms;
      Post *Seek( Location target )
         {
         // 1. Seek all the ISRs to the first occurrence beginning at
         //    the target location.
         // 2. Move the document end ISR to just past the furthest
         //    word, then calculate the document begin location.
         // 3. Seek all the other terms to past the document begin.
         // 4. If any term is past the document end, return to
         //    step 2.
         // 5. If any ISR reaches the end, there is no match.
         }
      Post *Next( )
         {
         return Seek( nearestStartLocation + 1 );
         }

   private:
      unsigned nearestTerm, farthestTerm;
      Location nearestStartLocation, nearestEndLocation;
   };
```

```cpp
class ISRAnd : public ISR
   {
   public:
      ISR **Terms;
      unsigned NumberOfTerms;
      Post *Seek( Location target )
         {
         // 1. Seek all the ISRs to the first occurrence beginning at
         //    the target location.
         // 2. Move the document end ISR to just past the furthest
         //    word, then calculate the document begin location.
         // 3. Seek all the other terms to past the document begin.
         // 4. If any term is past the document end, return to
         //    step 2.
         // 5. If any ISR reaches the end, there is no match.
         }
      Post *Next( )
         {
         return Seek( nearestStartLocation + 1 );
         }

   private:
      unsigned nearestTerm, farthestTerm;
      Location nearestStartLocation, nearestEndLocation;
   };
```

Phrase match

Terms must be in consecutive locations.

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

"quick brown fox"   (513   514   515)

Length of the match must equal to sum of the lengths of the terms.

If a match is to be made including any of the present set of ISR positions, it must include whichever post is furthest down the index.

Phrase match

Terms must be in consecutive locations.

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

"quick brown fox" (513 514 515)

Can phrase matches be overlapping?

Do you need to pay attention to document boundaries?

If it’s not a match, do all the ISRs have to move?

Phrase match

Terms must be in consecutive locations.

quick	10	27	105			513	518	520						
brown	28	50	62	70		514				790				
fox	87	106				515	550					1200		
#DocEnd					112				570		1006		1704	

"quick brown fox"   (513   514   515)

Can phrase matches be overlapping? *Yes, if beginning and ending terms match.*

Do you need to pay attention to document boundaries? *No, not if you skip a location number between documents. All phrase matches will always be within a single document.*

If it's not a match, do all the ISRs have to move? *No, you iterate, trying to move the nearest to correct position relative to the furthest.*

Phrase match

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

"quick brown fox"   (513   514   515)

So, what are the functions you might want? *Probably want both Next( ) and NextDocument( ).*

Phrase match

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

"quick brown fox" (513 514 515)

So, what are the functions you might want? *Probably want both Next( ) and NextDocument( ).*

Phrase match

quick	10	27	105			513	518	520							
brown	28	50	62	70		514				790					
fox	87	106				515	550						1200		
#DocEnd					112				570		1006			1704	

"quick brown fox"   (513   514   515)

To look for a match:

1. Pick the furthest ISR.
2. Advance the other ISRs to their first matches starting at exactly where they should appear to be a matching phrase.
3. If any ISR is past the desired location, pick the new furthest and continue searching.

```cpp
class ISRPhrase : public ISR
   {
   public:
      ISR **Terms;
      unsigned NumberOfTerms;
      Post *Seek( Location target )
         {
         // 1. Seek all ISRs to the first occurrence beginning at
         //    the target location.
         // 2. Pick the furthest term and attempt to seek all
         //    the other terms to the first location beginning
         //    where they should appear relative to the furthest
         //    term.
         // 3. If any term is past the desired location, return
         //    to step 2.
         // 4. If any ISR reaches the end, there is no match.
         }

      Post *Next( )
         {
         // Finds overlapping phrase matches.
         return Seek( nearestStartLocation + 1 );
         }
   };
```

NOTs

Terms that cannot appear in a matching document.

quick	10 27 105 513 518 520
brown	28 50 62 70 514 790
fox	87 106 515 550 1200
#DocEnd	112 570 1006 1704

brown -fox	790
-fox	*Not allowed*

A NOT matches anywhere the term doesn't appear, which is likely pretty nearly everywhere.

So we don't allow searches for nots alone and we don't check for exclusions until we've found an otherwise matching page.

Container ISRs

ISRs that must match and those that must not within a document.

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

brown -fox	790
-fox	*Not allowed*

(AND'ing is a special case of a container with no exclusion ISRs.)

```
class ISRContainer : public ISR
   {
   public:
      ISR **Contained,
          *Excluded;
      ISREndDoc *EndDoc;
      unsigned CountContained,
         CountExcluded;
      Location Next( );

      Post *Seek( Location target )
         {
         // 1. Seek all the included ISRs to the first occurrence beginning at
         //    the target location.
         // 2. Move the document end ISR to just past the furthest
         //    contained ISR, then calculate the document begin location.
         // 3. Seek all the other contained terms to past the document begin.
         // 4. If any contained erm is past the document end, return to
         //    step 2.
         // 5. If any ISR reaches the end, there is no match.
         // 6. Seek all the excluded ISRs to the first occurrence beginning at
         //    the document begin location.
         // 7. If any excluded ISR falls within the document, reset the
         //    target to one past the end of the document and return to
         //    step 1.
         };
```

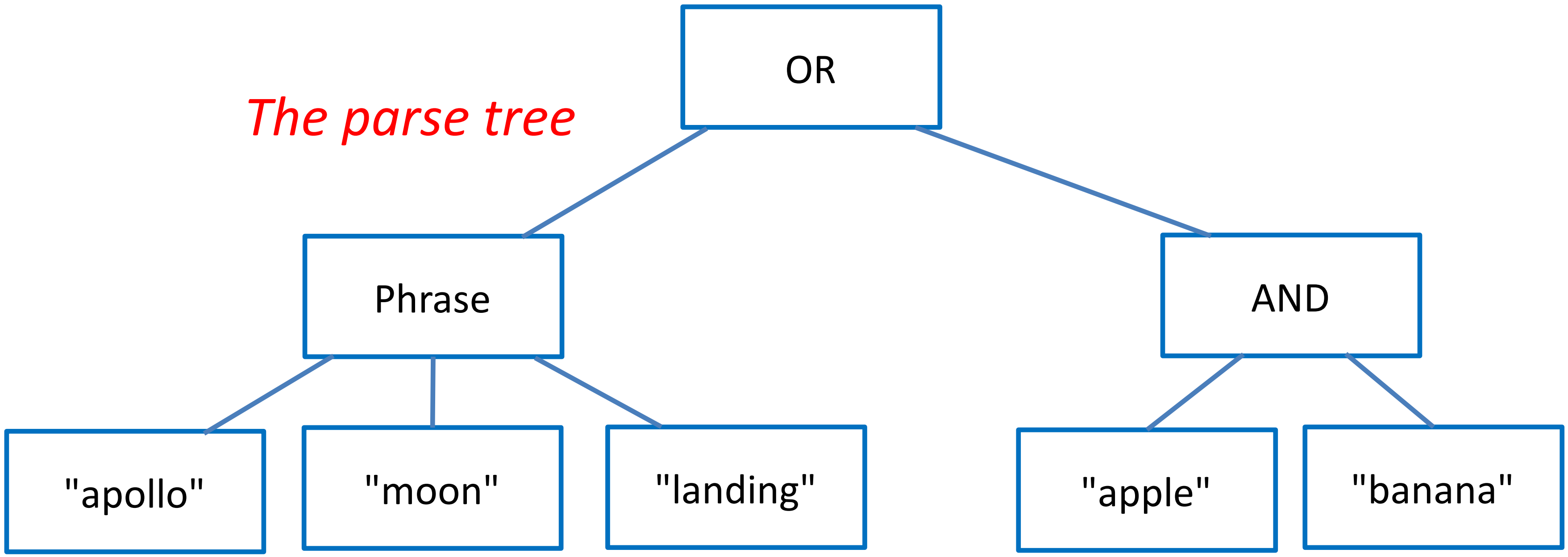
```
   Post *Next( )
      {
      Seek( Contained[ nearestContained ]->GetStartlocation( ) + 1 );
      }

private:
   unsigned nearestTerm, farthestTerm;
   Location nearestStartLocation, nearestEndLocation;
};
```

The query language and the ISRs can be recursive

"apollo moon landing" | (apple banana)

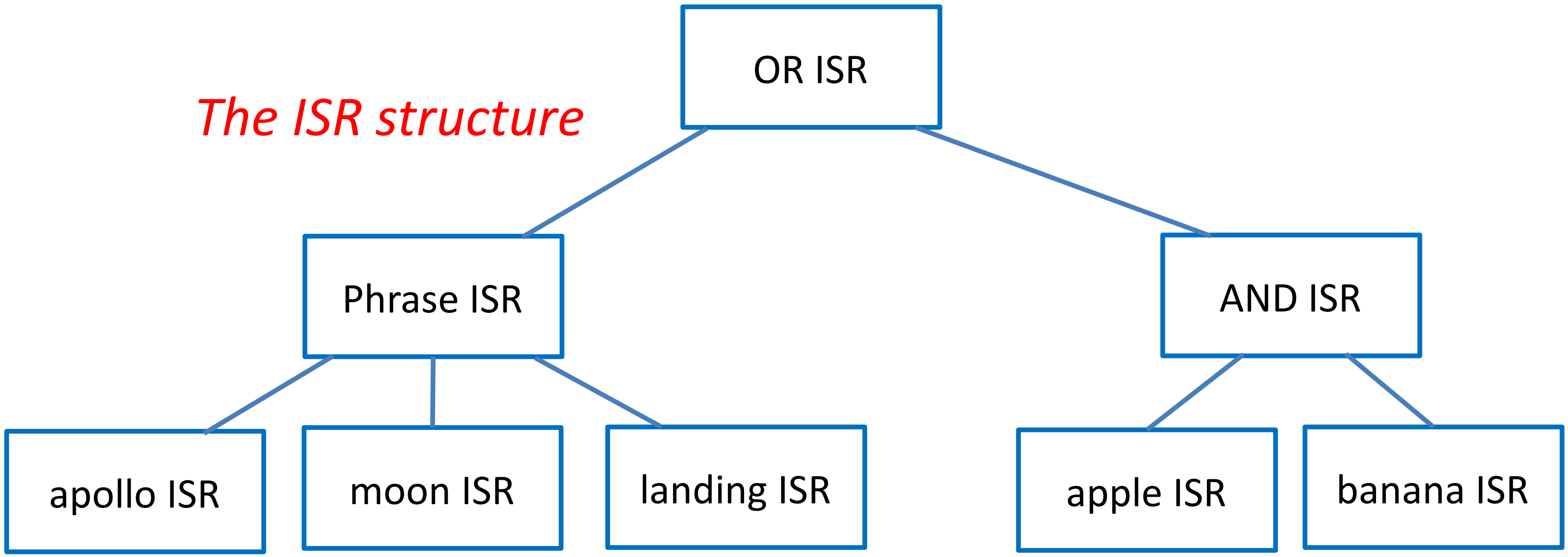
The parse tree



The query language and the ISRs can be recursive

"apollo moon landing" | (apple banana)

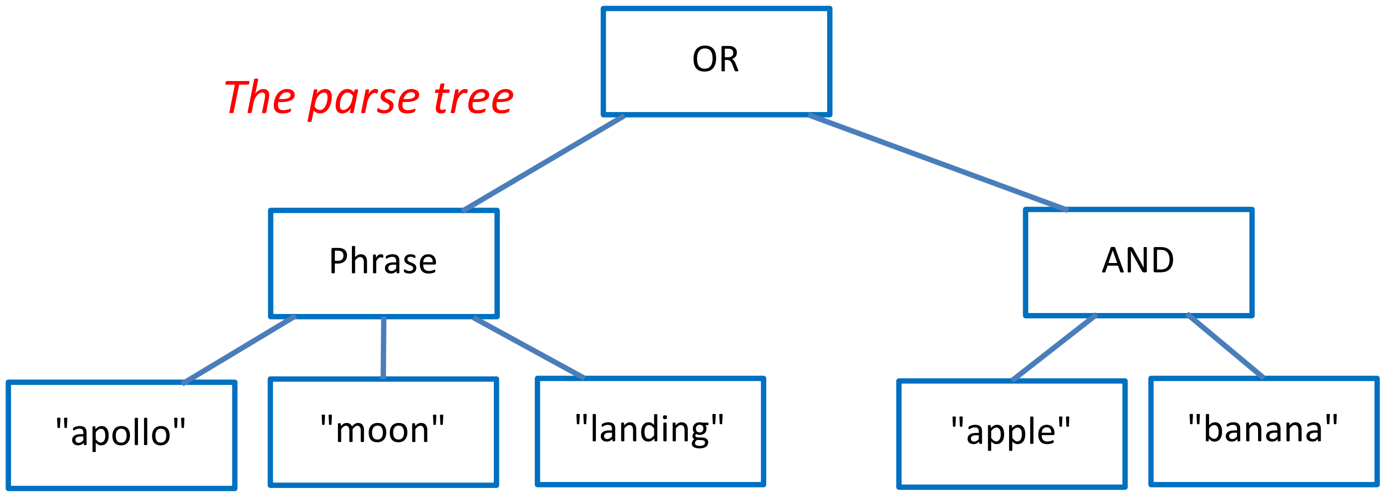
The ISR structure



"apollo moon landing" | (apple banana)

The trees are the same.

The parse tree



The ISR structure