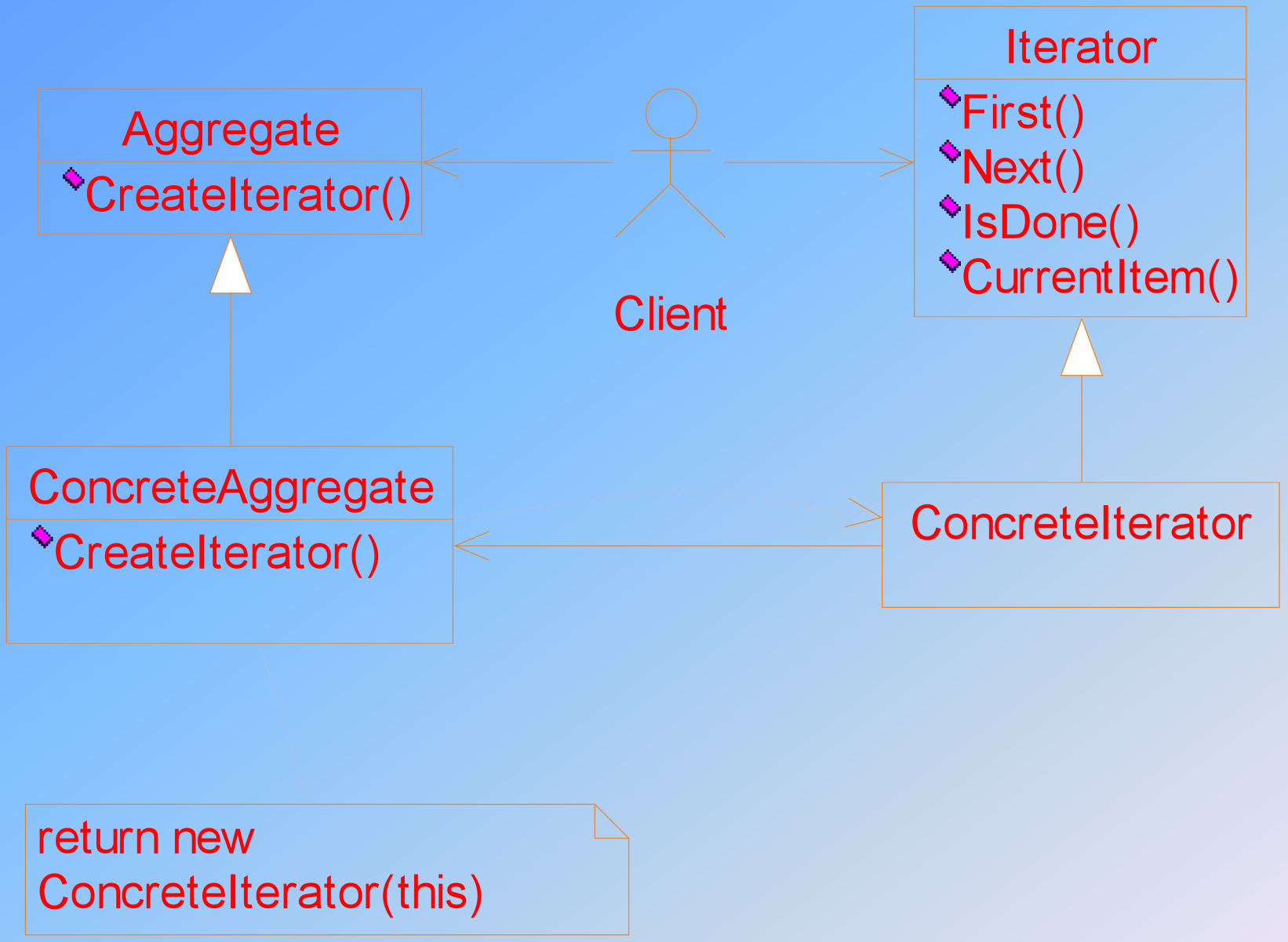
Overview

- Abstract factory pattern
  - The pattern
  - Implementation(singleton,make)
- Iterator & visitor pattern
  - Iterator
  - Proxy pattern

Outline

- Iterator Pattern
  - Robust Iterator
  - Pre-order traversing
- Visitor Pattern
- Builder Pattern

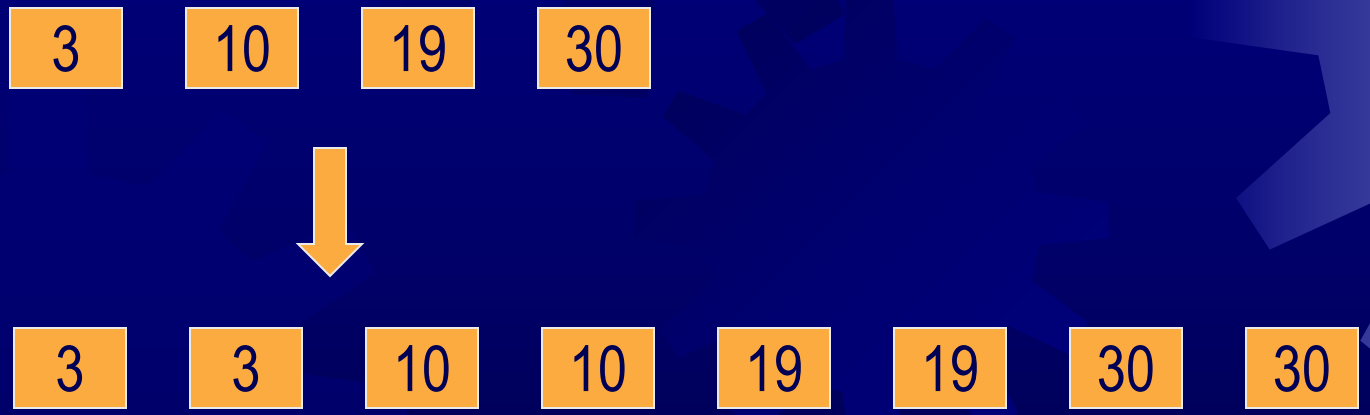


General structure of the iterator pattern

Implementation

- Robust iterator
  In general, it is dangerous to modify an aggregate object while you are traversing it.

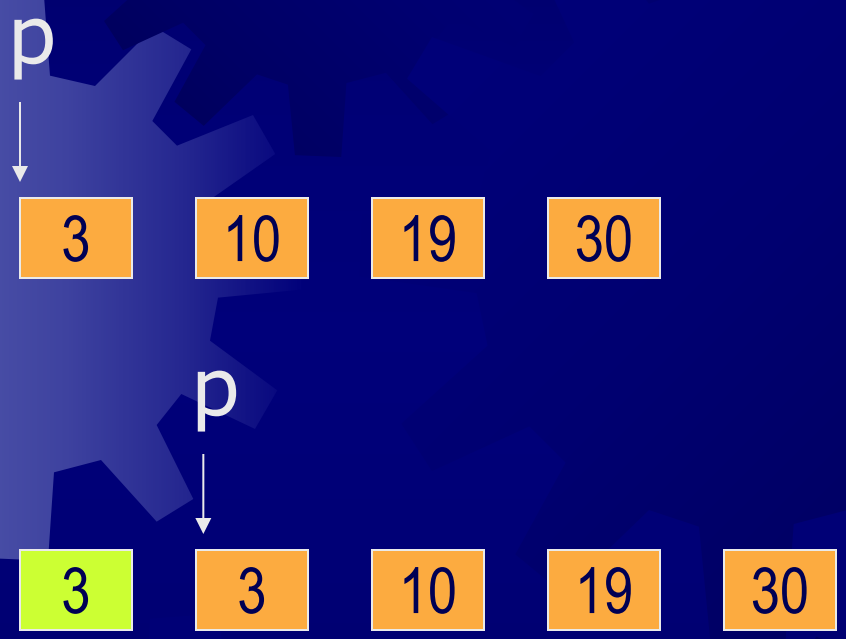
  An example:



```
#include <iostream>
#include <vector>
using namespace std;
main()
{
   vector<int> v;
   for (int i=0; i<10; i++)
         v.push_back(i);
   vector<int>::iterator p;
   for (p=v.begin(); p!=v.end(); p++)
         v.insert(p, *p);
}
```

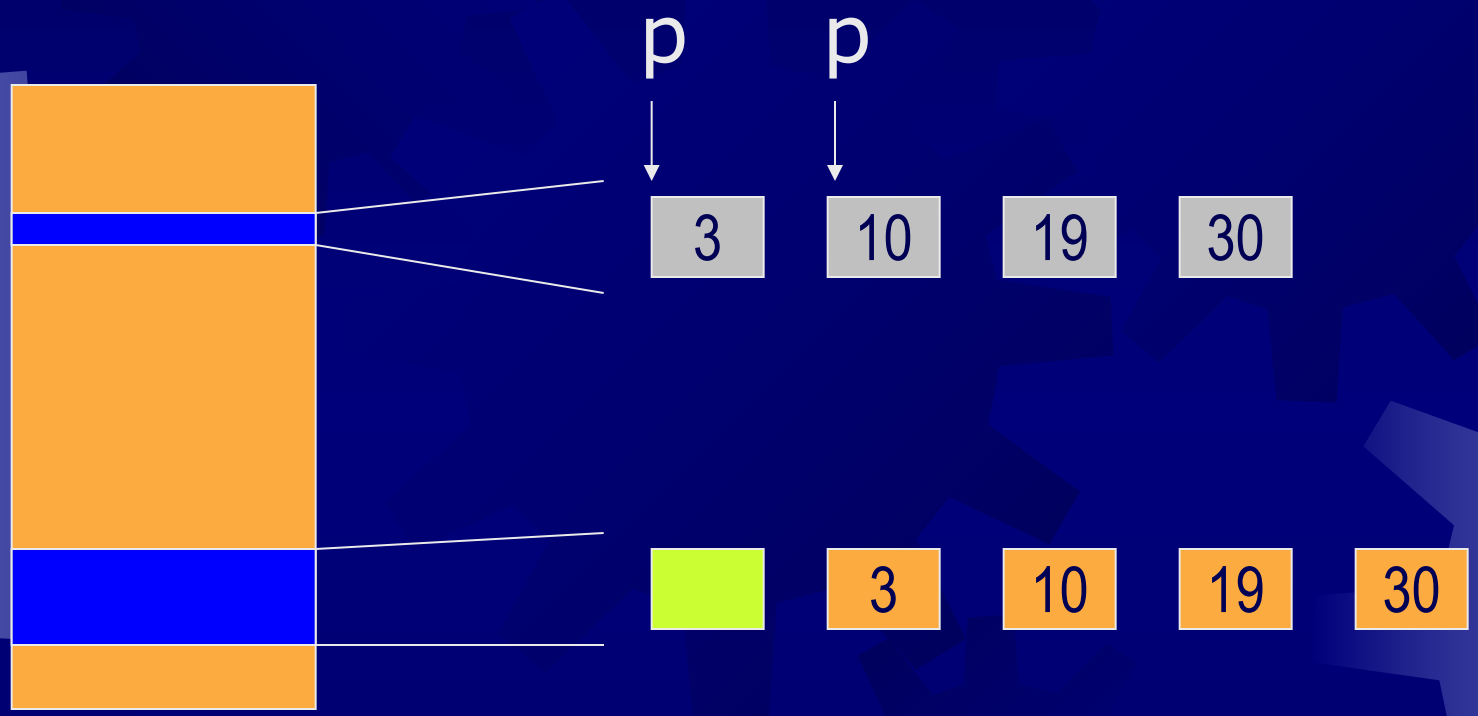
Why is the program wrong?

The first problem:before insertion, the elements after the insertion point are moved backward.



As a result, all new elements will be inserted to the head of the original sequence.

The second problem: the insertion operation may lead to re-allocation of memory

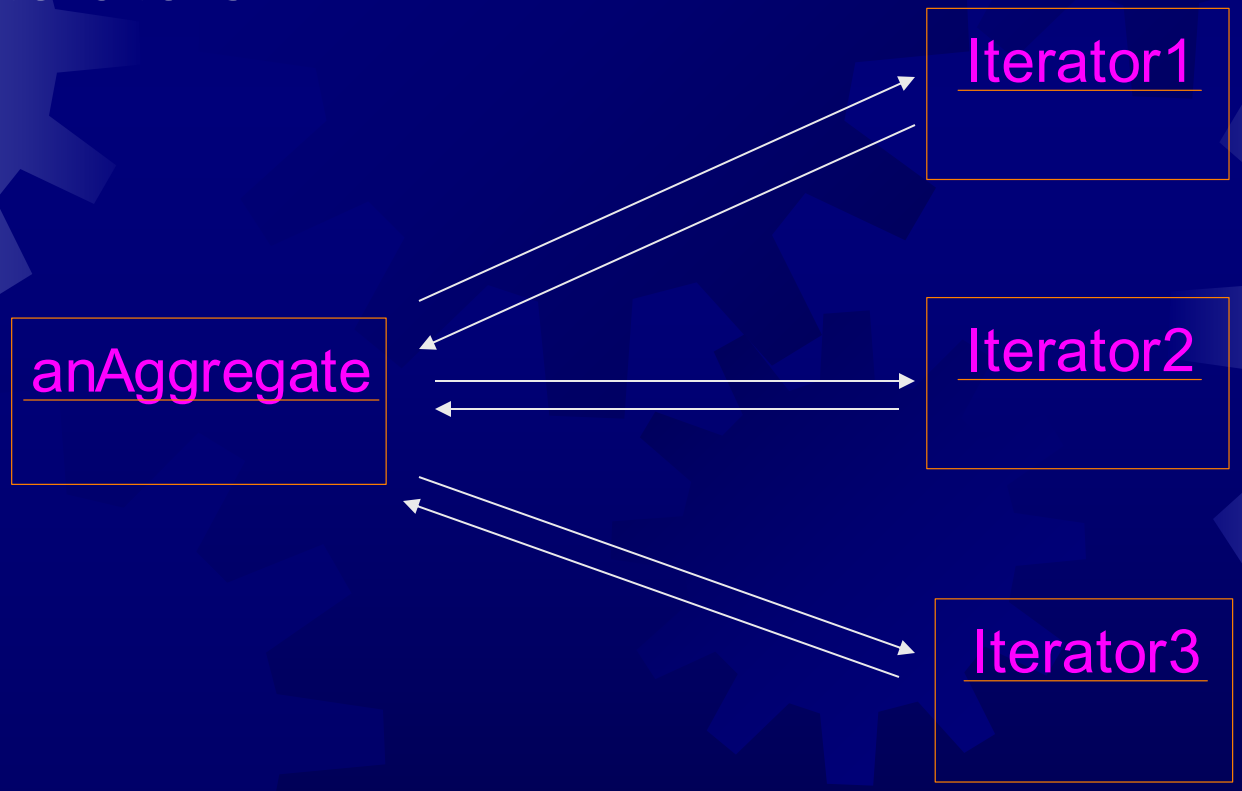


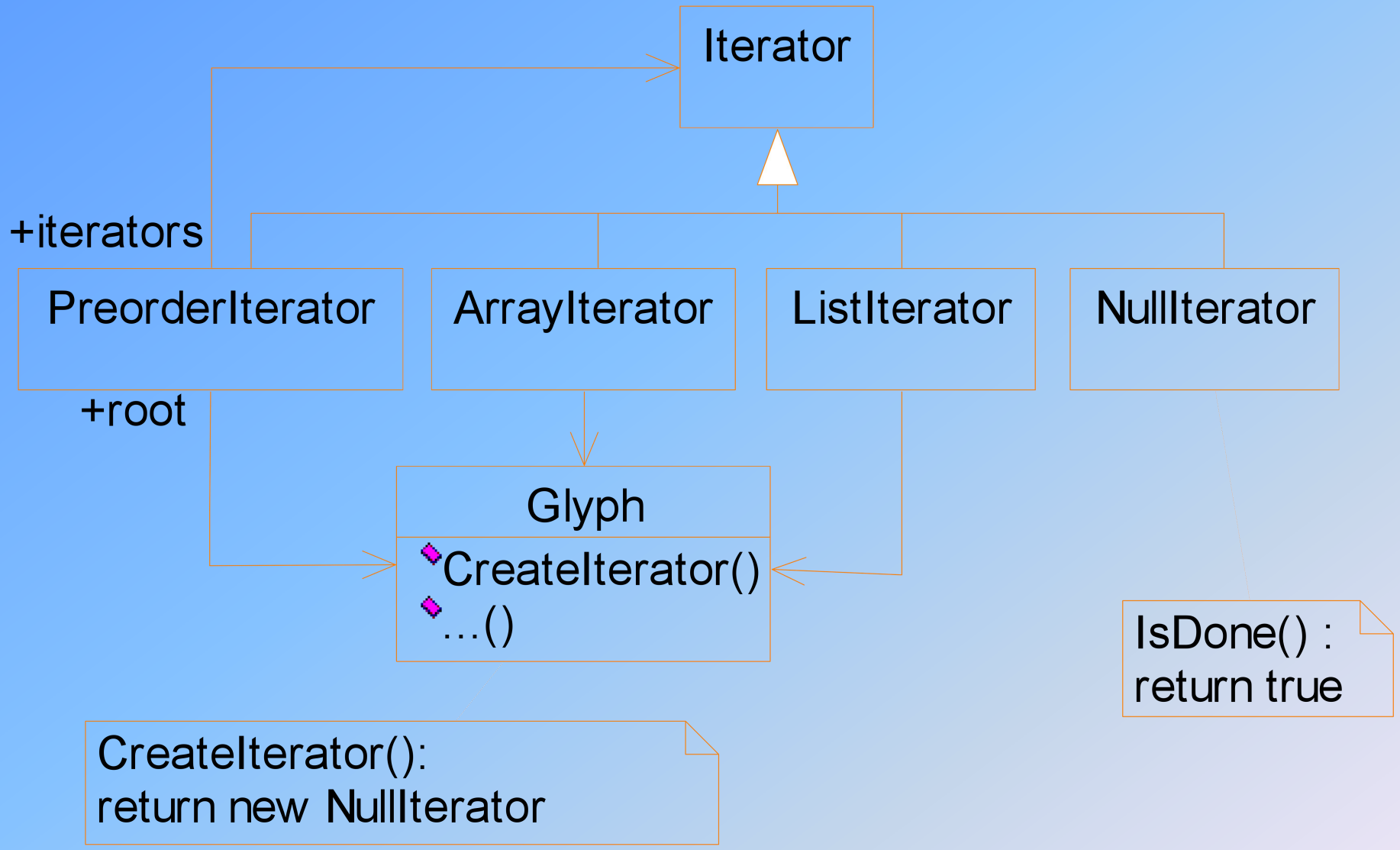
As a result, the iterator(pointer) points to an invalid memory block.

How to solve the problem?

Solution for robust iterator

When create an iterator, register it with the aggregate. On insertion and deletion, the aggregate adjust the internal state of the iterators





Applying the Iterator pattern in Lexi

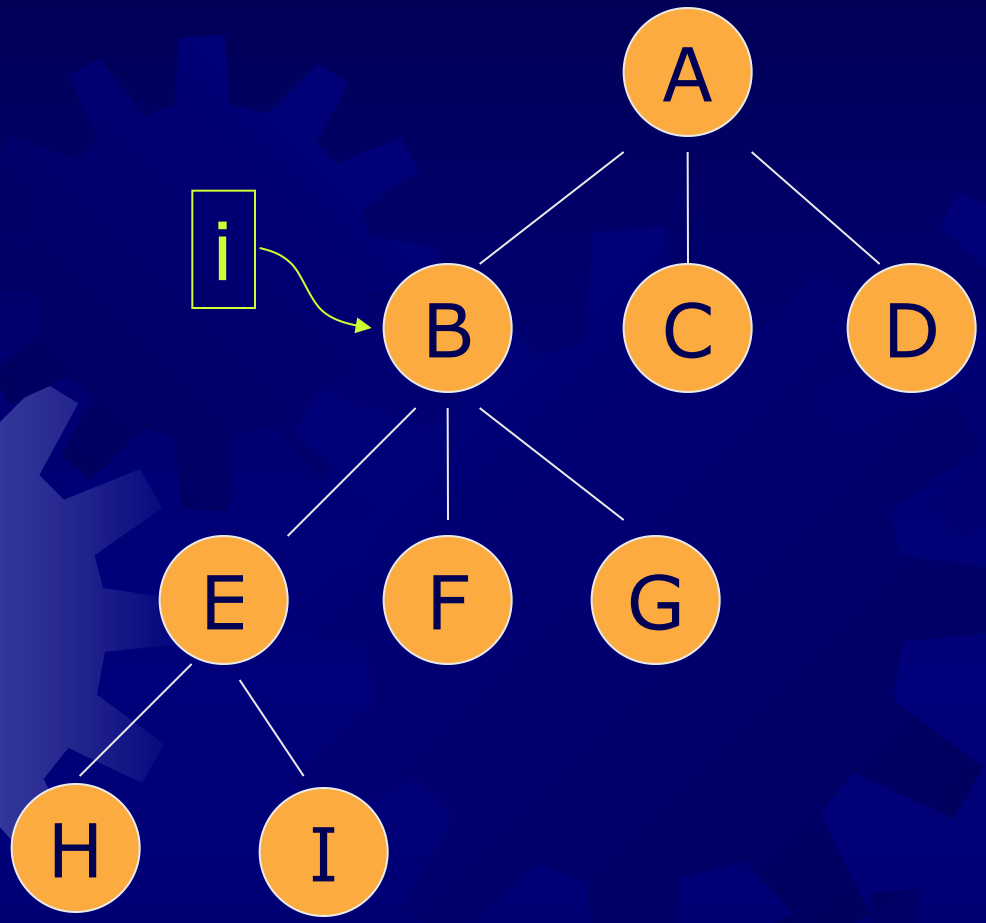
Traversing a Glyph element

```
void func( Glyph * g)
{
  Iterator<Glyph*>  *it = g->CreateIterator();
  // the Proxy pattern can be used here instead.

  for (it->First(); ! it->IsDone(); it->Next() ) {
      Glyph * child = it->CurrentItem();

      // do something with current child
  }

  delete it;
}
```



→ B

CurrentItem: B

PreorderIterator: Traversing a Glyph hierarchy tree in preorder

- What is preorder traversing?

- The “first” operation of a preorder iterator

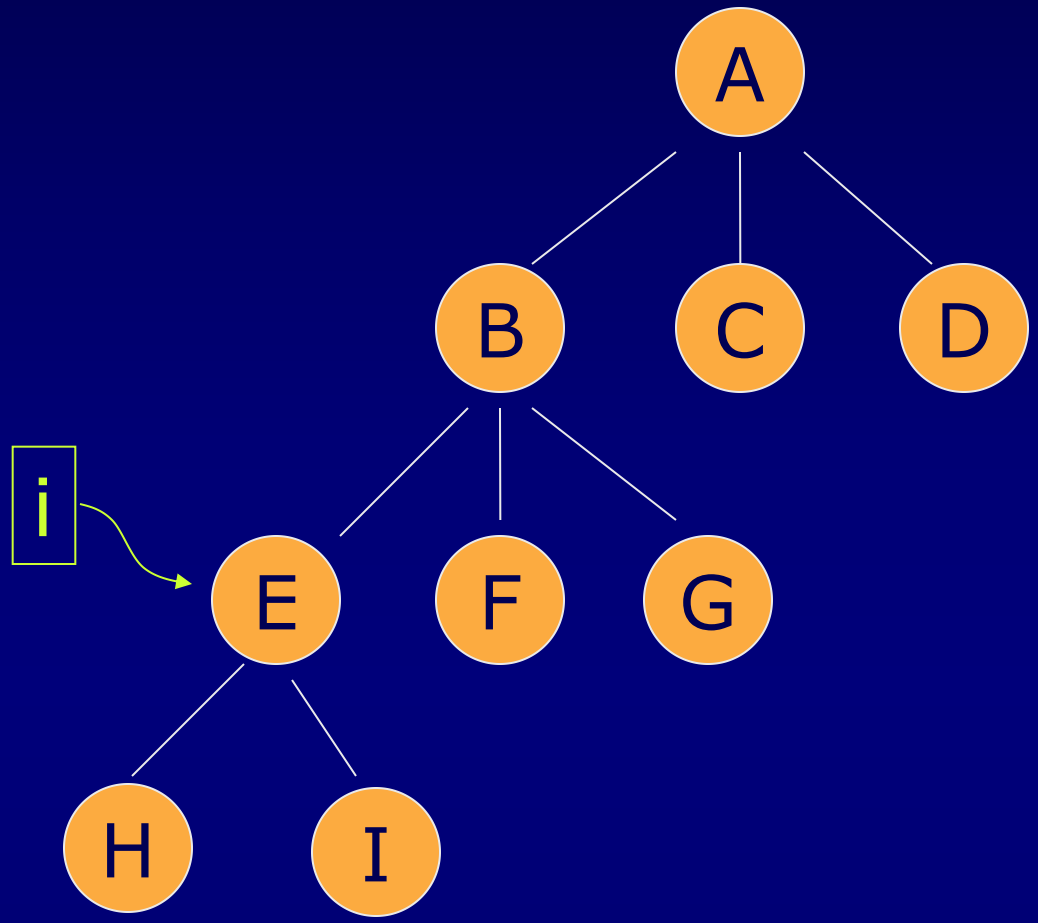
```
void PreorderIterator::First() {
    Iterator<Glyph*> * i = _root->CreateIterator();
    if (i) {
        i->First();
        _iterators.RemoveAll();
        _iterators.Push(i);
    }
}
```

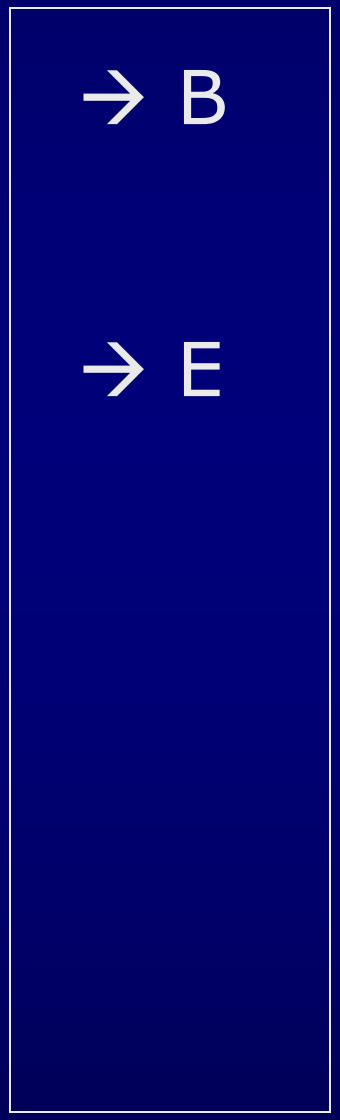
The next( ) operation of a pre-order iterator

- Use a stack to record the path previously reached: Each element of the stack is an iterator; The CurrentItems pointed by the iterators form the path
- If the current item is not a leaf node, simply let the iterator point to its left-most child.
- If we encounter a leaf node, we call Top()->Next();
- If we have searched all children of the top, delete the top, and call Top()->Next().

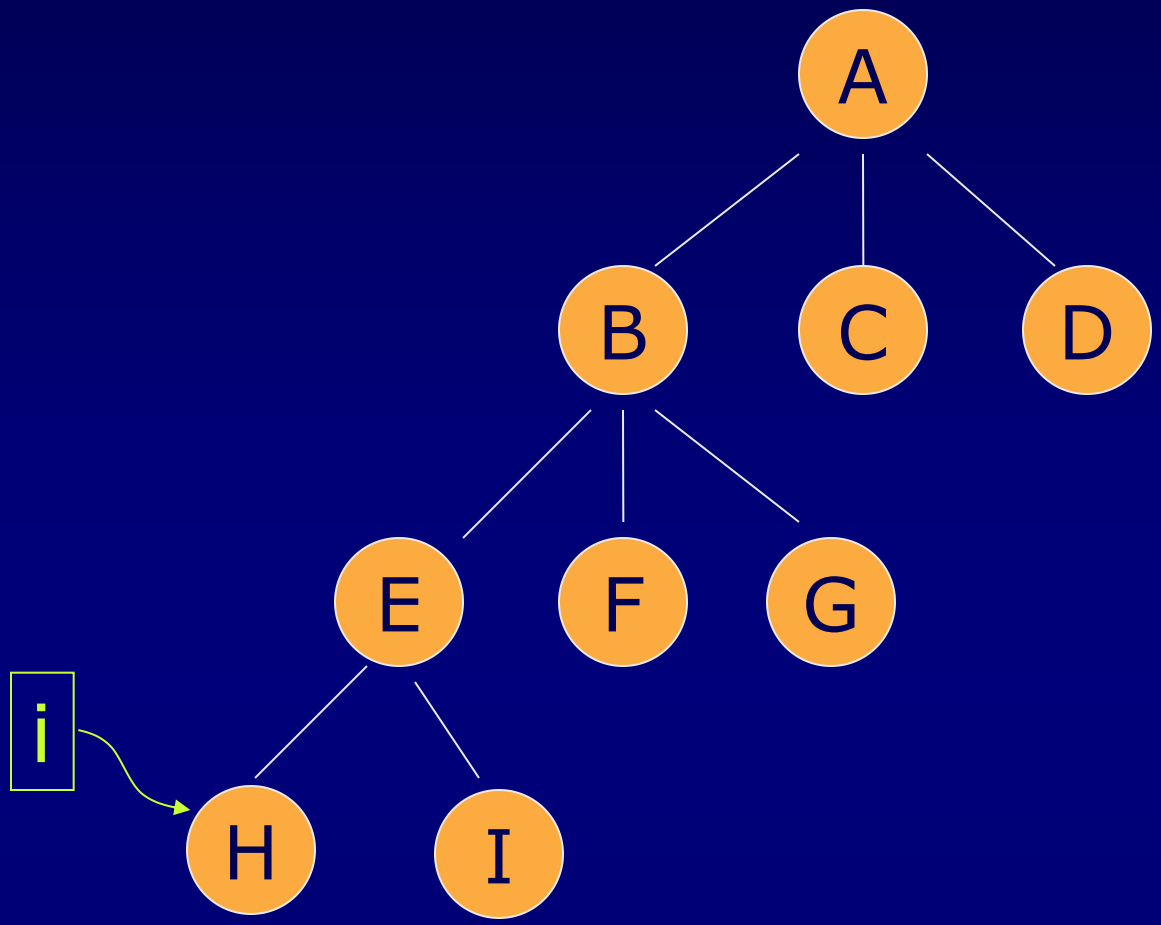
```
void PreorderIterator::Next() {
  Iterator<Glyph*> * i =
     _iterators.Top()->CurrentItem()->
         CreateIterator();
  i->First();
  _iterators.Push(i);

  while { _iterators.Size() > 0 &&
     _iterators.Top()->IsDone() ) {
         delete _iterators.Pop();
         _iterators.Top()->Next();
  }
}
```





CurrentItem: E

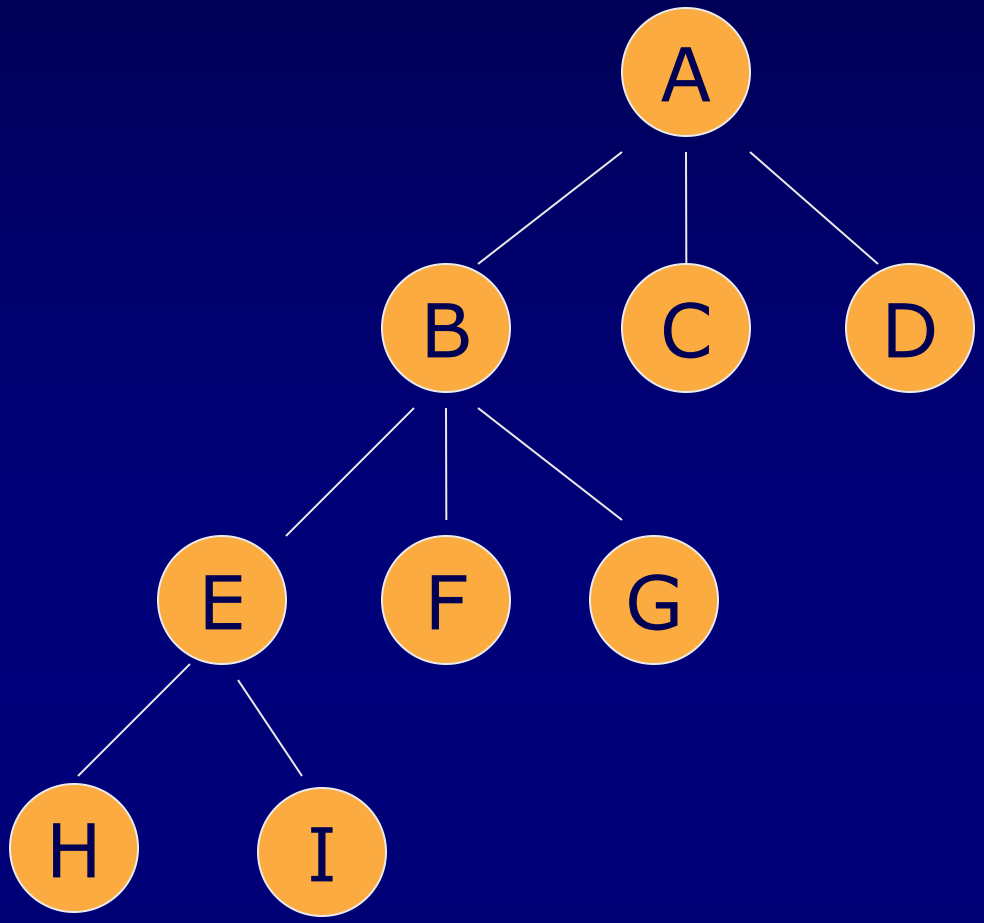


→ B

→ E

→ H

CurrentItem: H



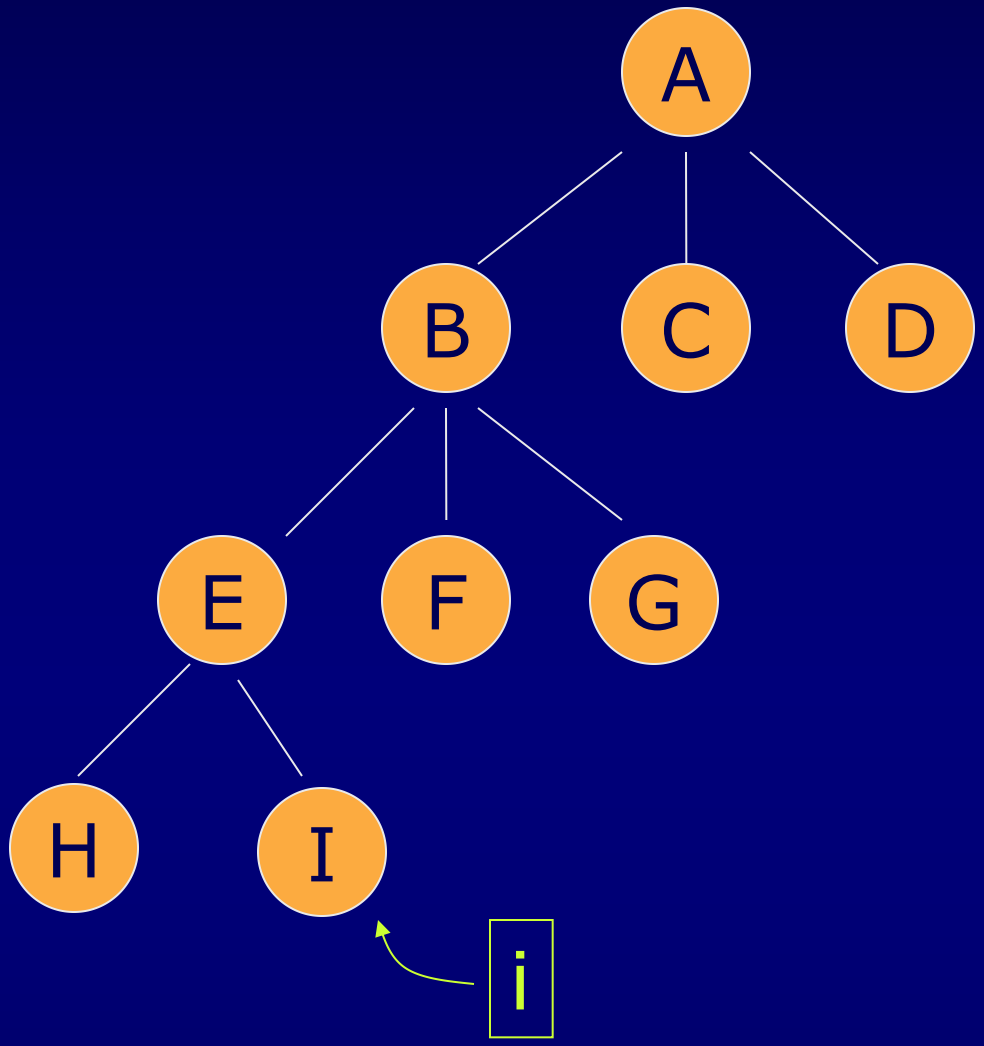
I:^

→ B

→ E

→ H

I:^

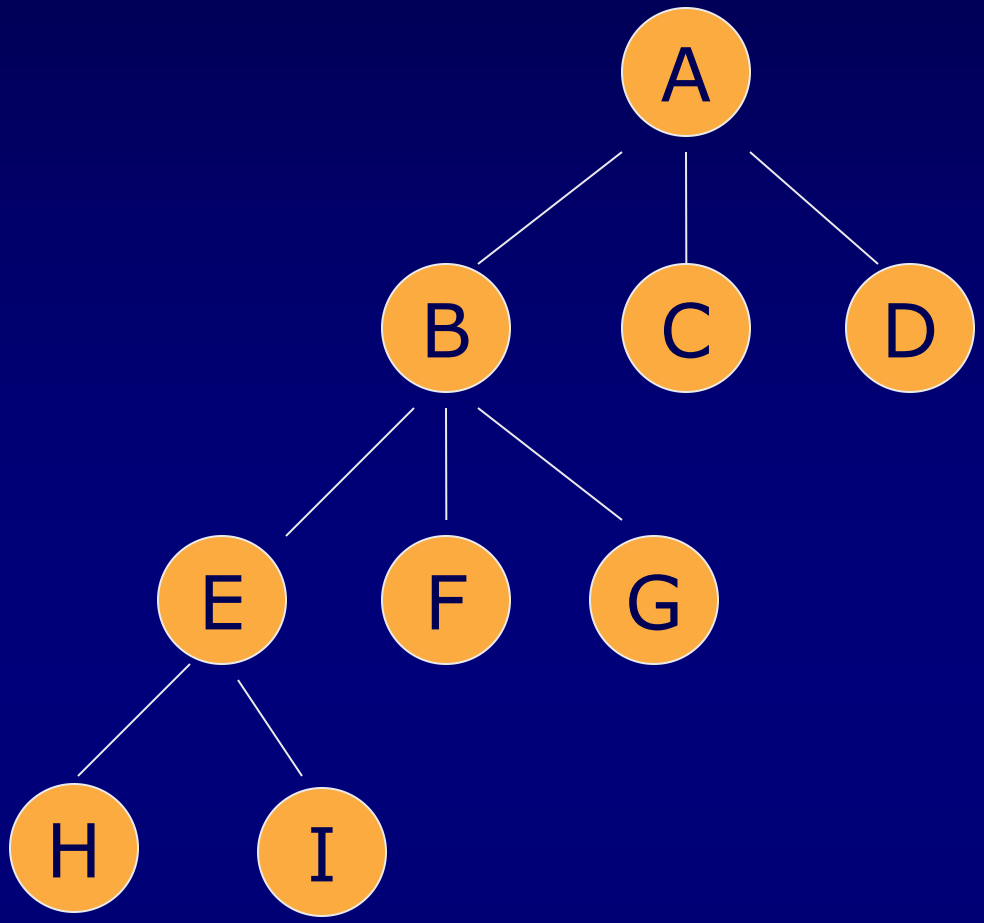


→ B

→ E

→ I

CurrentItem: I



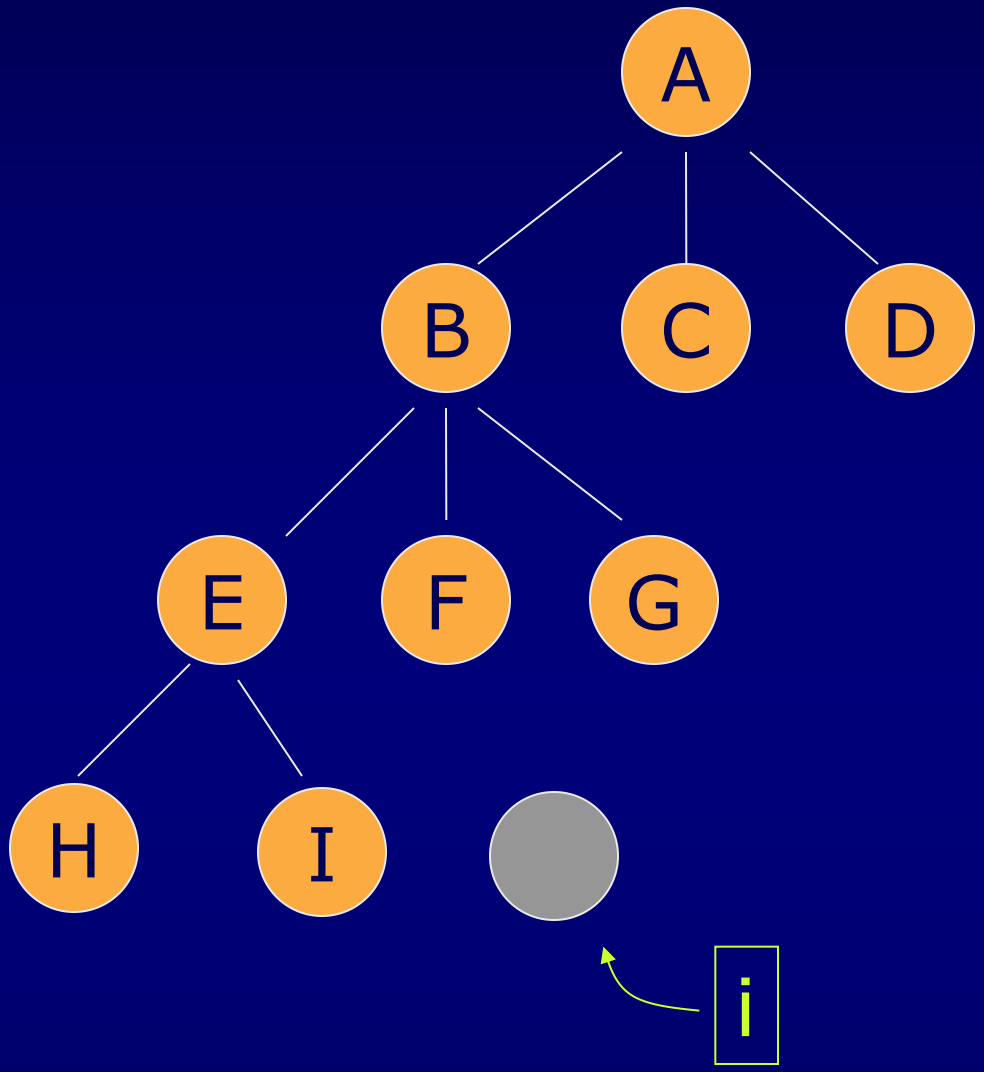
I:^

→ B

→ E

→ I

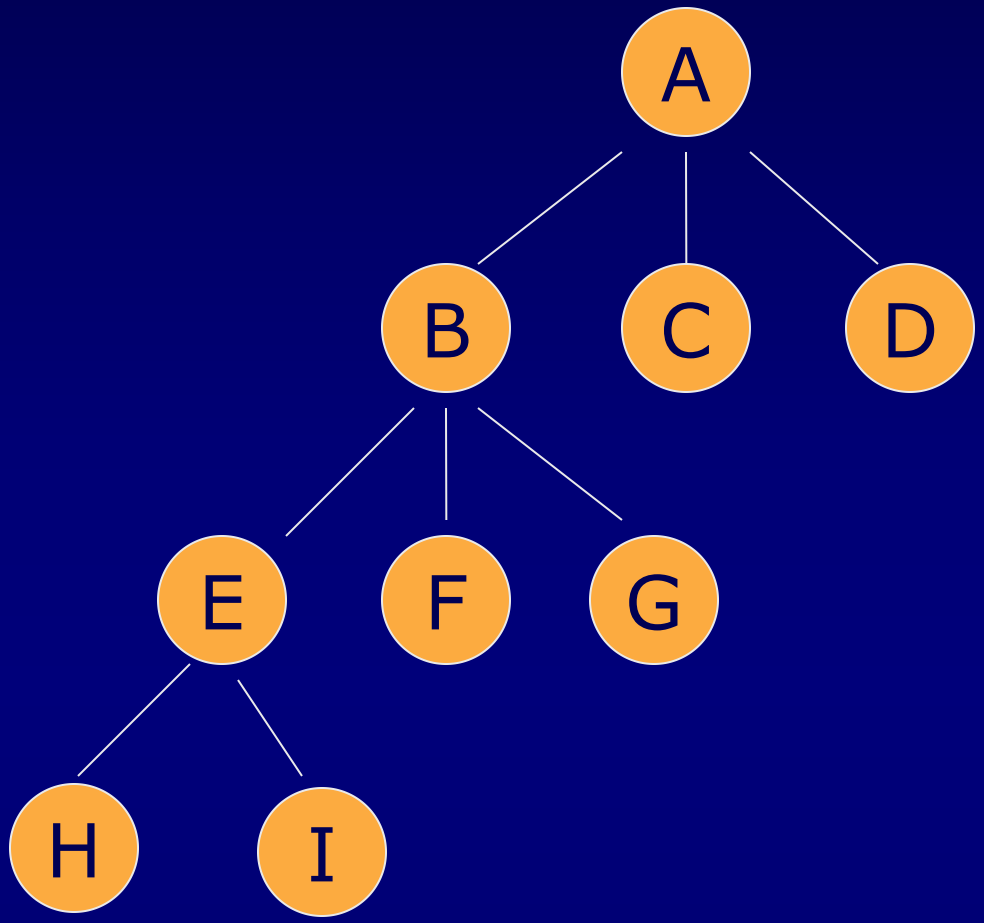
I:^



→ B

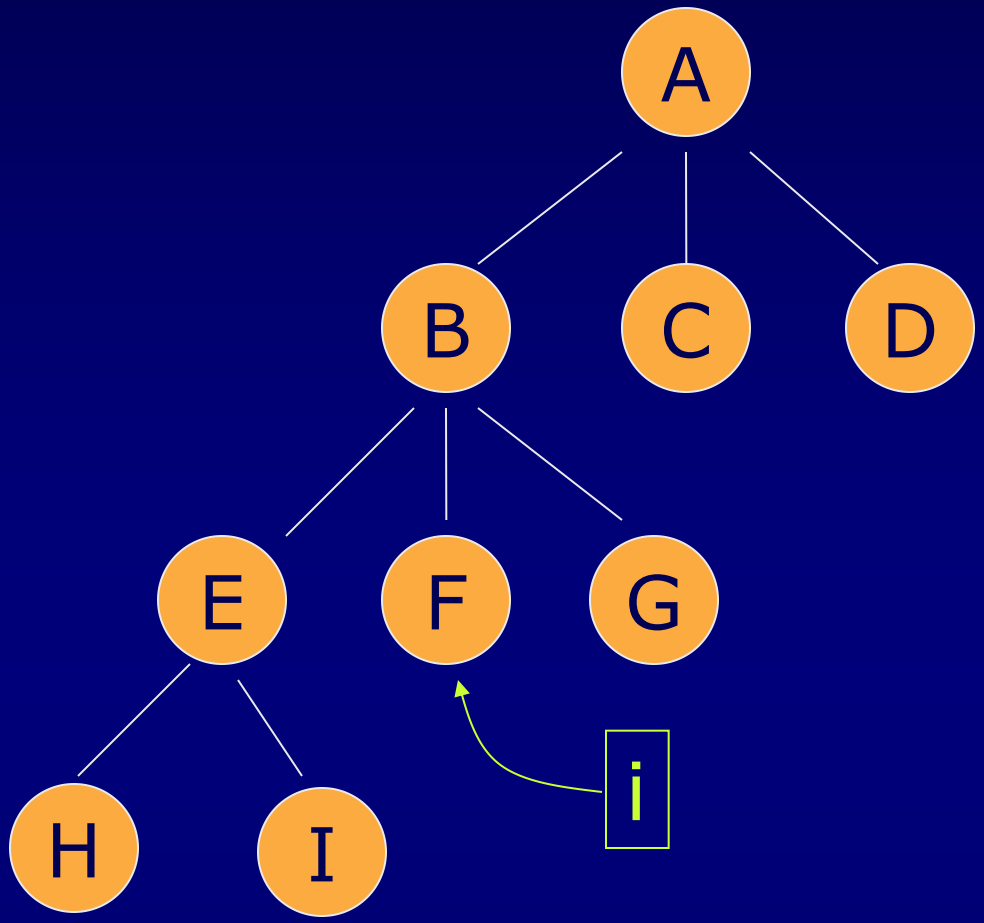
→ E

IsDone



→ B

→ E



→ B

→ F

Traversal versus Traversal actions

- Where to put the traversal actions
  e.g. spelling check, hyphenation
  - Integrate analyzers into the iterator
    ➔ Possibly there are many analyzers!
  - Separate the analyzers with the iterator.
- Different analyzers are interested with different type of aggregate objects
  E.g. spelling checker only processes characters
  ➔ how to judge the type of the aggregate

First step: encapsulating the analyzer

- During the iteration, each time we encounter a new Glyph, we **feed** it into the analyzer
- The analyzer can accumulate information (characters in this case)

Straight solution: using type-casting

```
SpellingChecker spellingChecker;
Composition *c;
Glyph * g;
PreorderIterator i(c);

for ( i.First(); ! i.IsDone(); i.Next() ) {
      g = i.CurrentItem( );
      SpellingChecker.Check( g );
}
```

```
void SpellingChecker::Check( Glyph * glyph )
{
	Character * c;
	Row * r;
	Image * i;

	if (c=dynamic_cast<Character*> (glyph)) {
		// analyze the character
	}else if (r=dynamic_cast<Row*> (glyph ) {
		// prepare to analyze r's children
	}else if (i=dynamic_cast<Image*>(glyph ) {
		// do nothing
	}
}
```

Safer solution: using special operation

- Add the operation
  void CheckMe(SpellingChecker & checker)
  into the Glyph class

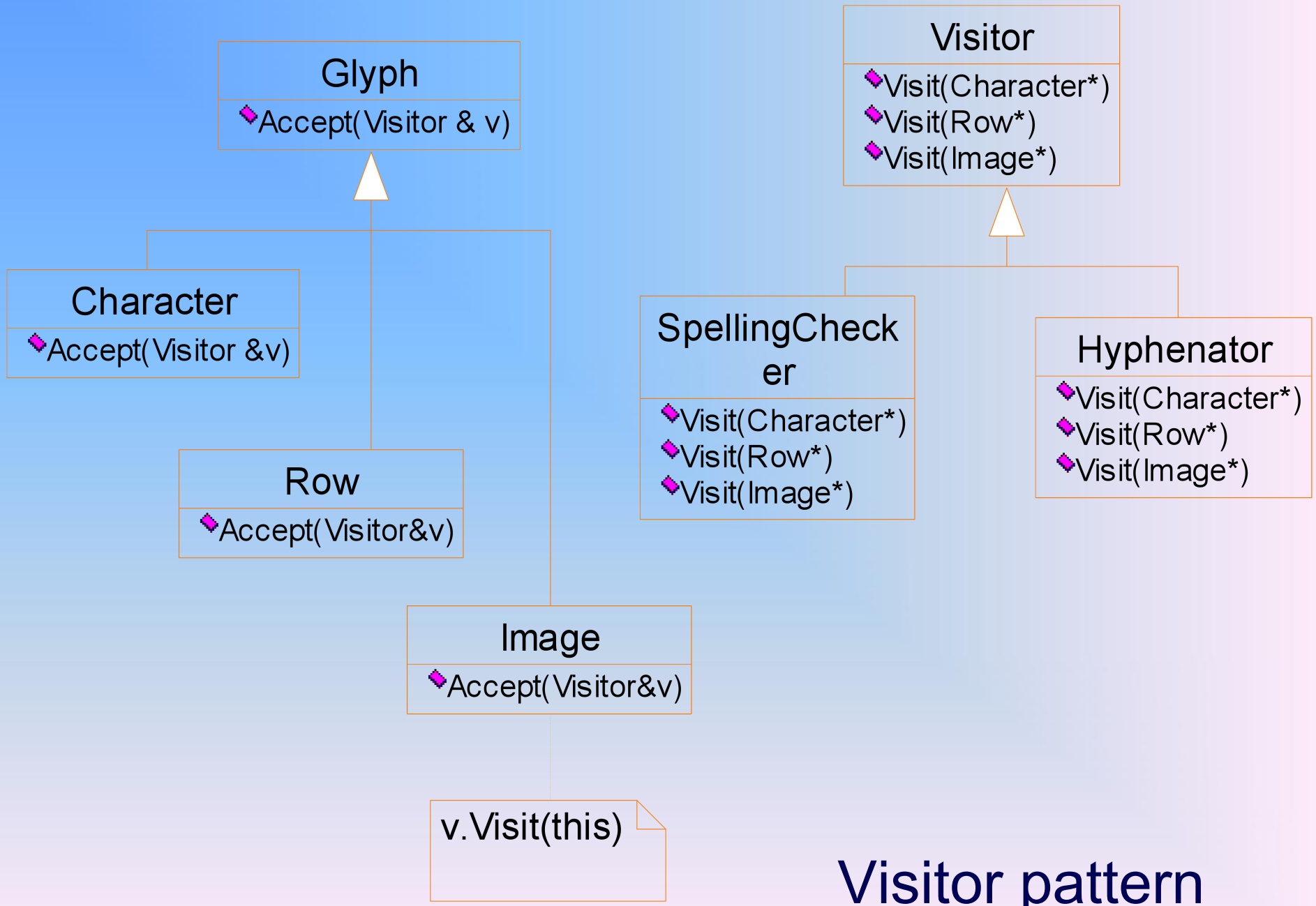
- For each subclass of Glyph, define the operation like:

```
void Subclass::CheckMe(SpellingChecker&
                                checker) {
    checker.Check(this);
}
```

```
class SpellingChecker {
public:
        SpellingChecker();
        void Check(Character*);
        void Check(Row*);
        void Check(Image*);
private:
        char _currentWord[ MAX_WORD_SIZE ];
}
```

```
SpellingChecker spellingChecker;
Composition *c;
Glyph * g;
PreorderIterator i(c);

for ( i.First(); ! i.IsDone(); i.Next() ) {
      g = i.CurrentItem( );
      g.CheckMe( spellingChecker );
}
```



Visitor pattern