Sparse arrays

About sparse arrays

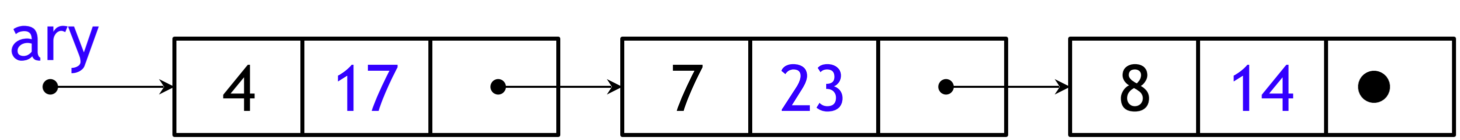
- A sparse array is simply an array most of whose entries are zero (or null, or some other default value)
- For example: Suppose you wanted a 2-dimensional array of course grades, whose rows are Penn students and whose columns are courses
  - There are about 22,000 students
  - There are about 5000 courses
  - This array would have about 110,000,000 entries
  - Since most students take fewer than 5000 courses, there will be a lot of empty spaces in this array
  - This is a big array, even by modern standards
- There *are* ways to represent sparse arrays efficiently

Sparse arrays as linked lists

- We will start with sparse one-dimensional arrays, which are simpler
  - We'll do sparse two-dimensional arrays later
- Here is an example of a sparse one-dimensional array:

	0	1	2	3	4	5	6	7	8	9	10	11
ary	0	0	0	0	17	0	0	23	14	0	0	0

- Here is how it could be represented as a linked list:



A sparse array ADT

- For a one-dimensional array of Objects, you would need:
  - A constructor: `SparseArray(int length)`
  - A way to get values from the array:
    `Object fetch(int index)`
  - A way to store values in the array:
    `void store(int index, Object value)`
- Note that it is *OK* to ask for a value from an "empty" array position
  - For an array of numbers, this should return zero
  - For an array of Objects, this should return `null`
- Additional useful operations:
  - `int length()` : return the size of the array
  - `int elementCount()` : how many non-null values are in the array
- Are there any important operations we have forgotten?

Implementing the operations

- ```
  class List { int index;          // the row number
               Object value;       // the actual data
               List next;          // the "pointer"
  ```
  - ```
    public Object fetch(int index) {
        List current = this;  // first "List" (node) in the list
        do {
            if (index == current.index) {
                return current.value;  // found correct location
            }
            current = current.next;
        } while (index < current.index && next != null);
        return null;        // if we get here, it wasn't in the list
    }
    ```
- ```
  }
  ```
- The `store` operation is basically the same, with the extra complication that we may need to insert a node into the list
  ```

Time analysis

- We must search a linked list for a given index
- We can keep the elements in order by index
- Expected time for both `fetch` and `store` is 1/2 the number of (nonzero/nonnull) elements in the list
- That is, O(n), where n is the number of actual (non-default) elements
  - For a “normal” array, indexing takes constant time
  - But for a sparse array, this isn’t bad
  - This is a typical example of a time-space tradeoff--in order to use less of one (space), we need to use more of the other (time)
- Expected time for the secondary methods, `length` and `elementCount`, is just O(1), that is, constant time
- We’re done, right?
- Unfortunately, this analysis is *correct but misleading*

What is the problem?

- True fact: In an ordinary array, indexing to find an element is the only operation we really need
- True fact: In a sparse array, we can do indexing reasonably quickly
- **False conclusion:** In a sparse array, indexing to find an element is the only operation we really need
- The problem is that in designing the ADT, we didn’t think enough about how it would be *used*

Example: Finding the maximum

- To find the maximum element in a normal array:
  - ```
    double max = array[0];
    for (int i = 0; i < array.length; i++) {
        if (array[i] > max) max = array[i];
    }
    ```
- To find the maximum element in a sparse array:
  - ```
    Double max = (Double) array.fetch(0);
    for (int i = 0; i < array.length(); i++) {
        Double temp = (Double) array.fetch(i);
        if (temp.compareTo(max) > 0) {
            max = temp;
        }
    }
    ```
- Do you see any problems with this?

Problems with this approach

- Since we tried to be general, we defined our sparse array to hold Objects
  - This means a lot of wrapping and casting, which is awkward
  - We can deal with this (especially if we use Java 1.5)
- More importantly, in a normal array, every element is relevant
- If a sparse array is 1% full, 99% of its elements will be zero
  - This is 100 times as many elements as we should need to examine
- Our search time is based on the *size* of the sparse array, not on the number of elements that are actually in it
  - And it's a *big* array (else we wouldn't bother using a sparse array)

Fixing the ADT

- Although "stepping through an array" is not a fundamental operation on an array, it is one we do frequently
  - Idiom: `for (int i = 0; i < array.length; i++) {...}`
- This is a very expensive thing to do with a sparse array
- This *shouldn't* be so expensive: We have a list, and all we need to do is step through it
  - *Poor* solution: Let the user step through the list
    - The user should not need to know anything about implementation
    - We cannot trust the user not to screw up the sparse array
    - These arguments are valid *even if the user is also the implementer!*
  - Correct solution: Expand the ADT by adding operations
    - But what, exactly, should these operations be?
    - Java *has* an answer, and it is the answer we should use

Interfaces

- An `interface`, in Java, is like a class, but
  - It contains only `public` methods (and maybe some `final` values)
  - It only *declares* methods; it doesn't *define* them
- Example:

```
public interface Iterator { // Notice: no method bodies
    public boolean hasNext( );
    public Object next( );
    public void remove( );
}
```

- This is an interface that is defined for you, in `java.util`
  - "Stepping through all the values" is something that you want to do for many data structures, so Java defines a standard way to do it
- You can write your own interfaces, using the above syntax
- So, how do you use this interface?

Implementing an interface

- To use an interface, you say you are going to implement it, then you define every method in it
- Example:

```
public class SparseArrayIterator implements Iterator {
    // any data you need to keep track of goes here
    SparseArrayIterator() { ...an interface can't tell you what
                    constructors to have, but you do need one... }
    public boolean hasNext ( ) { ...you write this code... }
    public Object next ( ) { ...you write this code... }
    public void remove ( ) { ...you write this code... }
}
```

Code for SparseArrayIterator

```
public class SparseArrayIterator implements Iterator {
    private List current; // pointer to current cell in the list
    SparseArrayIterator(List first) { // the constructor
        current = first;
    }
    public boolean hasNext() {
        return current != null;
    }
    public Object next()  {
        Object value = current.value;
        current = current.next
        return value;
    };
    public void remove() {
        // We don't want to implement this, so...
        throw new UnsupportedOperationException();
    }
}
```

Example, revisited

- Instead of:

```
Double max = (Double) array.fetch(0);
for (int i = 0; i < array.length(); i++) {
    Double temp = (Double) array.fetch(i);
    if (temp.compareTo(max) > 0) {
        max = temp;
    }
}
```

- We now need:

```
SparseArrayIterator iterator = new SparseArrayIterator(array);
Double max = (Double) array.fetch(0);
while (iterator.hasNext()) {
    temp = (Double) iterator.next();
    if (temp.compareTo(max) > 0) {
        max = temp;
    }
}
```

- Notice that we use `iterator` in the loop, not `array`

Not quite there yet...

- Our `SparseArrayIterator` is fine for stepping through the elements of an array, but...
  - It doesn't tell us *what index* they were at
  - For some problems, we may need this information
- **Solution #1:** Revise our iterator to tell us, not the value in each list cell, but the index in each list cell
  - Problem: Somewhat more awkward to use, since we would need `array.fetch(iterator.next())` instead of just `iterator.next()`
  - But it's worse than that, because `next` is defined to return an `Object`, so we would have to *wrap* the index
  - We could deal with this by overloading `fetch` to take an `Object` argument
- **Solution #2** (possibly better): Keep `SparseArrayIterator` as is, but *also* write an `IndexIterator`

IndexIterator

- For convenience, we would want `IndexIterator` to return the next index *as an* *`int`*
- This means that `IndexIterator` *cannot* implement `Iterator`, which defines `next()` to return an `Object`
  - But we can define the same methods (at least those we want)

Code for IndexIterator

```
public class IndexIterator {    // does not implement iterator
    private List current;       // pointer to current cell in the list
    IndexIterator(List first) { // constructor
        current = first;        // just like before
    }
    public boolean hasNext() {  // just like before
        return current != null;
    }
    public int next()  {
        int index = current.index; // keeps index instead of value
        current = current.next;    // just like before
        return index;              // returns index instead of value
    }
```

Wrapping the SparseArray class

- If we want a sparse array of, say, doubles, we can use the SparseArray class by wrapping and unwrapping our values
  - This is a nuisance
  - It's poor style to create another class, say SparseDoubleArray, that duplicates all our code
    - Reason: It's much easier and less error-prone if we only have to fix/ modify/upgrade our code in one place
  - But we can wrap SparseArray itself!

Code for SparseDoubleArray

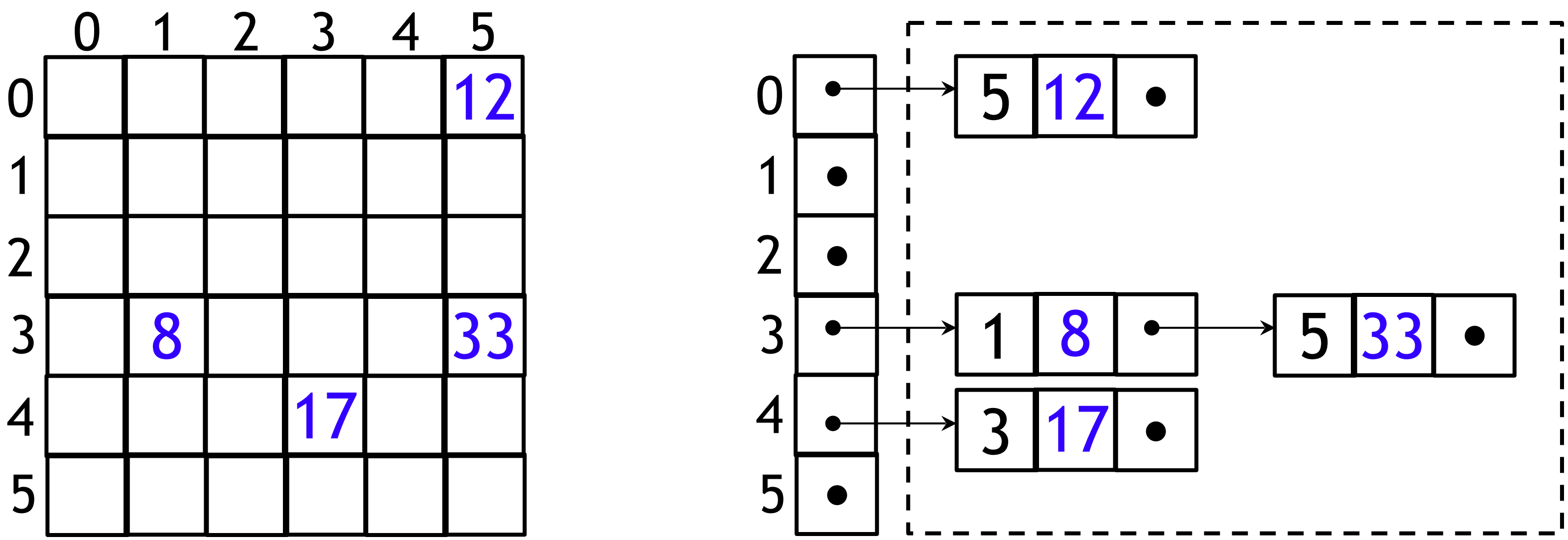
```
public class SparseDoubleArray {
    private SparseArray array;                  // the wrapped array
    public SparseDoubleArray(int size) { // the constructor
        array = new SparseArray(size);
    }
    // most methods we just "pass on through":
    public int length() { return array.length(); }
    // some methods need to do wrapping or unwrapping
    public void store(int index, double value) {
        array.store(index, new Double(value));
    }
    public double fetch(int index) {
        Object obj = array.fetch(index);
        if (obj == null) return 0.0; // gotta watch out for this case
        return ((Double) obj).doubleValue();
    }
    // etc.
```

Practical considerations

- Writing an ADT such as `SparseArray` can be a lot of work
  - We don't want to duplicate that work for `int`s, for `double`s, etc.
- If we write `SparseArray` to hold `Objects`, we can use it for anything (including suitably wrapped primitives)
- But—wrappers aren't free
  - A `Double` takes up significantly more space than a `double`
  - Wrapping and unwrapping takes time
- These costs may be acceptable if we don't have a huge number of (non-null) *elements* in our array
  - Note that what is relevant is the number of *actual values,* as opposed to the *defined size* of the array (which is mostly empty)
- Bottom line: Writing a class for `Objects` is usually the simplest and best approach, but sometimes efficiency considerations force you to write a class for a specific type

Sparse two-dimensional arrays

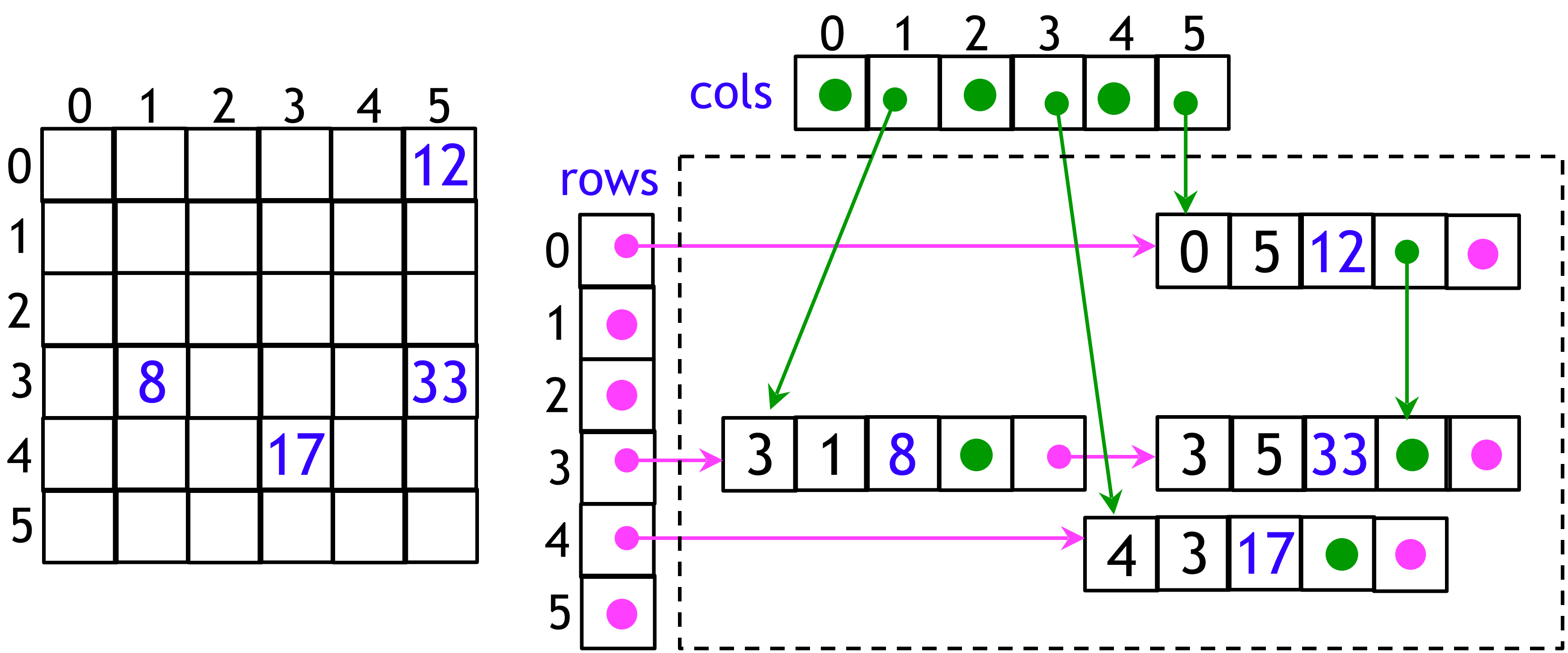
- Here is an example of a sparse two-dimensional array, and how it can be represented as an *array* of linked lists:



- With this representation,
  - It is efficient to step through all the elements of a *row*
  - It is expensive to step through all the elements of a *column*
  - Clearly, we could link columns instead of rows
  - Why not both?

Another implementation

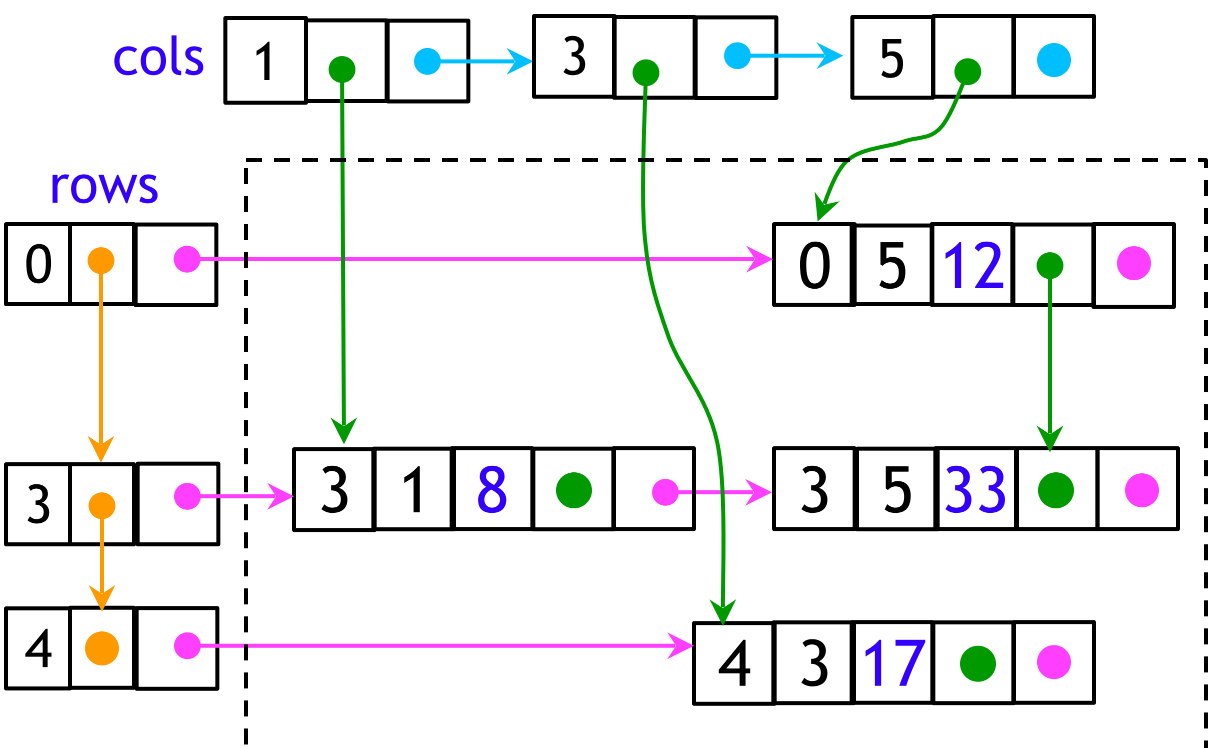
- If we want efficient access to both rows and columns, we need another array and additional data in each node



- Do we really need the row and column number in each node?

Yet another implementation

- Instead of arrays of pointers to rows and columns, you can use linked lists:



	0	1	2	3	4	5
0						12
1						
2						
3		8				33
4				17		
5						

- Would this be a good data structure for the Penn student grades example?
- This may be the best implementation if most rows and most columns are totally empty

Considerations

- You may need access only by rows, or only by columns
- You may want to access all the elements in a given row without caring what column they are in
  - In this case, you probably should use a Vector instead
- In the most general case, you would want to access by both row and column, just as in an ordinary array

Looking up an item

- The fundamental operation we need is *finding* the item at a given row `i` and column `j`
- Depending on how the array is implemented:
  - We could search a row for a given column number
  - We could search a column for a given row number
    - If we reach a list node with a higher index number, that array location must not be in the linked list
      - If we are doing a `fetch`, we report a value of zero (or `null`)
      - If we are doing a `store`, we may need to insert a cell into the linked list
  - We could choose whether to search by rows or by columns
    - For example you could keep a count of how many elements are in each row and each column (and search the shorter list)

A sparse 2D array ADT

- For a two-dimensional array of Objects, you would need:
  - A constructor:
    `Sparse2DArray(int rows, int columns)`
  - A way to store values in the array:
    `void store(int row, int column, Object value)`
  - A way to get values from the array:
    `Object fetch(int row, int column)`
- Additional useful operations:
  - A way to find the number of rows: `int getRowCount()`
  - A way to find the number of columns: `int getColumnCount()`
  - You may want to find the number of values in each row, or in each column, or in the entire array
  - You almost certainly want row iterators and column iterators

One final implementation

- You could implement a sparse array as a hash table
  - For the keys, use something like:

```
class Pair {
    private int row, column;
    Pair (int r, int c) { row = r; column = c; } // constructor
    public boolean equals(Object that) {
      return this.row == that.row && this.column == that.column;
    }
    public int hashCode( ) {
      return row + column;
    }
  }
```

- What are the advantages and disadvantages of this approach?

Summary

- One way to store sparse arrays is as linked lists
- A good ADT provides all the operations a user needs
  - The operations should be *logically complete*
  - They also need to be the *right operations* for *real uses*
- Java `interface`s provide standardized and (usually) well-thought out skeletons for solving common problems
- It is usually best and most convenient to define ADTs for `Object`s rather than for a specific data type
  - Primitives can be wrapped and used as Objects
  - For even more convenience, the ADT itself can be wrapped
  - The extra convenience also buys us more robust code (because we don’t have duplicate almost-the-same copies of our code)
  - Extra convenience comes at a cost in efficiency

The End